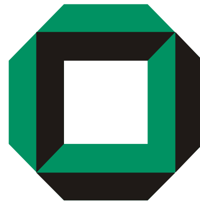
Universität Karlsruhe – Fakultät für Informatik – Bibliothek – Postfach 6980 – 76128 Karlsruhe

How dark should a component black-box be?
–
Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007), July, 31st, 2007, Berlin, Germany

Herausgeber:
Ralf Reussner, Clemens Szyperski,
Wolfgang Weck

Interner Bericht 2007-13



Universität Karlsruhe
Fakultät für Informatik

ISSN 1432 - 7864

Preface of the Workshop on Component-Oriented Programming (WCOP) 2007

WCOP 2007 is the twelfth event in a series of highly successful workshops, which took place in conjunction with every ECOOP since 1996.

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 recent years, including component technology standards, such as CORBA/CCM, COM/COM+, J2EE/EJB, .NET, and most recently software services, but also the increasing appreciation of software architecture for component-based systems, as in SOA, and 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 the end users, who are not able or willing to change components. This requires standards to allow independently created components to interoperate, and specifications that put the composer into the position to decide what can be composed under which conditions. 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.

After WCOP '96 focused on the fundamental terminology of COP, the subsequent workshops expanded into the many related facets of component software.

WCOP 2007 will discuss the black-box nature of components. On the one hand, for many, components became synonymously with the black-box building blocks of software. Technically, this means a component is described by the interfaces it provides and requires. On the other hand, for many reasons, an abstract description of specific aspects of the component's behaviour in addition to the mere interface specification is needed. These reasons include architectural dependency analysis, the description of non-functional properties or the verification of the absence of deadlocks. Therefore, in WCOP 2007 we explicitly ask for position statements discussing work related to the question:

"How dark should a component black-box be?"

This includes position statements dealing with components or component-based systems or component infrastructures that explicitly make use of information on components beyond mere provides and requires interfaces.

WCOP 2007 accepted 10 papers, which are organised according to the program below. The organisers are looking forward to an inspiring and thought-provoking workshop. The organisers like to 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
Probusweg 9
CH-8057 Zürich, Switzerland
E-mail: mail "at" wolfgang-weck.ch
Web: http://www.wolfgang-weck.ch

Programme WCOP 2007

9:00h Opening of WCOP by Organisers
9:15h Session I: Model-Driven Development and Adaptation of Components

- Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0 ?
  Javier Cubo (1), Gwen Salaün (1), Carlos Canal (1), Ernesto Pimentel (1), Pascal Poizat (2,3)
  1 University of Malaga, Spain; 2 Université d'Evry Val d'Essonne, France; 3, INRIA Rocquencourt, France

- On the benefits of using model transformations to describe components design process
  Eveline Kaboré and Antoine Beugnard
  ENST Bretagne, France

9:45: Session II: Component Performance Prediction

- Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions
  Klaus Krogmann
  Universität Karlsruhe (TH), Germany

- Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs
  Michael Kuperberg and Steffen Becker
  Universität Karlsruhe (TH), Germany

10:15 Session III: Aspects and Components

- Aspectual Dependencies: Towards Pure Black-Box Aspect-Oriented Composition in Component Models
  Bert Lagaisse and Wouter Joosen
  K.U.Leuven, Belgium

10:30h Coffee Break
11:00h Session III continued

- A Seamless Extension of Components with Aspects using Protocols
  Angel Núñez, Jacques Noyé
  EMN-INRIA, Nantes, France

- AOCI: An Aspect-Oriented Component Infrastructure
  Guido Söldner and Rüdiger Kapitza
  University of Erlangen-Nürnberg, Germany

11:30h Session IV: Component Nature:

- How dark should a component black-box be? The Reuseware Answer
  Jakob Henriksson and Florian Heidenreich and Jendrik Johannes and Steffen Zschaler and Uwe Aßmann
  Technische Universität Dresden, Germany

- Black & White, Never Grey: On Interfaces, Synchronization, Pragmatics, and Responsibilities
  Franz Puntigam
  Technische Universität Wien, Austria

- Components have no Interfaces!
  Richard Rhinelander
  University of Kitara, Australia

12:15h Planning of Break out Groups
12:30h Lunch break
14:00h Discussion in Break out Groups
15:30h Coffee Break
16:00h Presentations of Break Out Groups
16:45h Closing of WCOP
17:00h End of the 12th WCOP

Table of Contents

Session 1: Model-Driven Development and Adaptation of Components

Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0?
Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel, Pascal Poizat 9

On the benefits of using model transformations to describe components design process 14
Eveline Kaboré, Antoine Beugnard

Session 2: Component Performance Prediction

Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions
Klaus Krogmann 23

Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs
Michael Kuperberg, Steffen Becker 30

Session 3: Aspects and Components

Aspectual Dependencies: Towards Pure Black-Box Aspect-Oriented Composition in Component Models
Bert Lagaisse, Wouter Joosen 36

A Seamless Extension of Components with Aspects using Protocols
Angel Núñez, Jacques Noyé 46

AOCI: An Aspect-Oriented Component Infrastructure
Guido Söldner and Rüdiger Kapitza 53

Session 4: Component Nature

How dark should a component black-box be? The Reuseware Answer
Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, Uwe Aßmann 58

Black & White, Never Grey: On Interfaces, Synchronization, Pragmatics, and Responsibilities
Franz Puntigam 63

Components have no Interfaces!
Richard Rhinelander 68

Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0

Javier Cubo, Gwen Salaün,
Carlos Canal, Ernesto Pimentel
Dept. of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
Emails: {cubo,salaun,canal,ernesto}@lcc.uma.es

Pascal Poizat
INRIA/ARLES Project-Team, France, and
IBISC FRE 2873 CNRS – Université d'Évry, France
Email: pascal.poizat@inria.fr

***Abstract*—In this paper, we propose to relate model-based adaptation approaches with the Windows Workflow Foundation (WF) implementation platform, through a simple case study. We successively introduce a client/server system with mismatching components implemented in WF, our formal approach to work mismatch cases out, and the resulting WF adaptor. We end with some conclusions and a list of open issues.**

I. Introduction

Software Adaptation [1] is a promising research area which aims at supporting the building of component systems [2] by reusing software entities. These can be adapted in order to fit specific needs within different systems. In such a way, application development is mainly concerned with the selection, adaptation and composition of different pieces of software rather than with the programming of applications from scratch. Many approaches dedicated to model-based adaptation [3], [4], [5], [6], [7] focus on the behavioural interoperability level, and aim at generating new components called *adaptors* which are used to solve mismatch in a non-intrusive way. This process is completely automated being given an *adaptation mapping* which is an abstract description of how mismatch can be solved with respect to behavioural interfaces of components. However, very few of these approaches relate their results with existing programming languages and platforms. To the best of our knowledge, the only attempts in this direction have been carried out using COM/DCOM [5] and BPEL [8].

In this paper, we propose to relate adaptor generation proposals with existing implementation platforms. BPEL [9] and Windows Workflow Foundation (WF) [10] are very relevant platforms because they support the behavioural descriptions of components/services. Implementing BPEL services is possible with the Java Application Server included in Netbeans Enterprise. On the other hand, WF belongs to the .NET Framework 3.0 developed by Microsoft®. Here, we have chosen WF to achieve our goal because the .NET Framework is widely used in private companies whereas BPEL is a language that recently emerged and for which tool support is being released. In addition, WF can be used to implement Web services, as it is the case for BPEL, but also any kind of software component. WF makes the implementation of services easier thanks to its workflow-based graphical support. Last, by using with WF, most of the code is automatically generated, which is not the case with BPEL platforms.

The remainder of the paper is organised as follows. We give a quick overview of WF in Section II. We present in Section III a simple example of on-line computer sale, and the WF components on which it will rely on. In Section IV, we apply successively the main steps that are necessary to compose and adapt these WF components: extraction of behavioural interfaces from WF workflows, mismatch detection, writing of the mapping, generation of adaptor protocol, and implementation of the adaptor component from its abstract description. In Section V, we draw up some conclusions, and discuss issues that we will tackle in future work.

II. WF Overview

In this section we present the WF constructs that we use in this work: `Code`, `Terminate`, `InvokeWebService`, `WebServiceInput`, `WebServiceOutput`, `Sequence`, `IfElse`, `Listen` with `EventDriven` activities, and `While`. The reader interested in more details may refer to [10].

WF belongs to the .NET Framework 3.0, and is supported by Visual Studio 2005. The available programming languages to implement the workflows in Visual Studio 2005 are *Visual Basic* and $C\#$. In this work, $C\#$ has been chosen as the implementation language.

The `Code` activity is meant to execute user code provided for execution within the workflow. The `Terminate` activity is used to finalise the execution of a workflow. A WF `InvokeWebService` activity calls a Web service and receives the requested service result back. If such an invoke has to be accessed by another component C, it has to be preceded by a `WebServiceInput` activity, and followed by a `WebServiceOutput` activity. Hence, C will interact with this new service using these two input/output activities that enable and disable the data reception and sending, respectively, with respect to the invoked Web service. WF-based XML Web

services require at least one WebServiceInput and one or more WebServiceOutput activities. The input and output activities are related, thus each output activity must be associated with an input activity. It is not possible to have an instance of WebServiceInput without associated outputs, as well as having outputs without at least one WebServiceInput.

The Sequence construct executes a group of activities in a precise order. The WF IfElse activity corresponds to an *if-then-else* conditional expression. Depending on the condition evaluation, the IfElse activity launches the execution of one of its branches. If none of the conditions is true, the *else* branch is executed.

The Listen activity defines a set of EventDriven activities that wait for a specific event. One of the EventDriven activities is fired when the expected message is received. Last, the While construct defines a set of activities that are fired as many times as its condition is true.

III. Case Study: On-line Computer Sale

In this section we introduce a simple case study of on-line computer sale. The example consists of a system whose purpose is to sell computer material such as PCs, laptops, or PDAs to clients. As a starting point we reuse two components: a *Supplier* and a *Buyer*. These components have been implemented using WF/.NET, and their workflows are presented in Figures 1 and 2 respectively.

First, the *Supplier* receives a request under the form of two messages that indicate the type of the requested material, and the max price to pay (type and price). Then, it sends a response indicating if the request can be replied positively (reply). Next, the *Supplier* can terminate the session, receive and reply other requests, or receive an order of purchase (buy). In the latter case, a confirmation is sent (ack) pointing out if the purchase has been realised correctly or not.

The *Buyer* can submit a request (request), in which it indicates the type of material he wants to purchase and the max price to pay for that material. Next, once he/she has received a response (reply), the *Buyer* may realise another request, buy the requested product (purchase), or end the session (stop).

In both *Supplier* and *Buyer* we have split the workflows of Figures 1 and 2, presenting them into two parts. On the left-hand side, we show the initial execution belonging to the first request, and on the right-hand side we present the loop offering the possibility of executing other requests, performing a purchase or finalising. We identify the names of certain activities, whose functionality is the same, with an index (such as type_1 and type_2, or invokeType_1 and invokeType_2 in *Supplier*), because WF does not accept activities identified using the same name. Note that in the *Buyer* component, the messages with the code suffix, such as request_1_code, correspond to the execution of *C#* code. Last, some WebServiceInput and WebServiceOutput activities may be meaningless with respect to the component functionality, and appear in the WF

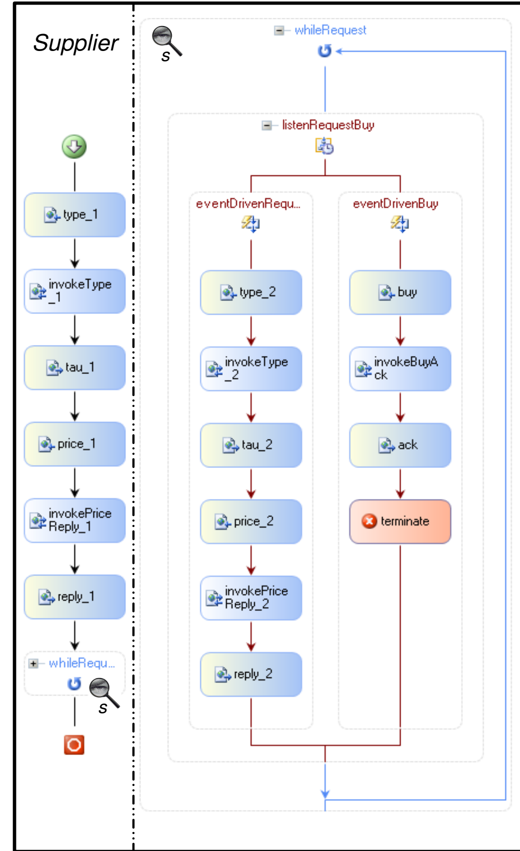


Fig. 1. WF workflow for the *Supplier* component

workflow only because WF obliges their presence before and after InvokeWebService activities. In Figures 1 and 2 these activities are identified with tau identifiers.

IV. Composition and Adaptation of WF Components

In this section, we focus on the composition and adaptation of the *Buyer* and *Supplier* components.

A. Extraction of the Behavioural Interfaces

First of all, we present in Figure 3 the LTS (Labelled Transition Systems) extracted from the workflow-based components presented in Section III. The main ideas of the obtaining of LTS from workflow constructs are the following:

- Code is interpreted as τ transition (internal);
- Terminate corresponds to a final state in LTS;
- InvokeWebService is split into two messages, one emission followed by a reception;
- WebServiceInput and WebServiceOutput messages are translated similarly in LTS;
- Sequence is translated so that it preserves the order of the involved activities in the resulting LTS;
- IfElse corresponds to a choice, that is two transitions outgoing from the same state, which encodes both parts of the conditional construct;

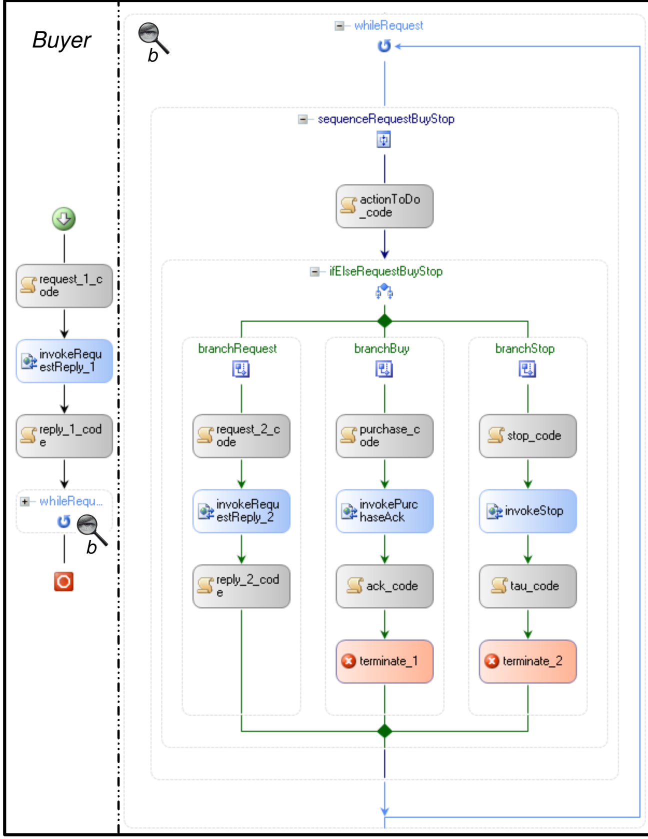


Fig. 2. WF workflow for the *Buyer* component

- `Listen` corresponds to a state with as many outgoing transitions as there are branches in the WF contruct; each transition holds a message that may be received;
- `While` is translated as a looping behaviour in the LTS.

LTS does not support the description of data expressions, consequently conditions appearing in `While` and `IfElse` constructs are abstracted during the LTS extraction stage. Likewise, `WebServiceInput` and `WebServiceOutput` activities identified with `tau` identifiers (see Figs. 1 and 2) are translated as τ transitions in the corresponding LTS.

Initial and final states in the LTS come respectively from the explicit initial and final states that appear in the workflow. There is a single initial state that corresponds to the beginning of the workflow. Final states correspond either to a `Terminate` activity or to the end of the whole workflow. Accordingly, several final states may appear in the LTS because several branches in the workflow may lead to the final state. Initial and final states are respectively depicted in LTSs using bullet arrows and darkened states.

The messages that appear in the *Buyer* LTS come from the output and input parameters that appear in its invoke activities. As far as the *Supplier* component is concerned, the invoke activities are made abstract because they correspond to interactions with external components (in charge of the material database), and are not of interest for the composition at hand. Therefore, the observable messages in this case are coming from the input and output messages surrounding the invoke activities. All the τ transitions in both LTSs corresponding to $C\#$ code in the *Buyer* workflow, and to `tau` `WebServiceOutput` activities in the *Supplier* one have been removed (by $\tau{*}.a$ reduction [11]) to favour readability. To identify unambiguously component messages in the adaptation process, their names are prefixed by the component identifier, respectively b for *Buyer*, and s for *Supplier*.

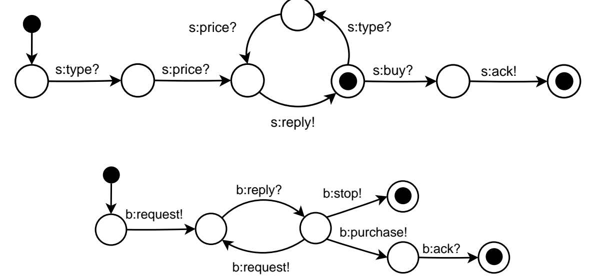


Fig. 3. LTS interfaces of *Supplier* (top) and *Buyer* (bottom) components

B. Mismatch Cases

In this simple example, we can emphasise three cases of mismatch:

1) name mismatch: the *Buyer* may buy the computer using `purchase!` whereas the *Supplier* may interact on `buy?`;
2) mismatching number of messages: the *Buyer* sends one message for each request (`request!`) while the *Supplier* expects two messages, one indicating the type (`type?`), and one indicating the max price (`price?`);
3) independent evolution: the *Buyer* may terminate with `stop!` but this message has no counterpart in the *Supplier*.

C. Adaptation Mapping

Now a mapping should be given to work the aforementioned cases of mismatch out. We use vectors that define some correspondences between messages. More expressive mapping notation exist in the literature, such as regular expressions of vectors [4], but with respect to the example at hand, vectors are enough to automatically retrieve a solution adaptor.

$V_{req} = \langle b{:}request!, s{:}type?\rangle$
$V_{price} = \langle b{:}\varepsilon, s{:}price?\rangle$
$V_{reply} = \langle b{:}reply?, s{:}reply!\rangle$
$V_{stop} = \langle b{:}stop!, s{:}\varepsilon\rangle$
$V_{buy} = \langle b{:}purchase!, s{:}buy?\rangle$
$V_{ack} = \langle b{:}ack?, s{:}ack!\rangle$

The name mismatch can be solved by vector V_{buy}. The correspondence between `request!` and messages `type?` and `price?` can be achieved using two vectors, V_{req} and V_{price}, where the second contains an independent evolution of component *Supplier*. The last mismatch is solved using V_{stop} in which the message `stop!` is associated to nothing.

D. Generation of the Adaptor Protocol

Given a set of component LTSs (Section IV-A) and a mapping (Section IV-C), we can use existing approaches (here we rely on [4]) to generate the adaptor protocol automatically. This is a strength of this proposal because in some cases, the adaptor protocol may be very hard to derive manually. Since the adaptor is an additional component through which all the messages transit, all the messages appearing in the adaptor protocol are reversed.

Figure 4 presents the *Adaptor* LTS. Note first that the adaptor receives the request coming from the *Buyer*, and splits the message into messages carrying the type and price information. This LTS also shows how the termination is possible along the `stop?` message, and how the adaptor may interact on different names (`purchase?` and `buy!`) to make the interaction possible.

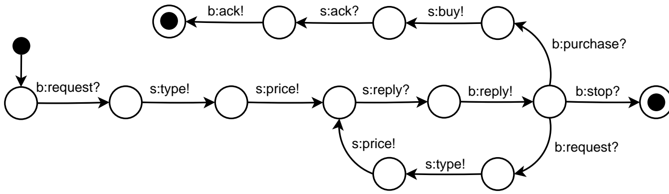


Fig. 4. *Adaptor* protocol for the case study

E. Implementation of the WF Adaptor

From the adaptor LTS presented above, a corresponding WF component is obtained following the reversed process that we have sketched in Section IV-A, *i.e.*, by generating a workflow from an LTS. Therefore, every emission followed by a reply is encoded as an `InvokeWebService` construct. Other input/output events are translated using `WebServiceInput/WebServiceOutput` activities. The decision of the *Buyer* is translated as a `Listen` construct, and the looping behaviour as a `While` activity. We present in Figure 5 the *Adaptor* workflow that has been encoded in WF.

Finally, we point out that the system presented in this section has been completely implemented using WF, and the *Buyer* and *Supplier* components works as required thanks to the use of the WF *Adaptor* component.

V. Conclusion

This paper has presented on a simple yet realistic example how existing model-based adaptation approaches can be related to implementation platforms such as WF in the .NET Framework 3.0. To make this work, we had to face and work out specificities of the WF platform such as the use of `tau WebServiceOutput` activities, or of several `InvokeWebService` activities in one session. This work is very promising because it shows that software adaptation is of real use, and can help the developer in building software applications by reusing software components or services.

We end with a list of future tasks we will tackle to make the adaptation stage as automated as possible:

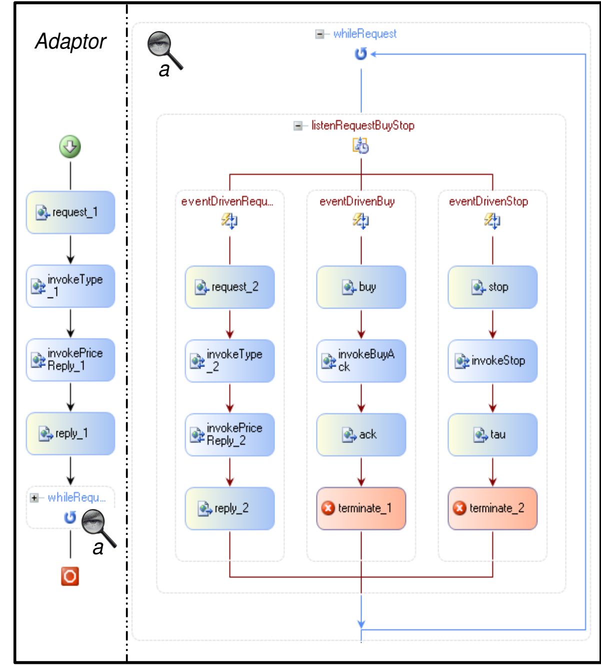


Fig. 5. WF workflow for the *Adaptor* component

- automating the LTS extraction from WF workflows;
- automating the mismatch detection, and generating the list of mismatch situations from a set of component LTSs;
- beyond mismatch detection, tackling verification of WF components;
- supporting techniques to help the designer to write the mapping out, and to generate automatically part of it;
- generating WF workflows from the adaptor LTS.

We would also like to carry out experiments on the implementation of adaptors using BPEL and the Netbeans Enterprise platform to compare on precise criteria the adequacy of both platforms to apply adaptation in practice.

Acknowledgment

This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and the project P06-TIC-02250 funded by Junta de Andalucía.

References

[1] C. Canal, J. Murillo, and P. Poizat, "Software Adaptation," *L'Objet*, vol. 12, no. 1, 2006, special Issue on Coordination and Adaptation Techniques for Software Entities.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Adisson-Wesley, 2003.

[3] A. Bracciali, A. Brogi, and C. Canal, "A Formal Approach to Component Adaptation," *Journal of Systems and Software*, vol. 74, no. 1, 2005.

[4] C. Canal, P. Poizat, and G. Salaün, "Synchronizing Behavioural Mismatch in Software Composition," in *Proc. of FMOODS'06*, ser. LNCS, vol. 4037, 2006.

[5] P. Inverardi and M. Tivoli, "Deadlock-Free Software Architectures for COM/DCOM Applications," *Journal of Systems and Software*, vol. 65, no. 3, 2003.

[6] H. W. Schmidt and R. H. Reussner, "Generating Adapters for Concurrent Component Protocol Synchronization," in *Proc. of FMOODS'02*. Kluwer, 2002.
[7] D. M. Yellin and R. E. Strom, "Protocol Specifications and Components Adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, 1997.
[8] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters," in *Proc. of ICSOC'06*, ser. LNCS, vol. 4294, 2006.
[9] T. Andrews *et al.*, *Business Process Execution Language for Web Services (WSBPEL)*, BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, Feb. 2005.
[10] K. Scribner, *Microsoft Windows Workflow Foundation: Step by Step*. Microsoft Press, 2007.
[11] H. Garavel, R. Mateescu, F. Lang, and W. Serwe, "CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes," in *Proc. of CAV'07*, ser. Lecture Notes in Computer Science, vol. 4590, 2007.

On the benefits of using model transformations to describe components design process

Eveline Kaboré and Antoine Beugnard
ENST Bretagne
Technople Brest-Iroise
CS 83818 - 29238 Brest Cedex 3 - France
Email: {Eveline.Kabore,Antoine.Beugnard}@enst-bretagne.fr

***Abstract*—In this paper we show how model transformations can be used to describe and automatize component design process throught an example of a communication component. Automatizing parts of the design with transformations allows to trace the design, defend such or such design choice and build non functional variants of the same component.**

I. Introduction

Models are widely used in sciences and have become an unavoidable tool for software designers and implementors. Models were used in many development methods such as SADT [1], JSD [2], etc. They allow to describe different aspects of a system: structural, functional, behavioral, temporal, etc. Models allow the description of the system to be developed at different stages with various levels of details. The Unified Modelling Language (UML) is the last avatar of a standard modelling notation. The way models are produced and elaborated is mainly beyond the scope of modelling; it is mainly good-practices, know-how and methods more or less formalized. One of the last great advances in software engineering was the introduction of patterns (especially design patterns) as a semi-formalization of good (or bad) practices as catalog of patterns expressed in a language of pattern. The formalization and the clarification of the process of elaborating models are the next challenge. Considering the processes of elaborating and refining models as an activity that can be described with a dedicated language is in our point of view, a revolution.

We show in this article how model transformations can be used to automatize the design and implementation process of a software component. Model transformation languages can hence be considered as a language dedicated to process modelling. In order to confront an idea to a real development process we have developed communication components using transformation techniques. Design choices are left to the designer, but the refinement process is implemented as transformations that are automatically applied once selected.

Communication components are components that are dedicated to communication and hence are naturally distributed. In [3], E. Cariou presented a manual transformation process that shows how the abstract specification can produce many implementation variants: one can be centralized, another be distributed without replication of data, and another with replication. The interest of this approach is that all variants are functionally substitutable but offer different non functional features. Communication components appear to be an interesting candidate to study the design process as a sequence of transformations. Due to its distributed deployed structure, many algorithmic variants may be chosen.

The rest of the article is organized as follows. The next section introduces the starting point of the process: the abstract specification of a communication component. Section III suggests the whole design process with models, meta-models and transformations. Section V evaluate the benefits of using model transformation to describe components design process. Some related works are presented before the conclusion.

II. Communication component: medium

In order to elaborate a better understanding of model transformations as design choices implementation, we have restricted their use to special communication abstractions called *mediums* [4].

a) Definition: A medium is a special component which implements any level communication protocol or system. A medium can implement, for example, a consensus protocol, a multimedia stream broadcast or a voting system. A medium includes classical component properties such as explicit interface specification, reusability or replaceability, but a medium is not a unit of deployment. A communication component is a *logical* architectural entity built to be *distributed*. An application is the result of inter-connecting a set of components and mediums. This is particularly interesting as it would allow the separation of two concerns: functional concern described by components and communication concern described by mediums.

b) Example: As an illustration, let us consider the example of an airline company with travel agencies located worldwide. A medium can implement the reservation system and offer services to initialize information on seats, to reserve seats and to cancel reservations. A reservation application can then be built by inter-connecting the reservation medium and components representing the company and the agencies as illustrated in figure 1.

c) Specification: The specification attempts to describe the medium contract from its user's point of view as illustrated in figure 1. As a 'classical' component, a medium is specified through a set of offered and required services. For each role, a medium defines an interface for offered services and an

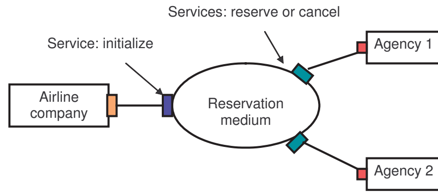


Fig. 1. An example of communication component: reservation medium

interface for required services. Each offered and required service should be specified.

Mediums are specified in UML. The specification of a medium in UML contains two aspects: *structural aspect* described by a class diagram and *behavioral aspect* described through other UML features such as interaction messages, statecharts and OCL. Figure 2 presents a specification of the reservation medium in a UML class diagram.

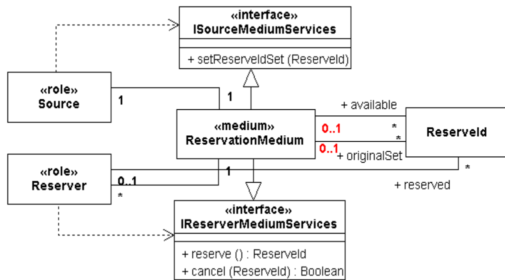


Fig. 2. Reservation medium abstract specification

d) Deployment target: In the previous section, we saw that, at the abstract level, the medium is represented by a single software component. The goal of the design process is to make distribution of this abstraction possible. The single software component which represents the medium at the abstract level is split into small implementation components called *role managers*. A different *role manager* is locally associated with each component and the medium becomes a logical unit composed of all the *role managers*. From a local point of view, each *role manager* implements the services used by its associated component. From a global point of view all the *role managers* communicate through middleware and cooperate to realize all the medium services.

Thus, at the deployment level the single software communication component which represents the medium at the abstract level disappears completely and the medium becomes an aggregation of distributed *role managers*. This architecture presents two main advantages: it allows several implementations of the same abstract medium model, depending on how *role managers* cooperate, and a good separation of functional and interactions concerns from specification to implementation.

The next section sketches out the full development process and its implementation with models and transformations.

III. Outline of the Design Process

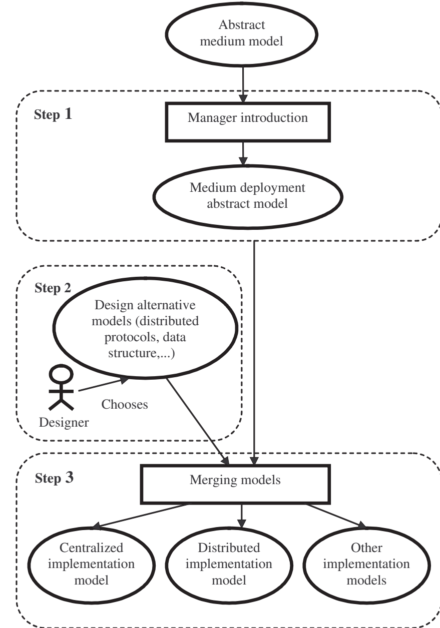


Fig. 3. An example of design alternatives

The entry point of the design process is the medium abstract specification model (figure 3). Our aim is to produce a distributed medium implementation model from this abstract model. This distributed implementation model should preserve all the functionalities of the medium and match all the deployment constraints of the medium. In order to reach this goal, we define a medium refinement process containing the three following steps.

A. Step 1: Introducing the distributed architecture in the medium model

The first step consists in transforming the single UML class which represents the medium in its abstract specification model into an aggregation of *Managers* in order to match the deployment architecture. This transformation is clearly described in [4]. Figure 4 illustrates the result of this transformation in the reservation medium abstract model (figure 2). In short, the transformation:

- associates a *Manager* to each role (*ReserverManager* for the role *Reserver* and *SourceManager* for the role *Source*);
- implements in each *Manager* all the services offered by the medium to its associated role (*setReserveIdSet* in *SourceManager*, *reserve* and *cancel* in *ReserverManager*);
- translates all the references of each role on its associated *Manager* (property *reserved* on *ReserverManager*).

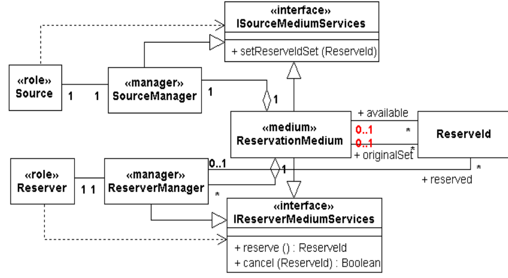


Fig. 4. Managers introduction into the reservation medium

B. Step 2: Specifying the medium actual design alternatives

1) Objective and principle: We propose a set of design alternatives to the medium designer. In the case of the reservation medium data management for example, the designer will have to choose among data structures (that will be used to express the communication semantic), distributed protocols (that will be used to define the repartition strategy of the medium data) and data representation formats (that will be used to represent the distributed data structure in each storage node). This is illustrated in the following example.

a) Example: design alternatives for the reservation medium:

- Example of a centralized implementation. For a small regional company, reservation management can be concentrated on a single role manager: the manager which is associated to the company (*SourceManager* in figure 4) for example. The other role managers associated to reservers act as proxies. In this case, the designer can represent the identifiers of reservation seats (property *available* in figure 2) by a list that will be stored in the *SourceManager*.
- Example of a distributed implementation. For a big company with hundreds of flights and thousands of travel agencies located worldwide, each agency can locally manage its reservations in order to increase the performance and the reliability of the application. In that case, the designer can use a set to represent the identifiers of reservation seats. It can distributed this set between reservers using the protocol *Chord* [5] and implements it using the algorithm defined by the *MIT* [6]. Finally, the designer can use a hashtable to locally represent the identifiers stored on each reserver manager.

Other design alternatives can be proposed to deals with non functional properties such as security, reliability, etc. Thus the designer chooses each design alternative according to its application features and constitutes the actual medium implementation strategy.

2) Modelling design alternatives: In order to facilitate the definition and the reuse of transformations, each choosen design alternative is defined in a model which is injected into the medium abstract model to produce the distributed medium implementation model. Hence, we define a metamodel for each design alternative. For the sake of brevity we only show in figure 5 a view of the distributed protocol definition metamodel.

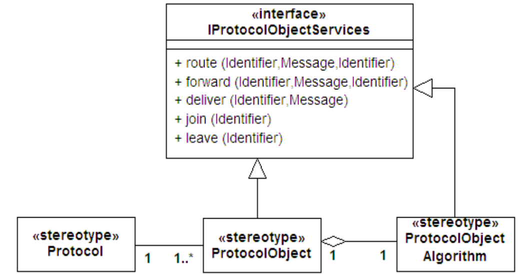


Fig. 5. A view of the distributed protocol specification metamodel

a) Distributed protocol metamodel.: We define a distributed protocol by a set of objects called *ProtocolObject* (figure 5). A *ProtocolObject* is an object that can execute the behaviour of a distributed protocol. Each *ProtocolObject* is implemented by a specific algorithm (*ProtocolObjectAlgorithm*). The main goal of the distributed protocol metamodel consists in defining a common interface for all distributed protocols that will be used in the context of mediums. Such interfaces are proposed in [7], [8], [9]. The *IProtocolObjectServices* interface exported by the distributed protocol definition metamodel is similar to the interface defined in [7]. This interface defines services for three main distributed application abstractions: the DHT (Distributed Hash Tables), the DOLR (Decentralized Object Location and Routing) and the CAST (group anycast/multicast). The *IProtocolObjectServices* interface offers the following services: *route* (to route a message), *forward* (to forward a message), *deliver* (to deliver a message), *join* (to join the distributed application) and *leave* (to leave the distributed protocol).

b) Use of the distributed protocol metamodel.: To define a distributed protocol model, we describe each *ProtocolObject* and its implementation algorithms and we implement the interface *IProtocolObjectServices* for this protocol by defining its services. This is illustrated in the following example.

c) Example: a view of the Chord *protocol model.:* In short, the initial specification of the *Chord* protocol defined in [5] exportes one role and provides fives services described in table I.

Figure 6 illustrates the structure of the *Chord* protocol according to the distributed protocol metamodel. In this figure 6, the class *ChordObject* represents the objects that can execute the behaviour of the *Chord* protocol. The *ChordObject* can be implemented using two different algorithms represented by the class *MITAlgorithm* and the class *MACEDONAlgorithm*.The class *MITAlgorithm* encapsulates the Chord protocol implementation algorithm proposed by the MIT. The class *MACEDONAlgorithm* encapsulates the Chord protocol implementation algorithm proposed by the MACEDON framework [8]. Other implementation algorithms of the Chord protocol can

Services	Description
insert (key, value)	inserts a key/value binding at r distinct nodes.
lookup(key)	returns a value associated with the key
update(key, newval)	inserts the key/newval binding at r nodes
join (n)	add a node to the *Chord* system
leave()	leave the Chord system

TABLE I
A VIEW OF THE CHORD PROTOCOL API [5]

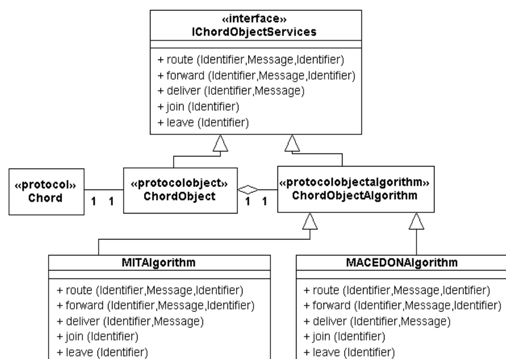


Fig. 6. A view of the Chord protocol specification

be added to the model.

In order to encapsulate each implementation algorithm of the *Chord* protocol, we implement the *insert, lookup, update, join* and *leave* services (described in table I) on top of the *IProtocolObjectServices* interface. As an illustration, a simple implementation of *insert* routes an *INSERT* message containing *value* and the local node's nodehandle[1], *S*, using **route(key, [INSERT,value,S], NULL)**. The key's root, upon receiving the message, stores the (key, value) pair in its local storage. In this definition of **route(key, [INSERT,value,S], NULL)**,

- the first parameter **key** represents the identifier of the local node;
- the second parameter **[INSERT,value,S]** represents the message. This message contains three information: the type of the message **INSERT**, the **value** to be inserted and the local node's nodehandle, **S**.
- the last parameter is an optional argument which is used to specify a node that should be used as a first hop in routing the message. No first hop node is specified in this example: the value is **NULL**

The *lookup, update, join* and *leave* services can be implemented in the same way. We note that each implementation algorithm of the *Chord* protocol has its own definitions of information contained in the messages, algorithms used in routing messages, etc.

C. Step 3: Merging the actual design alternatives in the medium model

The third step consists in merging the actual design alternatives defined in the previous step into the medium deployment abstract model obtained in the first step. We perform two kind of transformations to reach this goal: a structural transformation and a behavioral transformation. In short, each transformation is specified using a pre-condition, a post-condition and a set of actions. All the transformations are defined using the model tranformation language *kermeta* [10].

1) Structural transformation: The structural transformation introduces elements[2] which will be used to ensure each data distribution services. Two kinds of elements are created for this purpose: elements which ensure distributed data access services (small eclipse in figure 7) and elements which ensure data distribution services (big circle in figure 7). Figure 7 illustrates the introduction of these elements in the reservation medium model for the distributed implementation strategy illustrated in the previous section. For the sake of simplicity, we only present the new structure of the *ReserverManager*.

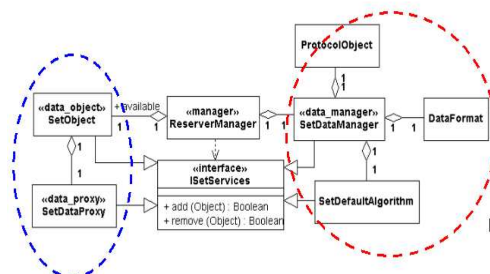


Fig. 7. Outline of data managers introduction in the reservation medium

- the class *SetDataManager* implements all the functionalities that are necessary to distribute a set. The distributed protocol functionalities are encapsulated in the class *ProtocolObject*. The set data access primitive definitions are encapsulated in the class *SetDefaultAlgorithm*. The definition of primitives that are used to read and/or write a piece of data stored in memory is encapsulated in the class *DataFormat*.
- The class *SetObject* implements all the functionalities that are necessary to access the distributed set.

Since the identifiers of reservation seats are distributed between reservers, the new structure of the *SourceManager* constains only elements which ensure data access services. No modification is performed on all the other classes of the medium model.

[1]A nodehandle encapsulates the transport address and identifier of a node in the system. The transport address might be, for example, an IP address and port (see [7] for more details).

[2]The technical details of these element definitions will not be given in this paper

2) Behavioral transformation: The behavioral transformation implements all abstract operations in the medium model obtained in the second step. Two kinds of transformations are performed for this purpose: algorithmic transformations and configuration transformations.

a) Algorithmic transformations: denote transformations that are used to generate abstract operation implementation algorithms. As an illustration, the following example show a view of the code which is automatically generated by an algorithmic transformation in order to implement the *add* operation of the type set.

```
class SetDefaultAlgorithm inherits
                    ISetServices { ...
  method add( element : Element) : Boolean
                from ISetServices is
  do
   if (self.dataManager.lookup(element)
                    == void)
   then
    result := self.dataManager.place(element)
   else result := false end
  end
}
```

In this example, the *add* operation invokes the *lookup* service defined in its associated data manager to check the existence of the current element in the set. If the element is already in the set, it returns false, otherwise, it adds the element in the set using the service *place* defined in its associated data manager.

b) Configuration transformations: denote transformations that are used to instanciate the appropriate objects according to the designer choices. Here is an outline of the result of the configuration transformation which is used to instanciate the appropriate objects for the reservation medium according to the distributed implementation strategy specified in the previous the section.

```
class IReserverManager {...
  operation connect() is
  do
   (1) available := SetDataObject.new
   (2) availableDataManager :=
                    SetDataManager.new
   (3) availableDataManager.protocolObject:=
                    ChordProtocolObject.new
   (4) availableDataManager.protocolObject.
     algorithm := MITAlgorithm.new
   (5) availableDataManager.setAlgorithm :=
                    SetDefaultAlgorithm.new
  end
}
```

In the *connect*[3] operation, instruction (1) instantiates a new *SetObject* and assigns it to *available* (figure 7). Instruction (2) instantiates a new *SetDataManager* and assigns it to *availableDataManager* (figure 7). Instruction (3) instantiates a new *ChordProtocolObject* and assigns it to the property representing the protocol object in *availableDataManager*. Instruction (4) instantiates a new *MITAlgorithm* and assigns it to the property which represents the chord and instruction (5).

[3]The method *connect* is invoked at the connexion of each manager

IV. Model transformations

Our approach of implementing the whole process outlined in the previous section consists in distributing information between metamodels and transformations.

A. Metamodels

The implementation of the whole process relies in 8 metamodels:

1) a medium abstract specification metamodel,
2) a medium deployment abstract metamodel,
3) a medium implementation metamodel,
4) a medium decision metamodel,
5) a distributed protocol metamodel,
6) a data representation format metamodel,
7) an abstract type metamodel,
8) a distributed abstract type metamodel,

The medium abstract specification metamodel specifies the concepts which are used to define a medium and the relationships between these concepts. These concepts and relationships are well described in [4]. Figure 8 illustrates the abstract structure. A medium offers and requires services.

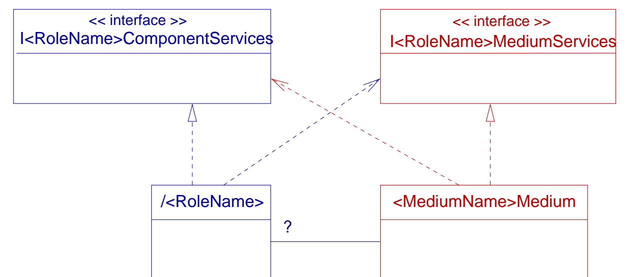


Fig. 8. A view of a medium abstract specification metamodel

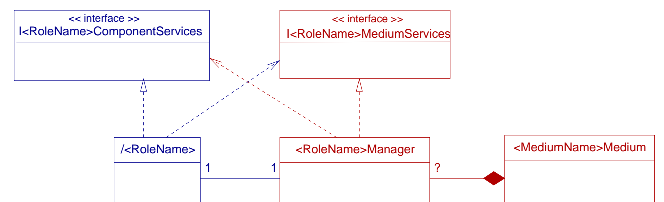


Fig. 9. A view of a medium deployment abstract metamodel

The medium deployment abstract metamodel describes the concepts which are used to define a medium abstract deployment architecture and the relationships between these concepts. The medium deployment architecture metamodel is also described in [4]. Figure 9 sketches out the abstract

structure of a medium deployment architecture: *Managers* are introduced in the medium abstract specification metamodel.

The medium implementation metamodel describes the concepts which are used to implement a medium and the relationships between these concepts. Figure 10 depicts the abstract structure of a medium implementation model: the medium class desapears.

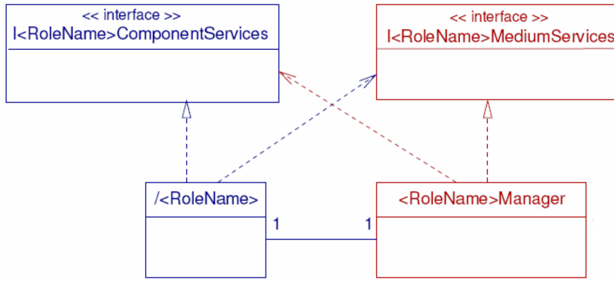


Fig. 10. A view of a medium implementation metamodel

The medium decision metamodel describes concepts which are used to specify the actual design alternatives of a medium.

The distributed protocol metamodel is outlined in the previous section.

The data representation format metamodel exports a common interface which defines abstract primitives that will be used to read and to write a data in a physical memory.

The abstract type metamodel defines concepts which are used to specify an abstract type such as a collection, a list, a set, a tree, a graph, a hashtable, etc.

The distributed abstract type metamodel defines concepts which are used to implement a distributed abstract type (such as a distributed collection, distributed list, distributed set, etc.) and the relationships between these concepts.

The data representation format, the abstract type and the distributed abstract type metamodels are defined in the same way as the distributed protocol metamodel. For the sake of brevity, we do not illustrate these metamodels in this paper.

In order to facilate models consistency checking, we associate a property model to each metamodel. The property model of a metamodel specifies all the constraints that a model should satisfy to conform to its metamodel.

B. Transformations

1) Definition: A transformation defines a process of conversion a model source into a target model [11]. In the context of mediums, we define each transformation through a set of preconditions, postconditions and steps (figure 11). The preconditions describe the constraints that a model should satisfy to be a valid source model of a transformation. The postconditions specify the set of constraints that a model should satisfy to be a valid result of a transformation. The steps specify the sequences of the transformation execution. Each step is defined by a set of actions or transformation rules

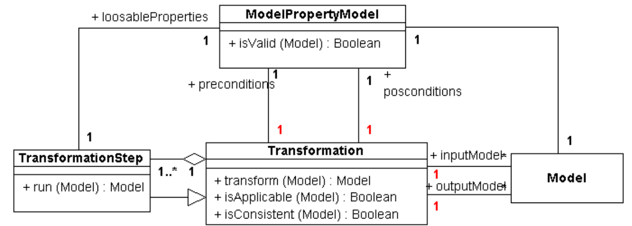


Fig. 11. Transformation specification

defining the process of converting a source model into a target model.

The constraints are organized in the form of structured properties. Each property contains a method called *isValid* which can be used to check the validity of this property in a given model. The organization of constraints in the form of structured properties allows to specify for each step of a transformation, the constraints that the source models must satisfy, the constraints that the target models must satisfy and the constraints which can be invalidated by the step.

2) Implementation: We implement each transformation using the model transformation language *Kermeta*. The source and target models of transformations are *Kermeta* files. The following piece of *Kermeta* code illustrates the transformations which are used to implement the step 1 of the design process (i.e to introduce the distributed architecture in the medium model).

```
class AddManagerTransformation inherits
                 Transformation {
  operation transform(inputModel :
     OrderedSet<Model>):OrderedSet<Model> is
  do
   if(self.preconditions.validate(inputModel))
   then
     .....
     outputModel := self.addManagerStep.
                         run(inputModel)
     if(self.postcondition.validate(
              outputModel))
     then
       result := outputModel
     end
   end
  end
}

class AddManagerStep inherits
                TransformationStep {
  operation run(inputModel:
     OrderedSet<Model>) :OrderedSet<Model> is
  do
   if(self.preconditions.validate(inputModel))
   then
     outputModel := self.createDependencyStep.
     run(self.createPropertyStep.run(
          self.createManagerStep(inputModel)))
     if (self.postconditions.validate(
          outputModel))
     then
```

```
        result := outputModel
      end
    end
  end
}

class CreateManagerStep inherits
               TransformationStep {...//code}

class CreatePropertyStep inherits
               TransformationStep{...//code}

class CreateDependencyStep inherits
              TransformationStep{...//code}
```

The *AddManagerTransformation* transformation definition contains a main step called *AddManagerStep* which introduces the *Managers* into the medium abstract model in order to produce the medium deployment abstract model. The *AddManagerStep* step definition contains:

- the *CreateManagerStep* step which creates a class representing the *Manager* of each role;
- the *CreatePropertyStep* step which updates the target model properties or references according to the medium deployment abstract metamodel definition;
- the *CreateDependencyStep* step which updates the target model dependencies according to the medium deployment abstract metamodel definition.

All the other transformations which are used to implement the second and the third steps of the design process are defined in the same way.

3) Validation: One of the main problem in using model transformations is the validation of the target models. In the context of mediums, the transformations are validated in three steps:

1) The first step of the validation is performed during the execution of the transformation and relies on the checks of pre- and post-conditions. It ensures the correctness of transformations.
2) After the execution of each transformation, we compile the target model using the kermeta compiler to checks types, inheritence, association multiplicities, etc. It ensures the UML correctness of models.
3) Finally, we instanciate and execute the generated implementation model of the medium in order to simulate the behavior of the services.

C. Use of transformations

1) Principle: The transformations are applicable to any communication component. In order to use them, the medium designer should specify the abstract model and the decision model of the medium. The medium abstract model should conform to the abstract medium metamodel and the medium decision model should conform to the medium decision metamodel. Each design alternative model specified in the decision model should also be defined according to its metamodel. We provide a library of design alternative models and a mecanism to extend this library. In the next section, we give some numerical data on the size of the the implementation models of some communication components using the transformations.

Initial model number of classes	SimpleList Medium 4		Reservation Medium 6		*Comments*
Manager introduction number of classes	SimpleList Medium 6		Reservation Medium 9		*1 + 1 per role*
Data type selection number of classes	List 9	List 9	List 15	Set 15	*+3 per attribute*
Distribution model number of classes	Centralized 11	Chord 13	Chord 19	Chord 19	
Final model number of classes	13	15	21	21	+2

TABLE II
NUMBER OF CLASSES PER MODEL

2) Empirical results: The realization of the design process relies on 8 metamodels and 18 transformations. The medium abstract specification metamodel contains approximately 30 classes and the medium deployment abstract metamodel contains approximately 35 classes. The first transformation which transforms a medium abstract model into a medium deployment abstract model (described in the previous section) is implemented with 516 lines of kermeta code. The pre-condition of this transformation contains 6 properties implemented with 285 lines of kermeta code. Its post-conditions contains 11 properties implemented with 350 lines of kermeta code. The whole set of transformations is 3565 lines of Kermeta code. In order to apply the process models have been developed: 2 communication component models, 4 abstract data type models, 4 distribution models, 3 data representation models. Many others can be added. New metamodels could be added in order to offer other design alternatives.

Table II shows the number of classes for some paths in the design. Adding classes is the basic transformation; many other model transformations are hidden such as association between classes, definition of methods, source code changes, etc. All these transformations are checked as consistent thanks to pre and post-conditions. Note that each transformation runs in a few seconds, including all verifications of pre and post-conditions.

V. BENEFITS OF THE APPROACH

The benefits of using model transformation to describe components design process is 6 fold.

1) The process is explicit. The process is described by the transformations and all models and meta models used.
2) The process is traceable. Being explicit, transformations could be memorized and could show the trace followed during the process. All models produced during the process can also be stored and retrieved when backtracking is required in the process.

3) The process is reusable. Applying the process to other initial models is direct. We applied our transformations to abstract data types considered as distributed memories with no difficulties.
4) The process is extensible. New models conforming to the concerns metamodels used can be added. We added new data type to implement association links very simply. Models to describe distribution variants have required more work. As long as the new models conform to the concern meta model, the extension is feasible.
5) The process can produce many implementation variants. From the same abstract specification of a medium we have produced many variants with very different non-functional features. Design choices are coded as transformations and can be reused on demand at the right moment in the process.
6) The process generates automatically the appropriate code. The structural, behavioural and configuration code of the medium can be generated according to its actual implementation strategy.

However, to obtain these benefits this approach requires a deep understanding of the domain. All metamodels constrain the set of products that can be built. All the transformations define the set of design choices that are applicable.

VI. Related works

Most methodologies are informaly described. They suggest a process which, in the most formalized cases, rely on contracts [12] or mathematical refinements like the B-method [13]. B defines a language and a refinement methodology. It is an algebraic specification language that is supported by tools that help refining specification safely. Each step of the process generates proof requirement the developer has to demonstrate, either manually or automatically. Some critical systems were developed in B (in 1998 the control system of line 14 of the Parisian subway was fully developed and proved in B). Our approach is more empirical and uses the so-called "semi formal" approach. It may be easier to learn and may tackle different kind of design problems such as distribution. We do not try to prove design steps, but just to automatize them and give enough confidence in the transformations thanks to pre and post-conditions.

In a recent paper, H. Sneed [14] criticizes the model driven approach. He argues that model-driven tools (1) magnify the mistakes made in the problem definition, (2) create an additional semantic level to be maintained, (3) distort the image of what the program is really like, (4) complicate the maintenance process by creating redundant descriptions which have to be maintained in parallel,(5) are designed for top-down development that creates well-known maintenance problems. These drawbacks are mainly associated with tools. All these criticisms have already been raised when assembly was replaced by high level programming languages. We agree tools are not mature. Our experiment shows that transformations may help explicit the process and simplify the maintenance, if models are defined well enough. Other experiments [15] tend to prove that model composition (hence a bottom-up approach) is possible.

This compositional approach looks like Aspect Oriented Modelling [16]. This approach recommends to separate concerns and offers an operation of weaving that composes/weaves each concern with the functional specification. Our approach differs since the "weaving" operation we use is a transformation that is adapted to the kind of concern composed. Instead of using a universal weaving operation we propose a more flexible approach (but less re-usable) were a balance may be found between the meta-model definition of the concern and its composition operation implemented as a transformation.

Model transformations are widely used on UML models. Most of them cover a small part of the development life cycle. Some transformations are dedicated to code generation. They usually produce the skeleton (structural part) of the source code that has to be completed manually. Another current use is applying design patterns [17]. Once again, the structural part is rather well implemented[4], but the collaboration one is still research in progress. Our experiment relies on all those works on model transformations and tries to demonstrate how (under which conditions) all these steps could be integrated in a full design process.

VII. Conclusion

We have shown in this article how model transformations can be used to automatize the design and implementation process of a software component. The application context was strongly constrained. The specification rules of communication components are used as preconditions of our first transformations. Models and metamodels used are also defined to work together. We wanted to demonstrate that when the domain is well defined and when the design process is well understood, it should be possible to automatize the whole design process with a set of model transformations. The example used is large enough and the deployment target complex enough to give a good indication of what transformation driven design process would look like.

Our experiment focuses on design, implementing design choices as transformations. We believe transformations should be considered in metrics computation, quality evaluation, test generation, . . . in all the development life cycle in short. Systems (viewed as a product) have their own dedicated languages: modelling or programming languages. We argue that transformation languages should be considered as dedicated process languages.

References

[1] D. A. Marca and C. L. McGowan, *SADT*, ser. McGraw Hill Software Engineering Series. McGraw-Hill, 1987, no. ISBN 0070402353.

[2] M. Jackson, *System Development*. Englewood Cliffs: Prentice Hall, 1983.

[4]Patterns purists would say that patterns are not dedicated to be automatically applied. In the absolute, we agree, but why not consider applying patterns in well defined contexts?

[3] E. Cariou, A. Beugnard, and J.-M. Jézéquel, "An archictecture and a process for implementing distributed collaborations," in *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002)*, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, September 17 - 20 2002.
[4] E. Cariou and A. Beugnard, "The specification of UML collaboration as interaction component," in *UML 2002 – The Unified Modeling Language*, ser. LNCS, J.M. Jézéquel, H. Hussmann, S. Cook, Ed., vol. 2640. Dresden, Germany: Springer Verlag, September 30 - October 4, 2002, pp. 352 – 367.
[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM Conference*, San Diego, 2001.
[6] Massachusetts Institute of Technology. (2004) lsd. http://www.pdos.lcs.mit.edu/chord/.
[7] F. Dabek, B. Zhao, P. Drushcel, J. Kubiatowicz, and I. Stoica, "Towards a common api for structured peer-to-peer overlays," in *2nd International Workshop on Peer-to-peer Systems (IPTPS'03)*, fevrier 2003.
[8] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat, "Macedon: Methodology for automatically creating, evaluating, and designing overlay networks," in *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.
[9] A. Rowstron and P. Drusche, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems." in *IFIP/ACM Middleware*, novembre 2001.
[10] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving executability into object-oriented meta-languages," in *Proceedings of MODELS/UML'2005*, ser. LNCS, S. K. L. Briand, Ed. Montego Bay, Jamaica: Springer, October 2005.
[11] OMG. (2003, juin) Mda guide version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf.
[12] D. D'Souza and A. C. Wills, *Objects, Components and Framework with UML: The Catalysis Approach*. Addison-Wesley, 1998. [Online]. Available: http://www.trireme.com
[13] J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996, no. ISBN 0521496195.
[14] H. M. Sneed, "The drawbacks of model-driven software evolution," in *Workshop on Model-Driven Software Evolution, IEEE - CSMR 2007 11th European Conference on Software Maintenance and Reengineering Software Evolution in Complex Software Intensive Systems*, Amsterdam, the Netherlands, March 20-23 2007.
[15] A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt, "On some properties of parametrized model application," in *First European Conference on Model Driven Architecture - Fondations and Applications (ECMA-FA'05)*, ser. LNCS, vol. 3748. Springer, November 2005, pp. 130–140.
[16] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales, "Aspect-oriented programming," *Lecture Notes in Computer Science*, vol. 1357, 1998.
[17] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel, "Design pattern application in UML," in *ECOOP'2000 proceedings*, ser. Lecture Notes in Computer Science, E. Bertino, Ed., vol. 1850. Springer Verlag, June 2000, pp. 44–62.

Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions

Klaus Krogmann
Institute for Program Structures and Data Organisation
Faculty of Informatics
University of Karlsruhe (TH), Germany
Email: krogmann@ipd.uka.de

***Abstract*—Predicting extra-functional properties of software systems requires knowledge about their architectures. For component-based software, it also requires understanding of extra-functional properties of each individual component. However, black-box components, as most commonly provided by current component models, do not provide sufficient details to predict QoS. We sketch two architectural design scenarios and describe which information about a component is needed to enable relevant analyses. Based on these considerations a component model is presented which defines such relevant information. We discuss reengineering approaches, namely static analysis and analysis combined with evolutionary algorithms to learn from monitored data to yield such information about a component. We conclude with a more refined view on the black-box-nature of components.**

I. Introduction

The black-box principle of software components is an established concept [1]. We find that several architectural analyses, and in particular the prediction of extra-functional properties, require a more refined black-box principle that includes knowledge on the interna of a software component which is most commonly not provided by component models.

Currently, various researchers are concerned with analysis using knowledge on component interna. For example, predicting QoS attributes like performance requires a detailed component model, a pure interface model is not sufficient (see discussion in section II). This poses two challenging questions: Firstly, what kind of information is needed to make components ready for QoS predictions for performance, while keeping the black-box principle? Secondly, how to realistically extract such information from components? This is particularly interesting in cases where existing components lack such information, as in legacy software.

As a contribution, this position paper discusses these challenges in detail. More concretely, we (a) use the Palladio Component Meta Model [2] to identify those information that is needed to reason on extra-functional properties (in particular performance; namely execution time and response time), (b) we discuss reengineering approaches to yield this information, and thus (c) gain a more refined view on the black-box nature of components.

The paper is organised as follows: In Section II, we discuss several relevant scenarios of extra-functional analysis in component-based systems which need more information in addition to mere interface specifications. In Section IV, we present those parts of the Palladio Component Model that are concerned with modelling components and their relationship to interfaces and the usage profile. Section V sketches feasible approaches to yield such models, for example static code analyses and machine learning from monitored traces. The conclusion in Section VI presents a more refined view on the black-box nature of components.

II. Information needed for Performance Predictions

Although it was perpetuated over the last years that a component is solely defined by its interfaces, some researchers already pointed out that certain analyses require additional information. *Architectural dependency analysis* identifies components of a system which could be affected by system changes, such as updated components, architectural changes, component failures, etc. [3]. Such information is needed to select test cases for regression tests or to halt all components affected by an update.

Another reason for specifying information on components beyond interfaces is the *automated adaptation* of components to restricted contexts. Here, a restricted context can be either a context where only a true subset of the component's implemented services (as specified in the provides interface) is needed and hence the component's required interface most likely can be weakened, or where – vice versa – a required interface of a component is not fulfilled by the context and hence, the component's provides interface must be restricted. Parametric contracts were originally introduced to automatically adapt component protocols in these scenarios [4].

A similar argumentation holds for the analysis of extra-functional properties of components. In such a case, any meaningful model of quantitative extra-functional properties of a component has to take dependencies between required and provided services into account [5]. Fig. 1 gives a short motivating example why an interface model is not sufficient for performance prediction. Assume the response time of service `a()` of Component A is going to be predicted. `a()` itself relies on the component-external calls of `b()`, `c()`, and `d()` which are provided by external components. In this case, one needs to describe the *dependencies* an invocation of `a()` has on the external services as the response time

of `b()`, `c()`, and `d()` (bindings to required services are not hard-coded). Such dependencies are not a property of an interface as it depends on the component implementation whether, for example, `b()` is called one or 100 times – the choice of algorithms for component implementation should not be expressed by interfaces.

Thus, to make modelled dependencies meaningful, one should introduce parametrisation over required services. Further parametric dependencies for the response time arise from the usage profile of a component (e.g. amount of data to process) and the underlying execution environment, which consequently need to be respected as well.

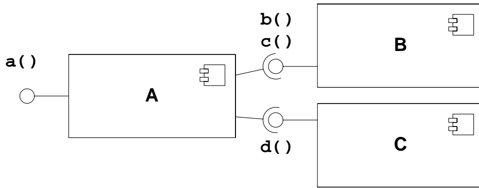


Fig. 1. Excerpt from a component architecture: Component A is requiring services of Component B and C

Our work is concerned with analysing component-based architectures at design-time to support the quality-driven selection (reasoning on extra-functional properties) of design alternatives before implementing the system. Consequently, we are dealing with models of components which can be composed with other components, leading to an architecture of a system. There are several reasons to make predictions based on abstract component specifications also for systems including components which are already implemented. This sounds counter-intuitive, as implemented components are already executable and all their internals are fixed. However, in this position paper we argue that

- implemented components are incorporated into many planned systems, due to component reuse. Still, the overall system is not executable by sole reuse of some components, and architectural analysis is a meaningful tool to drive the design.
- changes in the usage profile can require extensive analyses of existing systems to learn how the existing system will behave under different usage profiles (e.g., different load situations, etc.). Testing these effects is often not feasible due to the sheer size of the system.
- changes in the execution environment (i.e., component container, operating system, hardware, etc.) may require analyses of the system's end-to-end quality. Again, tests are often too expensive. In addition, such analyses often should be done early before the actual new execution environment is available.

The above scenarios clearly show the need to analyse existing component implementations to yield abstractions of the component which are needed to make analyses of extra-functional properties or simulations. The argument that such analyses could directly work on the code are true, but would internally always include a component abstraction, as the simulation of code execution is usually no more beneficial than executing the code itself. Therefore, using an abstraction of the component's internals, like the Palladio Component Model (PCM) from section IV, is required in any case.

Once abstractions of component's internals (the intended model) have been obtained, one can

- answer questions on extra-functional properties like execution time, without plugging "real" components together,
- evaluate different design decisions by reasoning via analysis and simulation results, without buying all required components,
- plan extensions of existing systems, without implementing new components, or
- answer sizing questions: finding appropriate hardware and middleware supporting desired loads before buying and configuring it.

III. Related Work

The approach presented in this paper is related to performance meta-models and component models. Three recent *performance meta-models* are compared by Cortellessa in [6]: The UML Profile for Schedulability Performance and Time (SPT) [7], the Core Scenario Model by Woodside et al. [8], and the Software Performance Engineering (SPE) meta-model [9] which are all designed for software systems in general – having no specific support for component-based approaches. KLAPER is a meta-model by Grassi et al. [10] designed for component-based software systems. To reduce complexity of models it introduces a unifying concept for resources and components. Opposite to this, the PCM used in our approach reduces modelling complexity by introducing developer roles, that only have to deal with sub-models (domain specific models).

Several other *component models* have been proposed, which do not focus on performance prediction. Each one has its own special focus on a set of particular aspects depending on the proposed application of the model. Models like EJB or COM are primarily aiming at industrial use. They have been designed to support component developers at an implementation level, while lacking the support for specifying or analysing extra-functional properties. Research oriented models (like SOFA [11]) are often bundled with a special analysis method for a set of system properties. In [12] a taxonomy of recent component models is given.

Architecture description languages like Rapide [13] facilitate an architectural description level of software systems including support for executable models. Rapide allows the specification of real-time constraints, but does not focus on component-based software systems. Darwin/Tracta [14] supports behavioral descriptions of software architectures including composition, but limits the description to finite state machines to check for example for unwanted behavior. Approaches like contract aware components introduced by Beugnard et al. [15] include behavioral and quality of service

level descriptions for components, but do not provide an implementation of the meta-model that is required for our performance prediction methods.

The need for information on components beyond interfaces has previously been stated by Büchi and Weck. They named their approach “greybox components” [16] and included parametric dependencies, but did not deal with extra-functional properties.

IV. The Palladio Component Model

The Palladio Component Model (PCM, cf. [2]) is a domain-specific modeling language to describe component-based software architectures. It is designed to enable design-time predictions for software architectures with respect to extra-functional properties like performance and reliability. Therefore, analysis and simulation methods exist that base on the PCM and calculate estimated behaviour (response times, resource utilisation, etc.) of a component-based software architecture.

Our analysis methods [17] are able to calculate metrics like the response time of provided services in a software system with respect to parametric dependencies within components and the actual usage profile of a software system. Simulation tools [2] generate simulation source code and scenarios, based on instances of the PCM. If, for example there are two design alternatives for a software system, both are generated into simulation code and afterwards evaluated in a simulation run. The outcome of such a simulation reveals preferable alternatives, pertaining to criteria like response time or bottlenecks, e.g. identified by overflowing simulated queues.

A rough overview on the PCM introduces the basic concepts: The PCM supports the definition of components including composite structures as well as the definition of provided and required interfaces. Additionally, assembly connectors describe the connections between required and provided interfaces. Components can be added to explicit component repositories and later be reused in different contexts to build new software systems. This enables the PCM to facilitate the reuse idea of component-based software engineering.

However, such an architectural model of a software system is not sufficient to make predictions on extra-functional properties: There are no specifications on the internals of components. As we pointed out in section II (Fig. 1) a description of component internals needs to include the dependencies an invocation of a provided service has on the external (required) services.

A. Service Effect Specification

As an essential part of the PCM, the so-called Service Effect Specification (SEFF) abstractly describes the internal behaviour (control and data flow) of a component, solving the problems stated in the previous section. For each provided service of a component (e.g. `a()` from Fig. 1), the SEFF specifies the effects on external required services (`b()`, `c()`) of that component. The aim of a SEFF is to provide an abstraction of the internal behaviour of a component. In particular, in addition to internal actions and external calls a SEFF can include control flow constructs (“actions”) like loops, branches, resource acquisition and release, and forking.

Fig. 2. Component for the SEFF given in Fig. 3

Figure 3 (grey background areas) shows a simplified example of a SEFF on the left-hand side for the component depicted in Fig. 2. The service `getListWithLittleEntropy` (`gLWLE()`) is provided by a component. This service itself requires the services `sort()` and `isEntropyLessThan()` which are provided by an external component `CollectionComponent`.

On the right-hand side of the figure, a code listing is given, showing the implementation of the service, while the left-hand side shows a SEFF notation (based on UML 2 activity charts [18]) of the service. If the shown service is called, a `while` loop is executed at first. This is depicted as a `LoopAction` node. Next, inside the `while` loop `gLWLE()` executes a `for` loop. As there are no component-external calls in the `for` loop, the code is merged into one `InternalAction`. SEFFs always merge actions into one `InternalAction`, if they are containing no component-external calls. Even, if there are hundreds of lines of code including loops etc., they are merged.

The following `if` statement results in a `BranchAction` (“diamond”). Within the branch and after it, the required services `sort()` and `isEntropyLessThan()` are called respectively. Those calls are provided by an external component and therefore result in an `ExternalAction`. It becomes clear that the SEFF allows to recognize dependencies between provided and required services of a component.

B. Resource Demanding SEFF

To enable more precise predictions for component-based systems, Resource Demanding SEFFs (RDSEFF) have been introduced [2]. They are an extension of the original SEFF-concept that adds two primary features: *Parametrisation and parametric dependencies* and *Resource usages for actions*.

1) Parametrisation and Parametric Dependencies: Parametrisation and parametric dependencies allow to specify how input parameters of a service call are passed to actions. In the above example, the input parameter `count` has an influence on the execution time of the `for` loop and thus has influence on the response time of the provided service. The larger `count` is, the more time the loop consumes. For external calls, parametrisation is supported, as well.

A RDSEFF’s primary aim is to provide an *abstraction* of the components behaviour to facilitate analyses. Of course, the source code of a component would be more precise than a RDSEFF, but analyses would suffer from complexity and component developers probably would not feel comfortable if the specification of their components (maybe available

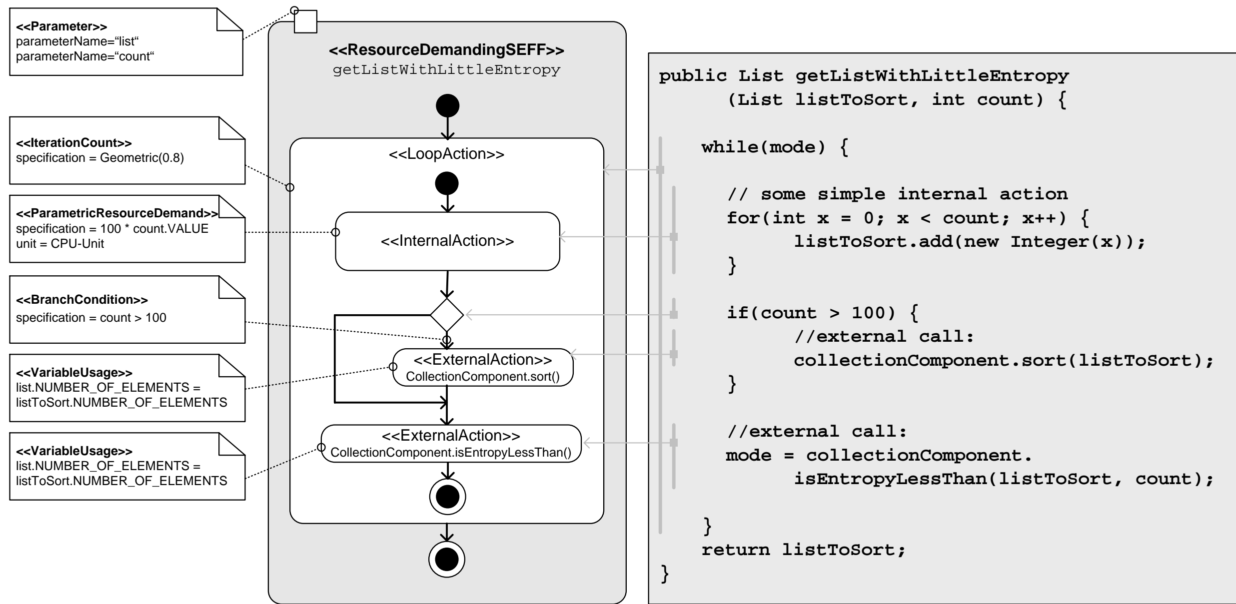


Fig. 3. Example of a simple SEFF: Abstract specification of a component service's behaviour

in an openly accessible component database as a white-box) contained the source code. Therefore, RDSEFFs merge component's internal behaviour and provide abstractions for parameters. For example, lists are not characterized by each of their elements but by the number of elements and/or the bytesize of the whole list.

2) Resource Usage: Still, there is lack of information for predicting metrics like the response time of a provided service. So far, the presented specification leaves out a mapping to the underlying hardware, for example, information on the actual usage of resources, although the execution environment has strong influence on the actual execution time of services.

To solve this problem, RDSEFFs allow the specification of (parametrised) resource usages. Fig. 3 lists such specifications. There is a `ParametricResourceDemand` of the `InternalAction` that defines a demand depending on the input parameter `count` and adds a mapping to the usage of *CPU-Units.*

3) Support of Stochastic Specifications: If one looks at the example given in Fig. 3 it becomes clear that it is not easy to determine the loop break condition (`mode`) of the `while` loop. The loop break condition depends on the boolean return value of the external service call to `isEntropyLessThan()` and the evaluated entropy strongly depends on the elements of the list. The parameter characteristics presented so far are not sufficient to specify the loop break condition in a satisfying way, because a functional description of the loop break condition would get too complex. Therefore the PCM includes support for stochastic expressions. The loop break condition is described by a distribution function that describes the probability of a certain number of loops (in the example `IterationCount` with a geometric distribution). The same applies to `BranchTransitions`. Fig. 3 includes one `BranchAction` within the `LoopAction`. In the example, there is a parametric specification (`GuardedCondition`) for selecting one or another branch, but the PCM supports stochastic definitions for branch transitions, too.

V. Reengineering Approaches

Section IV-A showed that SEFFs are required for predictions of extra-functional properties. An ideal SEFF should maintain a sufficient abstraction level to keep complexity of analyses bounded, while being as accurate as possible. The chosen abstraction level should enable analysis methods to finish within a reasonable amount of time.

If one thinks about a scenario in which components are offered in market places, predictions on extra-functional properties require exact component specifications as provided by SEFFs in the PCM. Software architects model their software architecture using the provided specifications of existing components. Based on meta-models like the PCM, the software architect could reason on design alternatives (and buy only those components that fit). In doing so, component developers could keep their intellectual property, as the SEFFs provide a sufficient abstraction of the source code, nevertheless supporting the re-use of their components. SEFFs keep the black-box view of components because they provide only on information that is required for external use.

Especially RDSEFFs tend to become complex constructs – a manual reconstruction would lead to a high effort for components developers which provide RDSEFFs. Tools supporting the automated reconstruction of component behaviour specifications are therefore desired. The benefit of tool-supported reengineering is ensuring the black-box principle of components. Though reengineering tools have a white-box view of components, their outcome (component behaviour specification) gives a black-box view.

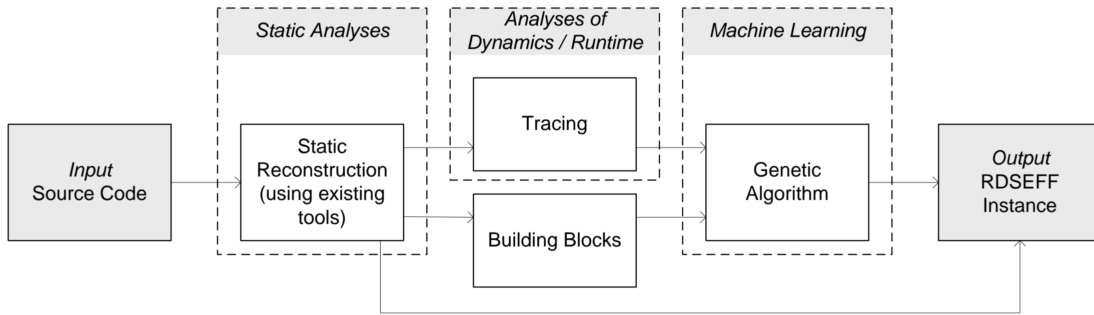


Fig. 4. Intended Process

We sum up the requirements to reengineering tools for RDSEFFs and propose our own approach: Reengineering must work in an automatic way to reduce the effort of gaining the specification and to keep the black-box principle. It must provide an abstraction of a component's source code. Therefore, it is required to find out important parameters and significant characteristics of complex parameters to represent an abstraction of real parameters. Parametric dependencies and resource usages must be revealed. Alternative to parametric dependencies, expressed in functional descriptions, stochastic data on the choice of branches and the number of loop executions should be available.

Current popular reengineering tools like Sotograph [19], Lattix [20], SonarJ [21] or even the academic Bauhaus [22] focus on reconstruction of static code structures. Those tools reconstruct classes, their inheritance, compute metrics like cohesion, cycles and many more – but they remain "class tools" that are not able to reengineer component's behaviour descriptions as needed for prediction of extra-functional properties. Other approaches like DiscoTect [23] emphasize the runtime monitoring but leave out data from static analysis. In any case, additional analyses are required to extract appropriate component behaviour specifications like RDSEFFs.

A. Methodology

Our idea for the reengineering of component behaviour descriptions (in our case RDSEFFs) combines static analyses, as supported by existing tools, with runtime tracing of components. The results of these static and dynamic analyses are fed into an evolutionary algorithm [24]. The task of the evolutionary algorithm is to find an abstraction of parametric dependencies – which are easier to analyse than the full source code, especially leaving out QoS invariant parameters and simplifying functional dependencies. The output of the evolutionary algorithm finally provides values for parametrising a RDSEFF. Fig. 4 sketches the intended process which needs to be done only once by the component developer.

The initial point for the reengineering is the source code of an application (left-hand side) – the reengineering target is an instance of a RDSEFF (right-hand side). There are three steps that are combined for reengineering (details are discussed below): static analysis of source code, analyses of runtime behaviour by tracing instances of a component, and machine learning.

The reengineerings process starts with static source code analysis for three reasons:

a) : The static structures of RDSEFFs (internal actions, external actions, loops, branches) should be reconstructed based on static analysis: An abstract syntax tree (a lot of tools for AST extraction exist) is very similar to the control flow part of a RDSEFF.

b) : For finding appropriate instrumentation points for tracing, it is required to know where loops, branches, external calls, etc. start and end in the code. If one would trace absolutely everything, the resulting trace would be too large to be evaluated efficiently. For example, for tracing it is important to find parameters that are meaningful. Constants that are passed as parameters will not improve knowledge on a component's behaviour. Consequently, there is no a priori knowledge on significant parameters. For sorting complex parameters and variables like `list`, it might be sufficiently to characterize the list by the number of its elements (`list.size()`), whereas in other cases like data streaming the list's bytesize is more meaningful. Tracing the whole data of every object would take ages or exceed the memory for logging in many cases. Finally, tracing results in monitored parameters and variables, the execution time between two measuring points, the resource usage, and in information on the probability of selected branches.

c) : The initial "building blocks" (the initial genome) for the evolutionary algorithm have to be extracted. For example, the execution duration of a `for`-loop in many cases is a linear function, depending on the starting size of an iterator. The evolutionary algorithm takes such functional blocks to improve the initial genome. For example, this could lead to an approximation of parametrised dependencies like $execution_duration = iterator_size * time_per_iteration + offset$.

An evolutionary algorithm's task is to judge on the fitness of genomes. To do so, the evolutionary algorithm takes two target functions to optimize the genome: (i) the evaluated results of the tracing step and (ii) the complexity to calculate functional dependencies found. The better the trade-off between easy to calculate functional dependencies and matching with the tracing results fits, the better a genome is.

Finally, we want to state why a mix of static and dynamic approaches appears necessary. Static analysis provides important information on the static structure of a RDSEFF. Without knowing these details, it would be hard to reconstruct a RDSEFFs control structure (based on tracing data only). The necessity of analyses of dynamics become clear from the example in Fig. 3. The loop exit condition of the while-loop (`mode`) is manipulated by the loop body itself. In the general case the halting problem would prohibit determining a loop exit condition by static analyses. Monitoring runtime behaviour nevertheless collects data on the loop exit condition.

B. Limitations and Assumptions

For the idea presented above, we assume that component boundaries and its interfaces are given. As SEFFs contain only component-external calls, this is a prerequisite for classifying methods calls into internal and external actions.

Our idea for reengineering is not intended to support the reconstruction of concurrency of components, though it heavily influences their extra-functional properties. Concurrency is neither fully supported by the PCM nor are our analysis methods supporting concurrency at present. Nevertheless, future work might deal with concurrency.

We are monitoring the runtime behaviour of components – a kind of testing approach. Therefore we need to feed test data into traced components. But of course test data will never be able to cover the whole input space of provided services of a component or cover all possible execution paths within finite time [25]. This implies that we need to find adequate test data to gain meaningful results.

For runtime tracing of a component, all it's required services need to be available. We assume implementations of those interface are available or can be simulated [26] or mocked.

Our RDSEFFs are a model of a specific component for the purpose of QoS predictions. This rises the question how to deal with component substitution, as for complex QoS specifications like RDSEFFs no simple substitutability check exists. We are currently researching formal substitutability checks for RDSEFFs.

VI. Conclusions

For many, the black-box principle of component usage means that a component does not expose *any* information on its interna. Accordingly, components are used at design- and run-time solely via their interfaces. The often cited definition of a component by Clemens Szyperski [1, p. 548] very much reflects this view. However, in this paper we have shown that for the analysis of extra-functional properties, one needs information on the interna of a component which is *not* given in the interfaces.

Since this trade-off between the black-box nature of a component on the one hand and the architectural and QoS analysis on the other hand seems to be unavoidable, we can ask for the reasons and benefits of the black-box nature of components. From our point of view, there are two reasons why the black-box nature of components should be kept in high esteem (in fact, anything else would drastically question the component-based approach):

1) The re-use of a component at design-time should be as easy as possible. In particular, one should not have to understand the component's interna to compose and deploy the component ("information hiding"). This is one of the major benefits of components compared to other software re-use approaches.
2) The "business" knowledge of the component creator should be protected. Unlike open source software, the creator of a re-usable component may base his or her business model on the assumption that the know-how which was used to create the component is protected and not "lost" when the component is released.

However, we take the position that these requirements do not conflict with the retrieval of additional information on a component beyond its interfaces. Therefore, we presented a component reengineering approach supporting information in addition to interfaces to enable the prediction of extra-functional properties – in our case performance. These models of components (RDSEFF) are going to be extracted using an approach combining static and dynamic analysis with an evolutionary algorithm.

Future work will deal with the evaluation of our approach. For a start we plan to trace parametric dependencies by monitoring primitive data types which then are analysed by existing implementations of machine learning algorithms, finally recovering functional dependencies.

References

[1] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York, NY: ACM Press and Addison-Wesley, 2002.

[2] S. Becker, H. Koziolek, and R. Reussner, "Model-based performance prediction with the palladio component model," in *Workshop on Software and Performance (WOSP2007)*. ACM Sigsoft, February5–8 2007.

[3] J. A. Stafford, A. L. Wolf, and M. Capuroscio, "The application of dependence analysis to software architecture descriptions," in *Formal Methods to Software Architects*, M. Bernardo and P. Inverardi, Eds., vol. 2804, 2003, pp. 52–62.

[4] R. H. Reussner, "Enhanced component interfaces to support dynamic adaption and extension," in *34th Hawaiin International Conference on System Sciences*. IEEE, 2001.

[5] R. H. Reussner and H. W. Schmidt, "Using Parameterised Contracts to Predict Properties of Component Based Software Architectures," in *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, I. Crnkovic, S. Larsson, and J. Stafford, Eds., 2002.

[6] V. Cortellessa, "How far are we from the definition of a common software performance ontology?" in *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, Palma, Illes Balears, Spain, 2005, pp. 195–204.

[7] O. M. G. (OMG), "UML Profile for Schedulability, Performance and Time," 2005. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2005-01-02

[8] M. Woodside, D. C. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (PUMA)," in *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, Palma, Illes Balears, Spain, 2005, pp. 1–12.

[9] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

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

[11] F. Plasil, D. Balek, and R. Janecek, "SOFA/DCUP: Architecture for component trading and dynamic updating," in *ICCDS98*, ser. IEEE Press. IEEE Press, pp. 43–52.

[12] K. K. Lau and Z. Wang, "A Taxonomy of Software Component Models," in *Proceedings of the 31st EUROMICRO Conference*, 2005, pp. 88–95.

[13] Stanford University, "The Stanford Rapide Project: Homepage," http://pavg.stanford.edu/rapide/, last retrieved 2007-05-12. [Online]. Available: http://pavg.stanford.edu/rapide/

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of ESEC '95 – 5th European Software Engineering Conference*, vol. 989, 1995, pp. 137–153.

[15] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, 1999. [Online]. Available: http://www.computer.org/computer/co1999/r7038abs.htm;http://dlib.computer.org/co/books/co1999/pdf/r7038.pdf

[16] M. Büchi and W. Weck, "The greybox approach: When blackbox specifications hide too much," Turku Center for Computer Science, Tech. Rep. 297, 1999. [Online]. Available: http://www.abo.fi/~mbuechi/publications/TR297.html

[17] H. Koziolek and V. Firus, "Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation," in *Proceedings of FESCA2006*, ser. Electronical Notes in Computer Science (ENTCS), 2006.

[18] Object Management Group (OMG), "Unified modeling language specification: Version 2, revised final adopted specification (ptc/05-07-04)," 2005. [Online]. Available: http://www.uml.org/{\#}UML2.0

[19] Sotograph homepage, "http://www.software-tomography.com/," last retrieved 2007-02-10. [Online]. Available: http://www.software-tomography.com/

[20] Lattix homepage, "http://www.lattix.com/," last retrieved 2007-02-10. [Online]. Available: http://www.lattix.com/

[21] SonarJ homepage, "http://www.hello2morrow.com/en/sonarj/sonarj.php," retrieved 2007-02-10. [Online]. Available: http://www.hello2morrow.com/en/sonarj/sonarj.php

[22] Bauhaus homepage, "http://www.iste.uni-stuttgart.de/ps/bauhaus/," retrieved 2007-02-10. [Online]. Available: http://www.iste.uni-stuttgart.de/ps/bauhaus/

[23] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 454–466, July 2006.

[24] J. R. Koza, *Genetic Programming – On the Programming of Computers by Means of Natural Selection*, 3rd ed. The MIT Press, Cambridge, Massachusetts, 1993.

[25] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence [program testing]," *Software Engineering, IEEE Transactions on*, vol. 16, no. 12, pp. 1402–1411, 1990.

[26] P. Parizeka and F. Plasil, "Specification and Generation of Environment for Model Checking of Software Components," in *Proceedings of Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006), Vienna, Austria*, ser. ENTCS, March 2006.

Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs

Michael Kuperberg and Steffen Becker
Institute for Program Structures and Data Organisation
Faculty of Informatics, University of Karlsruhe (TH), Germany
Email: {mkuper|sbecker}@ipd.uka.de

***Abstract*—Performance prediction of component-based software systems is needed for systematic evaluation of design decisions, but also when an application's execution system is changed. Often, the entire application cannot be benchmarked in advance on its new execution system due to high costs or because some required services cannot be provided there. In this case, performance of bytecode instructions or other atomic building blocks of components can be used for performance prediction. However, the performance of bytecode instructions depends not only on the execution system they use, but also on their parameters, which are not considered by most existing research. In this paper, we demonstrate that parameters cannot be ignored when considering Java bytecode. Consequently, we outline a suitable benchmarking approach and the accompanying challenges.**

I. Introduction

To meet requirements and expectations of users, modern software systems must be created with consideration of both functional and extra-functional properties. For extra-functional properties such as performance, early analysis and prediction reduce the risks of late and expensive redesign or refactoring if the required extra-functional properties are not satisfactory.

The performance (i.e., response time and throughput) of component-based software systems depends on several factors [1]:

a) the *architecture* of the software system, i.e. static structure of components and connections
b) the *implementation* of the components that comprise the software system
c) the *runtime usage* of the application (values of input parameters etc.) and
d) the *execution system* (hardware, operating system, virtual machine, middleware etc.) on which the application is run.

Making the influence of the execution system on performance explicit and quantifiable will help in different scenarios:

- **Redeployment of a component-based application in an execution system with different characteristics** (Fig. 1(a)): The execution system's characteristics will change when, for example, an operating system upgrade is conducted or when a more powerful server is bought. Assessing resulting performance changes *before* redeployment to compare benefit and costs beforehand is very reasonable.
- **Estimation of suitable execution system to fulfill changed performance targets for an existing software system ("Sizing")** (Fig. 1(b)): Changes in the usage profile (i.e., number of concurrent users, increased user activity, different input) of a business application may require adaptation of the application's performance that cannot be fulfilled with the original execution system. Performance prediction can be useful in choosing which execution system can fulfill the changed requirements.

The straightforward approach for predicting an application's performance on a new execution system would be to deploy the application there and then to test it, obtaining the resulting performance. While generally possible, testing requires installing the software on every concerned system and causes effort to provide the components implementing required services, an appropriate test workload and settings as well as further effort to take measurements etc.

To avoid the expensiveness and the inflexibility of the testing-based approach, the *building blocks* (such as bytecode instructions) which constitute component implementation can be used for performance prediction. Each building block's *performance* is obtained by benchmarking. The obtained performance is combined with the *frequency* of the building block, i.e. with the number of times the building block is executed during the execution of the component's services (cf. [2], [3]). In this paper, we target components that are compiled into Java bytecode and executed on a Java virtual machine, but the approach for .NET and other bytecode-oriented execution systems would be very similar.

The performance of such building blocks on an execution system can be captured in several ways. One possibility is to consider individual hardware and software resources that comprise the execution system and to evaluate the individual *resource consumptions* induced by the building block execution (for example, CPU time or memory usage). The results of these separate measurements must be integrated into a performance specification for the *entire* execution system. That is, the final result for the considered building block must be response time or another suitable metric.

Yet in general, such a bottom-up approach leads to a large

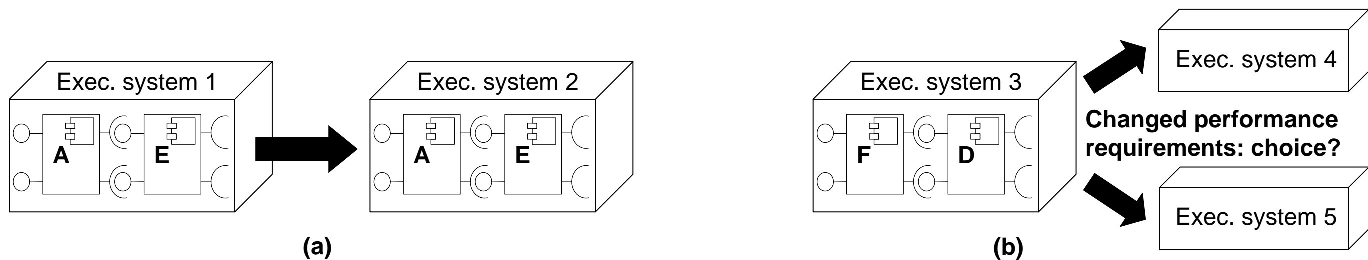


Fig. 1. Redeployment: scenarios for performance prediction

and complex model of the execution system. Additionally, the usage of the different resources of the execution system may lead to cross-influences. For example, memory-intensive computations may on some execution systems lead to unexpected hard disk usage due to swapping of virtual memory. Such cross-influences can impact several execution system resources and a detailed analysis of the resulting effects is required to evaluate the overall performance of a building block.

Hence, a simpler alternative is to consider the interaction of the components (i.e. their building blocks) with the entire execution system as interaction with one large black box, i.e. at a high abstraction level.

Furthermore, benchmarking-based performance prediction as utilised in this paper should also be faster than testing, as there is no need to run the considered component service for its entire wall-clock time duration. If the considered component service performs *external calls* to services of other components, their performance must be incorporated into the prediction. This can be done through simulation or analytical methods, and both should also be faster than testing.

Using bytecode instructions (or other building blocks) for predicting performance of software systems is not limited to component-based software. However, it appears suitable for components (especially given the number of commercial component architectures that rely on Java), and it can be applied to systems where the source code is not available (such as legacy systems). Once component services are annotated with performance descriptions (e.g. mean response times), model-based prediction methods such as KLAPER [4] or Palladio [5] can use them to predict the performance of a component-based software system that is reusing such components.

First approaches for utilising bytecode for performance prediction are described in [2], [3] and [6], but they ignore the parameters of bytecode instructions, assuming that parametric dependencies at bytecode level are not important. Hence, while each instruction's frequency is counted by observing the application at runtime, identification of the instruction parameters is not attempted and not recorded.

The contribution of this paper is a demonstration of the importance of bytecode instruction parameters and, correspondingly, a comprehensive bytecode benchmarking approach design that takes care of parametric dependencies. We outline the needed steps and present the resulting challenges that we plan to address in future work.

To provide the background for our work, we explain bytecode-level parameters and present our case study in section II. Building on the study, we discuss the details on benchmarking and performance prediction in section III-A. Challenges that must be addressed are summarised in section III-B. General limitations and assumptions of the resulting approach are described in section IV. In section V, we consider related work and compare it with our approach. Section VI concludes by describing future work.

II. Java Bytecode Instructions and their Parameters

A. *Java Virtual Machine, Java Bytecode and Java API*

Applications targeted for execution on the Java platform are compiled into *Java bytecode*, an intermediate language that is less machine-specific than pure native machine code. Java bytecode is executed on the *Java Virtual Machine* (JVM), which abstracts the specific details of the underlying software/hardware platform. The usage of bytecode makes compiled Java components portable across different platforms.

Through the standard *Java API*, application programmers have access to a number of libraries that are bundled with the JVM. These libraries implement frequently needed functionality, such as collections, inter-program communication etc. The functionality is accessed from Java components by calling the API methods and using API interfaces/classes.

From the component point of view, when a class that is part of a component makes an API call, this call could be considered as a call to another component. However, following the properties of components as described in Szyperski [7, p. 30], the JVM and the attached Java libraries should not be considered as a component. This is because the JVM is neither a unit of independent deployment (it cannot be deployed where other Java components are deployed, namely inside a Java virtual machine), but also because the JVM cannot be composed by a third party into another Java component.

Furthermore, components are composed by connecting their provides and requires interfaces, but direct use of the Java libraries means direct access of API classes' fields, and therefore is beyond the interfaces provided by the API. Therefore, in this paper, we consider calls to the Java API (that are done from the compiled bytecode) as part of the internal implementation of the component, and not as calls to another component.

B. Java Methods and Their Parameters

In bytecode, four instructions (`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`) are used to invoke *any* methods from bytecode. This means that these instructions are used both for API methods and for methods provided by the calling component itself or provided by other components. The method signatures are passed to `invoke*` as bytecode instruction parameters stored directly inside bytecode. Therefore, the performance of these four method-invoking instructions will depend on the actual method invoked, as well as on the parameters of that method.

Consequently, each combination of `invoke*`, method signature and method parameters potentially has its own performance. However, previous approaches to bytecode benchmarking did not consider bytecode instruction parameters to account for this fact (see section V). As a method's parameters are already on top of the JVM stack when the method-invoking bytecode instruction is called, the method parameters should be detected and used for performance prediction, which has to be considered during benchmark construction and execution.

Inside the JVM, a call to a single API method can result in many method calls and bytecode operations. Using a profiler, one can verify that for a single call to `System.out.println` method, many bytecode instructions but also several other API methods are invoked in a *call tree*. Some call trees can include native methods, so it is in general not possible to decompose a call tree into a set of bytecode instructions that does *not* include any method invocations. Hence, method invocations must be considered as building blocks in a bytecode-based performance prediction approach, and invocations of private and/or native methods that are not visible through the API must be considered as well.

One possibility to treat the resulting call tree of instructions *and* methods is to record each one of them and to use a benchmark to predict the performance of the call that is the root of the call tree. Obviously, this is not a favorable approach due to the resulting complexity. Additionally, propagation of measurement errors for the nodes of the call tree and other factors (e.g., need to circumvent Java security measures to benchmark private methods) favor a different approach, namely treating `System.out.println` and other API methods in an *atomic* way. Therefore, we prefer to remain at the highest abstraction level that is possible for components that are compiled to bytecode, and the Java API methods will not be decomposed any further in our approach.

Apart from method calls, parameters of other bytecode instruction can be expected to play an important role when working with collections. In the following section, we consider initialisation of collections and identify important parameter characterisations with respect to performance.

C. Influence of Bytecode Parameters on Performance of Instructions

For our case study, we have considered the initialisation of two data structure type: arrays and `ArrayLists`. Arrays in Java have a fixed size that must be specified at initialisation time while an `ArrayList` can grow dynamically and no initial size must be specified for it at initialisation time (although the Java API documentation specifies that the default initial capacity will be 10).

Initialisation of data structures includes allocating memory, and it is therefore logical to expect performance of the initialisation to depend on factors such as

a) collection's initial size
b) collection type
c) the type of the collection's elements (including the difference between value types and reference types) and
d) the size of each collection element (for example 8 bytes for a `double` vs. 4 bytes for an `int`)

To validate whether these dependencies really do exist for bytecode instructions, we have created arrays by using appropriate bytecode instructions such as `newarray` and `anewarray`. We then compared the results with initialisation of `ArrayLists`, which corresponds to a method call inside bytecode. The results in Fig. 2 show for each initial collection size (on the x-axis) the median of 100 measurements. In each measurement, 400 initialisations took place. The execution system was MS Windows Server 2003 SP1 on a single-core AMD Sempron™3100+ (1.80 GHz, 1 GB RAM) with Sun Java 1.5.0_11 and standard settings.

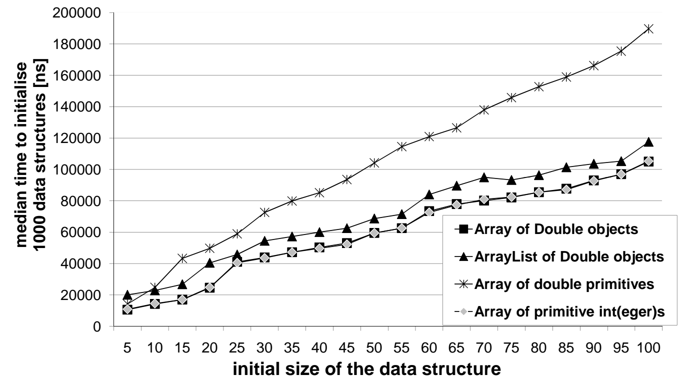


Fig. 2. Bytecode instructions' parametric dependencies in data structure initialisation

In fact, the measured values grow almost linearily with the initial collection size and all four expected parametric dependencies are exposed. This contradicts most existing approaches, which do not identify these impacts on bytecode instructions; at best, as in [3], only some few Java API methods where linear dependency is expected are measured and their performance is specified on a per-element basis.

III. Benchmarking of Java Bytecode and API

In this section, we present our approach for bytecode-based performance prediction of components and outline the consequences of considering the instruction parameters. The description of the methodology is followed by the discussion of the resulting challenges in section III-B.

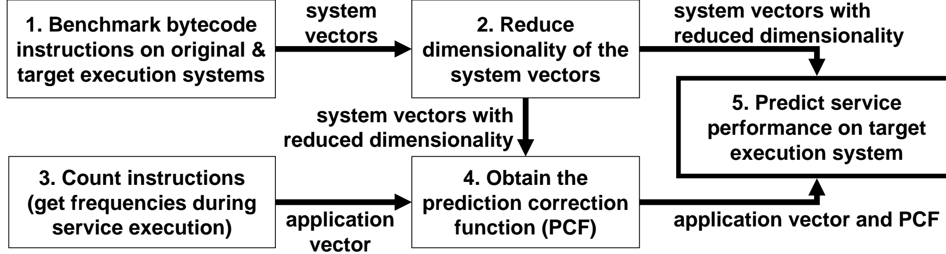


Fig. 3. Steps to predict performance using bytecode benchmarking

A. Methodology

We follow the notation in [2], where frequency of individual bytecode instructions is expressed by the *application vector* $\vec{P} = (p_1 p_2 p_3 \dots p_n)^T$. In $\vec{P}$, n denotes the number of available instructions and $p_i \geq 0$ is the execution frequency of instruction i. Correspondingly, performance of instructions is collected in the the n-dimensional *system vector* $\vec{S} = (s_1 s_2 s_3 \dots s_n)^T$. The performance prediction is obtained by $\vec{P} \cdot \vec{S}$ under the assumption that with the same usage (i.e., input parameters etc.), $\vec{P}$ remains the same accoss execution systems and component service executions. In Fig. 3, the steps of the proposed methodology are outlined.

Step 1. Benchmarking Bytecode

As we need to benchmark the bytecode instructions individually, a suite of microbenchmarks must be constructed. The microbenchmarks will fall into two groups: (a) the API methods that are called when instructions such as `invokevirtual` are executed and (b) the instructions that do not call the Java API.

When parameters have to be generated, attention must be paid to their representativeness and several microbenchmark runs must be performed if necessary (e.g. to discover linear dependencies). Parametric dependencies must be reflected by elements of $\vec{S}$. Considering the fine-granular nature of bytecode instructions, the microbenchmarks should measure a sufficient number of instructions with regard to timer resolution.

This will be problematic when it is not possible to separate the preparation of parameters passed over the JVM stack from the execution of the instruction itself. The performance of parameter preparation must then be benchmarked and subtracted from such results.

The benchmarking step has to be executed once for each execution system, and changed settings of the execution system require a repetition of this step. Also, the microbenchmarks should be executed several times to be able to eliminate outliers and to obtain statistically significant results.

Step 2. Reducing the System Vector's Dimensionality

The benchmarking step produces a very large system vector due to instruction parameters, the API methods and their parameters. We have envisioned two potential ways to prevent the size of the system vector $\vec{S}$ from becoming prohibitively expensive to work with: (a) usage of parametric descriptions in system and application vectors, for example duration per character in string conversion and (b) clustering of system vector's elements to group measurements with similar values. These methods could be combined as well.

Step 3. Obtaining the application vector

From the execution of the component's service on the original execution system, we obtain the frequencies p_i for the application vector $\vec{P}$. Since we assume that the component will be run on the target execution system with the same service parameters (i.e., runtime usage profile will remain stable), we assume that $\vec{P}$ will be valid there as well. For obtaining $\vec{P}$, we aim at bytecode instrumentation which has to be carried out on the original execution system only.

Step 4. Obtaining the prediction correction function

Hotspot compilation, JVM optimizations and other techniques might distort the prediction. To cope with this, we plan to introduce a *prediction correction function* (PCF). PCF could be obtained, for example, by an algorithm that uses machine learning to quantify the difference between prediction and reality (i.e., measurements). To learn, the algorithm takes a small fixed set of generic component services and compares prediction and measurement. From this comparison, a correcting function is derived that could be applied to the performance prediction of other component services. Of course, the application of PCF requires a representative learning set and a learnable correction function.

Step 5. Predicting the performance

The prediction is computed by $\vec{P} \cdot \vec{S}$, i.e. by $\sum_{i=1}^{|\vec{P}|} p_i \cdot s_i$. For this, we assume that elements of the system vector $\vec{S}$ contain only simple numerical values. However, API methods that sort a collection perform differently for *same* size of the collection because the sorting effort depends on the orderliness of the collection. If probability distributions are used to express the resulting variation of the performance behaviour, [8] describes the computation of $\vec{P} \cdot \vec{S}$ using convolution.

B. Challenges

Of course, not all instructions are as expensive as collection initialisation or API calls. Hence, some simple instructions may be orders of magnitude faster and/or parameter-independent. We aim at identifying the computationally expensive bytecode instructions and at fully benchmarking *only* them and API methods, while approximating the performance of the computationally "cheap" instructions.

For benchmarking the Java API, additionally to method parameter generation and parameter range coverage (representability), we must deal with runtime exception handling, inheritance/polymorphism, invocation target instantiation (for non-static methods) and some other challenges.

An implementation of the JVM (i.e., mapping of Java bytecode instructions and Java APIs to machine code) is not specified and varies across vendors. It is possible that an instruction's performance will be different for distinct JVMs that executed on the *same* underlying hardware/operating system. We can count for this by considering the JVM as a part of the execution system and then specify the bytecode instruction performance for each execution system.

Modern virtual machines start the execution of bytecode in interpretation mode while trying to detect performance hotspots. These hotspots are further compiled into native machine code on the fly if the result is an overall performance increase. The standard Sun Microsystems JVM features the so-called *just-in-time compilation* (JIT [9]) capability, and for long-running business applications that our research targets, such compilation will be relevant. Thus, identifying and quantifying hotspot compilation is a challenge for our research.

Pipelining [10] is a technique used by CPUs to increase execution speed. For our methodology, it means that a sequence of bytecode instructions may execute in less time than the sum of individual execution times, thus distorting the prediction. A study must be performed to evaluate whether pipelining is important at bytecode (JVM) level.

Background memory (de)allocation, i.e. the so-called Garbage Collection (GC), is another important factor that depends on the implementation of a JVM is which is visible through irregular or periodical events halting or slowing the program execution. Such interruptions and delays can distort the prediction, and we will try to understand the effects of GC on long-running business applications by inspecting its duration in different modes, as for example exposed by usage of `-verbose:gc` parameter of Sun Microsystems' JVM.

Granularity of the benchmark atoms is important both for complexity, accuracy and precision. Analysis of instruction tuples in [11] has shown that some pairs are very frequent while others do not occur at all; [12] uses sequential bytecode blocks instead of individual bytecodes. For our research, we may evalute the use of individual non-API bytecode instructions to separate parameter preparation for API calls from the actual API calls' performance.

Error and error propagation are general issues in benchmarking and prediction; additionally, we can expect simple bytecode instructions to be very fast. Consequently, special attention must be paid to benchmark such instructions with respect to timer resolution, confidence intervals, outliers and similar aspects.

IV. Assumptions and Limitations

For an initial implementation of the proposed approach, some helpful assumptions must be made which limit the complexity of the undertaking. For instance, a virtual machine can include different optimisations of bytecode execution, for example by replacing some instructions with semantically equivalent but faster ones in vendor-specific ways. Such behaviour would distort the performance prediction by altering the application vector $\vec{P}$. Therefore, we assume that no such optimisations take place at runtime.

A component service implementation can include sections that may be executed parallely (for example, by spawning a new thread). Counting the bytecode instructions and combining them with the results of the bytecode benchmark as our approach proposes will then result in wrong prediction results if the service is executed on a system that offers hardware-supported parallelism. For now, our approach assumes that the execution of the service is not parallelisable.

The state of the components may impact their performance. As this is hard to measure and hard to predict, separate provisions would have to be developed for this. For now, we aim to abstract from the state of the individual component and to investigate it in future work.

The performance of the component's required services must be included into prediction. We assume that such external calls can be detected and integrated into prediction, for example by using Sevice Effect Specifications (SEFFs, [13]).

V. Related Work

Sitaraman et al. [14] discuss parametrics of both space and time behaviour of collection initialisation from the implementation viewpoint, while our work focuses on capturing the resulting dependencies during measurement/benchmarking and during prediction.

In the Java Resource Accounting Framework (JRAF [15]), Binder and Hulaas use bytecode instructions counting for the estimation of CPU consumption. However, all bytecodes are treated equally, parameters of individual instructions (incl. API method names) are ignored, which contradicts our case study findings. In JRAF, it is assumed that invocations of API methods break down to invocations of elementary bytecode instructions in a platform-independent way, while we consider API calls as atomic.

In HBench:Java [3], Zhang and Seltzer build the system vector by separating high-level JVM "components" such as system classes (i.e., API implementation), memory management, JIT and control flow/primitive bytecode execution. However, the evaluation was performed by selecting and benchmarking only 30 particularly expensive API methods (some of them were found to show linear dependency on one parameter), and no absolute comparison between measured and predicted performance is provided. Therefore, it remains questionable

whether the such “components” can be combined into a suitable and application-independent prediction. Furthermore, it is not clear how an application vector can be obtained with respect to the JVM ”components”. For our approach, we aim to consider the JVM as a black box and to study execution of bytecode instructions.

In [2], Meyerhöfer and Lauterwald want to measure the application vector by using interceptors in an application server. Interceptors then instrument the Java classes when they are loaded and the executed bytecode instructions are counted. Although API methods are mentioned in [2] as a potential extension for that approach, method parameters are not considered and it is not discussed how the substantially larger number of the building blocks can be handled. To deal with JIT effects, the authors suggest to maintain two performance measurements per instruction (for interpreted and JIT-compiled modes). However, it is not described how such measurements can be obtained.

VI. Conclusions

This paper presents a detailed description of a substantially refined performance prediction methodology. This methodology relies on bytecode instruction benchmarking and learns from the execution of existing component services to enhance the performance prediction on new execution systems. A case study is presented that shows the importance of parametric dependencies for API methods and other bytecode instructions. We discuss the inclusion of calls to the API into bytecode benchmarking, since API calls are a very important part of bytecode execution.

Other important factors influencing both benchmarking and prediction, such as respect of hotspot compilation and garbage collection are also discussed. Several suitable scenarios for the usage of the prediction algorithm are described and the challenges as well as assumptions and limitations are presented. We also mention ways to handle the large size of benchmark results that is caused by analysis of parametric dependencies and by API benchmarking.

The proposed approach can be beneficial for developers and architects who wish to evaluate the performance of components. For this, the bytecode benchmark can be intergrated into component-oriented modeling languages, such as the Palladio Component Model [5]. This can be done by constructing the behavioural specifications of the component services and annotating them with performance predictions obtained through the methodology proposed in this paper.

Future work will start with constructing the API benchmark and by investigating whether benchmarking of a subset of remaining bytecode instructions is sufficient for acceptable precision of prediction. After the validation of the proposed prediction methodology, we plan to extend it to allow probabilistic descriptions of benchmark results. Ultimately, we plan to integrate our approach into the modeling and prediction tooling of the Palladio Component Model [5].

References

[1] S. Becker and R. Reussner, “The Impact of Software Component Adaptation on Quality of Service Properties,” *L’Objet* -, vol. 12, no. 1, pp. 105–125, 2006.

[2] M. Meyerhöfer and F. Lauterwald, “Towards Platform-Independent Component Measurement,” in *Tenth International Workshop on Component-Oriented Programming*, W. Weck, J. Bosch, R. Reussner, and C. Szyperski, Eds., 2005.

[3] X. Zhang and M. Seltzer, “HBench:Java: an application-specific benchmarking framework for Java virtual machines,” in *JAVA ’00: Proceedings of the ACM 2000 conference on Java Grande*. New York, NY, USA: ACM Press, 2000, pp. 62–70.

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

[5] S. Becker, H. Koziolek, and R. Reussner, “Model-based Performance Prediction with the Palladio Component Model,” in *Workshop on Software and Performance (WOSP2007)*. ACM SIGSOFT, 2007.

[6] A. Camesi, J. Hulaas, and W. Binder, “Continuous Bytecode Instruction Counting for CPU Consumption Estimation,” in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006)*, pp. 19–30.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, Reading, MA, USA, 1998.

[8] H. Koziolek and V. Firus, “Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation,” in *Proceedings of FESCA2006*, ser. Electronical Notes in Theoretical Computer Science (ENTCS), 2006.

[9] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, 1999.

[10] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.

[11] J. Lambert and J. F. Power, “An Analysis of Basic Blocks within SPECjvm98 Applications,” Department of Computer Science, National University of Ireland, Maynooth, Co. Kidare, Ireland, Tech. Rep. NUIM-CS-TR-2005-15, 2005.

[12] P. Wong, “Bytecode Monitoring of Java Programs,” *BSc Project Report, University of Warwick*, 2003.

[13] R. H. Reussner, H. W. Schmidt, and I. Poernomo, “Reliability Prediction for Component-Based Software Architectures,” *Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes*, vol. 66, no. 3, pp. 241–252, 2003.

[14] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, “Performance Specification of Software Components,” in *SSR*, 2001, pp. 3–10.

[15] W. Binder and J. Hulaas, “Using Bytecode Instruction Counting as Portable CPU Consumption Metric,” *Electr. Notes Theor. Comput. Sci.*, vol. 153, no. 2, pp. 57–77, 2006.

Aspectual Dependencies: Towards Pure Black-Box Aspect-Oriented Composition in Component Models

Bert Lagaisse and Wouter Joosen
DistriNet, Dept. of Computer Science
K.U.Leuven, Belgium
Email: {bert.lagaisse,wouter.joosen}@cs.kuleuven.be

***Abstract*—In this paper we present the DyMAC aspect-component model that supports the specification of *aspectual dependencies*: these are explicit dependencies between advised and advising components involved in an aspect-oriented composition. DyMAC advocates a pure black-box approach to aspect-oriented composition and aspectual dependencies further extend this approach. The component model avoids explicit dependencies of aspects on the implementation of a component: syntactically as well as semantically.**

We validate the concept in the context of one of the major application domains of aspect-component models: AO-middleware. Aspectual dependencies can express required middleware services as explicit dependencies in the component specification. They untangle the composition of the component with the middleware. Middleware services can be well-modularized in reusable advising components with explicit interfaces that specify the aspectual behaviour.

I. Introduction

Distributed applications are typically built on middleware platforms, a software layer on top of the operating system that offers a component model and execution environment for these applications. Practical middleware platforms nowadays have to support complex composition of application components and have to support a broad range of services that deal with the non-functional concerns in a distributed application. Due to the specific needs of applications and the different deployment contexts, middleware platforms need to be customizable and extensible[3], [14]. Aspect-oriented middleware has been put forward as a promising and relatively successful paradigm to improve usability, extensibility and customization capabilities of middleware platforms [8], [17], [13], [16]. In such middleware, traditional component containers evolved from a blackbox component with a declarative interface for a fixed set of services, to an open micro-container with an open-ended list of services that are composed by means of aspect-oriented composition. The component model[1], [2] that is offered by the AO-middleware has been extended to enable *aspect-oriented composition*; the result is a so-called *aspect-component model* [8], [16]. Before elaborating on the problem, we explain the basic concepts of aspect-oriented software development (AOSD) and aspect-component models.

Basic concepts of AOSD. To address the problem of crosscutting (often non-functional) concerns, aspect-oriented software development (AOSD[5]) often has been put forward as a possible solution. AOSD addresses this shortcoming by focusing on the systematic identification, modularization, representation and composition of (often crosscutting) concerns or requirements throughout the entire software development process. The core concept in AOSD is an aspect [5], [4]: a coherent entity that addresses one specific concern and that has the properties of a module that can be changed independently of other modules. An aspect defines behaviour that can be executed (so called advice) and defines composition logic to describe complex and dynamic dependencies between this behaviour and the rest of the software system. This composition logic is expressed using a joinpoint model. A common definition of a joinpoint refers to well-defined places in the structure or execution flow of a program [4], [5]. In any case, joinpoints represent dynamic, runtime conditions that arise during program execution. The occurrence of such a condition is an event that can trigger the execution of aspect behaviour (advice). A set of joinpoints can typically be specified with pointcut designators that address and describe the kind and context of the joinpoints [5]. By the *kind* of a joinpoint we mean for instance a method call or a field access, etc. By the *context* we refer to additional information that can be made available to constrain the condition, such as the method signature, type and identity of the caller or callee of a method, further credentials and properties of the caller etc.

Aspect-component platforms focus on combining the benefits of AOSD and CBSD[1], [2]. The state-of-the-art aspect-component models are typically characterized by two main properties:

1) The joinpoint model is *non-invasive*: the kind and context of the joinpoints that can be advised are limited to elements in the interfaces of the components. These are typically incoming invocations on the provided interfaces and outgoing calls on the required interfaces. Deep advice in the implementation of the component is not allowed.
2) Aspects have component semantics: this includes clearly defined interfaces and support for third party composition. In the state of the art this is typically supported by decoupling aspects (pointcuts +advice) into (1) an aspect-component containing the (reusable) advices as well-named methods, and (2) an aspect-oriented composition specification that binds the advices to a set of joinpoints.

A problem with aspect-components. State-of-the-art AO-middleware offers an aspect-oriented programming model that supports the concept of aspect-components to define new middleware services in the middleware platform. Consequently, AO-middleware must offer appropriate means to specify the composition of application components with such middleware services - as these services are typically offered as aspect-components. However, state-of-the art AO-middleware does not allow application components (possibly provided by a third party) to express essentially required middleware services unless these services are modeled and implemented as traditional (non-AO) components with provided interfaces. In fact, if services are implemented as aspect-components, then they cannot be specified in the required interface of an application component. Middleware services are often needed by the component in order to ensure correct behaviour of the component and of the application. A classical example of such middleware services are synchronization and transaction services.

If two components interact in a traditional system, then one component is a calling component - relying on a required interface - and the other is the called component - offering a provided interface. Required interfaces are dependencies of a component that need to be fulfilled in order to guarantee correct behavior of this component. The implementation of the component typically includes calls to these required interfaces. Figure 1 sketches a banking component that interacts with a storage component.

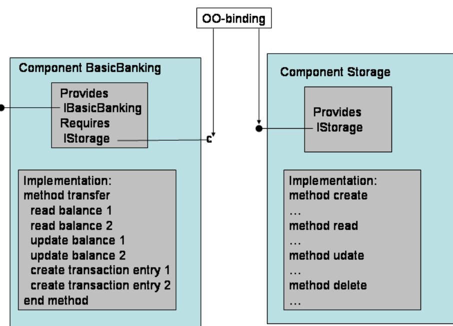


Fig. 1. Object-oriented composition

An AO-composition model deals with the complex, crosscutting composition of an aspect-component (the advising component) with other components (the advised components) in the application. These AO-compositions are typically specified in separate composition specifications such as the deployment descriptor of the application. Such a composition specification defines a set of middleware services that the middleware platform needs to offer for the execution of the application: (a) *essential* middleware services that are strictly required for the correct behaviour of the application components, as well as (b) middleware services that are specific for the given deployment context. An example of the former is transaction support, an example of the latter is an organization-specific authorization aspect. We illustrated such an AO-composition in Fig 2.

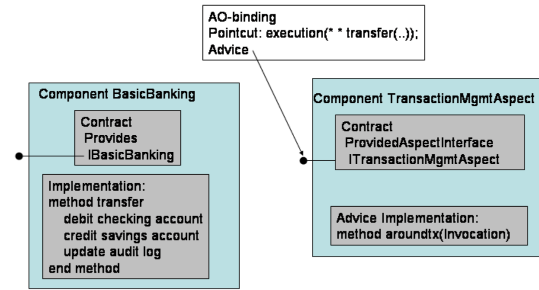


Fig. 2. Aspect-oriented composition

State-of-the-art component specifications are lacking support to specify explicit dependencies to (aspectized) middleware services that are essential for the correct behaviour of the application components. A traditional example of such a dependency is transaction management. Transactional behaviour is a crosscutting concern, that one wants to untangle from the component implementation, and specify in the component descriptor (cfr EJB[6]). The behaviour of the application component that needs to be advised by the transaction aspect needs to be specified by the component developer [18]. The component developer should never rely on the application composer or deployer to specify the required transaction management for the component[7]. Moreover, this would imply a (semantical) dependency to the implementation of the component, because the application composer needs to know what happens in the implementation in order to configure the transaction support.

This lack of information in the specification of components is an essential deficit to define components as self-containing black-box entities (that ideally, should be deployable by a third party). It also breaks the pure black-box approach to aspect-oriented composition: although the transaction-aspect composer does not have syntactical dependencies on the component implementation (because of the non-invasive joinpoint model), he has semantical dependencies on the implementation of the component in order to compose the necessary transaction support.

Furthermore, this lack of expressive power also makes it impossible for the AO-middleware to verify that all essential dependencies are satisfied. In practice an application composer or deployer cannot be aware of such unspecified required aspect-components. The resulting composition, in which each dependency *seems* to be fulfilled, can result in inconsistent application state at runtime, for instance - as suggested in the example above - due to the lack of transaction management.

Solution. In this paper we present the support for *aspectual dependencies* in AO-middleware, such that

1) Required middleware services can be specified as explicit dependencies in the component specification.
2) Middleware services can be well-modularized in reusable aspects, i.e. advising components with provided interfaces that explicitly specify aspectual behaviour.

This solution results in extensible and customizable AO-middleware that supports pure black-box aspect-oriented com-

position. We have implemented such a middleware platform and we have illustrated and validated our solution by supporting transaction management in a component-based application that is deployed on AO-middleware. In this validation we have tackled the crosscutting composition of the application components with the transaction service.

Note that required middleware aspects are inherently non-oblivious to the base components . A common - but incorrect - view on AOP assumes that components are always there first and aspect are independent extensions that are always imposed later on the unaware (oblivious) base components. Obliviousness is a property that can be realized by aspect-oriented composition, but it is not defining AO, and it also not a requirement for AO. A further discussion on obliviousness is out of the scope of this paper. Also note that in our approach we assume that required middleware aspects are known when defining the architecture of the distributed application. This also postulates that the interfaces of required aspects are defined at design time.

The remainder of the paper is structured as follows. In section 2 we define the basic concepts and the rationale behind our solution. In section 3 we describe how the DyMAC AO-middleware supports aspectual dependencies. Section 4 illustrates aspectual dependencies for the composition of a transaction service in DyMAC. Section 5 discusses related work. Finally, we conclude in section 6.

II. The concept of aspectual dependencies

In this section we present the essential concepts that are offered by our solution to support aspectual dependencies. First of all, the component description includes specific interface definitions for AO-compositions. Secondly, the component description explicitly describes aspectual dependencies for composition in an application. Third, dependencies have to be resolved at deployment time.

Interface definitions for aspect-oriented composition. In general, an interface definition specifies the behaviour of a component by means of provided and required interfaces. An aspect-oriented composition defines a composition between a set of advising components and components being advised. From the viewpoint of the component being advised, the advices can be essential (i.e. strictly required) for the correct behaviour of this component, or not. For instance, an advice that is only relevant for a particular application deployment context is not required from the viewpoint of the component being advised.

Interface definitions of application components need to support aspect-oriented composition in two cases: for *required aspects* in essential compositions and for *imposed aspects* in other compositions. This leads to the following types of specification in the interface definitions of the advising and advised components:

1) For required aspects:
   a) First, an aspect component (advising component) needs to be able to specify its provided functionality. This is specified in the provided interface of the advising component, which is an interface that is specific for aspect-oriented composition.
   b) Second, a component that needs advice must specify (1) its dependencies towards the AO-middleware: i.e the advising components that it expects in the middleware platform, as well as (2) its own corresponding joinpoints, i.e specifying *when* advice is needed.
2) For imposed aspects:
   a) An advising component should specify the interface it requires from the advised components: the kind of joinpoints the advising components wants to advise.
   b) A component that can be advised (i.e. a component that is subject to aspect-oriented composition) should specify the interface of joinpoints that it provides towards advising components. This element has actually been the focus until now of most related work on aspects and interfaces (open modules [19], XPI [21], aspect integration contracts[15]) - see section 5.

Our description therefore focuses on support for required aspects. Aspectual dependencies are the key concept that we put forward to bridge the gap.

Component specification with aspectual dependencies. An aspectual dependency defines an aspect that is required by a component. The component description therefore specifies these aspectual dependencies and makes the required interfaces of advising components explicit. An aspectual dependency consist of (1) a name for the dependency, (2) a pointcut that specifies the joinpoints, i.e. when and where the advice is required and (3) a required interface and advice-method that must be provided by a matching advising component. This obviously assumes that advising components can specify aspect behaviour in their interfaces.

The joinpoint model for defining the pointcuts is basically irrelevant for the nature of our solution. In practice, we have chosen for a model that specifies the kind and context of the joinpoints as follows. The kind of joinpoints are calls and executions of methods, which implies two kinds of pointcuts: call and execution pointcuts. The pointcuts can further evaluate on the context information that is associated with a joinpoint. This context information is of course limited to the context information about the component specifying the aspectual dependency. In case of a call pointcut the pointcut can evaluate on the required interface and method signature of the method call. In case of an execution pointcut the pointcut can evaluate on the provided interface and method signature of the method execution.

Resolving aspectual dependencies. Dependencies are resolved by specifying *bindings*. Our solution defines *aspectual dependency bindings* to map the aspectual dependencies to concrete advising components in the AO-middleware platform. These bindings are specified as an aggregation of (1) a set of dependencies, requiring the same interface, and (2) an advising

component that is selected to fulfill the required interface. These bindings are specified in the deployment descriptor of the application. They actually resolve the aspectual dependencies of the components that are contained in the component specification. An aspectual dependency and its binding is depicted in Figure 3.

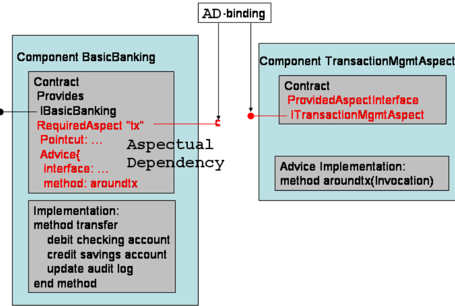


Fig. 3. Aspectual dependency

III. Aspectual dependencies in DyMAC

In this section we describe the support for aspectual dependencies in the DyMAC AO-middleware. First we focus on the specification of aspectual dependencies to express required aspects and on the specification of aspect interfaces.Second we discuss how DyMAC supports the two kinds of aspect binding: the binding of required aspects and the binding of imposed aspects.

A. Interfaces and dependencies in DyMAC

DyMAC components are object-based entities that separate interfaces and implementation. Components specify two kinds of interfaces: provided and required interfaces. The provided interfaces are declared in the component descriptor, and specify the behaviour that is provided for each kind of composition: i.e. their provided behaviour as called component in a object-oriented composition and their provided behaviour as advising component in an aspect-oriented composition. DyMAC components declare their required interfaces by means of dependencies in the component descriptor. A dependency can be a normal dependency or an aspectual dependency. We explain these different concepts and illustrate them shortly. A more detailed illustration is presented in the next section.

The provided interfaces of a DyMAC component consist of a *create-interface* and an *instance-interface*. A create-interface specifies how to instantiate a component, and an instance-interface specifies which methods are offered by a component instance. We illustrate this for the BasicBanking component in the next listing. [1]

```
interface IBasicBankingCreate{
 IBasicBanking Create();}
interface IBasicBanking{
 ...
 void Transfer(string from, string to, double amount,
                                    string msg);}
```

[1]The examples are implemented on the DyMAC.NET prototype and use plain C#.

The advice-methods of a component are also defined in the provided instance-interface of a component. The advice-methods represent the provided aspect interface of the component and they can be used as advice in an aspect-oriented composition. These methods need to have a special signature:

```
interface TransactionMgmtAspect{
  void aroundtx(RuntimeJoinPoint rjp);}
```

The dependencies of a component express the required behaviour of other components that the component is composed with. Normal, object-oriented dependencies specify a required interface on which the component explicitly calls method invocations. Such a dependency is defined by a dependency name and a required interface that is expected to be bound to the dependency by means of OO-composition.

Aspectual dependencies extend the notion of dependencies for required aspects. They are defined by a name, pointcut and an advice of a required aspect interface. The joinpoint model for the pointcuts in aspectual dependencies supports calls on the required interfaces, including the advices of required aspect interfaces. It also supports executions of methods in the provided interface of the component, including the provided aspect interface of the component. The advice specified in an aspectual dependency must specify at least an AspectInterface and advice method. AdviceType and InstantiationScope are optionally, and can be left open to specify in the binding.

The component descriptor defines the component name, the provided interfaces, the implementation file and the dependencies of the component. We illustrate the component descriptor of BasicBanking with the aspectual dependency towards the *aroundtx* advice.

```
Component BasicBanking{
  ...
  Aspectual-dependency transactions{
    Pointcut{
      Kind: execution
      Interface: IBasicBanking
      MethodMessage: * transfer(..);
    }
    RequiredAdvice{
      AspectInterface: ITransactionMgmtAspect
      AdviceMethod: aroundtx;
      AdviceType: Around;
      Scope: PerInstance;
    }}
```

B. Binding components in DyMAC

The composition of the application and the middleware is defined in an application descriptor. The application descriptor first defines a name for the application, then it defines the set of components that is used, by referring to their descriptor. Then the compositions of the components are defined. These compositions can be (1) normal OO-bindings between OO-dependencies and components, or they can be (2) aspectual dependency bindings of required aspects, or they can be (3) aspect-oriented compositions of *imposed* aspects. In this way an application is constructed on a custom middleware platform with the required middleware services and the imposed application-specific middleware services. We now focus on the bindings of required aspects and imposed aspects.

1) Aspectual dependency bindings: are defined in the application descriptor of DyMAC applications. They bind a set of aspectual dependencies to a concrete advising component in the application. This set of aspectual dependencies typically requires the same aspect interface. This is illustrated in the next listing. Optionally, the advice type and instantiation scope can be defined.

```
Application BankingApplication{
 ...
 Aspectual-dependency-binding{
  Dependency: BasicBanking.transactions, ...;
  Component: TransactionMgmtAspect.ITransactionMgmtAspect;
 }}
```

2) Binding imposed aspectual middleware services: Aspect-oriented compositions of imposed, application-specific middleware services are defined in the application descriptor. This is only discussed briefly in this paper and is presented more thoroughly in our previous work [9]. Aspect-oriented compositions consist of two main parts: a pointcut expression and a set of advices. We first discuss pointcut expressions, then we discuss advices. The next listing specifies the stucture of such an aspect-oriented composition as a guidance to the textual explanation.

```
AO-composition{
  Pointcut{
    Kind: [call|execution];
    Signature: <method-pattern>;
    Caller{
      [<property>: <string-pattern>;]*}
    Callee{
      [<property>: <string-pattern>;]*}}
  [Advice{
    AdvisingComponent: <component-name>;
    Scope: [Singleton|PerHost|PerComponent|...];
    AdviceType: [Before|After|Around];
    AdviceMethod: <advice-method>;
  }]*
}
```

Pointcut expressions. Pointcuts are logical expressions that evaluate on the kind and context of the joinpoints. The kind of joinpoints that can be evaluated on are calls and executions of method invocations. The pointcut expressions to support that are a *call and execution* pointcut. The context on which the pointcuts can evaluate are the message signature, the dependency name of the sending component, the interface of the receiving component, and the names of the sending and receiving component. If pointcuts do not specify a value for a certain property, it has the default value *all*.

Advices. An advice is described by (1) the component that provides the aspect behaviour, (2) its instantiation scope, (3) an advice method of the advising component and (4) an advice type (before, after or around).

IV. ILLUSTRATION

In this section we illustrate the composition of required middleware services in DyMAC. First we present a description of the example. Then we elaborate on the composition of the middleware services in DyMAC.

A. Situating the example

We use an example of a webbanking application with which a customer can manage his checking accounts and perform financial transactions on the checking accounts. The core business component is BasicBanking, which is a component offering operations to manage checking accounts, and execute financial transactions. This component is located at the application server. The customers use the WebClient component at the webserver. The webclient sends the requested operations to the BasicBanking service at the application server. Each financial transaction involves updating multiple data sources, but also involves updating the same data by multiple concurrent clients. Software transactions ensure data integrity in case of concurrent access or failure. Software transactions rely on three standard operations: begin, commit and rollback. Each transaction groups together a number of operations on transactional objects, encapsulating application data. For example the transfer operation of basic banking is listed in the following pseudocode:

```
method transfer
  begin transaction
    credit source account
    debit destination account
    update transaction logs
  commit transaction
  on_error: rollback transaction
end method
```

Input validation of the information entered by the customer is also an essential non-functional concern in the application. For example, the customer could insert SQL statements when entering a message in order to withdraw the money from another account[22]. A third party component SqlDetector is used to detect SQL injection in input strings from the customer. BasicBanking should be defined as a self containing entity, reusable in a third party composition, and thus deployable in a third party environment. Therefore it has to clearly define its dependencies. The aspectual dependencies of the BasicBanking component are transaction management and SQL detection.

In the deployment descriptor of the banking application, other *imposed* aspects will be composed with the BasicBanking component, for example bank-specific authentication and authorization components. These aspects enforce security policies that are bank-specific and provide credential validation and access control of the called method invocations on the BasicBanking component.

B. Composing middleware services in DyMAC

The custom-built middleware platform for the application offers support for the required aspects and the other imposed aspects. We discuss the composition of the application with the middleware as follows. First we define the interfaces of the basicbanking component and we discuss the aspectual middleware services *TransactionMgmtAspect* and *SqlDetector*. Next, we illustrate the definition of the aspectual dependencies and finally we illustrate the composition specification of the required aspects and the other imposed aspects.

The BasicBanking component provides the operations that can be executed in the webbanking environment: transferring money, inspecting customer information and updating customer information.

```
interface IBasicBanking{
 void updateCustomerInfo(string id, string info);
 string getCustomerInfo(string id);
 void transfer(string from, string to, double amount,
                                      string msg);}
```

Support for transactions with demarcation based on method executions is satisfactory for most of the applications. Middleware containers offering declarative support for transactional methods have been widely used, e.g. in EJB [6]. Despite of the performance drawbacks, the reduced development efforts for transactional methods are considered more beneficial.

AO-middleware can offer a similar kind of reusable transaction support: around-advices offer a reusable composition pattern that calls the begin, commit and abort operations of transactions. These around advices can be composed with method executions to enforce transactional behaviour[18]. We illustrate this in DyMAC. The *TransactionMgmtAspect* provides an advice-method *aroundtx* that can be executed around operations to manage the transactional behaviour. The implementation of that behaviour is specified in the class *TransactionMgmtAspectImpl*: it starts a transaction before the operation, then proceeds, then commits after the operation or does a rollback in case of an exception. We illustrate the interface and implementation of the aspect in the next listings.

```
Interface ITransactionMgmtAspect{
  //the provided advice method
  void aroundtx(RuntimeJoinpoint rjp);
  //provided methods for oo composition
  string getTxId();
  void beginTx(ComponentInstance c,
               MethodMessage msg, string id);
  void commitTx(ComponentInstance c,
                MethodMessage msg, string id);
  void abortTx(ComponentInstance c,
               MethodMessage msg, string id);
}
```

```
class TransactionMgmtAspectImpl:
     ITransactionMgmtAspect, ComponentInstance{
  public void aroundtx(RuntimeJoinpoint rjp){
    Component c = rjp.Callee.Instance;
    MethodMessage msg = rjp.Message;

    string id = getTxId();
    try{
      beginTx(c, msg, id);
      rjp.proceed();
      commitTx(c, msg, id);
    } catch {
      abortTx(c, msg, id);
      throw;
    }}
  ...
}
```

The SqlDetector component provides an advice-method to validate the input arguments of the method execution. All arguments that are a string are verified for sql injection.

```
interface ISqlDetector{
  void VerifyInput(RuntimeJoinpoint rjp);}
```

```
class SqlDetectorImpl{
 void VerifyInput(RuntimeJoinpoint rjp){
   foreach(object o in rjp.MethodMessage.Args){
     // if o is string, check for sql
   }}}
```

The specification of the BasicBanking component defines aspectual dependencies for the required aspects. It requires a transaction around the transfer method and input validation on all methods. The component developer specifies which methods require transaction support by means of aspectual dependencies. In this approach, the person who provides the component also specifies the transactional methods.

```
Component BasicBanking{
  // define provided interface
  provides: IBasicBanking
  ...
  Aspectual-dependency transactions{
    Pointcut{
      Kind: execution
      Interface: IBasicBanking
      MethodMessage: * transfer(..);
    }
    RequiredAdvice{
      AspectInterface: ITransactionMgmtAspect;
      AdviceMethod: aroundtx;
      AdviceType: Around;
    }
  }
  Aspectual-dependency inputvalidation{
    Pointcut{
      Kind: execution;
      Interface: IBasicBanking
      MethodMessage: * *(..);
    }
    RequiredAdvice{
      AspectInterface: ISqlDetector
      AdviceMethod: VerifyInput;
      AdviceType: Before;
    }
  }
}
```

The application composer needs to fulfill the two aspectual dependencies that are defined in the specification of BasicBanking. We illustrate the composition of the transaction dependency with the TransactionMgmtAspect component. Additionally, aspect-oriented compositions impose the authentication and authorization aspect.

```
Application BankingApplication{
 Aspectual-dependency-binding{
  Dependency: BasicBanking.transactions;
  Component: TransactionMgmtAspect.ITransactionMgmtAspect;
 }
 ...
 AO-composition{
   //impose authentication and authorization aspects
   Pointcut{
     Kind: execution;
     MethodMessage: * *(..);
     Callee{
       Component: BasicBanking;
     }
   }
   Advice{
     //authenticate user
   }
   Advice{
     //authorise user
   }
 }
}
```

V. Related Work and Discussion

In this section we discuss other approaches to tackle crosscutting concerns in distributed applications. First we discuss two non-AO approaches to middleware, two state-of-the-art AO-middleware approaches, and two aspect languages that have been proposed to deal with crosscutting concerns in distributed applications. Afterwards we discuss related work in the domain of aspect interfaces.

In the introduction we defined two requirements for extensible and customizable middleware. First, composition logic between application logic and the middleware services must be untangled and middleware services must be well modularized in a reusable component. Second, components deployed on the middleware must be able to specify explicit dependencies towards the middleware. We discuss these requirements in the different technologies for the transaction service and the input validation.

A. Non-AO-middleware

The first kind of technologies that we discuss are the middleware approaches that do not focus on offering AO-programming support. The first type of non-AO-middleware is a *naked* ORB for distributed, object-based components. The second type is container-based middleware that offers a fixed set of declarative services for distributed object based components. Examples of the first type are .NET remoting or Java RMI. Example of the second type are the COM+ container of .NET (with *ServicedComponents*) or the EJB container of J2EE platforms.

1) Object Request Brokers: In a *naked* ORB, middleware services such as transaction support or security services need to be enforced by means of API calls (e.g the JAAS authentication and authorization API). The composition of the transactional behaviour and security services is intermingled with the application logic. Note that in such an approach, there is an explicit dependency in each component descriptor towards the required OO-interface of the middleware service. These interfaces are typically used in the implementation to explicitly call the transaction behaviour and SQL input validation. This results in the crosscutting composition though.

```
ITransactionMgmt txMgmt;
ISqlDetector sqlDetector;

public void transfer(string from, string to,
                     double amount, string msg){
  sqlDetector.validate(from);
  sqlDetector.validate(to);
  ...
  string txId = txMgmt.getTxId();
  try{
    txMgmt.begin(txId);
    //check amount and new balance,
    //do withdrawal and deposit
    //update transaction logs
    txMgmt.commit(txId);
  }catch(Exception e){
    txMgmt.rollback(txId);
  }}
```

2) Container-based Middleware: We will discuss the EJB container as an example of container-based middleware. The j2ee specification contains a declarative interface to configure the transactional behaviour that is needed in an EJB component. The EJB components contain explicit dependencies to the transaction interface of the EJB specification.

A j2ee vendor provides an implementation for the EJB specification, which includes the implementation of the transaction behaviour in the EJB container. The crosscutting composition of the begin(), commit() and rollback() operation, as well as the implementation is dealt with in the transaction implementation in the container.

However, the containers that implement the EJB specification only support limited configuration. They are hard to customize or extend for application or deployment specific contexts. Therefore, input validation (e.g. sql injection detection) has to be implemented as an explicit call in the BasicBanking component. This results in a tangled implementation and crosscutting composition. The next two listings illustrate the untangled transaction management in EJB and the explicit input validation in the transfer implementation.

```
ISqlDetector sqlDetector;
...
public void transfer(string from, string to,
                     double amount, string msg){
  sqlDetector.validate(from);
  sqlDetector.validate(to);
  sqlDetector.validate(msg);
  //check amount and new balance,
  //do withdrawal and deposit
  //update transaction logs
}
```

```
<container-transaction>
  <method>
    <ejb-name>BasicBanking</ejb-name>
    <method-name>Transfer</method-name>
     ...
        </method>
  ...
</container-transaction>
```

B. AO-middleware

Many AO-middleware platforms offer similar support for interface specification - although they are quite different from other viewpoints. These platforms include JBoss AOP[23], JAC[8], AspectJ2EE[17] and Spring[24]. We first discuss JBoss AOP as a representative example of this group. Only provided interfaces are supported in these technologies. Next we discuss CAM/DAOP as it offers a more extended support for interface specification.

1) JBoss AOP: JBoss AOP is an AO-middleware approach that defines aspects as pure java objects. Advices are methods that have a special signature. Aspects can provide multiple advice methods. The binding of the aspects with the other classes are defined in separate xml-based composition descriptors.

The component provider of the TransactionAspect can modularize the transactional behaviour as an aspect that provides the aroundtx advice method. The application composer deals with the binding of the transaction aspect on the *BasicBanking* component, and thus has to specify the transactional methods.

There is no possibility to express the explicit dependency in the basic banking component towards the required transactional behaviour (or input validation).

```
interface ITransactionMgmt
  public Object aroundtx(Invocation invocation);}
class TransactionAspect implements ITransactionMgmtAspect{
  public object aroundtx(Invocation invocation){
    string id = getcurrenttx();
    try{
      begin(id);
      object o = invocation.proceed()
      commit(id);
      return o;
    }catch(Exception e){
      abort(id);
      throw e;
    }}
```

```
<bind pointcut="*␣*␣BasicBanking->transfer(..)">
<advice aspect="TransactionAspect"
        name="aroundtx"/>
</bind >
```

2) CAM/DAOP: CAM/DAOP has two component types: OO-components and aspect components. The provided interface of OO-components (for OO-composition) and aspects (for AO-composition) can be described in the component specification. The required interfaces of OO-components towards other OO-componentns can also be made explicit in the component specification. Required aspect components can not be specified. CAM/DAOP offers the notion of mandatory aspects. But that term refers to the fact that advice that is executed, must evaluate to true and may not throw any exception in order to continue the execution of the application.

The approach for modularizing the input validation as aspect component in CAM/DAOP is quite similar to the approach in JBoss AOP. However, for the transaction management there is a difference. CAM/DAOP does not support around advice. Therefor the AO-composition of the transaction has to be specified in terms of before, after and after-throwing advice by the application composer. This involves a more complex composition between application logic and middleware service. Moreover, this composition needs to be specified by the application composer.

C. Language support for service composition

The modularization and composition of crosscutting, non-functional concerns in distributed applications by means of aspect languages has been described in [11], [18], [10]. Aspect/J and Caesar[12] are two languages that have been put forward to tackle such crosscutting concerns and modularize them in aspects.

1) AspectJ: AspectJ offers abstract aspects with abstract pointcuts to define aspects with reusable advices. An inheriting aspect defines the concrete pointcut to bind the aspect behaviour to a concrete set of java classes.

The provider of the TransactionMgmt aspect can use abstract aspects to deal with the modularization of the transactional behaviour in a reusable aspect. The application composer deals with the binding of the transaction aspect on the basic banking component by means of an inheriting aspect that specifies the transactional methods. However, there is no possibility to express the explicit dependency in the basic banking component towards the required transactional behaviour (or input checking). We illustrate the abstract and concrete aspect for transaction management in the next two listings.

```
abstract aspect TransactionMgmt{
 abstract pointcut transactions();
 around: transactions(){
  try{
   // begin transaction
   proceed();
   // commit transaction
  }
  catch(Exception e){
   // rollback transaction
  }}}
```

```
aspect BasicBankingTransactionMgmt
       extends TransactionMgmt{
  pointcut transactions(): execution(* *.transfer(..));}
```

2) Caesar: Caesar offers the concept of provided and expected methods in the aspect collaboration interfaces. These interfaces define the interfaces for OO-composition between the components. Provided methods are implemented in the aspect implementation. Expected methods are implemented in the aspect binding. Advice cannot be specified in the collaboration interface. It is always specified and implemented in the aspect binding. Therefore, CaesarJ does not support the specification of advice in the provided interface of the advising component or the specification of a required aspect in the advised component.

The composition of the transaction behaviour can be untangled from the application code in an around advice. However that around advice that deals with this crosscutting composition has to be specified in the binding, thus by each provider of an application component that needs transactional behaviour. It is not possible to specify the around advice signature and implementation in the aspect implementation (as in the AO-middleware approach). Therefore there is no support to define required aspects. There is only an explicit OO-dependency to the OO-interface of the TransactionMgmt component. Additionally, the definition of the BasicBanking class has to be specified in the binding so that it cannot be provided separately, without the explicit OO-dependency. This is illustrated in the next listings.

```
interface TransactionProtocol{
  interface TransactionalObject{}
  interface TransactionMgmt{
    provided string getCurrentTxId();
    provided begin(id);
    provided commit(id);
    provided rollback(id);
  }}
```

```
// aspect implementation
class TransactionProtocolImpl
      implements TransactionProtocol{
    class TransactionMgmt{
       //implements provided methods
    }}
```

```
//aspect binding
class TransactionProtocolBinding
    binds TransactionProtocol{
  class TransactionalBasicBanking
        binds TransactionalObject
        wraps BasicBanking{
  class BasicBanking{
    // define and implement basic banking in the binding
  }
  ITransactionMgmt txMgmt;
  pointcut transactions(): execution(* *.transfer(..));
  around(): transactions(){
   // cfr previous approaches.
  }}
```

```
deployed class CTP
 extends TranactionProtocol<TransactionProtocolBinding,
                            TransactionProtocolImpl>{};
```

We can conclude that AO-composition in CaesarJ does not support the specification of advice in the provided interface of the advising component or the specification of a required aspect in the advised component.

D. Black-box AO-composition

In this section we present related work on black-box AO-composition that goes beyond the traditional non-invasive joinpoint models. Non-invasive joinpoint models basically allow to advise any invocation that is part of the provided and required (OO) interfaces. However, the AO-technologies in this section support the definition of an explicit interface of an advisable component towards an advising component [15], [19], [20], [21]. These interfaces specify explicitly the kind of joinpoints they provide as well as concrete joinpoints they provide for (possible) aspect-oriented compositions.

Aspect Integration contracts [15] support expressive power in the component contracts to specify the set of joinpoints that is available for AO-composition. The contracts can specify which state and behaviour they provide towards concrete aspects or groups of aspects. The pointcuts for the AO-composition itself are still defined outside the component contract in a composition specification.

Open modules [19] focuses on the same problem as Aspect Integration Contracts. Open modules define an extended approach to define interfaces of classes by exporting certain private joinpoints by means of pointcuts. This opens up certain parts of the private state of modules and implementation of methods as possible joinpoints. Only the explicitly exported joinpoints are advisable. XPI [21] is an approach that abstracts the internal behaviour of a set of collaborating modules by exposing abstract pointcuts on the collaborating modules. These abstract pointcuts are advisable by aspects. In [20], a module defines pointcuts that hide the joinpoints that must not be advised.

We conclude that the exposed joinpoints in the discussed related work are part of the provided interface for ao-composition. These joinpoints *can* be advised by an aspect. Aspectual dependencies also expose a set of joinpoints in the abstract behaviour of a component, however the dependencies also *require* well specified advice. Therefore they are part of the required interface for ao-composition.

VI. Conclusion

Traditional middleware containers suffer from monolithic design and limited configurability and extensibility. AO-middleware opens up this black-box for the application developer. The container evolved from a black-box component with a declarative interface for a fixed set of services, to an open micro-container with an open-ended list of services that are composed by means of aspect-oriented composition.

However, the application component specifications are lacking support to specify explicit dependencies to (aspectized) middleware services that are essential for the correct behaviour of the application components. This lack of information in the specification of components is an essential deficit to define components as self-containing black-box entities. It also breaks the pure black-box approach to aspect-oriented composition: semantical dependencies on the implementation of the component are introduced.

In this paper we presented *aspectual dependencies* in the context of AO-middleware, such that required middleware services can be specified as explicit dependencies in the component specification. In that way, implicit semantical dependencies towards the component implementation by the application composer can be omitted. Aspectual dependencies also define the mininal set of services that is required of the open container by the application components in order to guarantee correct execution.

References

[1] C. Szyperski. Component software: beyond object-oriented programming. Second Edition. ACM Press/Addison-Wesley.

[2] G. Heineman & W. Councill. Component-based Software Engineering. Addison-Wesley.

[3] G. Blair, G. Coulson, et al. The design and implementation of OpenORB version 2. IEEE DS Online Journal, 2(6), 2001

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold.An Overview of AspectJ. In Proc. ECOOP'01.

[5] R. Filman, T. Elrad, S. Clarke & M. Aksit. Aspect-Oriented Software Development. Addison-Wesley, 2004.

[6] Sun Microsystems. Inc. Enterprise Java-Beans (EJB) Specification v2.0, 2001.

[7] Sun Microsystems. Java 2 Platform, Enterprise Edition BluePrints, 2001.

[8] R. Pawlak, L. Seinturier, L. Duchien & G. Florin. JAC: A Flexible Solution for Aspect-oriented Programming in Java. In Proc. Reflection'01.

[9] B. Lagaisse & W. Joosen. True and Transparent Distributed Composition of Aspect-Components. In Proc. Middleware'06.

[10] R. Bodkin et al. Applying AOP for Middleware Platform Independence. Practitioner Reports, AOSD 2003.

[11] A. Colyer et al, Large-scale AOSD for middleware. In Proc. AOSD'04.

[12] M. Mezini & K. Ostermann. Conquering Aspects with Caesar. In Proc. AOSD'03.

[13] M. Fleury & F. Reverbel. The JBoss extensible server. In Proc. Middleware 2003.

[14] E. Truyen, B. Vanhaute, B.N. Jorgensen, W. Joosen & P. Verbaeten. Dynamic and Selective Combination of Extensions in Component-Based Applications. In Proc. ICSE'01.

[15] B. Lagaisse & W. Joosen. Component-Based Open Middleware Supporting Aspect-Oriented Software Composition. In Proc. CBSE'05.

[16] M. Pinto, L. Fuentes & J.M. Troya. A Dynamic Component and Aspect-Oriented Platform. The Computer Journal,2005.

[17] T. Cohen & J.Y. Gil. AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework. In Proc. ECOOP'04.

[18] J. Kienzle, R. Guerraoui. AOP: Does it Make Sense? The Case of Concurrency and Failures. In Proc. ECOOP 2002.

[19] J. Aldrich. Open Modules: Modular Reasoning about Advice. In Proceedings of the European Conference on Object-Oriented Programming, volume 3586 of LNCS, pages 144–168. Springer, 2005.
[20] D. Larochelle, K. Scheidt and K. Sullivan, Join Point. Encapsulation. In Proceedings SPLAT Workshop, AOSD 2003.
[21] W. G. Griswold, et al., Modular Software Design with Crosscutting Interfaces, IEEE Software, vol. 23, no. 1, 2006, 51-60.
[22] S. Boyd and A. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proc. of the 2nd Appl. Cryptogr. and Netw. Secur. Conf., volume 3089 of LNCS, June 2004.
[23] JBoss AOP homepage, http://labs.jboss.com/jbossaop
[24] Spring Framework website. http://www.springframework.org/

A Seamless Extension of Components with Aspects using Protocols

Angel Núñez, Jacques Noyé
Project OBASCO, EMN-INRIA, LINA
4 rue Alfred Kastler, 44307, Nantes, France
{angel.nunez,jacques.noye}@emn.fr

***Abstract*—This paper shows how components and aspects can be seamlessly integrated using protocols. A simple component model equipped with protocols is extended with *aspect components*. The protocol of an aspect component observes the service requests and replies of plain components, and possibly internal component actions, and react to these actions (possibly preventing some base actions to happen as is standard with AOP). A nice feature of the model is that an assembly of plain and aspect components can be transformed back into an assembly of components. All this is done without breaking the black-box nature of the components (dealing with internal actions requires to extend the component interface with an *action interface*).**

I. Introduction

Aspect-Oriented Programming (AOP), initially developed in the context of Object-Oriented Programming (OOP), has shown that classes are not enough to properly modularize all the concerns of an application. The use of classes alone leads to so-called *crosscutting concerns*, scattered in the various classes that build the application. AOP makes it possible to collect these scattered parts of the concern in a new modularization construct: an *aspect*, and leave the set of classes to which the aspect applies, the *base program*, free from any code for the concern. It is then the job of the compiler to *weave* the aspect and the base program, *i.e.*, to introduce concrete connections between the aspect and the classes, using the aspect *pointcut* and *advice*. The *pointcut* is a predicate defining the *join points*, *i.e.*, the execution points in the base program which should be affected by the aspect. The *advice* defines the new behavior to be inserted at the join points, including calls to base program methods. An abstract way of considering *weaving* is to see it as a transformation back to the scattered and tangled code that would have been written by hand using plain OOP (in practice, however, weaving is not a source-to-source transformation, but a direct transformation to lower-level code, typically bytecode). Apart from improving the modularity of the application, AOP also allows incremental programming: the base program can be developped independently from the aspects, which can be developped at a later stage.

The situation is not really different when moving from OOP to plain Component-Oriented Programming (COP). Crosscutting concerns have to be dealt with. In a strict black-box model, incremental programming is not possible. The crosscutting concern has to be implemented as a (collection of) component(s). Connection code has to be introduced in the implementation of the base components, which must also be equipped with the proper interfaces.

This paper deals with improving on this situation by showing how AOP and COP can be seamlessly integrated. We start with a simple component model where components are defined as a set of (structural) interfaces describing their provided and required services and a protocol, describing the behavior of the components in terms of service requests and replies as well as internal actions. We then extend this model by adding *aspect components*, which are also defined as a set of interfaces and a protocol. This protocol has however a slightly different meaning than a standard component protocol. It corresponds to the definition of a *stateful* concurrent aspect [1], [2], [3], which can observe various base actions (service requests and replies, internal actions) and react accordingly. This includes the possibility of preventing a base action from happening, a standard feature of AOP. In this model, weaving can be seen as a transformation of the initial system of plain components and aspect components into a system of plain components.

Section II gives more details on our approach. Section III describes our simple reference model. Section IV extends this model with aspect components. Section V shows how weaving transforms an initial system with aspect components into a system of plain components. Section VI illustrates the approach with a small example. Section VII discusses related work. Finally, Section VIII concludes.

II. The approach

As explained in the introduction we integrate the notion *(class, aspect)* of AOP with the notion *(component, aspect component)* in a seamless way. For doing that, we use Baton [3], a language for programming concurrent stateful aspects in Java. This language is based on the Concurrent Event-based AOP (CEAOP) approach [2] that models concurrent base programs and concurrent aspects as Finite State Processes (FSP). CEAOP models the weaving of aspects into the base program as FSP composition of the corresponding FSPs. Baton implements these ideas in the OOP world.

In order to implement the integration of AOP and COP, we evolve Baton into a language for programming aspect components that applies to component-based applications. The weaving of aspect components written in Baton into an application is implemented as the generation of a plain component

representing the aspect component, which is connected to the rest of the components of the system.

For the time being, this paper just considers the case of weaving a single aspect into a component-based system. However, we lay the foundations for the full support of the concurrent aspects modeled by CEAOP.

In the following sections, we describe a simple component model used as a reference model. Then, we present the syntax of Baton and we describe the weaving of aspect components.

III. A SIMPLE COMPONENT MODEL

This section describes a very simple component model with the basic features assumed by our aspect-component language.

We consider a minimal component model, whose components are *black boxes* equipped with interfaces and a protocol. Furthermore, the model allows the definition of *primitive* and *compound* components.

A primitive component declares its interfaces and its protocol with the following syntax:

```
Component   ::= component Id implements
                  Interfaces { Behavior }
Interfaces  ::= Id ( , Id )*
```

The definition of the interfaces is done outside the component. Each interface declares provided and required services with the following syntax:

```
Interface   ::= interface Id { IntBody }
IntBody     ::= ( Action ; )*
Action      ::= Mod Name ( Params? )
Mod         ::= provides
Mod         ::= requires
```

Action represents a service that can be provided or required, *Name* is the name of the service, and *Params* are optional parameters declared by the service (parameters are used for message passing purposes).

The protocol defines the behavioral interface of the component using an FSP. We assume a protocol with the following syntax:

```
Behavior    ::= ProcDef ( , ProcDef )* .
ProcDef     ::= ProcId = Body
Body        ::= ( Prefix ( | Prefix )* )
Prefix      ::= ( ActLabel -> )* ProcId
ActLabel    ::= Name ( Params? )
```

The label of each transition (*ActLabel*) consists of the name of a service declared in some interface (*Name*) and its optional parameters (*Params*). We say that the transition *refers* to the service. The semantics of each transition depends on the type of service the transition refers. If a transition refers to a provided service, then the semantics of the transition is that the component receives a request for the service. If a transition refers to a required service, then the semantics of the transition is that the component sends a request for the service.

A compound component is an assembly of subcomponents. Its interfaces are formed by interfaces *exported* from subcomponents. An exported interface is such that it has not been bound or it only defines provided services. The protocol of a compound component is obtained from the protocols of its subcomponents by performing FSP composition.

Components are connected through their interfaces. We just consider binary communication (one sender, one receiver). When connecting two interfaces, services are bound through name matching. The condition is that each required service of one interface is provided by a service of the second interface.

A recent approach introduces the notion of *open modules* [4], which can be used to expose internal actions of a black-box component. We extend the component interface with an *action interface* in order to include this notion. Then a primitive component may not only declare the standard interface of provided and required services, but also action interfaces.

The action interface defines abstract internal actions that are made observable from outside the component and are included in the component protocol together with provided and required services. The syntax of an action interface is as follows:

```
Interface   ::= interface Id { ActIntBody }
ActIntBody  ::= ( ExpAction ; )*
ExpAction   ::= exposes Name ( Params? )
```

IV. A LANGUAGE FOR PROGRAMMING ASPECT COMPONENTS

We seamlessly integrate the notion of aspect in AOP into the notion of aspect component. For doing this, we present Baton as a language for programming aspect components. This section describes the syntax of the language.

A. Aspect components

An aspect component, as the name implies, is an aspect with a component flavor. Like a component, it is defined using a set of interfaces and a protocol. Its protocol has however a slightly different meaning than a standard component protocol. It corresponds to the definition of a stateful concurrent aspect. The concrete syntax of an aspect component (see below) is very similar to the syntax of a plain component, the differences are in the definition of the interfaces and the protocol.

```
Component   ::= aspect Id implements
                  Interfaces { Behavior }
Interfaces  ::= Id ( , Id )*
```

An interface is defined by the following syntax, which is very similar to the syntax of a plain-component interface:

```
Interface   ::= interface Id { IntBody }
IntBody     ::= ( Action ; )*
Action      ::= Mod Name ( Params? )
Mod         ::= event
Mod         ::= skippable event
Mod         ::= action
```

Whereas a plain-component interface declares required and provided services, an aspect-component interface declares abstract actions representing base-program actions. Actions denoted with the keyword `event` represent actions that the aspect component observes in the base program. Actions denoted with the keyword `skippable event` represent actions that the aspect component observes and can make the base program skip. Actions denoted with the keyword `action` represent actions that the aspect component requires to implement its advices.

The syntax of the aspect-component protocol is as follows:

```
Behavior    ::= ProcDef ( , ProcDef )* .
ProcDef     ::= ProcId = Body
Body        ::= ( Prefix ( | Prefix )* )
Prefix      ::= ( ActLabel Advice? -> )* ProcId
ActLabel    ::= Name ( Params? )
```

As we can see, the syntax of the aspect-component protocol is almost the same as the syntax of the plain-component protocol, except that the former allows the definition of *advised* transitions (i.e. transitions including an aspect advice, represented by the non-terminal *Advice*). For both, advised and non-advised transitions, *ActLabel* corresponds to an abstract action declared in the interface, more precisely, it corresponds to an action declared with the keyword `skippable event` for advised transitions, and to an action declared with the keyword `event` for non-advised ones. The semantics of a non-advised transition is that the aspect changes its state with the occurrence of the corresponding base action. The semantics of an *advised* transition is that, in the context of the corresponding base action, the aspect may execute advices and may prevent the action from happening. The syntax of *Advice* is as follows:

```
Advice      ::= > Before PS After
Before      ::= ( ActLabel ; )*
After       ::= ( ; ActLabel )*
PS          ::= skip | proceed
```

Advices (*Before*, *After*) are sequences of abstract actions. Each of these abstract actions is an action declared with the keyword `action` in the aspect-component interface. The semantics of each action is that the aspect component sends a request for the corresponding service in some component of the system.

The parameters (*Params*) declared in the syntax of the aspect component are used for passing information from the base program to the aspect component. These parameters are available in the scope of each transition (*Prefix*).

B. Connectors

A connector binds abstract actions declared in the interface of an aspect component with concrete actions declared in the interface of the base components. The syntax of a connector is as follows:

```
Connector   ::= connector { Connection* }
Connection  ::= connects Action to Pattern ;
```

Action is an action declared in the interface of an aspect component. *Pattern* corresponds to a pattern that permits to match actions declared in the interface of a component.

V. Weaving

Weaving an aspect component into a component-based system corresponds to generating a system with plain components. This is done by transforming the aspect component into a *plain aspect component* (PAC) and connecting it to the rest of the components of the system.

A. The aspect component as a plain component

This section describes how an aspect component is implemented as a plain component. In the remainder we describe the generation of the interfaces and the protocol of this component.

1) Generation of the protocol: The protocol of the PAC is the result of transforming the protocol of the aspect component. As previously explained, the aspect component observes actions of the component-based application, this is implemented in the PAC as the reception and sending of synchronization events (equivalent to the events introduced by the CEAOP model). These events are implemented as component services. We obtain the protocol of the PAC by applying the following transformations:

```
T( name(params) -> P ) =
  eventB_name(params) -> eventE_name() -> P
```

```
T( name(params) > before; ps; after -> P ) =
   eventB_name(params) -> before ->
     psB_name() -> psEname() -> after ->
     eventE_name() -> P
```

The first transformation describes that taking into account a base program action *name(params)* is implemented as the reception of an event `eventB_`*name(params)* indicating that the action is about to be executed (the `B` in `eventB` is for begin) followed by the reception of an event `eventE_`*name()* indicating that the action has been executed (the `E` in `eventE` is for end).

The second transformation describes that a transition that introduces advices, and can make the base program skip an action *name(params)*, is programmed through the following communication between the PAC and a base component:

i. The PAC receives the event `eventB_`*name(params)* from a base component when the action is about to be executed.

ii. Then, it executes the sequence of actions denoted by *before* and emits either the event `skipB_`*name()* or the event `proceedB_`*name()* to indicate to the base component whether the action has to be skipped or not.
iii. The base component receives the last event, skip the action or proceed, and emits either the event `skipB_`*name()* or the event `proceedB_`*name()* indicating whether the action has just been skipped or not.
iv. The PAC receives the last event, executes the sequence of actions denoted by *after* and emits the event `eventE_`*name()* to indicate to the base component that this base component can continue with its computation.
v. The base component receives this event and continues.

2) Generation of the interfaces: The interfaces of the PAC are derived from the interfaces of the aspect component. They basically consist of the declaration of the synchronization events used in the PAC protocol.

An action declared as `event` *name(params)* in the aspect-component interface is used in non-advised transitions of the aspect-component protocol. It generates the following interface in the PAC:

```
interface SyncA_name {
  provides eventB_name(params);
  provides eventE_name();
}
```

In an analogous way, an action declared as `skippable event` *name(params)* in the aspect-component interface is used in advised transitions of the aspect-component protocol. It generates the following interface in the PAC:

```
interface SyncA_name {
  provides eventB_name(params);
  requires eventE_name();
  requires proceedB_name();
  provides proceedE_name();
  requires skipB_name();
  provides skipE_name();
}
```

Finally, an action declared as `action` *name(params)* in the aspect-component interface is used in the advice of advised transitions and generates the following interface in the PAC:

```
interface A_name {
  requires name(params);
}
```

B. Connecting plain components

Once the PAC has been generated, the second part of the weaving process is to connect the PAC to the rest of the components of the system. The Baton connector tells us which base component should be connected to the PAC, more precisely, which concrete actions from the base components should be connected to which abstract actions of the aspect component.

A Baton connector matches services and internal actions declared in the interface of a base component. If a service or internal action *s(params)* is matched, then there is an association with an abstract action used by the aspect component (to simplify things, we suppose that for each abstract action only one service or internal action is matched in all the component hierarchy). If the abstract action has been declared as `event` *name(params)* or `skippable event` *name(params)*, then the PAC implements an interface `SyncA_`*name*. A complementary interface, namely `SyncB_`*name*, is introduced in the component to make the connection. Furthermore, the necessary modifications in the protocol of the base component are performed.

We define two transformations to the base-component protocol:

```
T(s(params) -> P) =
 eventB_name(params) ->
  s(params) ->
  eventE_name() -> P
```

```
T(s(params) -> P) =
 eventB_name(params) -> proceedB_name() ->
  s(params) ->
  proceedE_name() -> eventE_name() -> P
 |eventB_name(params) -> skipB_name() ->
  skipE_name() -> eventE_name() -> P
```

The first transformation applies if the abstract action *name(params)* has been declared as `event`. Then the component has to generate one event before the execution of the concrete action and another event after. The second transformation applies if the abstract action has been declared as `skippable event`. Then the component has to generate events that introduce the possibility of skipping the action (as seen in the generation of the PAC protocol).

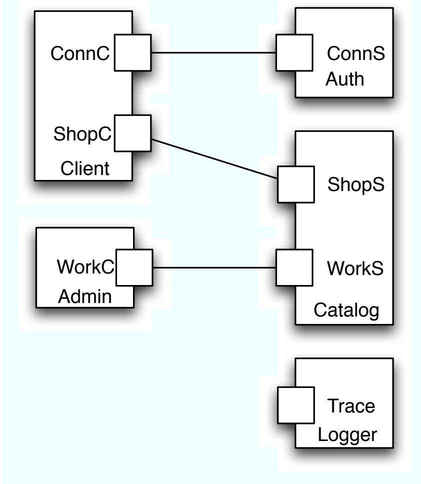
If the abstract action has been declared as `action` *name(params)*, then there is an interface `A_`*name* that is connected to the interface of *s(params)*.

We have introduced a language for programming aspect components and shown how these components can be implemented as plain components. Afterwards, this section has described the process of weaving. The next section presents an example to illustrate the approach.

VI. Example

To illustrate the approach we use a simple example based on e-commerce applications. Clients connect to a website and must login to identify themselves, then they may browse an on-line catalog. The session ends at checkout, that is, as soon as the client has paid. In addition, an administrator of the shop

can update the website at any time by publishing a working version. Consider the application has been programmed as the component-based system of the following figure:



The application consists of five components: `Client` represent a client, `Admin` an administrator, `Auth` an authorization entity, `Catalog` the on-line catalog, and `Logger` a component that implements a logging functionality. The interfaces `Client.ShopC` and `Catalog.ShopS` declare a service `getItems()`, which is used by the client to browse the catalog, and a service `pay(List items)`, used to make a payment. The interfaces `Client.ConnC` and `Auth.ConnS` declare a service `login(Credential credential)`, which is used by the client to identify itself. The interfaces `Admin.WorkC` and `Catalog.WorkS` declare a service `addItem(Item item)`, which is used by the administrator to update the catalog.

As an example we show the definition of the component `Admin`:

```
component Admin implements WorkC {
  Begin = ( addItem(Item item) -> Begin).
}


interface WorkC {
  requires addItem(Item item);
}
```

Let us now consider the problem of canceling updates to the client-specific view of the e-commerce shop during session, e.g. to ensure consistent pricing to the client. We can define a suitable aspect component, which we call `Consistency`, to solve this problem. The aspect component programmed in Baton is as follows:

```
aspect Consistency implements ConsistencyI
{
  OutSession =
    ( login() -> InSession ),
  InSession =
    ( update() > skip; log() -> InSession
    | checkout() -> OutSession ).
}

interface ConsistencyI {
  event login();
  event checkout();
  skippable event update();
  action log();
}
```

This aspect initially starts in state `OutSession` and waits for a `login()` action from the base program (other actions are just ignored). When the `login()` action occurs, the base program resumes by performing the `login()`, and the aspect proceeds to state `InSession` in which it waits for either an `update()` or a `checkout()` action (other actions being ignored). If `update()` occurs first, the associated advice `skip; log()` causes the base program to skip the `update()` action and the `log()` action is performed. Then the base program resumes and the aspect returns to state `InSession`. If `checkout()` occurs first, the aspect returns to state `OutSession`. Since `update()` actions are ignored in state `OutSession`, updates occurring out of a session are performed, while those occurring within sessions (state `InSession`) are skipped.

In order to weave the `Consistency` aspect component, we define the following Baton connector, which binds the abstract actions declared in the interface of the `Consistency` aspect component with concrete actions declared in the system:

```
connector Connector {
 connects login() to *.*.login(..);
 connects checkout() to *.*.pay(..);
 connects update() to *.*.addItem(..);
 connects log() to Logger.Trace.log();
}
```

In the weaving of the aspect component into the application, the PAC is generated and connected to the corresponding components of the system. The code below shows the definition of the resulting PAC:

```
component PAC implements SyncA_login,
         SyncA_checkout, SyncA_update,
         A_log
{
 OutSession =
 ( eventB_login() -> eventE_login() ->
   InSession),
 InSession =
 ( eventB_update() -> skipB_update() ->
   skipE_update() -> log() ->
   eventE_update() -> InSession
 | eventB_checkout() -> eventE_checkout()
   -> OutSession ).
}
```

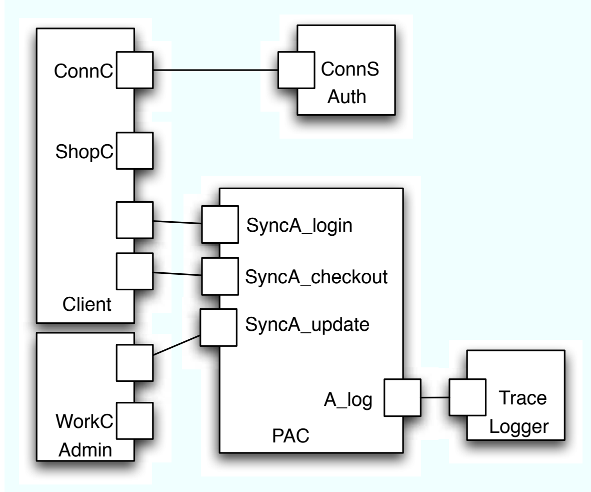
```
interface SyncA_login {
  provides eventB_login();
  provides eventE_login();
}


interface SyncA_checkout {
  provides eventB_checkout();
  provides eventE_checkout();
}


interface SyncA_update {
  provides eventB_update();
  requires eventE_update();
  requires proceedB_update();
  provides proceedE_update();
  requires skipB_update();
  provides skipE_update();
}


interface A_log {
  requires log();
}
```

The following figure illustrates the PAC with its corresponding interfaces connected to the rest of the system (for the sake of space, we have hidden the component `Catalog`).



The weaving also produces the instrumentation of some base components.As an example the component `Admin` becomes equivalent to:

```
component Admin implements
                     WorkC, SyncB_update
{
 Begin =
 ( eventB_update() -> proceedB_update() ->
   addItem(Item item) -> proceedE_update()
   -> eventE_update() -> Begin
 | eventB_update() -> skipB_update() ->
   skipE_update() -> eventE_update() ->
   Begin ).
}


interface WorkC {
  requires addItem(Item item);
}


interface SyncB_update {
  requires eventB_update();
  provides eventE_update();
  provides proceedB_update();
  requires proceedE_update();
  provides skipB_update();
  requires skipE_update();
}
```

VII. Related Work

The work on open modules [4] suggests that module interfaces should be extended with pointcut names to be used by aspect implementors in order to advise the aspect as well as by the module implementor who, in case of an evolution of the module, may have to update the definition of the pointcut. We do something very similar with action interfaces, which, together with the component protocol, is an abstract description of the execution points within the component that an aspect may affect.

FuseJ [5] aims at achieving a symmetric, unified component architecture that treats aspects and components as uniform entities. Then, it addresses the problem of properly configuring connections between components implementing a concern and the rest of the system. FuseJ proposes a powerful *configuration language* to program component connections that support crosscutting connections. This is conceptually similar to a Baton connector.

Fractal Aspect Component (FAC) [6] introduces a general model for components and aspects. FAC decomposes a software system into regular components and *aspect components* (ACs), where an AC is a regular component that embodies a crosscutting concern. An *aspect domain* is the reification of the notion of a pointcut: the components picked out by an AC. Furthermore, the implicit relationship between a woven AC and the component in which the aspect component applies is a first-class entity called an *aspect binding*. A posterior work [7] introduces the notion of open modules to FAC.

None of these approaches support the definition of connections between components implementing advices and base components that depend on a global shared state. Baton permits to program this kind of smart connections that corresponds to stateful aspects in the AOP terminology. Furthermore, none of these approaches seamlessly integrate AOP into COP as Baton does.

VIII. Conclusion

This paper proposed a solution of the problem of modularizing crosscutting concerns in component-based system. Our main contribution is to show how AOP and COP can be seamlessly integrated. The tuple *(class,aspect)* of AOP has been introduced into COP as the tuple *(component, aspect components)*. In concrete terms, Baton, a language for programming concurrent aspects in Java, has been evolved into a language for programming aspect components that are applied to a component-based system.

We have shown how weaving an aspect component and plain components can produce a system with only plain components. This can actually be extended to the weaving of many aspects composed together using composition operators [2]. The operators are then also translated into plain components. The action interface makes it possible to deal with aspects, including some form of incremental development, without breaking the black-box property of components. Indeed, the action interfaces have to be anticipated and made part of the component interface at design time but aspect weaving can still take place at deployment time (this implies that component implementations can be instrumented at deployment time, which is for instance the case when these implementations are provided as Java bytecode).

As future work, we plan to extend these ideas to a more realistic component model including, for instance, multi-ary communication. In this regard, we could combine efforts with works on component models with explicit protocols such as [8]. We also plan to integrate support for distributed aspects in the line of AWED [9].

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments and mention that this work has been partly supported by the project AMPLE: Aspect- Oriented, Model-Driven, Product Line Engineering (STREP IST-033710).

References

[1] R. Douence, P. Fradet, and M. Südholt, “Composition, Reuse and Interaction Analysis of Stateful Aspects,” in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, K. Lieberherr, Ed. Lancaster, UK: ACM Press, Mar. 2004, pp. 141–150.

[2] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt, “Concurrent Aspects,” in *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE’06)*. Portland, USA: ACM Press, Oct. 2006, pp. 79–88.

[3] A. Núñez and J. Noyé, “Baton: A Domain-Specific Language for Coordinating Concurrent Aspects in Java,” in *Proceedings of the 3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA07)*, Toulouse, France, Mar. 2007.

[4] J. Aldrich, “Open modules: Modular reasoning about advice,” in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, ser. Lecture Notes in Computer Science, M. Odersky, Ed., vol. 3586. Glasgow, UK: Springer-Verlag, Jul. 2005, pp. 144–168.

[5] D. Suvée, B. D. Fraine, and W. Vanderperren, “A symmetric and unified approach towards combining aspect-oriented and component-based software development.” in *CBSE*, ser. Lecture Notes in Computer Science, I. Gorton, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, Eds., vol. 4063. Springer, 2006, pp. 114–122.

[6] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien, “A Model for Developing Component-based and Aspect-oriented Systems,” in *Proceedings of the 5th International Symposium on Software Composition (SC06)*, ser. Lecture Notes in Computer Science, vol. 4089. Vienna, Austria: Springer-Verlag, Mar. 2006, pp. 259–273.

[7] ——, “A Safe Aspect-Oriented Programming Support for Component-Oriented Programming,” in *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming (WCOP06)*, R. Reussner, C. Szyperski, and W. Weck, Eds., Nantes, France, Jul. 2006, technical Report 2006-11, Universität Karlsruhe, Fakultät für Informatik.

[8] F. Fernandes, R. Passama, and J. C. Royer, “Components with Symbolic Transition Systems: A Java Implementation of Rendez-Vous,” in *Proceedings of the Communicating Process Architecture Conference*, 2007.

[9] L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée, “Explicitly distributed AOP using AWED,” in *OOPSLA 2006, Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, P. L. Tarr and W. R. Cook, Eds. Portland, Oregon, USA: ACM Press, Oct. 2006.

AOCI: An Aspect-Oriented Component Infrastructure

Guido Söldner and Rüdiger Kapitza
{guido.soeldner,rrkapitz}@cs.fau.de
Dept. of Comp. Sciences, Informatik 4
University of Erlangen-Nürnberg, Germany

***Abstract*—Component-based middleware provides already mature support for non-functional properties (e.g. load balancing, transaction support or persistence) offered by the component container. However, to support context-aware applications enabling a flexible dynamic reconfiguration and adaptation, further techniques are required.**

This paper proposes an Aspect-Oriented Component Infrastructure (AOCI) that allows the dynamic weaving according to application and environment demands while preserving component integrity. This is achieved by offering a greybox component concept exporting potential points for extension together with an ontology-based metadata description. In contrast to related systems the proposed solution promises a better support for runtime adaptation and eases the development of adaptable components.

***Index Terms*—Component-based Middleware, Dynamic Adaptation, AOP.**

I. Introduction

Software components [19] are usually considered as black boxes that offer specific services provided by interfaces. Furthermore, components can be easily combined with each other, thereby forming complex applications. However, to support component-based and context-aware applications enabling dynamic reconfiguration and adaptation, the component model and the infrastructure should further ease the development. To tackle these issues, recent systems [1], [6], [18] support the use of Aspect-Oriented Programming (AOP) [14]. AOP aids the programmer in the separation of concerns, specifically crosscutting concerns, as an advance in modularization.

A crosscutting functionality represents code that cannot be located in one file but is scattered throughout many different files of an application. The concept of *aspects* improves modularity by locating these scattered elements in one place. Implementing an aspect consists of providing *advice* code and a *pointcut*. While the advice code defines the behavior of an aspect, the pointcut specifies a set of *joinpoints* where this behavior is applied in the application.

In this paper, we explore how to enhance Enterprise Java Beans (EJB) [10] by means of AOP and ontology-based metadata, thus enabling to provide components that can be dynamically adapted at runtime by the environment.

Traditionally, black box components and AOP do not match together, as components inherently forbid changing their implementation or otherwise loose major benefits like encapsulation or information hiding. In the past, several systems [9], [17] have been proposed to overcome this issue by exporting additional points for adaptation. However, they do not sufficiently address scenarios, where dynamic reconfiguration and context-awareness is demanded. We propose AOCI, a context-aware aspect-oriented component infrastructure, which is built upon EJB. Developers using AOCI have the possibility to export potential points for extension matching extended pointcuts of a component for use with AOP, while enforcing encapsulation and preserving the concept of a black box to a certain degree. The extended pointcuts represent the developers domain-specific knowledge of the component, which is expressed by means of the Resource Description Framework (RDF) [15]. Thus, our system supports a greybox component [3] approach, allowing a dynamic adaptation of components at runtime on behalf of the exported pointcut information together with additionally attached metadata. We call this combination a *greybox annotation*. Based on the provided metadata and environment-specific rules the infrastructure is able to integrate non-functional properties, provided by advices, during run-time. This is achieved by dynamically weaving the program elements, which match the rule configuration of the context, together with the component.

Compared to standard component-based solutions [16] we see several advantages. As our solution is based on AOP, components can separate business logic from non-functional properties and issues related to dynamic adaptation do not need to be addressed. Therefore, neither interfaces have to be implemented, nor is a custom code generation tool needed. This significantly reduces the complexity of developing components. Additionally, adaptations can happen without the explicit knowledge of the source code, as the component developer provides the relevant information. Finally, our solution is compatible with every container, which supports the EJB 3.0 specification.

The remainder of this paper is organized as follows. Section 2 gives an example scenario as a motivation for our architecture. In Section 3 we describe the proposed solutions along with the technologies which are needed for the implementation. An overview of related work can be found in Section 4, followed by the conclusion in Section 5.

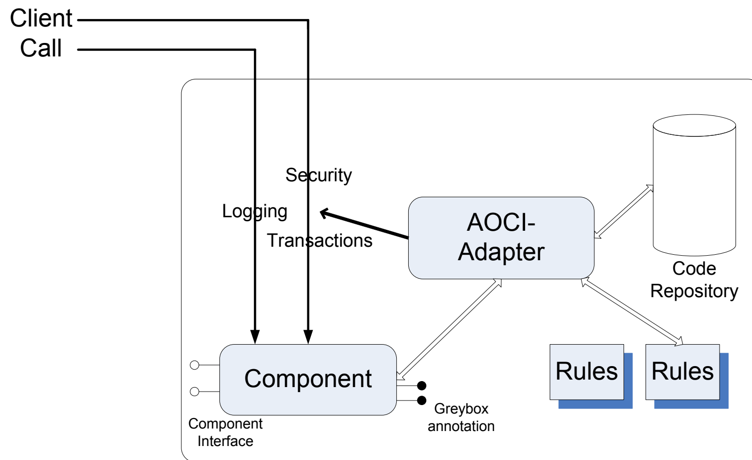


Fig. 1. The AOCI architecture

II. Motivation

In our scenario, an instant messaging software for a company is deployed within the AOCI component infrastructure. Frist, the plain instant messaging component is used without applying any constraints. Clients can dynamically discover the service, log on with their credentials and use it. After a certain time, the administrator of the software discovers, that conversations between users are sniffed from time to time. Therefore, it is decided that all the messages sent by the instant messaging software should be encrypted. This is possible, because the component developer has exported the program elements where adaptation can happen, together with the metadata offering information where security related issues could be applied. The administrator changes the rules of the AOCI-infrastructure, defining that the network traffic is secured by using the Secure Sockets Layer (SSL). Afterwards, the infrastructure intercepts all the calls of the component to a plain socket and replaces the socket by a secure socket (e.g., *java.net.ssl.Socket*) thus enabling a secure communication. After a while, the executive board of the company decides that no private conversations may be held within the company. To guarantee this, all employees at the company are informed that the communication is being logged within the company. Again the administrator changes the rules to apply logging for each message being sent within the company. If clients use the instant messenger at home, the container will recognize the context switch and turn logging off. Furthermore, the company plans to further ease the communication with people from abroad. Therefore, each incoming message should automatically be transferred into the user's native language and each outgoing message be transferred to the language of the communication partner. To fulfil this requirement, a small piece of software is written which accepts a message, transmits it to a webservice and returns the translated message. By attaching it with metadata defined in the ontology, it is quite easy to plug it into the existing software without any greater modification of the instant messenger.

As outlined, AOCI-aware components can be dynamically deployed without knowing in advance which non-functional constraints are demanded at a specific target location. The application developer concentrates only on the business logic, while the constraints are imposed by execution context at runtime.

While some of the functionality could also be fulfilled by the container software, AOCI offers much more flexibility in adapting the software. With the help of RDF and an ontology, it is even possible to use AOP technology together with functional requirements.

III. Proposed solution

Our AOCI architecture proposal is based on EJB. Therefore we suggest an AOP implementation that is tightly integrated into EJB and allows dynamic weaving of aspects. For our prototype implementation, we use the JBOSS [12] application server as it already provides basic support for AOP, allows the dynamic weaving of aspects and is available as Open Source. Figure 1 shows the overall architecture of the AOCI infrastructure. The main components are the deployed component with its greybox annotations, the rules, the *Code Repository*, which stores the nonfunctional advices, and the core component provided by the *AOCI-adapter*. In the following we will shortly discuss the underlying technology RDF, followed by different architectural elements of the solution.

A. RDF-Ontology

In our approach the developers provide means for later adaptability by exporting metadata about the component and thus providing the component as a greybox. This information allows the infrastructure to dynamically weave code to the component. To enable the dynamic binding between an extended component with context-sensitive code from the infrastructure we use RDF, which is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata model using XML. The underlying structure of

a RDF expression is a triple in the form of subject-predicate-object. A collection of RDF statements intrinsically represents a labeled, directed pseudo-graph. The benefits of RDF are its simple data model and the ability to model disparate, abstract concepts. Using this technology we are be able to define an ontology that represents a set of concepts within a domain and the relationships between those concepts. The metadata in the rules and in the components corresponds to the nodes in the ontology. This allows the infrastructure to have a common understanding of the greybox components, the rules and the advices, which are stored in the Code Repository, thereby enhancing the infrastructure to dynamically adapt the software in a context-sensitive environment. To outline the proposed mechanism we present a partial definition of a domainspecific taxonomy targeting security as an example for a non-functional requirement (Figure 2).

IV. Extended Pointcuts

There are two ways to advise elements of a component. Either a central XML-configuration file, where all possible modifications can be specified, is used or Java annotations as a hint for the infrastructure. In the following, the annotation-based approach will be outlined (Figure 3), in which the method `sendMessage` is marked.

```
public class InstantMessenger {
        @AOCI(SecurityMechanism.
            SecurityProtocol.NetSecurity)
        @AOCI(SecurityMechanism.Auditing)
        public void sendMessage(String text,
            String receiver){ }
}
```

Fig. 3. Annotation-based extended sample pointcut

With this information the AOCI infrastructure knows, that this method is a place, where encryption like SSL or auditing can be applied.

To determine how the component should be dynamically adapted, defining the extended pointcuts is not enough. The infrastructure must know which code must be added or replaced. Therefore rules are needed on behalf the infrastructure can match existing extended pointcuts with possible code fragments. The notion of the rules is very similar to the extended pointcuts and also based on RDF. The following snipped (see Figure 4) determines that communication must be encrypted with the help of SSL.

A. AOCI Adapter

The AOCI-Adapter is the part of the framework which analyzes and adapts the components. It has to fulfill several steps (Figure 5). First of all, the extended pointcuts of the greybox are read. It then checks, if the syntax of the extended pointcuts is right and corresponds to a node in the RDF tree. On success an appropriate implementation is searched. If no such is found, there are different possibilities how to proceed. Either the infrastructure can throw an error, can forget about

```
<xml-version= 1.0   encoding=UTF-8 >
<AOCID>
<entry>
<key> SecurityMechanism.SecurityProtocol.
    NetSecurity </key>
<value> SSL </value>
</entry>

</AOCID>
```

Fig. 4. Example rule defining a security constraint

the missing implementations and just proceed or can take a standard implementation for the extended pointcut. The infrastructure then dynamically builds an AOP fragment with the implementation. Within the last step the point of the extended pointcut is taken as a hook and the infrastructure weaves the newly created fragment with the current implementation.

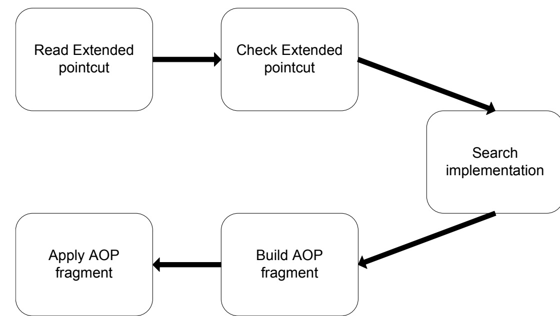


Fig. 5. The AOCI-Adapter

While RDF is a flexible and extensible way to represent information about the resources on a platform, there is still need for a standardized way to retrieve these data. Therefore we propose the use of SPARQL [5], which consists of the syntax and semantics for asking and answering against RDF graphs. Furthermore it is possible to query by triple patterns, conjunctions, disjunctions, and optional patterns. Constraining queries by source RDF graph and extensible value testing is supported as well as the possibility that results of SPARQL queries can be ordered, limited and offset in number. As a framework for RDF and SPARQL we propose Jena [13], which allows to dynamically compose and evaluate these queries.

V. Related Work

Recently, several infrastructures targeting the adaptation of component-based software have been developed. Most of them concentrated on providing mechanisms to adapt components, only few use the benefits of AOP and make use of an ontology-based metadata description.

Pessemier et al. [17] show that AOP can safely be supported by Component-Oriented- Programming (COP). In their paper, they explain which problems they have encountered when using AOP with components. To overcome these issues they have introduced the greybox, which marks a compromise

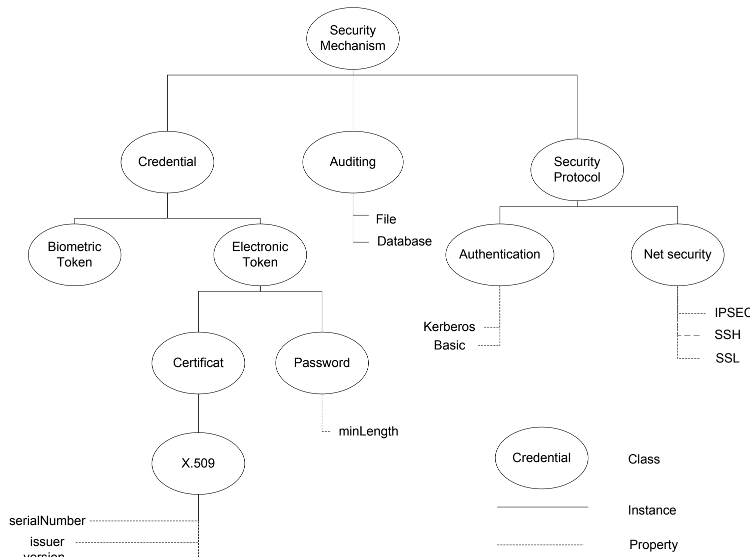


Fig. 2. Security ontology

between modularity and openness. However there is no support for building context-aware applications using the support of an ontology.

In their work, Rho et al. [18] describe that services in a Service Oriented Architecture (SOA) are general in their nature, which allows them to fit the needs of as many clients as possible. On the other hand, these systems cannot address each single requirement. Therefore they introduced the concept of context-sensitive service aspects which can be changed dynamically at runtime. Their framework is called Ditrios, which is based on OSGI, allowing a context-sensitive weaving. Although there is support for AOP, there is no concept like a greybox.

Duclos et al. [6] outline that both, component-based applications and AOP address the same separation of concerns issue. While containers are only proposing a fixed set of services, AOP is working at object level. Additionally they present an approach to get both the advantages of AOP and component based systems. Furthermore, they provide a simple language for using aspects on applications. However, the software allows the adaption of components, but no context-awareness is supported.

The Rainbow framework [7] provides a software architecture and infrastructure to support the self-adaption of software systems. To fulfill these issues, the architecture uses an abstract model to monitor the systems runtime properties, and provide means to react upon events and adapt the software. However, Rainbow neither addresses the concept of greyboxes nor AOP and therefore misses concepts for dynamic injection of custom code.

Another component-based approach is presented in the K-Component-Model [4]. The work is based on architectural reflection. To guarantee integrity and a safe dynamic software evolution a graph transformation is proposed. Furthermore adaption-specific code is separated from functional code by encapsulating it in reflective programs called adaption contracts. In contrast to AOCI, KComponent does not address AOP and greybox and has no support for an ontology.

CAMidO [2] is an architecture for supporting development and execution of context-aware component applications. It provides an ontology metamodel, which facilitates the description of contextinformation. Additionally the platform provides the possibility to write interpretation and adaption rules, which are used by the CAMidO-compiler. However, there is no support for either AOP or ontology.

Toa Gu et al. present a Service Oriented Context-Aware Middleware (SOCAM) [8], which provides efficient support for acquiring, discovering, interpreting and accessing various contexts to build context-aware services. A formal context model is introduced, which is based on ontology using Web ontology Language to address issues including semantic representation, context reasoning, context classification and dependency. The infrastructure has no support for components.

Mügge et al. [9] present a solution for an adaptive middleware in the area of ubiquitous computing and describe a new approach combining aspect-oriented programming with structural metadata. However, they do not address context-aware applications.

Figure 6 shows an overview of the capabilities of the presented related work. Our approach is the only one supporting context-awareness, AOP, components, the greybox concept and semantic ontology. As we want to weave different aspects and components together, which do not know each other until runtime, we attach importance to the fact that the infrastructure supports both the concept of a greybox and is capable to deal with an ontology.

System	greybox	AOP	Context-aware	Component-based	Adaptive	Ontology
Pessemier et al.	X	X		X	X	
Rho et al.		X	X		X	
Duclos et al.		X		X	X	
Rainbow			X	X	X	
K-Component			X	X	X	
Camido			X	X	X	X
SOCAM			X		X	X
Mügge et al.		X		X	X	
AOCI	**X**	**X**	**X**	**X**	**X**	**X**

Fig. 6. Overview of capabilities of related approaches

A. Conclusion

In this paper, we presented an architecture based on the EJB component model, which allows the dynamic and automated adaptation of components. Because this usually breaks the modularity of components, we have introduced the greybox-concept, which allows annotating certain program elements to be adapted by the infrastructure.

At the moment, we are working on a prototype of the described infrastructure. As already mentioned, it is based on JBoss as an EJB container and uses the AOP support provided by JBoss. In the near future we will define an ontology, which contains preferably most functionality needed for effective adapting components. This will demand for extended case studies to figure out the right balance between generality and practical usability of the terms defined by the ontology. Additionally, we consider enhancing our infrastructure by using OSGi [11] for flexible loading and unloading of aspect as well as component code.

Additionally we are still evaluating some issues. One question is to find out, if there is already a standardized ontology which could possibly be adapted by AOCI and would ease the further development of the infrastructure. Furthermore, we will investigate, if it is possible to attach current software with the metadata model of AOCI, which would allow to easily integrate them with the infrastructure. Another field of examination is to find out, how to combine the infrastructure with other technologies like web services. As a rule, AOP based system are restricted to deal with non-functional crosscutting concerns. Based on our infrastructure and the ontology we will evaluate, if it is possible to extend the use of AOP to some functional requirements within AOCI.

References

[1] D. B. A. Gal, M. Franz. Learning from components: Fitting aop for system software. In *Proc. of the Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2003.

[2] N. B. Behlouli, C. Taconet, and G. Bernard. An architecture for supporting development and execution of context-aware component applications. In *ACS/IEEE Int. Conf. on Pervasive Services*, pages 57–66, 2006.

[3] M. Büchi and W. Weck. A plea for grey-box components tucs technical report no.122, 1997.

[4] J. Dowling and V. Cahill. Dynamic software evolution and the k-component model. Technical Report TCD-CS-2001-51, Trinity College Dublin, 2001.

[5] A. S. e. E. Prud'hommeaux. Sparql query language for rdf, w3c working draft. http://www.w3.org/TR/rdf-sparql-query, 2006.

[6] F. D. J. Estublier and P. Morat. Describing and using non functional aspects in component based applications. In *AOSD '02: Proc. of the 1st Int. Conf. on Aspect-oriented Software Development*, pages 65–75. ACM Press, 2002.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[8] T. Gu, H. K. Pung, and D. Q. Zhang. A service-oriented middleware for building context-aware services. *J. Netw. Comput. Appl.*, 28(1):1–18, 2005.

[9] H.Mügge, T.Rho, and A. Cremers. Integrating aspect-orientation and structural annotations to support adaptive middleware. In *Proceedings of the 1st workshop on Middleware-application interaction: in conjunction with Euro-Sys 2007, Lisbon, Portugal*, 2007.

[10] S. M. Inc. Javabeans component architecture specification revision 1.0.1. http://www.java.sun.com, 1998.

[11] O. S. G. Initiative. Osgi service platform version 3. http://www.OSGi.org, 2003.

[12] JBoss. JBoss EJB Server. http://jboss.org.

[13] Jena 2. A Semantic Web Framework. http://www.hpl.hp.com/semweb/jena2.htm, 2006.

[14] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In ECOOP 97: Proceedings of the 11th European Conference on Object Oriented Programming, number 1241 in Lecture Notes in Computer Science, pages 220-242, Jyvskyl, Finland, 1997.

[15] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. Technical report, W3C, 2004.

[16] OMG. Corba component model specification version 4.0. Technical Report formal/06-04-01, 2006.

[17] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. A safe aspect-oriented programming support for component-oriented programming. In *Proc. of the 11th Int. ECOOP Workshop on Component-Oriented Programming (WCOP'06)*, volume 2006–11 of *Technical Report*, Nantes, France, jul 2006. Karlsruhe University.

[18] T. Rho, M. Schmatz, and A. B. Cremers. Towards context-sensitive service aspects. In *Work. on Object Technology for Ambient Intelligence and Pervasive Computing*, 2006.

[19] C. Szyperski. Component software: Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., 2002.

How dark should a component black-box be? The Reuseware Answer

Jakob Henriksson and Florian Heidenreich and Jendrik Johannes and Steffen Zschaler and Uwe Aßmann
Technische Universität Dresden
Email: {jakob.henriksson|florian.heidenreich|jendrik.johannes|steffen.zschaler|uwe.assmann}@tu-dresden.de

Short answer: Jet-black with plenty of holes, some of which are not visible to everyone.

Long answer:

I. Introduction

The Software Technology Group at TU Dresden has long experience with component-based software development and techniques. For a recent addition to the public debate, see the book entitled *Invasive Software Composition* [1]. Currently, the group is involved in projects (e.g. European NoE REWERSE, IP ModelPlex, feasiPLe etc.) addressing composition for *declarative languages*. More precisely, languages important for the development of the Semantic Web and in software modeling are addressed. Such languages include, for example, rule languages (Xcerpt, R2ML), Web query languages (XQuery), ontology languages (OWL, Notation3) and general modeling languages (MOF, UML, Ecore). To enable component-based development for such languages, the composition framework *Reuseware*[1] is being developed [3], both as a conceptual framework and as a tool.

Szyperski [4] defines a software component as follows:

> ”A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [4]

This definition calls for components to be black-box components where no information can be inferred beyond the explicitly specified interfaces of the component. Such an approach enforces strong encapsulation and is very useful for reuse of components by third parties as these third parties need only rely on the—relatively little—information provided in the interface specifications.

For declarative languages, a pure black-box approach cannot always be taken. We currently see two reasons for this. First, not all declarative languages describe processing entities (e.g. ontology languages). As such, there is not even a notion of well-defined inputs and outputs to interface components, which is an assumption made for black-boxes. Thus, a different composition paradigm is needed to address certain declarative languages. We argue that the grey-box and fragment-based component paradigm is more suited for these languages.

[1] http://www.reuseware.org

The second reason is related to our desire to reuse existing tools for the different languages. To achieve this, all referenced components need to be composed before applying the tools. As the tools are assumed to not have a prior understanding of components (in the fragment-based sense of the word), they do not understand the need for their restricted interaction (essential for proper component encapsulation). For this reason the components needs to be *opened up* such that their interactions can *statically* be ensured in the composition result. Again, this is not an idea supported by black-box component environments, but possible in composition systems based on grey-box approaches.

For some general-purpose languages (e.g. C++, C#, Java), components can be described on different abstraction-levels—either as run-time entities or as static source-code snippets (as done for aspects in Java). But for most languages used in software modeling or on the Semantic Web we do not have much choice. Thus, for the languages in these important fields, components necessarily consist of source artifacts—snippets of descriptions—which can play roles in more complex, complete, coherent and usable descriptions or declarative programs. However, one should take care not to meddle with one of the most powerful notions in component-based development: *the power of abstraction*. Thus, it is of utmost importance to properly encapsulate components—hide their details—and to access them via *well-defined composition interfaces*. Here, again, we are in line with Szyperski's definition from above: All access to the component should occur through well-defined interfaces, all dependencies should be explicit. We shall return to the issue of interfaces shortly.

Our work has its roots in Invasive Software Composition (ISC) [1]. ISC takes a grey-box composition approach where components, or *fragments*, are static source-code entities with well-defined interfaces using the notion of *hooks*. A hook is a location in a component which may effectively be replaced by another component, thus, composed. As such, the hooks of a component define its interface. The replacement of a hook with some existing component constitutes the basic composition technique of ISC. One of the conclusions from work on ISC was the identification of a set of *primitive composition operators* implementing the described composition technique. Two of the identified primitive composition operators are *bind* and *extend*, where *bind* replaces a hook with some component once and *extend* possibly multiple times. The composition technique and operators defined for ISC are very general

and applicable to many different languages and situations. It should be noted that ISC is able to realize existing composition approaches and techniques, such as aspect-orientation, view-based programming, hyperspaces etc.

Our experience with ISC and component-based development for declarative languages has refined our requirements for composition interfaces. Most importantly, we argue that components for declarative languages shall indeed be grey-boxes, but with tailored and *refined composition interfaces* to answer the call from language-specific needs and language-specifically developed composition operators.

II. Dedicated composition systems and environments

Many domain-specific languages in software modeling and on the Semantic Web do not provide sufficient constructs for defining reusable entities—components. Many languages do have some form of abstraction and reuse idea, but it is often limited, inflexible and most of all—fixed. For example, rule languages on the Semantic Web often allow rule chaining; the possibility of sequencing rules in different chains of computations. As such, the notion of the *rule* is the level of reuse made possible by the language itself. No other entities are reusable; there is no other level of *abstraction*. That is, the set of abstractions provided by the language is fixed. Thus, once the language has been designed and its relevant tools have been developed, the language as such is very inflexible to be changed for new abstractions. It should be noted that the expressiveness provided by languages is usually adequate, since the languages certainly were developed against use-cases and specific requirements. We exploit the fact that appropriate expressiveness is provided for by reusing existing tools developed for the different languages when processing the composition results. However, we will address the issue that a flexible level of abstraction is not to be found.

We argue that instead of redesigning an individual language, additional levels of abstraction, and thus reuse, can be provided via *composition*. We propose to layer a *light-weight dedicated composition system* (LWDCS)[2] on top of a targeted *core language* and its tools (see Figure 1) to provide richer abstractions and allow programmers to think about their programs in new and interesting ways. The composition system is dedicated because it addresses issues for a single targeted language, and light-weight since once developed and deployed it is assumed to be operable without its users directly being aware of it. The LWDCS injects a core language with additional constructs, giving users the possibility to define reusable components and to compose them in desired ways, all tailored for the need at hand. The LWDCS is responsible for interpreting the newly introduced constructs and for composing specified components into programs or descriptions of the core language. Thus, the existing and already developed tools are reused and the semantics of the core language is appropriately retained (as mentioned, we deem the core languages already capable of expressing what they should). Furthermore, a LWDCS is type-safe, ensuring that resulting descriptions (programs) are (syntactically) valid with respect to the underlying core language. Ensuring semantically correct results is also possible, but not further discussed here.

[2]Pronounced *low-deeze*; pl. LWDCSs (*low-deezes*).

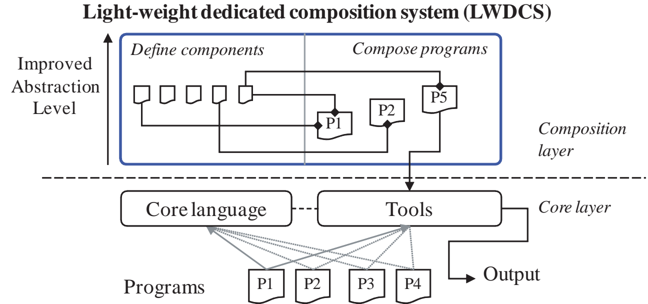


Fig. 1. A light-weight composition system can be layered on top of a language and its associated tools to improve and add the level of abstraction provided by the underlying language itself.

A composition system can be seen as a triple consisting of: a *component model,* a *composition language,* and a *composition technique* [1]. We argue that a LWDCS should be constructed as a refinement of a generic composition system, where the triple comprising the system is specialized for the task at hand—indeed tailored (see Figure 2). As can be seen from Figure 2, the dedicated composition language is adapted for the specialized task and refined from a more general-purpose language. Furthermore, the dedicated component model references an *upper-level component model* where general composition system concepts are modeled. Finally, instead of including a general composition technique and generic operators in the dedicated composition system, it is shipped with a set of predefined, specialized, composition operators.

The most important detail to notice in Figure 2 in order to answer the question about the desired darkness of components is the relationship between the set of dedicated operators and the dedicated component model. The detail to notice is that the component model, which effectively determines the darkness of the components used in a composition system, heavily depends on the specific composition operators included in the LWDCS. We shall return to this issue with a more detailed discussion in Section IV. First, before describing our notion of refined composition interfaces, we will briefly describe how a dedicated composition system may be semi-automatically generated. In particular, how a component model may be derived from a core language.

III. Generated component models

We intend to build upon the language-independent composition technique introduced in ISC. This means that components may contain hooks that can be replaced by other components. We refer to these positions in components by the more general term *variation points*. Thus, the variation points declared in components define the components' interfaces. From these

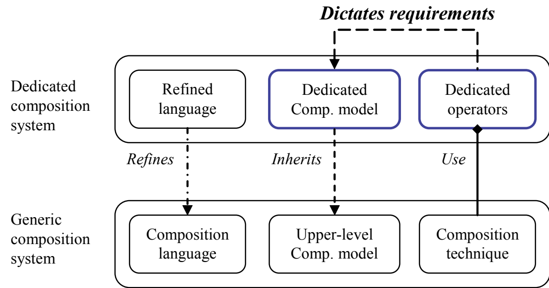


Fig. 2. A dedicated composition system is a specialization of a generic composition system where the tailored composition operators addressing particular issues in a declarative language dictate the form and detail of the dedicated component model and thus, the components' interfaces.

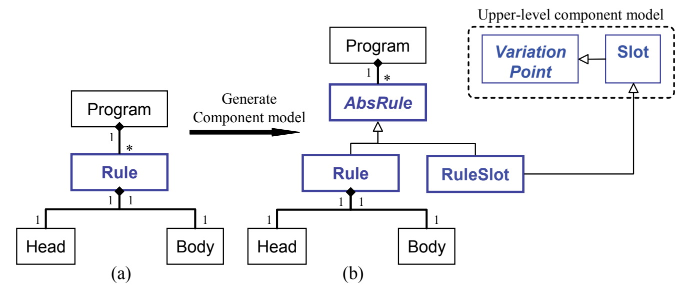


Fig. 3. The abstract syntax description of a simple rule language on the left-hand-side can be extended into the abstract syntax description on the right-hand-side to allow programs to be underspecified with unknown rules, to be composed into the program at a later point.

requirements, one can automatically generate a component model from a *core language* description (grammar or model based) [3]. Figure 3 (a) shows a simple (partial) model of some rule language. The model states that programs of the rule language consist of one or more rules, which in turn are composed from a head and a body (what they are in detail has been left out here).

Assume the simple case that we want to be able to write rule programs in our rule language where certain rules are not explicitly given, but left *unspecified* (the program as a whole is *underspecified*). The underspecified program is a component with an interface, given by the variation points programmed into the component. In order to be able to declare variation points we inject the core language with constructs for this purpose. This modification can be seen in Figure 3 (b). The concept of the rule has here been made *variable*. Rule programs may now consist of normal rules (concept `Rule` in Figure 3 (b)) and rule *slots* (concept `RuleSlot` in Figure 3 (b)). The abstract super-concept `AbsRule` is introduced to represent this choice. A `Slot`, as can be seen in the upper-level component model is a kind of variation point. Here it is assumed that a slot has some concrete syntax such that variation points can explicitly be declared in components. The model in Figure 3 (b) properly describes what our simple rule components look like and defines how the composition technique is allowed to modify the components (by replacing variation points with other suitable components). The only access points to the components are the declared variation points (expressed using slots), everything else is properly encapsulated. As such, the derived language model in Figure 3 (b), along with references to the upper-level component model, is the component model for our simple components. It is possible to automate such transformations.

IV. Refined and Controlled Composition Interfaces

It is useful to be able to reuse common composition techniques across different dedicated composition systems targeting different languages (see relationship between the generic composition technique and the dedicated operators in Figure 2). This is beneficial since the basic technology does not have to be reimplemented for each composition system and targeted language. However, in order to support and realize the appropriate kinds of reuse abstractions, different languages require special-purpose composition operators. Thus, it is desirable that the dedicated composition operators are defined in terms of, that is, reuse, the primitive composition operators implementing the general composition technique (see Figure 4). Furthermore, if a specific reuse abstraction concept is desired for different languages, using the same basic and underlying composition technique is again advantageous for practical reasons. Examples of reuse concepts not limited to a specific language are modules and aspects (disregarding the exact detail of their purpose and how they look in the specific languages).

As our work extends that of ISC, which provides a very general composition technique, we aim at reusing this technique and its primitive composition operators for creating LWDCSs. While a single operation of a primitive operator only can describe a *low-level* composition step, a properly defined sequence of such primitive composition operators can achieve a more advanced desired effect on a set of fragments—a *high-level* composition step. If such a high-level sequence is found useful for different fragments, one would like to be able to encapsulate the sequence as a single reusable atomic composition operator. We call such an operator a *complex composition operator*. Thus, a complex composition operator is able to encapsulate and realize a *non-obvious reuse abstraction*. This notion gives us the possibility to develop language-tailored composition operators to be included in LWDCSs.

One thing to notice about complex composition operators is that they may not only encapsulate a sequence of primitive operators, but also components. That is, some composition operators may require internal components, needed for the realization of the (abstraction) construct they are implementing. Such components are not visible, or indeed known, to programs using the operators; they are completely encapsulated within the operator definitions.

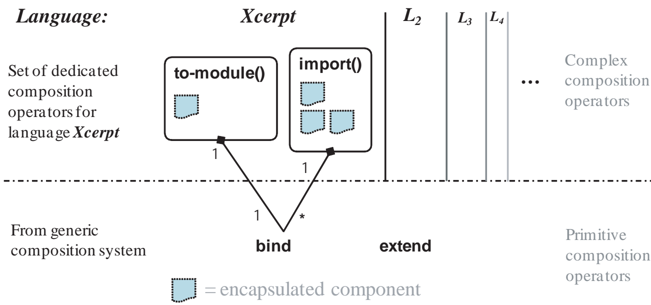


Fig. 4. Complex and dedicated composition operators for a dedicated composition system are defined in terms of general composition operators and techniques from invasive software composition. The operators to-module() and import() address specific issues in the core language Xcerpt.

A. Example – Modules for the Web Query Language Xcerpt

We have practical experience with targeting and building a LWDCS for the Web query language Xcerpt [2]. Xcerpt is a powerful rule-based language following the logic programming paradigm for querying different kinds of semi-structured data. An Xcerpt program consists of a finite set of rules. What differentiates Xcerpt from some other well-known Web query languages, e.g. XQuery, is that Xcerpt programs (i.e. their rules) have a clear separation between data query parts and data construct parts. As in other logic programming languages, Xcerpt rules consist of a head and a body. The body of a rule can match existing data, resulting in variable bindings. The variable bindings produced by successful matching of the body of a rule can then be applied to the head of the rule in order to derive new data. As such, the rule bodies represent the queries and the rule heads the construct parts.

An identified and desired (but so far lacking) abstraction for Xcerpt was the notion of Xcerpt *modules* (much in the style of other logic programming systems). An Xcerpt module consists of a set of rules, which can be imported and reused in different programs. A good example of a useful module is a set of rules able to perform simple reasoning on ontology documents (e.g. OWL). An example of such reasoning is to derive implicit subclass-of relationships from explicitly declared class-hierarchies.

As a module consists of a set of rules, they should all be included in the importing program at composition-time, such that they are available to the Xcerpt interpreter when the composition result is executed. However, properly realizing the module system is more subtle and complicated than just executing the merger of different rule-sets. Since a module from our point of view is a *component*, certain parts of the module should be able to be encapsulated. From a usage perspective, a module can almost be seen as a black-box with an *input* rule and an *output* rule. The input rule is passed data to process (possibly constructing intermediate results for rules encapsulated in the module) and eventually data to be used by the importing program is constructed by the output rule. At the level of composition, however, we cannot consider modules as black-boxes. In order to allow modules to be encapsulated, one must ensure that inappropriate rule dependencies do not occur when programs and modules are merged before being executed. That is, programs should only have access to certain rules in imported modules, and vice versa. This encapsulation can be realized by transforming the heads and bodies of the rules of the imported module in appropriate ways. The details are left out since it is not relevant exactly how this is realized. What is clear is this: If rules in modules are to be transformed in some way at composition time, the way they are transformed, and thus *accessed by composition operators*, must properly be reflected in the relevant component model.

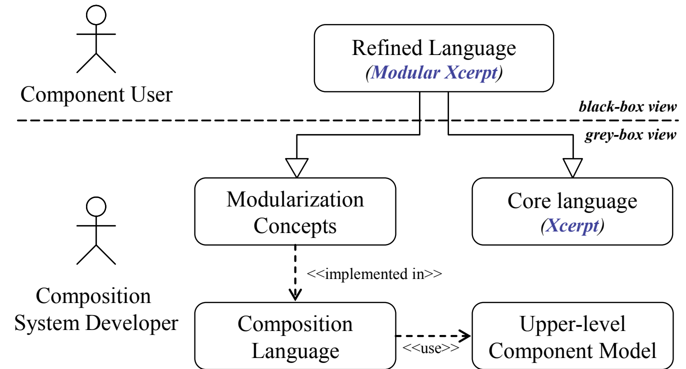


Fig. 5. The *Component User* and the *Composition System Developer* are working with different parts of the composition system and have different views on the system.

To understand the requirements of the component model, it is helpful to distinguish two different roles—or view-points—with respect to a LWDCS (containing the component model). Figure 5 illustrates these different view-points.

1) *Component user role* Users of the above-described Xcerpt module system only want to be able to define (encapsulated) modules and import already existing ones. The constructs for doing so should appear to be first-class constructs of the core language rather than added composition operators. As such, one should not require the module programmers and users to define precisely how and where their modules must be transformed during composition. That is, they should not be required to describe how the underlying encapsulated composition operators realize the module system and, thus, access the modules (components).
2) *Composition system developer role* The composition system seen from the view of the system developer is however much different. The system developer cannot assume the black-box view of the users, but rather a *grey-box view* in line with our arguments of this necessity for declarative languages. The system developer must develop the complex composition operators responsible for realizing the module system and provide an appropriate component model reflecting the intended interfaces of the components. We recall from Section I

the argument for the need to ensure proper component interactions statically. Hence, the components *do* need to be opened up in the deployment of the module system and this responsibility lies on the composition system developer.

To support these different roles—considered attractive for the users—the development of the specific composition operators and the composition system as a whole dictate requirements for the component model. We therefore need to transform the core language model in a slightly different way (Figure 6) as to what was done in Figure 3. As can be seen in Figure 6 (b), in place of the head construct we introduce a head variation point (`HeadVP`), which forms part of the interface of components adhering to the component model. At the variation point, either the original head construct can directly be programmed in its place (as a *default value* for the variation point), or a concrete variation point (slot) can be used. This means that regardless of whether the head of some rule consists of a core language head construct (`Head`) or is left unspecified (using the introduced `HeadSlot` construct), the component model describes it as accessible, as part of the component's interface.

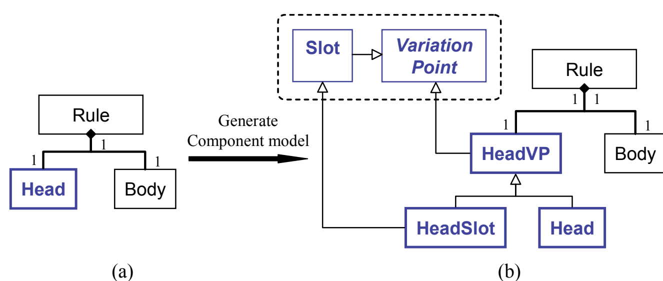


Fig. 6. The original construct (here `Head`) must at composition time be accessible to certain composition operators. Thus, the construct must be part of the interface of the component, which is realized by making the variation point (`HeadVP`) a super-class of Head.

The same kind of transformation can be done for the body of rules. Again, it should be stressed that module programmers working against the component model in Figure 6 (b) do not have to express via some special mark-up that the rule heads are part of the interface. They can write rules as they normally would, but still expect the programmed modules to be usable in the LWDCS realizing the module system.

So, the module system is realized by a set of dedicated composition operators (transforming rules), along with a dedicated component model adjusted to the needs of the operators. Along with a composition language (not discussed here) we can create a LWDCS for Xcerpt realizing *additional abstractions*, in this case the possibility of authoring encapsulated modules and using them in Xcerpt programs.

The critical notion is the following: due to the encapsulation of the complex composition operators, we find it necessary to refine our notion of composition interfaces. This is a consequence from the fact that was remarked upon earlier: the set of dedicated composition operators included in a targeted composition system dictates the form of the associated component model, that is, how components look and interact (see Figure 2).

In a similar fashion one can identify needed abstractions for other declarative languages. Instead of re-designing the language and its tools one can realize the abstraction by implementing the necessary additional constructs as complex composition operators in a LWDCS and generate an appropriate and tailored component model with the *appropriate shade of darkness*.

V. Conclusion

In this paper we have presented the Reuseware approach to Invasive Software Composition in an attempt to answer the question “How dark should a component black-box be?” for components in declarative languages or in situations where composition occurs on the source-code level.

Our short answer has been “Jet-black with plenty of holes, some of which are not visible to everyone.” In the long answer we showed that this means that we require encapsulated components where composition can only occur in well-defined places—hence they are “jet-black”. At the same time, however, component developers and users should not have to worry about all the details of the composition interface relating to encapsulated composition operators. Rather, this part of the interface should be described in the relevant component model and taken advantage of by the complex composition operators available in the dedicated composition system for which the components have been written. Hence, components “have plenty of holes”, but they are “not visible to everyone”. More specifically, some parts are visible to developers of dedicated *composition systems* (LWDCS), while *component* developers and users only have to care about the part of the interface relevant to them.

Acknowledgment

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net), as well as through the 6th Framework Programme project Modelplex contract number 034081 (cf. http://www.modelplex.org) and by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. http://www.feasiple.de).

References

[1] U. Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[2] F. Bry and S. Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 295–310. Springer-Verlag, London, UK, 2003.

[3] J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware – adding modularity to your language of choice. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.

[4] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley Publishing Company, second edition, 2002.

Black & White, Never Grey: On Interfaces, Synchronization, Pragmatics, and Responsibilities

Franz Puntigam
Technische Universität Wien
Institut für Computersprachen
Argentinierstraße 8, A-1040 Vienna, Austria
E-mail: franz@complang.tuwien.ac.at

***Abstract*—When composing systems from components we have to deal with involved aspects like synchronization or non-functional properties like performance. It is nearly impossible to clearly specify such aspects in interfaces. Looking behind the interfaces (into grey boxes) does not solve the problem because of lost substitutability. In this paper we explain on the example of synchronization, first, how pragmatic descriptions solve the problem in usual cases and, second, that moving and splitting responsibility for synchronization between components is helpful in further cases. We argue that there is a general pattern applicable to many functional and non-functional aspects behind this solution.**

I. Motivation and Overview

The most important concept in object-based programming is data abstraction, this is the encapsulation of data and routines in objects together with data hiding. We regard objects as black boxes the internal structure of which is hidden from the rest of the world. Since users of an object do not see and (more importantly) have no direct access to implementation details, we can safely change these details and even substitute the object for another similar one without breaking code. On this substitutability property we build successful concepts like inclusion polymorphism (with subtyping and dynamic binding) in object-oriented programming as well as the composition of systems from components.

While we use concepts and tool based on data hiding we often experience the fact that data hiding poses limits on the usability of objects and components. We are led into temptation to look inside the box and make use of some parts we find there, this is, we regard the box as being grey instead of black. Doing so usually gives us an immediate advantage: We have to write less code, get better performance, and can show useful properties. Often the pain follows later when we detect that our system does not work properly after substituting objects or components. Then, we remember why our tools required black boxes. Even worse, especially when composing components there are situations where we intentionally give up the idea of black boxes and use grey boxes instead, knowing that it will be difficult to replace them later on; sometimes the only other alternative would be to program everything from scratch. Such situations give raise to the question whether black boxes are really that successful when dealing with components.

Actually no component is completely black because interfaces give needed information to users. This information is considered to be stable and, therefore, we regard interfaces as white boxes. In this paper we distinguish between components as

- black boxes with white boxes at their entries where interfaces clearly specify which portions of the components are white and stable, and the unstable black rest is completely hidden from the user,
- and grey boxes where even unstable portions can become visible.

Of course, a balanced combination of black and white is preferable over grey since we can get substitutability and the needed stable information at the same time.

Conventional interfaces (= signatures) cannot give all the information needed by users. For example, users need information about synchronization in the component to avoid synchronization conflicts that can lead to deadlocks and livelocks. Hence, we often think of grey boxes as something we cannot avoid in practice.

In the author's opinion, rich interfaces and pragmatic models can be very effective to reduce the need of grey boxes. It is not even necessary to wait for new technologies (although new techniques slowly coming up promise to be quite helpful in this respect) because clear contracts and component descriptions written in a natural language are often all we need. However, the basics of many contracts in software are still not understood well enough. For example, even seemingly harmless synchronization primitives somewhere deep in the code of a component can cause synchronization conflicts. If we solved this problem by putting all synchronization information into the interface, we would have to consider much more program parts as stable than we like to, causing many useful component substitutions to be impossible. We will discuss such topics in this paper.

In Section II we point out problems with synchronization in black boxes and how they are usually addressed in practice. In Section III we argue that it can be quite useful to move responsibility for synchronization from a component to its user. Next, in Section IV we generalize what we have seen so far to further aspects. Finally, in Section V we give some concluding remarks.

II. Synchronization and Informal Interfaces

We deal with synchronization whenever we require an action to be performed before another one as well as when we simply perform actions in sequence. In sequential programs, synchronization is implicit in the control flow, and in concurrent programs we handle it explicitly.

Synchronization is a very complex topic because it can be conflicting. For example, if a component performs an action A only before another action B while a user insists on B to be performed before A (by invoking B before A) we have a deadlock. The user must know the behavior of the component to avoid such conflicts. Unfortunately, the requirement of "A before B" may not be statically determinable; it may occur only under certain conditions like full buffers that are not predictable in practice.

From playing with this and similar simple examples we can learn the following:

- We expect good components not to cause deadlocks. Unfortunately, this statement is more of a wish than reality: Deadlocks often result from the interaction between components. We usually find no deadlocks when testing components in isolation. Of course, good components make it more unlikely to experience deadlocks in realistic environments.
- Static deadlock prevention is usually no option because of its restrictive nature. Actually it is easy to find a set of syntactic constraints to ensure programs to be free of deadlocks. However, each such approach restricts the set of programs conforming to the constraints in a severe way that is not acceptable in most cases. Statically ensuring other liveness properties (like livelock prevention) restricts conforming programs even further.
- In the few cases where we absolutely need static deadlock prevention we have to design the whole system (or subsystem) with this goal in mind. This means, components can be used only if designed with the same goal in mind; then, interfaces statically specify the synchronization behavior. A simple analysis of a component (as a grey box) is rarely helpful in this respect.
- Without static deadlock prevention we will sometimes (hopefully not too often) experience deadlocks. To find out the cause of them it is helpful to inspect the source code of the component as a grey box. Without a proper documentation and without access to the source code it can be quite difficult to find the reason for the deadlock. Once we know the problem we can usually avoid it without any need to change the component (except if we found a problem in the component itself).
- It is difficult to describe all cases that can lead to a deadlock when interacting with a component. Most interface specifications do not tell much in this respect. One of the simplest ways to specify all possible deadlocks is to describe the complete control flow by, for example, an automaton. However, such approaches turn black into nearly white boxes.
- Without clear interface specifications it is always an open question who is responsible for a problem, the component or its user.

These considerations do not give much hope that we can avoid grey boxes by giving appropriate interfaces in the context of synchronization although such interfaces are important to arrange responsibilities. Other liveness properties (like livelocks) are even more difficult to handle. Fortunately, there is another side of the coin as we will see below.

Deadlocks do not occur that often in practice. Obviously, users and developers of components already employ some sort of (informal) interface specification that includes synchronization behavior. In general we have a description that tells users what a component does and how to use it (often in the form of an application programmer interface). Even if the description does not specify the component's synchronization behavior in detail, it shows the overall structure, and experienced users are able to derive the most likely synchronization behavior just from this description. Only in cases of unexpected synchronization patterns the documentation gives hints on them. This pragmatic and mostly implicit kind of communication between developers and users works surprisingly well.

Simplicity of synchronization patterns and their pragmatic use is an important precondition for this communication. Theoretically we can imagine a huge number of possible synchronization patterns, but only a handful of them actually occurs. For example, with a mechanism to ensure mutual exclusion ("synchronized" in Java) and another one to avoid underflows and overflows we cannot just develop buffers, but almost everything that needs synchronization. Furthermore, it is not necessary to mention mutual exclusion in an interface of a component: If the code running in mutual exclusion terminates in finite (and for practical reasons short) time, then mutual exclusion cannot cause synchronization conflicts [10]. (Of course we must specify in interfaces if a routine expects to be invoked in mutual exclusion and does not ensure mutual exclusion by itself.) Synchronization to avoid overflows and underflows in a component can be in conflict with (often unintended and accidental) synchronization by the user. Fortunately, it is usual to describe in the interface what happens in the case of an underflow or overflow although this description often does not refer to synchronization.

By using "wait" and "notify" we can produce arbitrarily complex synchronization patterns. However, it is better not to do so. The implicit communication between developers and users works well in the simple cases mentioned above (and in some further simple cases), but not in general. It is extremely difficult to describe complex synchronization patterns in interfaces. Moreover, complex synchronization patterns are often less stable. Since they should be specified in interfaces so that users can avoid conflicting synchronization, there is a danger that we have to either keep inappropriate synchronization behavior in future versions or change interfaces and thereby lose substitutability. Good components keep their synchronization behavior simple and as usual (this is, pragmatic).

There is a tendency to support more restrictive synchroniza-

tion primitives that can be better expressed in an interface. For example, in Polyphonic C# [2] (based on the Join calculus [3]) we combine routines like "put" and "get" in a buffer to a chord to be executed as a single unit. Users see in the interface how routines in a chord are synchronized. Since only one routine in a chord is executed synchronously and all other routines are asynchronous, synchronization with chords is less expressive but more visible than that with "wait" and "notify". However, even a component written in Polyphonic C# can have synchronization not visible in the component interfaces. The interfaces show only synchronization that happens when invoking component routines from outside, but not that occurring when invoking them from inside. Such languages help to avoid the worst cases. However, they cannot solve the problem.

We can argue that it is absolutely necessary to specify the complete synchronization behavior of a component in its interface (thereby turning a black box into a nearly white one) because users need this information. Once users rely on such information we cannot easily change the communication structure anyway. Perhaps there is much truth in this argumentation if we consider arbitrarily complicated communication patterns. Fortunately, current software engineering practice shows that in many situations users have all needed synchronization information without turning black into white boxes. It is an open question whether we have to show all synchronization behavior in the remaining cases. Of course, we want to avoid that if possible. In the next section we deal with required synchronization as a way to specify partial synchronization information (as opposed to complete synchronization information) in an interface. It is our goal to keep as much information as possible hidden in the black box.

III. Required Synchronization

Under required synchronization we understand the synchronization that a component requires from the user when invoking routines.[1] For example, many components require from a user to invoke an initializing routine before any other routine. This is clearly a kind of synchronization although we need no synchronization primitive like "synchronized", "wait", and "notify" to ensure it.

Users can provide the required synchronization only if they know exactly in which ordering routines are invokable. The synchronization depends only on data available to both the component and the user. Thus, it is much easier to specify required synchronization in interfaces than other kinds of synchronization. Required synchronization is specified in the interface and usually only there. Such properties cannot be derived from the implementation (since there are no explicit synchronization primitives), and the component is always a black box in this regard.

[1] Required synchronization has nothing to do with require interfaces of components. There is just an unintended similar naming. In this paper, a require interface does not differ from any other interface and, hence, can also specify required synchronization. Required synchronization in a require interface specifies synchronization to be ensured by the component itself because the component acts as client of the unit referred to by the require interface.

Required synchronization can be very expressive. We can specify arbitrary sequential orders of routine invocations, alternatives (where always the user selects the alternative to be taken), and of course arbitrary interleavings thereof. This means, we can express each possible prefix-closed set of routine invocation sequences (or trace set). The expressiveness need not depend on the language we use in the interface – a plain natural language or any formal language able to express trace sets. Substitutability of required synchronization is also easy and well-defined: A subtype must essentially support all routine invocations in all orderings supported by the supertype, this is, the trace set corresponding to the subtype includes that corresponding to the supertype (just set inclusion).

The difficulty is rather at the side of the user who must ensure that routines are invoked only in an ordering specified in the interface. At a first glance this seems to be easy since the user has all needed information. However, in most cases there is not just one component with a single well-defined user. In systems composed from components, each component can provide services to and use services from several other components. If several users interact with the same component, the users must coordinate themselves to invoke routines in a required linear ordering. There are several ways to ensure linearity:

- The simplest solution is to require only a single user whenever there are constraints on the ordering of routine invocations. In simple cases (like initializing routines to be invoked before others) this solution is absolutely appropriate. It is easily enforceable and expressible in a natural language.
- In the case of several users we can ensure that at each point in time only a single user interacts with the component. This solution requires more effort on coordinating users, but it is doable. However, without tool support and without a clear concept how to do it the approach is error prone. An obvious disadvantage is the strict linearity (this is, a single user at a time) even for independent routine invocations.
- To improve the approach we add support for splitting a single linear thread into several independent linear threads and combining them again when threads become dependent. In this context a thread can be an independent instance of any mechanism to ensure linearity, and it can also be a concurrent thread of control in concurrent programs. This improved approach is the most flexible one, but it is difficult and error prone without proper tool support.

Tool support has been proposed for single as well as multiple linear threads [1], [4], [5], [6], [7], [8], [9], [10]. However, such support is currently not widely available. At the moment it is only possible to simulate the corresponding techniques by writing annotations in natural language into the users' code and check them by hand.

Although the most advanced approaches are possibly not yet mature, required synchronization is in many cases preferable

over more complex synchronization using synchronization primitives in components (except of mutual exclusion and a handful simple synchronization patterns). In a large majority of cases we need to ensure linearity only in small scale (which is doable by hand).

As a simple example let us consider a window on a screen that is shown either in full or as a small icon. The window component supports two methods (`iconify` acceptable only when the window is shown in full, and `uniconify` acceptable only when being an icon) changing the state of the window. To have a further level of complexity we assume the methods to take parameters and return results so that we cannot simply ignore invocations that occur when the window is in the wrong state. Synchronizing executions of such methods in the window component is not simple because of the state dependence. Users need detailed information about the synchronization and the window's state to avoid undesirable program behavior like no immediate reaction and some undesirable delayed reaction when pressing the "iconify" button. This information must be expressed in the interface. If we require the synchronization to be provided by users (this is, users have to ensure that method invocations occur only in appropriate window states), then we need no synchronization at all in the component, and it is probably easy for the users to ensure proper synchronization: We have to ensure linearity (this is, always only one user can invoke `iconify` or `uniconify`) and let the corresponding user know the window's (initial) state. It is quite natural to provide this required synchronization simply by having only a single "iconify" button while the window is shown in full and a single "uniconify" button while the window is an icon; we need no synchronization primitives at all to get proper synchronization.

Systems based on required synchronization

- avoid the grey-box view of components,
- clearly regulate responsibilities for synchronization,
- usually do not suffer from deadlocks,
- and are often simpler and promise higher performance.

In the author's opinion, few safe synchronization patterns together with required synchronization are almost always sufficient to achieve the desired synchronization behavior. There is no need to regard a component as a grey box for synchronization. The desire to look inside a component to determine its synchronization behavior is often just a sign of bad component design.

IV. How to Avoid the Grey Area

Availability of source code is independent of regarding a component as grey box. For example, offering customers a look at the code (or supplying open source code) can be an effective way of establishing trust into software. Code inspection need not cause a black box to become grey. The code can tells us something about the developers, company, or product line, and it is a back-up in case the software is no longer supported. However, modifications adapting code to own needs or just making use of implementation details decouples the software from further developments and is undesirable.

Sometimes we regard a component as a grey box even without looking at its code. This happens, for example, if we measure the performance (or any other non-functional property) of a system composed of components. Even if the performance is satisfactory we cannot rely on it after substituting a component with a new one. Although there exist approaches to specify non-functional properties we are currently far away from being able to specify them as part of a practical software contract. Of course, it is always possible to give some kind of guarantee based on average usages. However, it remains an open question if the one application we are concerned with fits into the class of average usages.

In this scenario we can apply a similar pattern as we did in Section II (based on the pragmatic support of usual cases): The description of the component clearly explains what the component is supposed to be used for and how to use it. Most users who correspond to this rather narrow description will most likely experience no dramatic performance decrease with future releases; they fit into the class of average usages. However, each other user is let alone. Hence, we have a pragmatic solution for the usual cases and an unpleasant position in other cases. We can improve the situation by a deep analysis of the problem and the development of appropriate tools, especially tools to improve the communication and to shift responsibility from components to their users (similarly as we did in Section III). As a simple example, we can allow users to influence the performance of components by setting parameters or providing specialized routines.

Probably this is a general pattern for a large number of aspects:

- We are able to describe the aspect in an informal interface in a very pragmatic way. The notion of pragmatic description means that component developers and users have some common understanding of the aspect and the usual solution space for corresponding problems; the description mentions only which solutions have been taken (if there are several alternatives) as well as deviations from the usual solutions. Interface specifications can be very brief (or even non-existing) especially in usual cases. Quite often such specifications are abstract in the sense that they only refer to some notion and give no details at all. For example, an interface specifies that a component behaves as a "buffer" without clarifying what this notion means. Although we can imagine many different kinds of buffers this simple word often gives us enough information to correctly use the component. Furthermore, an interface often explicitly specifies a component to be usable as replacement of another component without giving details. Such specifications relate abstractions based on common understanding. To have a common understanding is the basic idea behind the "simulation of the real world" which is a key concept in object-oriented programming. When dealing with involved aspects like synchronization we need a common understanding even

if there is no direct counterpart in the real world.

- The pragmatic approach has its limits. It works fine as long as all taken solutions are close to the usual solutions (this means that there is a common understanding) and the users need no information about other aspects than usually considered in the component description. However, it breaks down in unusual cases. For example, the pragmatic approach fails where we need static deadlock prevention because this aspect is usually not considered in the description.
- Tools and formal techniques (based on a deep problem analysis) can help in cases where the pragmatic approach fails. For example, we can apply a specific type checker to ensure deadlock-free programs. Such tools and techniques are very specific to aspects they consider and require a deep understanding of the aspects, and their use often costs much time and/or severely restricts the flexibility of the solutions. Therefore, they are often not widely used. They will be used only if the limits of the pragmatic approach begin to hurt.
- There is a common pattern behind many of these tools and techniques: Primarily they describe components in greater detail so that users get more information through interfaces. For example, type checkers to ensure deadlock-free programs impose specific program structures and make them visible in the interface. However, because of needed substitutability it is important to keep implementation details hidden. For this reason it is often no good idea to specify the complete synchronization behavior in an interface.
- To avoid unnecessary exposion of implementation details we take an additional way to strengthen connections between components and users: We move responsibilities from components to users, for example, by introducing required synchronization as we did in Section III. Then, users can take advantage of all knowledge about the environment they have. It is likely that these responsibilities rather belong to components than to their users in the usual understanding; otherwise we would not have considered them to be part of the components. Hence, the tools and techniques are mainly concerned with the support of responsibility for aspects at places where they do not naturally belong to without these tools and techniques. They give us more freedom in moving things around. They also allow us to split (otherwise not divisible) responsibility so that we can deal with each sub-aspect where we have the needed information. For example, techniques mentioned in Section III handle aliases and ensure linearity to allow us to move synchronization between components and to divide responsibility for it between all users of a component.

Historically, many developments in object-oriented and component-based programming had one common goal: Support programmers in arranging (or moving) code and data such that it becomes easy to add new parts and replace existing parts without any need to change unrelated parts. The application of tools and techniques to arrange or move further aspects (besides code and data) seems to be the natural next step. Aspect-oriented programming (as often advertised today) can probably not achieve this goal because it is not specific enough to the particularities of certain aspects. We will have to put effort into each aspect separately. Much hard work remains to be done in this area. Unfortunately, most results of such work will only be used in small scale because they are especially useful in exceptional cases where pragmatic solutions fail. Nonetheless, such work is important. Today object-oriented and component-based programming are already well-established. Progress will be made by many small improvements, not by radically new ideas as was the case years ago. Sometimes new techniques will become standard and part of the pragmatic solution.

V. Conclusions

The black-box view of components is practically important and we shall avoid to give it up even if it causes troubles. In most cases it is possible to develop components based on a common understanding of how to handle certain aspects. Interface specifications in a natural language are all we need in these cases to keep the black-box view. In more unusual situations it often helps to move responsibility for certain aspects to other components and/or to divide responsibility to several components. However, depending on the considered aspects, today we have only few tools and techniques that support us in doing so. That is an open area of research in the tradition of object-oriented and component-based programming.

References

[1] Farhad Arbab. Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming*, 55(1–3):3–52, 2005.

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):269–804, September 2004.

[3] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[4] Naoki Kobayashi, Benjamin Pierce, and David Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[5] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3):210–237, August 2004.

[6] Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In *Proceedings FMOODS'97*, Canterbury, United Kingdom, July 1997. Chapman & Hall.

[7] Franz Puntigam. Type specifications with processes. In *Proceedings FORTE'95*, Montreal, Canada, October 1995. IFIP WG 6.1, Chapman & Hall.

[8] Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, number 1241 in Lecture Notes in Computer Science, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.

[9] Franz Puntigam. *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany, 2000.

[10] Franz Puntigam. Internal and external token-based synchronization in object-oriented languages. In *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, number 4228 in Lecture Notes in Computer Science, pages 251–270, Oxford, UK, September 2006. Springer-Verlag.

Components have no Interfaces!

Richard Rhinelander

Department of Computing Science
University of Kitara
Australia
Email: rr@kit.edu

Abstract—[12] **While the discussion on "what is a component" seems to finally come to a (wrong) conclusion, this paper argues heavily against one of the most dramatic misconceptions of this still immature field, i.e., that software components are specified by their interfaces. We explain what problems result from this misconception and we discuss requirements for a component model which avoids the problems discussed. Most interestingly to the rest of the community, such a component model is not an interface model, but much more a model where components are *not* defined by their interfaces.**

a) keyword (selected): predictable assembly of components, performance/efficiency and reliability of component-based systems, systems for the description and prediction of non-functional component properties, deployment attribution / constraints, COP and Model-driven Development (MDA), role of composition frameworks, interoperation among component frameworks, dynamic composition of component-based systems, component-oriented development processes, relating architectural principles/approaches to component software, architecture description languages suitable to guide COP, addressing variability requirements in component-based solutions, system design for independent extensibility, system design for the use of third-party components, component versus application evolution, components in distributed embedded systems, incl. mobile phones and PDAs, domain-specific (vertical) standards, organizational aspects, business aspects, what worked / what didn't work in practice and lessons learned, plus some other keywords, not mentioned here explicitly, but including SOA, metamodelling, UML and various other politically incorrect terms.

I. Introduction

One of the major driving forces and motivations of the whole field of component based software engineering[3] is the fundamental question of "what is a component". The discussion was certainly made "interesting" by views ranging from "anything but reusable" [HC91] to a principally undefinable entity [CE00]. Particularly damaging is the list of component properties from Clemens Szyperski (most often cited as a definition, hence we do not cite it here) that a component "is a unit of third party deployment, has contractually specified interfaces and that is has no state". For teachers, in particular the latter property is still a pain in the neck. However, for the author (of this position statement, not the property list) the second property shows the irresponsibility of the author (of the list, not this paper) much more clearly. Not alone with letting us alone what "contractually specified" means (a term that was most often used inconsistently and wrongly in the community, until a hallmarking paper [RS02] (by one of the most overrated researchers in the community)); much worse (back again with the other irresponsible author (the one of the property list, not the hallmarking paper), the whole idea that a component has interfaces (what so ever specified) is totally wrong and results in various misconceptions. One of the more severe problems caused was a review comment on a paper of the author (of this position paper, but a review comment was on another paper), that rejected the idea of a novel component model because the reviewer insisted that a component model is in fact the same as an interface model. Beyond this dramatic mislead of at least one reviewer, there are various lighter problems of less concern (in fact, all relating to topics listed in the Call for Papers).

However, as (of course totally by accident) the organisers of this year's WCOP explicitly asked for position statements on how dark is the component black-box, the author clearly sees a feeling of guilt in the workshop organisers attitude and is willing to forgive and to share his more fundamental insights in the true nature of components. (Attention to reviewers:) The contribution of this paper is the clarification of the relation between components and interfaces, in particular interfaces mistakenly pretending to specify component-QoS-properties plus a sketch of a solution to save the world ("world" in the broader meaning of the word: "universe"). The remainder of the paper is organised as follows: in section II, interfaces and components, we (unexpectedly) discuss the terms interfaces and component.[4] In section III we conclude that components have no interfaces and conclude. Related work (of course of a much lesser depth of thought as the author's work (the author of the this paper, not the ones of the related work)) is presented in section IV. Section V than again states the main message of

[1] Author names not in alphabetical order.

[2] The German Research Council would not be happy to hear that this work is supported by it. Therefore, various grant references are suppressed.

[3] In fact formerly, component phrased software engineering, as the majority of self-declared component experts simply declared objects or classes as components and re-submitted their old concepts. However, we now beat back in declaring components as services.

[4] We intentionally use interface in its plural form while using the term component in its singular, as one component can have several interfaces. (oh shit. In fact, they don't.)

this position paper, but now in a form that you can cut of of the paper and wear it as a sticker (which less humble characters may do to demonstrate their intellectual superiority).

II. Interface(s) and Component(s)

Message number one is: An interface is an abstraction of an encapsulated software unit. In case of a procedure (as software unit) the signature would be the interface, but that is a little pathologic, as interfaces are mostly known as an abstraction of a module (as a software unit), as introduced by Parnas 1972 [Par72]. Argued in an exemplary way, modules defined to encapsulate design decisions. Interfaces are used as the sole way to access modules. The idea is, that the module implementation can change (if the design decision is changed to use an alternative implementation), but the interface is stable. Therefore, it is clear that the interface must be an abstraction of the module. If it were not, than any change of the module would also affect the interface. Although this sounds simple, Parnas also stated that an interface should contain sufficient information to use the module and to implement the module. Interestingly, this is in direct contradiction to the "interface is an abstraction"-idea, as an abstraction intentionally omits information and, hence, may not for all cases provide sufficient information for usage or implementation. However, in any case, the interface is used to describe a module. As a module is always supposed to be in a fixed environment (at least if it is seen as a way to encapsulate design decisions and not as a means for software reuse), one has not to worry whether the module really provides the functionality promised in the interface. However, things are different for components (which are delivered without context and can be deployed in different contexts). Therefore, we continue with:

Message number two is: Interfaces are first class entities. They can exist without any component. (For example, this is the case in domain standards, where interfaces are defined and the semantics of the service [5] signatures listed in the interfaces is described.) As a consequence, between two first class entities (namely components and interfaces) there can be several relationships:

implements:
: that is what we know from Java. The meaning of "A implements B" is that a class (err, component) A contains the code of the service signatures of the *implemented* interface B.

depends:
: that is also easy. The meaning of A requires B means that component A contains code, that depends on external services as specified in interface B.

provided:
: totally meaningless for isolated components: what functionality a component actually provides depends not only on the component itself, but also on several factors, in particular whether external components which the component depends on are connected and which properties they have. Hence, the provided relationship between a component and an interface can only occur in architectural diagrams where a component is put into its context (which includes the wiring to other components).

required:
: totally meaningless for isolated components: what functionality a component actually requires depends on the functionality one component is used for in a specific context. Without knowing what functionality of a component is actually called, one cannot specify which functionality is really required. Like above, the required-relationship between a component and an interface only occurs in architectural diagrams where a component is put into its context.

The message number three is: What information is actually specified in an interface, is given by the interface model (actually interface meta model). The even more important massage is: an interface model is not a component model, as, trivially, an interface is not a component and both are independent first-class entities.[6] Various interface model classifications have been published, one of the most often cited is the one from Beugnard et al [BJPW99]. In this classification the authors interestingly omit the dimension of classification and just list classes (of probably increasing complexity): "Syntactic level, behavioural level, synchronisation level, QoS level". In fact, it is clear why the classification dimension is omitted, because in fact, it were two dimensions intermixed. Therefore, in a brilliant work[7], Becker et al presented at CBSE 04 a landmarking paper about an interface model classification with *two* dimensions, namely, the level of granularity you are specifying properties for (single services of an interface or the interface itself) and whether you talk on functional or extra-functional properties.[8] While moving from functional signatures to functional protocols (as a behaviour specification you stay constant in the functional-extra-functional axis, but move on in granularity. Differently, when going from behaviour to QoS specifications. Here you move back in granularity but more to extra-functional properties. The basic message however, is, what is specified in an interface is not fixed. Although mist time it are service signatures, it was firstly argued by Nierstrasz in 1993 that a list of service descriptions is insufficient and one needs in addition a specification of valid call sequences [Nie93].[9] In a different dimension you

[5]The use of the term "service" instead of "method" or even "procedure / function" is a concession to the zeitgeist.

[6]Listen! Listen carefully, thou reviewer of a recent conference who rejected one of my papers because this difference was not clear to you. Dopey!

[7]Probably not as brilliant as the paper which got rejected, as the reviewers where still able to grasp the concepts of the paper.

[8]In case you are always confused with a clear borderline between functional and extra-functional properties: functional properties are all those you can specify with a Turing machine, those you cannot, are extra-functional. However, this is an academic statement: I got to know that in consulting and industrial practice Turing machines are sometimes omitted in functional specifications.

[9]Yes, it was Nierstrasz, not TomH in 2000. In between even the author argued for specifying component protocols, at least one year before some (suitably chosen) others.

can extent interfaces by specifying the quality of services. In practice, one is mainly interested in performance metrics (such as response time or throughput) or reliability metrics (mean-time-to-failure or availability). To summarise: although not used in Java, there are interfaces which describe not only the signature of services but also some of their quality properties, and this is perfectly ok with the interface concept. (In particular, there is no need to introduce new terms, like "gates".)

III. Components and Interfaces

Unlike modules, components are to be used in different contexts. Therefore, they have to make dependencies to the context explicit. Naively, one would think that a component uses one or several interfaces to specify what functionality they expect from the environment (to be implemented). While this is naive even for purely functional interfaces, the whole tragedy comes most clear when considering QoS-specifying interfaces. What QoS of a required service our component should ask for? For example, assume that our QoS-specifying interfaces allow to specify response time. What response time a component C should ask for a specific external service e for? The answer is simple: this depends on what the component's C services (who are using e) promise as a response time to callers. However, this is not the right answer, because what the response time of C's services is also not clear. In fact, it depends from the C callers, hence it is context dependent. Altogether: as long as we do not know, which QoS is expected from C, we cannot specify the "required" interfaces of C. With the same line of argumentation, we also cannot specify what QoS C will provide, as long as we do not know, which QoS will be provided by the required external service e to our component C. Although not such seductively clear, the argumentation holds for functional properties: should our component C always ask for all external services which are referenced in its code? Of course, it could (and has to be (type-) safe), but that would restrict reusability in all cases, where actually not all of C's implemented functionality is used by callers. In practice, this is often the case and it exactly should be the case. As it should be cheaper to reuse a component which implemented actually more functionality than I want to use in a specific context is perfectly ok. And in fact, if such a reuse is made unnecessarily expensive by formally required external services which in fact are never used, the whole component reuse idea gets questionable.

From the above argumentation, we can derive the following two requirements for component models:

A distinction between unbound components and components which are wired into a context. As discussed in the introduction of this paper, the central question of the whole research field of component orientation centers around the question of what is a component.Much confusion exist, as it is usually not differentiated for which level of abstraction the component is defined for. In fact, a clear distinction of component abstraction levels is lacking. From the above discussion on the relation between interfaces and components, we learned that at least two different levels of components exist. The first level is concerned with unbound components which can implement on interfaces or depend on interfaces. The second level are architecturally bound components, which provide and require interfaces. You can refine these two layers of abstraction even further, by distinguishing between the following layers:

Provided component type:
: A component in a coarse grained architecture: you just describe that a component exists which will implement some functionality. As in this level of coarse grained architectural design you are mainly interested in decomposing a system (i.e., identifying components) you should not forced to specify interfaces the component depends on. Hence, implementations of this type can depend on interfaces which are not specified in the provided type.

Complete component type:
: This component already includes a specification of the interfaces it implements and those it depends on. An implementation of this type must not depend on more interfaces as specified in the complete type.

Implementation component type:
: while the words "implementation" and "type" sound contradicting, the point is that this type contains additional information which may relate very strongly to its implementation. However, it is still a specification of an unbound component which abstracts from the actual implementation. Which information is specified in this in particular is discussed in the next subsection.

Deployed component:
: This component is deployed. This means it is linked to its resources and the interfaces it references to are itself connected to external components.

Run-time component:
: A lump of code in the memory, we do not discuss any further.

The next section is unnecessary and therefore clearly identifies itself of having as a sole reason of existence the addressing a concern of a hasty reviewer.

Note that although a component has no interfaces, it can implement interfaces and can also depend on interfaces. The point is, that a until deployment time it is rather unclear whether all interfaces implemented are actually provided and, likewise, whether all interfaces the component implementation depends on are actually required. The "provides"-relation and "requires"-relation between a component and an interface are deployment-context specific (including the usage-profile), while the "implements"-relation and "depends-on"-relation between a component and an interface are component implementation specific.

A different way for specifying unbound components than interfaces. As the discussion shows, one has to specify the

relation between what a components gets from the environment (as specified by the required interfaces the component is bound to at deployment) and what the component provides (as specified by by the provided interfaces the component is bound to at deployment). Now three different scenarios exist:

1) A component is bound to required interfaces, hence it has to be computed what the component in this context can actually provide (given the required interfaces, what would be ”maximal” provided interfaces, i.e., the interfaces which describe all functionality at the best QoS the component can provide with the required interfaces given.)
2) A component is bound to provided interfaces, hence it has to be computed what the component in this context hast to actually ask for (given the provided interfaces, what would be ”minimal” required interfaces, i.e., the interfaces which describe the least functionality at the worst QoS the component has to ask for to provide the functionality specified by the provided interfaces given.)
3) A mixed case: some provided and some required interfaces are bound. Now the question is what is a suitable combination of provided and required interfaces for the unbound ports, if such a combination exists at all.

Note that in any case, one can ask all questions also for protocol specifying interfaces. (The answer for questions one and two for protocol specifying interfaces was even so academic (not to say: confusing) that it was rewarded with a dissertation in a until than well-regarded south-western German university. [Reu01])

IV. Related Work of lesser depth of thought

Omitted to to space restrictions of the paper and time restrictions of the author. Although the paper is accepted and some additional pages were granted according to its outstanding importance, the author did still not consider writing this section.

V. Conclusions

Cut of here and place it in your name badge of the conference. Wearing it during the conference dinner is supported by the SUSIMSSRUC (Society to use superior innovative marketing to support scientific results of unclear character), pronounced (soozie-hmmm-ruk.)

Lessons learned:

---

1) Components have no interfaces. This is true for any unbound component, and particularly obvious for QoS-specifying interfaces.
2) It is ok to say, that a component implements an interface, if the interface is not specifying any QoS-properties.
3) It is ok to say, that a bound component provides and interface, even if the interface is specifying QoS-properties.
4) It is not ok to say that a component requires an interface, unless the component is bound to its calling component(s).
5) It is not ok to forget to cite parametric contracts.
6) A component model is not an interface model.

Acknowledgements

As this paper did not went through the regular review process of the research group, the reputation of the other members is not affected by this publication. However, they also lack the credit.

References

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[HC91] J. W. Hooper and R. O. Chester. *Software Reuse – Guidelines and Methods*. Plenum Press, New York, NY, USA, 1991.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, 1993.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Reu01] Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, 2002.