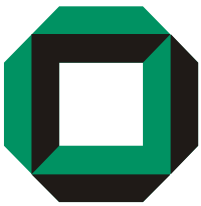
Universität Karlsruhe – Fakultät für Informatik – Bibliothek – Postfach 6980 – 76128 Karlsruhe

Advances in Component-Oriented Programming
–
Proceedings of the 11th International Workshop on Component Oriented Programming (WCOP 2006), July, 3rd, 2006, Nantes, France

Herausgeber:
Ralf Reussner, Clemens Szyperski,
Wolfgang Weck

Interner Bericht 2006-11



Universität Karlsruhe
Fakultät für Informatik

ISSN 1432 - 7864

Preface

WCOP 2006 is the eleventh event in a series of highly successful workshops, which took place in conjunction with every ECOOP since 1996.
Component oriented programming (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, and .NET, but also the increasing appreciation of software architecture for component-based systems, 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 2006 emphasizes reasons for using components beyond reuse. While considering software components as a technical means to increase software reuse, other reasons for investing into component technology tend to be overseen. For example, components play an important role in frameworks and product-lines to enable configurability (even if no component is reused). Another role of components beyond reuse is to increase the predictability of the properties of a system. The use of components as contractually specified building blocks restricts the degrees of freedom during software development compared to classic line-by-line programming. This restriction is beneficial for the predictability of system properties. For an engineering approach to software design, it is important to understand the implications of design decisions on a system's properties. Therefore, approaches to evaluate and predict properties of systems by analyzing its components and its architecture are of high interest.
To strengthen the relation between architectural descriptions of systems and components, a comprehensible mapping to component-oriented middleware platforms is important. Model-driven development with its use of generators can provide a suitable link between architectural views and technical component execution platforms.
WCOP 2006 accepted 13 papers, which are organised according to the program below. The organisers are looking forward to an inspiring and thought provoking workshop. The organisers thank Jens Happe and Michael Kuperberg 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

Program of WCOP 2006

Session I: Aspects and COP

Session II: Component adaptation

Session III: Component Composition and Deployment

On the Benefits of using Aspect Technology in Component-Oriented Architectures

Maarten Bynens, Wouter Joosen

DistriNet, KULeuven
Dept. of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium
maarten.bynens@cs.kuleuven.be

***Abstract*— Aspect-oriented programming (AOP) has been explored to support the development of complex software systems that expose many interdependencies (a.k.a. crosscutting concerns). In maturing AOP the research community has been investigating how concepts from AOP can be enhanced to offer the benefits of CBSE. In this position paper, we investigate an alternative approach to the combination of AOSD (AO Software Development) and CBSE, by identifying the essential (yet minimal) differences that AOSD could add to the core concepts of CBSE. We compare two alternatives and we show how aspect-based composition can enhance a basic component model to support sophisticated compositions that create, express and manage complex interdependencies.**

I. Introduction

Recently, there has been a large interest in the principles of aspect-oriented software development (AOSD, [4]). These principles focus on the systematic identification, modularization, representation and composition of (often crosscutting) concerns or requirements throughout the entire software development process. The core concepts of AOSD are *concerns*, *aspects* and *weaving*.

Concerns are similar to requirements in a broad sense of the word, ranging from high-level requirements that are articulated in an early stage of the software project, to additional - often detailed - requirements that are generated when performing detailed design and implementation. In that sense a concern often corresponds with a feature, capability or quality-of-service level that is important for a stakeholder in the software project. At the programming level, an aspect is a modular unit that implements such a concern. An aspect definition contains (a) behavior (code that must be executed) which is called *advice* and (b) a specification that expresses when, where and how to invoke the advice; this specification is called a *pointcut*. A pointcut is conceptually defined as a predicate that evaluates over *join points*. A join point is a well-defined place in the structure or execution flow of a program where additional behavior can be attached. Finally, *weaving* is the process of composing core functionality modules (typically application components) with aspects, thereby yielding a working system.

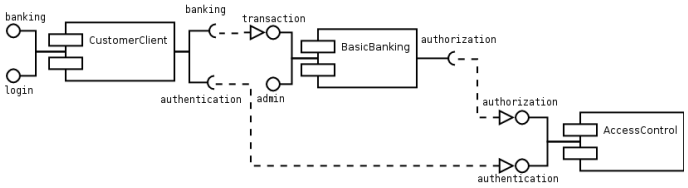


Fig. 1. Small part of a basic banking system.

Introducing the concepts of component-based software development (CBSD) to aspect-oriented programming (AOP) is one of the key challenges to make the use of aspects worthwhile and has received much attention ([9], [6]). Although this type of extension is an important research track, it may not focus on preserving the key concepts of CBSE, but rather on enhancing AOSD with a CBSE flavour.

However, in the opposing direction, introducing the concepts of AOSD in component models, has received less attention, maybe because the CBSE community expects no benefits from the AO paradigm, or maybe because the integration efforts so far have shown complicated and confusing. In this paper we will give an overview of how the principles of AOSD can be introduced to a component model and what the benefits will be of doing so. In the next section we present a basic component model and the integration of aspects into this model. Section 3 will evaluate the benefits of using the aspect mechanisms in the component model with respect to both reconfigurability and reusability. Finally, section 4 will summarize our main points.

II. Incorporating aspects in a component model

Based on the definition of a component [10], we can consider a basic component model as consisting of components, interfaces and connectors. A component exports interfaces, describing either provided or required functionality, and connectors serve as ”glue” to bind components with matching interfaces.

Consider for example a basic banking system, a small part of which is presented in figure 1. It consists of three components. The *CustomerClient* component offers a user interface to the customer of the bank for logging in and doing

transactions on his accounts. This component requires banking and authentication functionality from the environment. The *BasicBanking* component offers the basic operations for managing accounts and for doing transactions on those accounts. Because most of the banking operations need to be authorized, it needs some access control logic from the environment. The last component is the *AccessControl* component that offers both authentication and authorization functionality. Each component exports some provided and required interfaces and matching interfaces are bound with a connector (in the figures a connector is represented with a dashed arrow).

There is no single accepted definition of an aspect model. The most widespread model is that of AspectJ[5], which extends traditional classes with pointcuts and advice, using the keyword *aspect*[1]. Looking closer into this aspect model, we see that the real contribution is in the *pointcut language* and in the *aspect environment*[2] that composes the aspect behavior (advice) at the locations (join points) specified by the pointcuts. The advice itself, on the other hand, does not necessarily need a new mechanism, it corresponds to a method that is executed before, after or around a set of join points. Around advice is actually executed instead of the service it is bound to, but is able to call the original service[3].

So, concretely, when mapping the concepts of AOP to the component model, the concepts that must be considered are join points, the pointcut language, advice, aspects and the weaving mechanism. The aspects themselves are important as the module that contains the pointcuts and the binding with the crosscutting functionality.

In the component model, the composition and all interaction of the components goes through the interfaces. Extending this idea to aspect composition means that the only available join points are the entries on the interfaces.

Concerning aspects and the weaving mechanism, the aspect model can be incorporated into the component model in more than one way. In this paper we will now describe two of those. The first option is to consider aspects as a new kind of component, much like AspectJ was designed, the other option is to consider aspects as a new kind of connector. With the former option, the component will absorb the full behavior of the aspect and will be called an aspect component. With the latter option, most of the aspectual behavior will be absorbed by the connector and will be called an aspect connector.

An alternative mechanism for dealing with crosscutting behavior in component-oriented architectures, is the use of containers, like for example in the Enterprise JavaBeans technology [7]. Each component is deployed in a container,

[1]AspectJ also introduces inter-type declarations, which are however less representative for AOP. Also, the contribution of inter-type declarations to the component model is rather limited.

[2]A run-time environment is the most flexible, but also compile-time and load-time environments exist.

[3]In AspectJ, this is done using the keyword `proceed()`

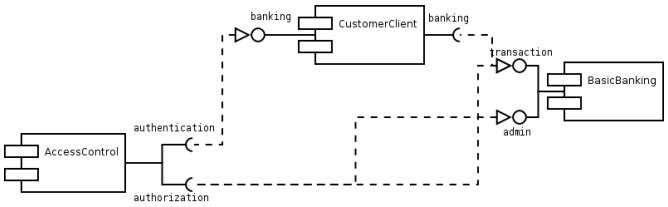


Fig. 2. The basic banking example using aspects as components

which means that for each component, a deployment descriptor needs to be defined. In return the container provides some functionality that would otherwise crosscut the components in the container. Typical functionality that is provided by a container is persistence, transactions, security and lifecycle operations. Although this is a practical way to encapsulate some frequently needed crosscutting functionality, the services provided by a container are fixed and cannot be extended by the component developer.

Since we are looking for mechanisms to encapsulate any crosscutting functionality, containers are not considered here and we will focus on the two approaches to incorporate an aspect model into a component model, sketched above.

A. Incorporating aspects as components

Besides providing the functionality as specified by the provided interfaces, based on the functionality on the required interfaces, an aspect component includes crosscutting behavior that should be exported in terms of provided and required interfaces. Because a pointcut is a predicate over join points and a join point is an entry on the interface of a component, a pointcut could be exported as a required interface of the aspect component. The semantics of such a required interface are that the aspect component will add crosscutting behavior to all the components connected to this required interface. The advice included in the aspect component may or may not be exported as normal functionality on a provided interface.

Going back to the example, the *AccessControl* component could be available as an aspect component (see figure 2). It will weave functionality at the join points as specified by its required interfaces, which are bound to the relevant interfaces. In this case, authentication functionality will be woven in when a customer uses the client and authorization will be woven in before all banking operations.

B. Incorporating aspects as connectors

Given that advice is nothing more than a normal method woven in at the right join points, it is a valid approach to put some of the aspectual behavior in the connectors instead of in the components themselves. The definition of the pointcuts and the binding of those pointcuts with the advice will be specified in a connector, while the advice functionality is available as a normal service on the interface of a component. Besides normal connectors, binding matching provided and required interfaces, we then get aspect connectors, which are able to bind some provided functionality before, after or around a set of join points.

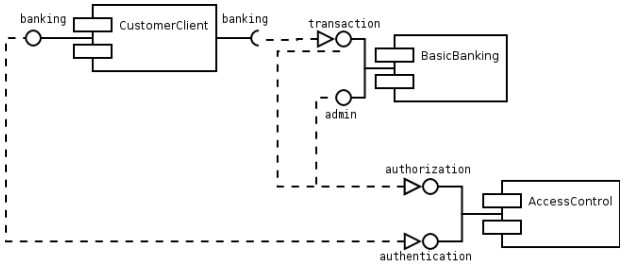


Fig. 3. The basic banking example using aspects as connectors. The services of *AccessControl* are all bound as before advice.

In the basic banking example, we see (figure 3) that the *AccessControl* component didn't change in comparison to figure 1, because all the aspectual behavior is now in the connectors. All the aspect connectors in this example (these are the arrows connected to the *AccessControl* component) will bind the advice before the pointcut, for example authorization must occur before the banking operation.

Incorporating aspects like this shows some resemblances with the composition filters programming model[1]. Composition filters intercept incoming and outgoing messages of an object and are able to a.o. redirect, substitute or dispatch them. In the extended composition filters model[3] it is possible to superimpose filters on other objects. Although these filters could be used to implement the aspect connectors described above, there is no real integration between composition filters and a component model, because it is mainly focused on a programming level.

Incorporating aspects as connectors offers more flexibility than incorporating aspects as components. All components can be developed as usual, the developer doesn't have to choose between developing a normal component or an aspect component. Only at composition time, it will be determined whether a component is used as an aspect component or not. This will make the component more reusable because it hasn't been decided whether the functionality offered by the component is crosscutting or not. The aspect connectors can be considered as some kind of composition descriptor to configure the components to interact with each other.In the following section we will take this approach as the basis for examining the benefits of using aspect technology in a component model.

One last note: because components can contain advice as normal services, this is also true for around advice. Around advice can call the original service around which it is composed, but because it is modelled as a normal service, it is not necessarily used as around advice. What happens when this service is bound using a normal connector? A possible solution is to specify around advice that needs the original service in both a provided and a required interface. To use this service as around advice the same way as it would be used in AspectJ, the provided service is bound around some base service with an aspect connector and the required interface is bound to this same base service using a normal connector.

III. Mastering component dependencies with AO composition

One of the advantages of using components instead of traditional classes, is that dependencies between components are made explicit through the use of required interfaces. The remaining dependency a component has with external functionality is twofold. On the one hand, the semantics of the external functionality needs to be specified in a required interface. On the other hand, the developer of the component needs to know the exact location or execution point of where external functionality needs to be attached. The join point concept of AOP has been created to enable the specification of such an execution point.

This was already a significant improvement over traditional classes, where a class that uses external functionality needs to know exactly the correct method to call with the correct parameters on the correct instance of the correct class. Using aspect-based composition can give the developer even more control over these dependencies. Although making the dependencies of a component explicit by exporting a required interface is a good thing, it may be a burden to the developer to specify the complete semantics of this interface. Also, the exact location of the use of the external functionality doesn't have to be specified by the client component if this location can be expressed using the join points on the interfaces.

The use of Open Modules[2] is a very promising approach to make the interface of a component more suitable for aspectual composition. Open Modules make it possible to add arbitrary pointcuts to the interface of a component. It is a new module system that is intended to be open to extension with advice but modular in that the implementation details of a module are hidden.

Looking at the example, we see that the banking components no longer have specified the need for authentication or authorization. The need for this functionality and the composition of this functionality with the banking components all happen at composition time, making the banking components more reusable.

IV. Increasing component configurability with AO composition

A component can be considered as a mapping from required functionality to provided functionality. The (re)configuration of a component is the adaptation of this mapping to what functionality is available or needed in the environment.

The more (re)configurable a component, the more reusable it will be. This comes from the fact that if the environment does not need all the provided functionality or cannot deliver all required functionality of a certain component, it cannot be reused unless it can be configured to fit in the current context. Approaches to deal with the (re)configuration have

been proposed before[8], but AOSD can also resolve this matter.

Using aspectual composition it is not necessary to make the banking components from the example, configurable with respect to the need for access control or not. With aspectual composition, optional functionality can be encapsulated in a separate component, that may or may not be composed into the application.

V. Summary

In this position paper, we have studied an approach to enhance CBSE with AO concepts, by identifying the essential (yet minimal) differences that AOSD could add. We have compare two alternatives: (1) the use of aspect components that become first class components and combine behaviour and composition semantics; and (2) the use of aspect connectors that focus on composition semantics only. We have argued that the latter is more suitable to create the minimal but essential extension of a component model (i.e. rich connectors) that integrates the key ingredients of AOSD with CBSE without duplicating concepts.

References

[1] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, page 144. Springer-Verlag, 2005.

[3] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[4] Filman et al. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[5] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[6] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[7] Sun Microsystems. Enterprise javabeans technology, http://java.sun.com/products/ejb/index.jsp.

[8] R. Reussner. The use of parameterised contracts for architecting systems with software components, 2001.

[9] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[10] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

A Safe Aspect-Oriented Programming Support for Component-Oriented Programming

Nicolas Pessemier[(1)], Lionel Seinturier[(1)],
Thierry Coupaye[(2)], Laurence Duchien[(1)]
[(1)] INRIA Futurs - LIFL, Project Jacquard/GOAL
59655 Villeneuve d'Ascq, France
[(2)] France Telecom R&D, 28 chemin du Vieux Chêne,
38243 Meylan, France
{pessemie,seinturi,duchien}@lifl.fr
thierry.coupaye@rd.francetelecom.com

***Abstract*— In this paper we show that Aspect-Oriented Programming (AOP) can be safely supported by Component-Oriented Programming (COP) by providing a way to control the openness of a component with regards to AOP techniques. Our proposal reconciles the intrusive nature of AOP with the "black box property" of components in COP. We build a compromise between modularity and openness applying the open modules approach to components. The experiment has been achieved on FAC, our model that unifies the notions of component and aspect. We show that most of open modules principles are directly available within our approach, we then study requirements for others. Once all these principles integrated, we are able to tune the accessibility of the content of a component to AOP during system runtime. Thus, components become grey boxes with dynamic variation points accessible to AOP techniques.**

I. Introduction

Component-Oriented Programming (COP) proposes to enhance object-oriented programming by separating concerns into clearly defined entities, called components. Reusable components with contractually specified interfaces are defined and composed together [13]. Nevertheless, whatever the decomposition adopted to represent a system, it has been shown that some concerns are mixed within a same component (code tangling), and that some concerns are scattered across several components [3], [6]. These concerns which are called crosscutting concerns hinder the reusability, the maintainability, and the evolvability of applications.

To tackle these issues, some approaches have proposed a support for Aspect-Oriented Programming (AOP) in component-based systems [5], [7], [12]. AOP is a well-known paradigm to overcome this issue by modularizing crosscutting concerns using aspects [4]. The main issue of these approaches is that AOP is applied regardless of the components themselves, the aspects are woven on the objects which implement the components. This intrusiveness breaks the encapsulation property of components and consequently their implicit contracts. This appears to be a major issue in COP where contracts and encapsulation are fundamentals.

At the object level, solutions have been proposed to overcome the issue of intrusiveness of AOP. For example, Aldrich introduced the notion of open modules, a new module system to open a program to AOP while keeping modularity by hiding implementation details of the module [1]. A module is defined as a set of entities which share a set of access points for the join points (points in the program execution flow where aspects will apply) exported by the module. Using this module system, the content of a module can be preserved by designating only the variation points where aspects can act.

In this paper we propose to push the open modules approach a step further by applying it to COP. The objective is to provide a safe way to support AOP in COP by controlling the openness of a component with regards to AOP. This control over variation points of a component can be seen as a compromise between modularity and openness. Our study focus on an extended component model for components and aspects which is presented in [10]. Our model, named FAC, unifies COP and AOP notions together by representing AOP notions as component ones. We show that when components and aspects are unified, some open modules properties are directly handled as first-class entities in our model. We have then extended our model to handle all open modules properties. A component can declare its variation points. Since our model is dynamic, this declaration can evolve at runtime.

The remainder of this paper is organized as follows. Section II provides some background on our unified model for aspects and components and on the open modules approach. Section III shows how we have applied open modules to FAC. Finally, Section IV concludes.

II. Background

This section provides some background on our previous work [10] on the unification of AOP and COP, and introduces the principles of the open modules approach.

A. FAC: An unification of AOP and COP towards COP

Since COP fails in supporting crosscutting concerns [3], [6], our motivation was to give a support to AOP in COP but also to take advantage from the strong encapsulation property of COP in AOP. Therefore, our proposal is built as a twofold integration of AOP and COP which has the benefit of representing AOP notions using COP ones. Thus, we introduce

three main concepts which are related to the general notions of component, binding and composite-component that generally appear in COP [13].

- An **Aspect component** is the representation of an aspect as a component. It offers as a provided interface a piece of advice code (the additional behavior to weave on other components). Basically, an aspect component applies around incoming and outgoing calls on component interfaces. Because we represent an aspect as a component we call our approach symmetric. Aspects and components are components, and can interact together using bindings (Figure 1 represents an aspect component connected to other components using various types of bindings). Traditionally in AOP two dimensions are considered: the base and the aspect dimension. Aspects are woven on the base and the base if oblivious of the aspect dimension. In our approach these two dimensions are unified to facilitate the interactions between components and aspects and their evolution.
- An **Aspect domain** is the representation of the domain of action of an aspect. It is represented as a composite component which contains the aspect component and the components on which it is woven. The notion of aspect domain can be seen as a reification of the notion of pointcut in AOP. Usually a pointcut is a description of a set of join points on which an advice code is woven. In our model we reify this notion as a first class entity (a composite-component) which contains the aspect component woven, and all the components affected by the aspect component. This clarifies the domain of impact of an aspect component in a system. This explicit relationship between advised code and aspects is a notion currently missing in AOP. Figure 1 gives an example of the aspect domain of the `transaction Aspect Component` which is woven on components `C`, `D` and `E`.
- An **Aspect binding** is the representation of the implicit link which exists between an aspect and a component on which the aspect is woven. The aspect binding notion can be seen as a more fine-grained notion than the aspect domain to capture the interaction between a component and a particular aspect component. Our philosophy is to consider only one dimension (aspects are components) and two types of interactions (regular bindings and aspect bindings). Aspect bindings are used to connect an aspect component with a component. By this way, each component can locally manage the aspects applied on its incoming and outgoing interfaces.

We have successfully mapped this general model with its three notions to the Fractal component model. Fractal [2] is a reflective and extensible component model, where bindings can be set and unset dynamically (at runtime); reflection is available through the use of special kinds of meta-interfaces called control interfaces.

We have extended this model by introducing our three main notions. This extension is called Fractal Aspect Component (FAC for short). We have used the provided notions of component and composite-component to represent our notions of aspect component and aspect domain. We have introduced a new control interface called the *weaving interface*. This interface, which appears on each component of the system, is in charge of setting/unsetting aspect bindings and of weaving aspect components. It has the benefit of locally managing the ordering of aspects for a component.

In addition to the advantages mentioned in the description of our three notions, the mapping onto the Fractal component model allows setting/unsetting aspect bindings dynamically. This makes our weaver dynamic and this opens the way to dynamic adaptation [9].

The complete description of our general model for component and aspect and its mapping to the Fractal component model [2], named FAC is beyond the scope of this paper and can be found in [10]. The next sub-section introduces the open modules approach which allows controlling the degree of openness of a module to AOP.

B. Open modules approach

The concept of *open modules* has been introduced by Aldrich to limit the access to join points of a system, which are accessed intrusively in AOP. With this approach any exposed join points has to be declared within a module, a special set of classes, to be accessed by aspects. The *open modules* approach is defined by Aldrich as follows: *"Open Modules describes a module system that:*

- ***Rule 1** allows external advice to interactions between a module and the outside world (including external calls to functions in the interface of a module)*
- ***Rule 2** allows external advice to pointcuts in the interface of a module*
- ***Rule 3** does not allow external modules to directly advise internal events within the module, such as calls from within a module to other functions within the module (including calls to exported functions).*"

We intentionally add rule numbers on items to facilitate the discussion in the following sections. The complete description of this module system can be found in [1].

More recently, the concept of open modules has been applied to AspectJ [8]. In this study, authors have extended the concept with new interesting features. Most of them are specifically related to AspectJ in order to support AspectJ pointcuts. Nevertheless it seems to us that some can be generalized out of the context of AspectJ. Thus, the most important feature with regards to our application of open modules to COP is the ability to open and not only to reduce the visibility on join points of a module. This can become extremely useful when using for instance debug aspects, or when considering dynamic adaptation using AOP. Among new features provided by this study, an interesting one is the ability to designate to which a pointcut is exposed to using a pattern language based on package hierarchy.

To help the discussion of the next section we will call the ability to designate which aspect has access to a module ***Rule 4***, and the ability to open a module by exposing join points ***Rule 5***.

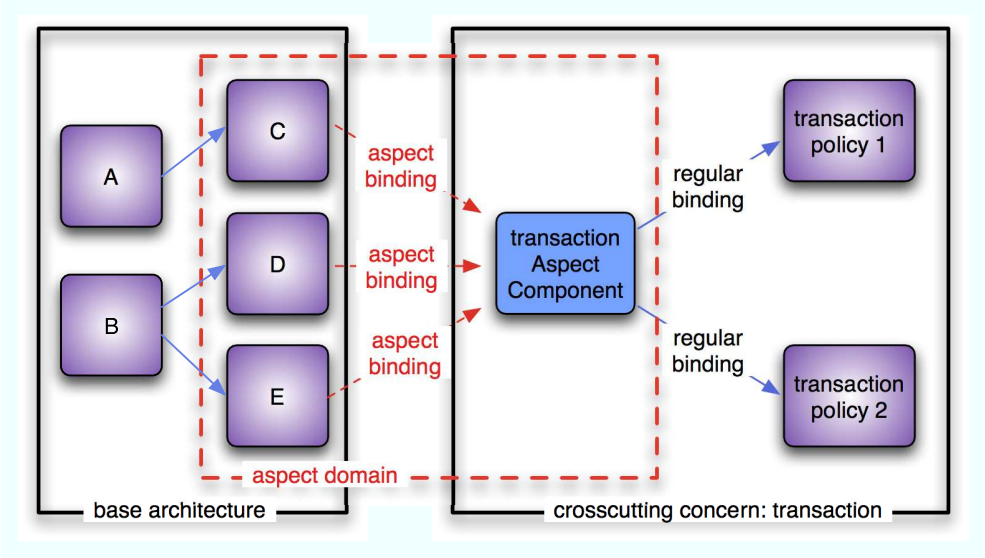


Fig. 1. Aspect binding best practice

III. Applying Open Modules to FAC

In this section we detail how we have applied the open modules approach to FAC. Some of the rules defined in open modules are directly mapped to existing notions of our model (Section III-A), some others require the introduction of new features to be correctly managed (Section III-B).

A. Similarities

The first obvious similarity is related to the notion of a module and a component. A module in the open modules approach is a collection of classes which share a set of access points to AOP. A component in COP is a contractually specified entity which provides and requires services by means of interfaces. A component is a black box which naturally hides its implementation details as required by *Rule 3* of open modules. Given that in FAC, join points are incoming and outgoing calls on component interfaces, the definition remains correct with regards to *Rule 1*, *2* and *3*. *Rule 1* and *2* are understood in FAC by the fact that an aspect component only applies to client and server interfaces (*Rule 2*). Following the definition of *Rule 1*, an aspect component interacts between a module (component) and the outside world (other components). *Rule 3* is preserved as soon as we do not want to break encapsulation in FAC. Join points are not points inside a component. However, when applying aspect component behavior on component external interfaces, we may consider that the original behavior of the component is altered by the aspect component. Thus, the behavior expected from a given component may be different. It seems important that the weaving of aspect component on provided and required interfaces of a component should be better controlled in order to provide a safer integration of crosscutting concerns. We elaborate more on that particular point in Section III-B.

The second similarity is related to *Rule 4* which has been defined to the particular use of AspectJ but can be also used within our system. This rule allows to clearly designate which aspect can apply on a given exported join point of a module. A regular expression is given as a parameter of the `expose to` declaration which designates a set of packages that are authorized to access the module. In FAC we have a very similar notion: the aspect binding. An aspect binding is set between a component and an aspect component using the weaving interface of the component. Because the weaving interface is a kind of meta interface, we can consider that the access policies defined on components are of the same type of meta-informations than the ones corresponding to the `expose to` definitions used in the extension of AspectJ supporting open modules. This means that we can consider *Rule 4* as naturally handled by each individual component which are able to choose the aspect to be impacted by.

At this point we have seen that *Rule 1*, *2*, *3*, and *4* are naturally handled by our model and its mapping to FAC. In the next section we study the requirements to manage remaining rule, Rule *5*.

B. Dissimilarities

We have seen that in our model and in its mapping to Fractal, FAC, the considered join points are incoming and outgoing calls on component interfaces. Thus, *Rule 3* is implicitely preserved when considering components as modules. However, this also means that the join points inside the component are not exposed. The original idea of open modules is to define some pointcuts and join points and make them available by means of interfaces of a module. In our case, the content of a component is implicitly protected, but we still need a support to give an access to other join points, *i.e.*, join points inside a component. Nevertheless, associating the level of implementation of components and the level of interaction between components (more architectural) takes part in our global vision of what really means applying AOP to COP that we have exposed in [9]. The unification of these two levels will allow us to look inside components and to externalize some

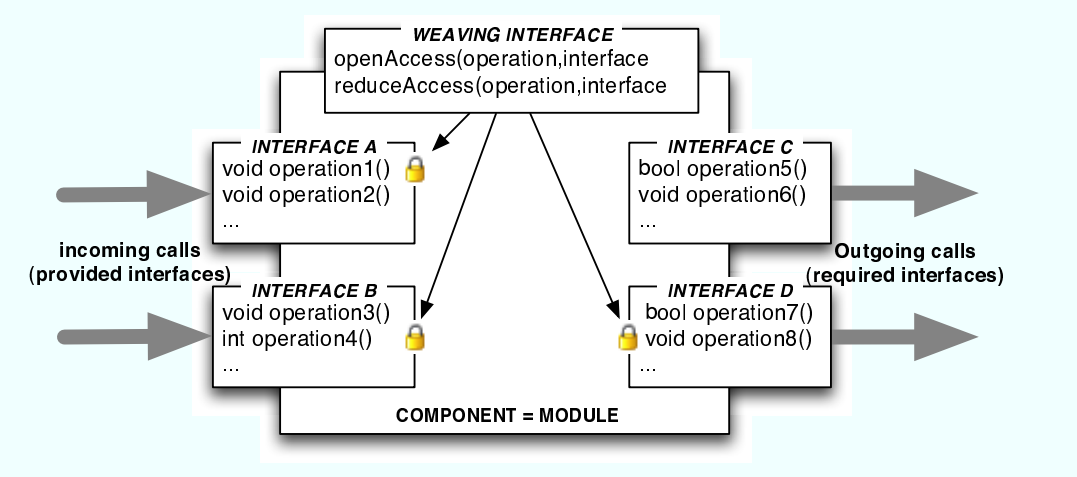


Fig. 2. A conceptual view of Open Modules applied to FAC. A component is a module which controls the access to its provided and required interfaces by means of the weaving interface. The content of the component is not exposed to aspects.

join points. We believe that these internal join points must also be controlled by the weaving interface. Thus, we would be able to fulfill needs for accessing internal join points, while preserving *Rule 3* by means of the weaving interface to control what is accessed or not. This approach has limitations with regards to legacy components that would not have be able to be instrumented to support AOP techniques. In our approach, we only consider a full-fledged component and aspect approach, where the design of the system follows the same formalism, the same design model.

The remaining rule (*Rule 5*) states that the open modules should not be limited to reducing the access to aspects but also to opening it. In Section II-A we have defined the role of the weaving interface as an interface to manage the setting/unsetting of aspect bindings, the weaving and the ordering/re-ordering of aspect components. We have extended the role of this interface to also manage the openness of a component to AOP. A conceptual view of the role of the weaving interface is presented in Figure 2. The weaving interface is able to prevent the weaving of aspects on a particular operation of an interface. Because FAC is a fully dynamic framework, these policies can be changed during runtime. A component can then be adapted to open or reduce the access to its join points. Moreover, since we have discussed it in Section III-A, weaving an aspect component on component external interfaces may change the expected behavior from other components as soon as it intercepts communications between interfaces. The idea of controlling the external join points which are accessible or not by other components seems interesting even if it was not originally considered by Aldrich in the first version of open modules.

IV. Concluding remarks

We have seen that AOP and COP can be reconciled on the particular issue of the intrusive property of AOP versus the strong encapsulation property of COP. To do so, we have applied the open modules approach to FAC, a unified model for components and aspects. Following the open modules philosophy our approach is able to open a component to AOP while keeping its content hidden from the outside. This compromise opens the way to a safe integration of AOP in COP. It is safe in the sense that the intrusiveness of AOP is finely managed on each individual component. Moreover in the case of FAC, we have seen that the openness of a component to AOP can be managed at runtime, allowing a component to adapt to unanticipated requirements. Black box components become grey box components providing variability points where AOP can access.

This integration of open modules approach to FAC takes part of our overall vision of applying AOP to COP presented in [9]. This vision is based on three levels: (1) An architectural level which is achieved with FAC where aspects notions are mapped on component ones; (2) A control level where aspects can be used to inject the control level of components into the remaining level, (3) the level of implementation of components. In [11] we show how AOP can be used for (2). In this paper we present a link between the architectural and the implementation level: Opening a component to expose internal join points and then, weaving architectural aspects to this internal join points (link between (1) and (3)). Our next step is to make all this three levels work together, allowing a coherent weaving of aspects whatever the chosen level.

Acknowledgments

This work was partially funded by France Telecom under the external research contract number 46 131 097.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586, pages 144–168. Springer, 2005.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering*, Edinburgh, Scotland, May 2004.

[3] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 65–75, New York, NY, USA, 2002. ACM Press.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[5] B. Lagaisse and W. Joosen. Component-based open middleware supporting aspect-oriented software composition. In *CBSE*, pages 139–154, 2005.

[6] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[7] M. Mezini and K.Ostermann. Conquering Aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 90–100. ACM Press, March 2003.

[8] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to Aspectj. In *Proceedings of the 5nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press, March 2006.

[9] N. Pessemier, O. Barais, L. Seinturier, T. Coupaye, and L. Duchien. A three level framework for adapting component-based systems. In *Second International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT05)*, Glasgow, Scotland, July 2005.

[10] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, Lecture Notes in Computer Science. Springer, Mar. 2006.

[11] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE06)*, Lecture Notes in Computer Science, Stockholm, Sweden, jun 2006. Springer.

[12] D. Suve, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29. ACM Press, 2003.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

Leveraging Component-Oriented Programming with Attribute-Oriented Programming

Romain ROUVOY
JACQUARD Project – INRIA Futurs
LIFL – University of Lille 1
59655 Villeneuve d'Ascq Cedex, France
Email: romain.rouvoy@inria.fr

Philippe MERLE
JACQUARD Project – INRIA Futurs
LIFL – University of Lille 1
59655 Villeneuve d'Ascq Cedex, France
Email: philippe.merle@inria.fr

***Abstract*— Component-oriented programming has achieved wide acceptance in the domain of software engineering by improving productivity, reusability and composition. This success has also encouraged the emergence of a plethora of component models. Nevertheless, even if the abstract models of existing component models are quite similar, their programming models can differ a lot. This drawback limits the reuse and composition of components implemented using different programming models.**

The contribution of this paper is to introduce a reification of an abstract model common to several component models. This reification is presented as an annotation framework, which allows the developer to annotate the program code with the elements of the abstract component model. Then, using a generator, the annotated program code is completed according to the programming model of the component model to be supported by the component runtime environment. This paper shows that this annotation framework provides a significant simplification of the program code by removing all dependencies on the component model interfaces. These benefits are illustrated with the OpenCOM and Fractal component models.

I. INTRODUCTION

Component-Oriented Programming (COP) has achieved wide acceptance in the domain of software engineering by improving productivity, reusability and composition. This success has also encouraged the emergence of a plethora of component models. These component models can now be applied at any software level, from operating systems (*e.g.*, Think [1]), to middleware (*e.g.*, OpenCOM [2], Fractal [3]), to applications (*e.g.*, EJB [4], CCM [5], SCA [6]). Usually, each of these component models defines their own *abstract model* and *programming model*. The abstract model defines the general concepts provided by the component model (*e.g.*, component, port/interface, binding/connection, composition/assembly). The programming model applies these concepts to a particular programming language, while introducing some technical code specific to the component model. Thus, this *technical code* is tangled with the *business code* of the application. Furthermore, if the abstract models of existing component models are quite similar, their programming models can differ a lot. This drawback limits the reuse and composition of components implemented with different programming models.

A convenient way to address this issue is to use *Attribute-Oriented Programming* (@OP) techniques [7], [8], [9]. @OP proposes to mark program code with metadata to clearly separate the business logic from the domain-specific logic (typically technical properties). @OP is gaining popularity with the recent introduction of annotations in Java 2 standard edition (J2SE) 5.0 [10] or in XDoclet [11], and attributes in C# [12]. Recently, the Enterprise Java Bean (EJB) 3.0 specification extensively uses annotations to make EJB programming easier [4]. The Service Component Architecture (SCA) component implementation model provides a series of annotations that can be placed in the code to mark specific elements of the implementation to be used by the SCA runtime environment [6]. Still, these annotations are specific to each component model. Therefore, annotated EJB code can not be used in a SCA runtime environment and vice versa.

In this paper, we introduce an abstract model common to several component models. This abstract model is reified as an annotation framework, which allows the developer to mark the program code with the elements of the abstract component model. Then, using a generator, the annotated program code is completed by the programming model of the component model that is supported by the desired component runtime environment, such as OpenCOM or Fractal. We show that this annotation framework provides a simplification of the program code by removing all dependencies on the component model interfaces. As a consequence, this approach protects the annotated program code from evolutions in the component models. Finally the annotated code can be executed with various component models.

The remainder of this paper is structured as follows. Section II introduces the problem of technical and business code tangling as the motivation of this paper. Section III presents the Attribute-Oriented Programming (@OP) approach and its relevance to Component-Oriented Programming (COP). Section IV provides an overview of the annotation framework defined to represent the abstract component model. Section V illustrates generators for two programming models: OpenCOM and Fractal. Section VI compares our approach with related work. Finally, Section VII concludes and provides some perspectives on this work.

II. MOTIVATION

Although COP provides more modularity, configurability, and reusability to applications, the use of a given component

model introduces also more complexity, more verbosity and redundancy in the information expressed by the developer compared to object-oriented programming practices. However, this complexity derives from the underlying programming model used to develop an application. This programming model maps the abstract model concepts to constructs of the programming language used to develop components. Therefore, the abstract model concepts are seamlessly drowned in the program code. By having introduced dependencies on the component model, the program code is no longer only concerned with business properties, but also with technical properties.

Developing an application using components requires taking into account concerns that are not always related to the business of the application. We illustrate these concerns with the application HelloWorld depicted in Figure 1. This application is composed of two components: Client and Server. These components are linked by a common contract — usually defined as an interface — named Service. Finally, the component Server can be configured via an attribute header to modify its display, which is implemented by a Logger.

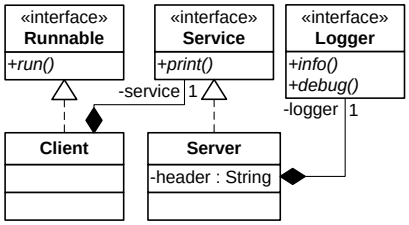


Fig. 1. Application HelloWorld.

```
1 public interface Service {
2   void print(String message);
3 }
```

Lst. 1. Interface Service.

```
1 public class Client implements Runnable {
2   protected Service s;

4   public void run() {
5     this.s.print("hello world");
6   }
7   public void setService(Service s) { this.s = s; }
8   public Service getService() { return this.s; }
9 }
```

Lst. 2. Component Client.

To clearly exhibit the problems related to COP, we write the program code of this application using a simple component model targetting the Java programming language [10]. Listing 1 presents the method `print` in the interface `Service` (Line 2). Listing 2 presents the implementation of the component Client. The class `Client` defines the required interface `Service` as an internal field (Line 2). The business code

```
1 public class Server implements Service {
2   protected Logger logger;
3   protected String header;

5   public Server() {
6     this.logger = JavaLog.logger("Server");
7   }
8   public void print(String msg) {
9     this.logger.info(this.header + msg);
10  }
11  public String getHeader() { return this.header; }
12  public void setHeader(String h) { this.header = h; }
13 }
```

Lst. 3. Component Server.

```
1 public class HelloWorld {
2   public static void main(String[] args) {
3     Server server = new Server();
4     Client client = new Client();

6     client.setService(server);
7     server.setHeader("-->");
8     client.run();
9 } }
```

Lst. 4. Application HelloWorld.

of the component is located in the method `run` (Lines 4–6). The methods `setService` and `getService` (Lines 7–8) are used to set the reference of the required interface `Service`. Listing 3 presents the implementation of the component Server. The class `Server` implements the interface `Service` (Line 1). Then, it declares a field `logger` that refers to an internal service provided by the Java platform (Line 2). It declares the field `header` to store the attribute header provided by the component Server (Line 3). The reference of the logging service is retrieved in the constructor of the class `Server` (Line 6). The business code of the component is defined in the method `print` (Lines 8–10). Finally, the methods `getHeader` and `setHeader` are defined to configure the attribute header (Lines 11–12).

Listing 4 presents the assembly code defined to build the application HelloWorld. The method `main` of the class `HelloWorld` defines the variables `client` and `server` to specify the use of the Client and Server (Lines 3–4) components. Then, it connects the component Client to the component Server, and configures the value of the attribute `header` of the component Server (Lines 6–7). When executing the application HelloWorld, the components are automatically created and configured before executing the method `run` of the component Client (Line 8).

The drawbacks that arise from such a component-oriented implementation of the application HelloWorld are located in the technical part of the program code. This technical part is usually tangled with the underlying component model used to implement the application. For example, in Listing 2, the business method of the component Client (Lines 4–6) is combined with methods that handle the required interfaces (Lines 7–8). Similarly, in Listing 3, the business method of the component Server (Lines 8–10) is mixed with the initialization code of the logger (Line 6) and the methods that handle

the attribute header (Lines 11–12). These technical methods can differ from one component model to another because they are defined by the programming model of each component model. Besides, when considering this part, it appears that the abstract component model is tangled with the programming model. The fields that are used by the technical methods are implicitly required by the component model because they store the concepts of the abstract component model. The declaration of these fields appears to be common the component models that can be used to develop the application. The remainder of this paper shows that annotating these fields can provide an abstraction of the programming model.

III. Attribute-Oriented Programming

Attribute-Oriented Programming (@OP) is a program-level marking technique. Basically, this approach allows developers to mark program elements (*e.g.*, classes, methods, and fields) with *annotations* to indicate that they maintain application-specific or domain-specific concerns [9], [7], [8]. For example, a developer may define a logging annotation and associate it with a method to indicate that the calls to this method should be logged, or may define a web service annotation and associate it with a class to indicate that the class should implement a Web Service. Annotations separate application's business logic from middleware-specific or domain-specific concerns (*e.g.*, logging and web service functions). By hiding the implementation details of those semantics from program code, annotations increase the level of programming abstraction and reduce programming complexity, resulting in simpler and more readable programs. The program elements associated with annotations are transformed to more detailed program code by a supporting generation engine. For example, a generation engine may insert logging code into the methods associated with a `logging` annotation. Dependencies on the underlying middleware are thus replaced by annotations, acting as weak references —*i.e.*, references that are not mandatory for the application. This means that the evolution of the underlying middleware is taken into account by the generation engine and let the program code unchanged.

@OP can also be used to provide *continuous integration* in Component-Based Software Engineering. Continuous integration allows a developer to generate the middleware artifacts at any step of the component development. Developers concentrate their editing work on only one source file per component. The deployment metadata are continuously integrated without worrying about updating them. When the development of a component consists of several files, @OP allows the developer to maintain only one of them while the other files are generated automatically. Besides, working with only one file per component gives a better overview of the program code to the developer. Therefore, the developer can concentrate on the business logic and reduce the development time drastically.

@OP has been applied in several object-oriented frameworks to ease the process of configuring applications (*e.g.*, Hibernate, Struts, Castor [11]), and it has been applied by several J2EE application servers to simplify the configuration of Enterprise Java Bean (EJB) components (*e.g.*, JOnAS, WebSphere, JBoss). Nevertheless, these annotations target only the configuration of the EJB components, and are specific to an application server.

Recently, the EJB 3.0 specification has introduced extensive annotations to make EJB programming easier [4]. The annotations defined in this specification address either EJB component configuration or program code generation concerns. Nevertheless, this specification presents two weaknesses. Firstly, it is dedicated to EJB components. This means that the strengths of this specification are not directly applicable to other component models. Secondly, the EJB specification does not focus on the EJB abstract model but abstracts the EJB programming model. Thus, the annotations defined in the EJB specification are tightly coupled to the EJB programming model.

This @OP approach provides an useful formalism to introduce a higher-level semantics into the artifacts of existing programming models. In particular, @OP can be applied to represent the abstract component model using annotations. These annotations remove all the technical code that is required by a given programming model and that is tangled with the business code. As a side effect, the use of @OP allows the developer to write a program code compliant with several component models. To achieve this, it is necessary to identify the core concepts that are usually defined in the component models. The specificities of existing component models are reified in some extensions of the annotation framework.

IV. Component Annotations

This section introduces our annotation framework that leverages the common COP practices. After identifying the core concepts involved in several component models, our abstract component model is represented as a set of five annotations. Finally, the application HelloWorld is revisited using our annotations.

A. Identification of the Component Model Concepts

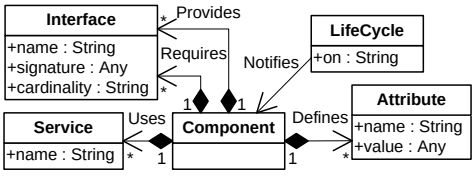


Fig. 2. Component core concepts.

Most existing component models (*e.g.*, OpenCOM [2], Fractal [3], JavaBean [13], EJB [4], CCM [5]) rely on some common core concepts, as summarized in Figure 2. A Component is defined as an entity that Provides and Requires some interfaces. These interfaces are usually identified by a *name* and a *signature* —*i.e.*, a set of operations. A *cardinality* is also specified to define that a Component requires several interfaces of the same type. A Component can additionally define

a set of Attributes to support configuration. An Attribute is initialized with a *value* when the Component is loaded. A Component can furthermore require some Services provided by the runtime structure executing it. These Services often provide to the technical properties required by a component (*e.g.*, logging). Finally, the component models define the concept of Lifecycle. This concept allows the component to be aware of its current state (*e.g.*, created, started, stopped, destroyed).

B. Overview of the Component Annotation Framework

Each core component concept previously identified is defined as an annotation applicable to a piece of the program code. The resulting annotations are summarized in Table I. The annotation @Provides applies to an interface that is provided by a component. An attribute *name* can be specified when using this annotation. The attribute *signature* is used when the interface signature cannot be inferred from the program code. The annotation @Requires applies to the reference to an interface required by a component. The attribute *name* (resp. *signature*) is defined to override the name (resp. the signature) of the field marked by the annotation. The attribute *cardinality* is useful to indicate whether the field refers to an optional reference or to a collection reference. The annotation @Attribute applies to the declaration of a field. If no attribute *value* is defined, the attribute is defined when composing the components together. The annotation @Service applies to the reference to a service provided by the component runtime environment (*e.g.*, logging). The attribute *name* refers to the identifier of the service. The annotation @Lifecyle applies to a method defined in the component code. This method defines treatments that should be executed at a given transition of the component lifecycle (*e.g.*, from stopped to started) using the attribute *on*.

C. Revisiting the Application HelloWorld with Annotations

This section revisits the application HelloWorld introduced in Section II. To illustrate the benefit of @OP, the program code of this application is reengineered to replace all the technical code by the previously defined annotations. Listings 5 to 7 show that the original business code is preserved. The interface `Service` is marked with the annotation @Provides to define `s` as its identifier (Line 1 of Listing 5). This information was previously defined in the ADL descriptor of the component. The reference to the interface `Service` in the class `Client` is marked with the annotation @Requires (Lines 3–4 of Listing 6). As a consequence the program code related to service reference handling becomes useless. The logging support required by the component `Server` and provided by the runtime environment is replaced by the annotation @Service, which takes in charge the configuration of the logging service (Lines 2–3 of Listing 7). The original constructor of the class `Server` is dropped because its only use was to retrieve the logging service. Finally, the attribute `header` is marked with the annotation @Attribute to reify this field as a component attribute, enabling the removal of the methods that get and set its value (Lines 4–5 of Listing 7).

```
1 /** @Provides name="s" */
2 public interface Service {
3   void print(String message);
4 }
```

Lst. 5. Interface Service.

```
1 /** @Provides name="r" signature="Runnable" */
2 public class Client implements Runnable {
3   /** @Requires */
4   protected Service s;
5   /** @LifeCycle on="start" */
6   public void run() {
7     this.s.print("hello world");
8 } }
```

Lst. 6. Component Client.

```
1 public class Server implements Service {
2   /** @Service name="logging" */
3   protected Logger logger;
4   /** @Attribute */
5   protected String header;

7   public void print(String msg) {
8     this.logger.info(this.header + msg);
9 } }
```

Lst. 7. Component Server.

The annotations enhance the program code elements —*i.e.*, classes, methods, and fields— with the core concepts of a component model —*i.e.*, requires, provides, attribute etc.— without the programmer having to worry about the specificities of its programming model. These specificities are automatically injected into the program code by generators that are specific to each targetted component model.

V. Implementation of the Generators

This section first introduces the architecture of a generation engine used to support the programming model of a given component model, and then illustrates our approach on two existing component models developed in Java: OpenCOM and Fractal.

A. Multi-Component Generation Process

Using our annotations, the annotated program code can be instantiated according to the programming model of various component models, as illustrated in Figure 3. The annotated program code is considered as Platform-Independent Code (PIC). Using generators, the PIC becomes Platform-Specific Code (PSC), which complies with a given programming model (*e.g.*, Fractal, OpenCOM). Furthermore, the same PIC can support the evolution of a given programming model in a seamlessness manner. This PIC can also be executed on various component models. This approach allows developers to support other programming models by implementing additional generators.

Annotation	Code Element	Parameter	Description	Cardinality	Default Value
@Provides	`Class` or `Interface`	*name* *signature*	provided interface name provided interface signature	optional optional	interface name interface signature
@Requires	`Field`	*name* *signature* *cardinality*	required interface name required interface signature required interface cardinality	optional optional optional	field name field signature `1..1`
@Attribute	`Field`	*name* *value*	component attribute name component attribute value	optional optional	field name -
@Service	`Field`	*name*	component service name	required	-
@Lifecycle	`Method`	*on*	lifecycle state to handle	required	-

TABLE I
OVERVIEW OF THE COMPONENT ANNOTATION FRAMEWORK.

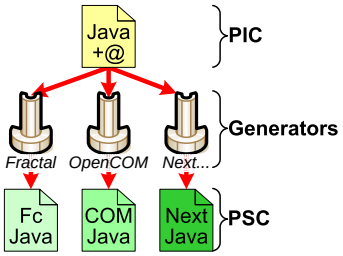


Fig. 3. MDE approach.

B. Architecture of the Generation Engine

Figure 4 presents the architecture of the generation engine used to produce the various component artifacts. This generation engine reifies the annotated program code as a model in memory. The generator uses this program code model to produce the Java program code (1). Then, the handwritten and generated program code are compiled by the Java compiler (2). Finally, the compiled code is executed by the component runtime environment (3).

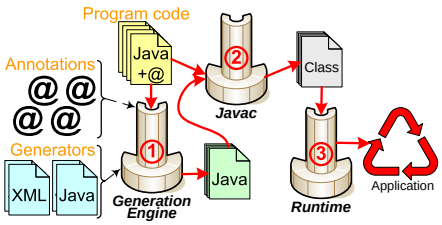


Fig. 4. Generation engine.

Two functionally equivalent implementations of the annotation framework have been developed. The first defines XDoc annotations and uses the XDoclet generation engine [11] to produce the various artifacts required by the component model. The second defines Java 5 annotations and uses the Spoon transformation tool [7] to enhance the handwritten program code with the technical properties of the component model.

C. OpenCOM Component Generator

This section presents the generator defined for the OpenCOM programming model [2]. After introducing the OpenCOM component model, we present the available generator, and the generated components.

1) OpenCOM Component Model: OpenCOM is a lightweight, efficient, reflective component model that uses the core features of Microsoft COM to underpin its implementation; these features include the binary level interoperability standard, Microsoft's IDL, COM's globally unique identifiers and the IUnknown interface [2]. Recently, a Java version of OpenCOM has been developed to provide platform independence, and to ease the developments of applications on top of OpenCOM.

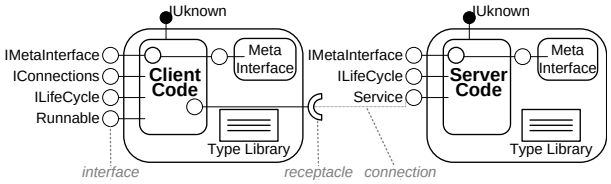


Fig. 5. OpenCOM assembly HelloWorld.

The key concepts of OpenCOM are *interface*, *receptacle* and *connection*. Each component implements a set of interfaces and receptacles, as shown in Figure 5. An interface expresses a unit of service provision, a receptacle describes a unit of service requirement and a connection is the binding between a receptacle and an interface of the same type. Among the possible interfaces provided by an OpenCOM component, the IUknown interface provides the reference of the component and an operation to navigate through the component's interfaces, the IMetaInterface interface provides operations to introspect the component, the IConnections interface is used to connect the component receptacles to interfaces, and the ILifeCycle interface supports the lifecycle of the component. OpenCOM provides a standard runtime substrate per address space that manages the creation and deletion of components, acts upon requests to connect/disconnect components and provides service interfaces for reflective operations. The runtime substrate dynamically maintains a graph of the components currently in use. The maintenance of dynamic dependencies between components is relevant for the introspection and reconfiguration of component configurations.

2) Overview of the OpenCOM Generator: In the context of OpenCOM, only one Java program code generator is available:

Component glue is a Java generator that produces, for each OpenCOM component, the technical code required by the OpenCOM programming model. This technical code requires to implement the methods of the interfaces `IUnknown`, `IMetaInterface`, `ILifeCycle`, and `IConnections`.

3) Generation for OpenCOM: Listings 8 and 9 present the program code generated by the *component glue* generator. The generated class `OCMClient` extends the original class `Client` and implements the interfaces `IUnknown`, `IMetaInterface`, `ILifeCycle` and `IConnections` (Listing 8). The method `QueryInterface` provides the reference of the interfaces provided by the component (Lines 11–18). The methods defined in lines 20–28 manage both the component meta-level and the component attributes. The methods defined in lines 30–31 notify the component of its lifecycle evolution. Finally, the methods defined in lines 33–42 allow OpenCOM to connect the references of the required interfaces to those provided by the `Server` component. In particular, the reference to the interface `Service` defined in the annotated program code is automatically updated in these methods. The generated class `OCMServer` extends the original class `Server` and implements the interfaces `IUnknown`, `IMetaInterface`, and `ILifeCycle` (Listing 9). The behaviour associated with the implementation of these interfaces is the same as that of the `OCMClient`.

D. Fractal Component Generators

1) Fractal Component Model: The hierarchical Fractal component model uses the usual *component*, *interface*, and *binding* concepts [3]. A component is a runtime entity that conforms to the Fractal model. A *primitive component* encapsulates a unit of computation described in a given programming language. An interface is an interaction point expressing the provided or required methods of the component. A binding is a communication channel established between component interfaces. Furthermore, Fractal supports *recursion with sharing* and *reflective control* [14]. Recursion with sharing means that a *composite component* can be composed of several subcomponents at any level, and a component can be a subcomponent of several components. Reflective control means that an architecture built with Fractal can be reified at runtime and can be dynamically introspected and managed. Fractal provides also an Architecture Description Language (ADL) [15], named *Fractal ADL*, to describe and automatically deploy component-based configurations.

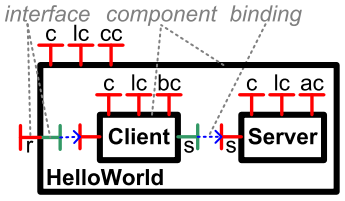


Fig. 6. Fractal component HelloWorld.

Figure 6 illustrates the different entities of a typical Fractal component architecture. Thick black boxes denote the *controller* part of a component, while the interior of the boxes corresponds to the *content* part of a component. Arrows correspond to bindings, and tau-like structures protruding from black boxes are internal or external interfaces. Internal interfaces are only accessible from the content part of a component. External interfaces appearing at the top of a component represent reflective control interfaces such as the component controller (c), the lifecycle controller (lc), the binding controller (bc), the content controller (cc), or the attribute controller (ac) interfaces.

2) Overview of the Fractal Generators: In the context of Fractal, two Java program code generators and one XML definition generator are available. The number of generators depends on the kind of artifacts that are generated.

Attribute controller is a Java generator that produces, for each Fractal component defining at least one attribute, an interface that handles the attributes of the component. This interface contains a getter and a setter method for each attribute.

Component glue is a Java generator that produces the technical code for each Fractal primitive component. This technical code requires to implement the methods of the attribute controller interface generated previously, the methods defined in the binding controller interface, and the methods required by the lifecycle controller.

Primitive definition is an XML generator that produces a Fractal ADL definition for each Fractal primitive component. This definition comprises the interfaces provided and required by the component, the attributes with their initial value, and the name of the content class implementing the component.

3) Generation for Fractal: The execution of the *attribute controller* generator provides the interface `ServerAttributeController`, which is used by the Fractal runtime environment to handle the component attributes (see Listing 11). The execution of the *component glue* generator provides the classes `FcClient` (see Listing 10) and `FcServer` (see Listing 12). The generated class `FcClient` extends the original class `Client` to introduce the interface `BindingController` required by Fractal to handle the component client interfaces. The generated class `FcServer` extends the original class `Server` to implement the interface `ServerAttributeController` and the interface `Loggable`, which allows Fractal to initialize the server's logger. The execution of the *primitive definition* generator provides the Fractal ADL definitions of the component Client (see Listing 13), the interface Service (see Listing 14), and the component Server (see Listing 15).

E. Conclusion

To conclude, the code produced by the generators (see Listings 8 to 15) would have been written by developers. Thus, our approach combining @OP and generative programming improves the development of components drastically.

```
public class OCMClient extends Client implements IUnknown, IMetaInterface, IConnections, ILifeCycle {
  private OCM_SingleReceptacle ocm_s = new OCM_SingleReceptacle(Service.class);
  private MetaInterface _meta_;
  public OCMClient(IUnknown binder) {
    OCM_SingleReceptacle ocm = new OCM_SingleReceptacle(IOpenCOM.class);
    ocm.connectToRecp(binder,"OpenCOM.IOpenCOM", 0);
    _meta_ = new MetaInterface((IOpenCOM) ocm.m_pIntf, this);
  }
  // IUnknown Interface
  public Object QueryInterface(String name) {
    if (name.equalsIgnoreCase("Runnable")) return this;
    Vector query = new Vector();
    _meta_.ReadInterfaceNames(this.getClass(), query);
    for (int i=0; i<query.size(); i++)
      if (name.equalsIgnoreCase(query.get(i).toString())) return this;
    return null;
  }
  // IMetaInterface Interface
  public int enumIntfs(Vector ppIntf) { return _meta_.enumIntfs(this,ppIntf); }
  public int enumRecps(Vector recp) { return _meta_.enumRecps((IUnknown)this,recp); }
  public boolean SetAttributeValue(String id, String kind, String name, String type, Object val) {
    return _meta_.SetAttributeValue(id, kind, name, type, val);
  }
  public TypedAttribute GetAttributeValue(String id, String kind, String name) {
    return _meta_.GetAttributeValue(id, kind, name);
  }
  public Hashtable GetAllValues(String kind, String id) { return _meta_.GetAllValues(kind, id); }
  // ILifeCycle Interface
  public boolean startup(Object pIOCM) { super.run(); return true; }
  public boolean shutdown() { return true; }
  // IConnections Interface
  public boolean connect(IUnknown itf, String signature, long id) {
    boolean r = ocm_s.connectToRecp(itf, signature, id);
    if (r && signature.equalsIgnoreCase("Service")) super.s = (Service) ocm_s.m_pIntf;
    return r;
  }
  public boolean disconnect(String signature, long id) {
    boolean r = ocm_s.disconnectFromRecp(id);
    if (r && signature.equalsIgnoreCase("Service")) super.s = null;
    return r;
}  }
```

Lst. 8. OpenCOM component Client.

```
public class OCMServer extends Server implements IUnknown, IMetaInterface, ILifeCycle {
  private MetaInterface _meta_;
  public OCMServer(IUnknown binder) {
    OCM_SingleReceptacle ocm = new OCM_SingleReceptacle(IOpenCOM.class);
    ocm.connectToRecp(binder, "OpenCOM.IOpenCOM", 0);
    _meta_ = new MetaInterface((IOpenCOM) ocm.m_pIntf, this);
    super.logger = JavaLog.logger("Server");
  }
  // IUnknown Interface
  public Object QueryInterface(String name) {
    if (name.equalsIgnoreCase("Service")) return this;
    Vector query = new Vector();
    _meta_.ReadInterfaceNames(this.getClass(), query);
    for (int i=0; i<query.size(); i++)
      if (name.equalsIgnoreCase(query.get(i).toString())) return this;
    return null;
  }
  // IMetaInterface Interface
  public int enumIntfs(Vector ppIntf) { return _meta_.enumIntfs((IUnknown) this, ppIntf); }
  public int enumRecps(Vector recp) { return _meta_.enumRecps((IUnknown) this, recp); }
  public boolean SetAttributeValue(String id, String kind, String name, String type, Object val) {
    if (name.equalsIgnoreCase("header")) super.header = (String) val;
    return _meta_.SetAttributeValue(id, kind, name, type, val);
  }
  public TypedAttribute GetAttributeValue(String id, String kind, String name) {
    if (name.equalsIgnoreCase("header")) return super.header;
    return _meta_.GetAttributeValue(id, kind, name);
  }
  public Hashtable GetAllValues(String kind, String id) { return _meta_.GetAllValues(kind, id); }
  // ILifeCycle Interface
  public boolean startup(Object pIOCM) { return true; }
  public boolean shutdown() { return true; }
}
```

Lst. 9. OpenCOM component Server.

```
1 public class FcClient extends Client
2   implements LifeCycleController, BindingController {
3   // LifeCycleController interface
4   public String getFcState() { return null; }
5   public void startFc() { super.run(); }
6   public void stopFc() { }
7   // BindingController interface
8   public String[] listFc() {
9     ArrayList _itf_ = new ArrayList();
10    _itf_.add("s");
11    return (String[]) _itf_.toArray(new String[0]);
12  }
13  public Object lookupFc(String itf) {
14    if (itf.equals("s")) return super.s;
15    return null;
16  }
17  public void bindFc(String n, Object itf) {
18    if (n.equals("s")) super.s = (Service) itf;
19  }
20  public void unbindFc(String itf) {
21    if (itf.equals("s")) super.s = null;
22 } }
```

Lst. 10. Fractal component Client.

```
1 public interface ServerAttributeController
2         extends AttributeController {
3   String getHeader();
4   void setHeader(String header);
5 }
```

Lst. 11. Fractal attribute header.

```
1 public class FcServer extends Server
2   implements ServerAttributeController, Loggable {
3   // Loggable interface
4   public Logger getLogger() { return super.logger; }
5   public void setLogger(Logger l) { super.logger = l; }
6   // ServerAttributeController interface
7   public String getHeader() {
8     return super.header;
9   }
10  public void setHeader(String header) {
11    super.header = header;
12 } }
```

Lst. 12. Fractal component Server.

```
1 <definition name="Client">
2   <interface name="r" role="server"
3              signature="Runnable"/>
4   <interface name="s" role="client"
5              signature="Service"/>
6   <content class="FcClient"/>
7 </definition>
```

Lst. 13. Fractal ADL definition Client.

```
1 <definition name="Service">
2   <interface name="s" role="server"
3         signature="Service"/>
4 </definition>
```

Lst. 14. Fractal ADL definition Service.

```
1 <definition name="Server" extends="Service"
2       arguments="header">
3   <content class="FcServer"/>
4   <attributes signature="ServerAttributeController">
5     <attribute name="header" value="${header}"/>
6   </attributes>
7 </definition>
```

Lst. 15. Fractal ADL definition Server.

VI. Related Work

This section compares our work with existing approaches such as Aspect-Oriented Programming and Model-Driven Engineering. We also compare to the existing technologies that use Attribute-Oriented Programming to leverage the management of technical properties.

Attribute-Oriented Programming has already been applied in the context of COP. The EJB 3.0 [4] and the Service Component Architecture (SCA) [6] specifications extensively use annotations to make programming easier but these approaches provide no complete abstraction of their programming model. [16] presents an *a posteriori* approach that extends an ADL to mark components with annotations. Nevertheless, this work is limited to the introduction of additional technical properties, such as the property of *Deny of Service* detection, to legacy components. Our work is an *a priori* approach to leverage COP using an annotation-based abstraction of the programming model of the component model.

Aspect-Oriented Programming (AOP) provides an partial solution to the problem of technical code abstraction. In [16], the annotations defined at the architectural level are consumed in the program code by aspects defined with AspectJ. The annotations mark potiential victim interfaces and are consumed to inject the *Deny of Service* detection code. Similarly, the AOKell implementation of the Fractal component model provides an aspect that automatically inject the technical code related to the handling of the component client interfaces [17]. However, AOP is not able to generate additional artifacts such as the attribute controller interface or the component definitions. Our approach provides Java and XML generators to support both program code and ADL definition generation.

Model-Driven Engineering (MDE) promotes the use of Platform-Independent Models (PIM) to define the business concern of an application. The PIM can be transformed into different Platform-Specific Models (PSM) that take into account the specificities of a given platform (*e.g.*, a given component model). Our approach follows the same idea at the program code level rather than at the model level. Indeed, the annotated program code can be considered as a Platform-Independent Code (PIC) composed of the business concern of the application and the annotations related to COP. Then, the generators produce the program code compliant with a given programming model as a Platform-Specific Code (PSC). Our approach is a practical application of MDE for the programming level, as a consequence it appears as an interesting solution to provide component model independency. In [8], the authors combined UML stereotypes and tagged values to simulate an annotation mechanism when modeling an application. The stereotype and the tagged values are thus mapped to annotations when generating the application code. This interesting approach brings annotations to the model level but it does not try to abstract the diversity of underlying platforms as proposed in this paper.

VII. Conclusion & Perspectives

This paper has presented a reification of an abstract component model as an annotation framework. This framework gathers the core concepts of the abstract component model manipulated by the developers. Using the five annotations defined in the framework, the developer can write a program code that contains only the business concern of the application, making it more lisible. The handwritten program code becomes simpler while being free of all the technical code. The compliance with the programming model of a given component model is ensured by generators that consume the annotations to produce the technical code required by a component model. The use of generators provides an interesting solution to the problem of the evolution of component models because a modification of the programming model of the component model impacts only the generators and no more the application. Finally, an annotated program code can be executed on various component-oriented platforms. In this paper, the generators for the OpenCOM and Fractal component models are illustrated and show the benefits of our approach.

Among the possible evolutions of this approach, the support of additional component models must be considered (*e.g.*, JavaBean [13], EJB [4], CCM [5], SCA [6]). Besides, the annotation framework can be easily extended to handle crosscutting concerns (*e.g.*, transaction, persistency). When considering the specificities of existing component models (*e.g.*, asynchronous communications in CCM), the annotation framework can easily be extended to include new annotations and generators. Nevertheless, the use of such annotations would restricted the list of target component models. Finally we are interested in the definition of a common composition language to describe component-oriented architectures. In particular, Fractal ADL provides an open interpretation engine to associate various behaviours to Fractal ADL concepts (*e.g.*, configuration verification, deployment plan generation, dynamic component deployment). We think that Fractal ADL can be considered for describing abstract compositions of components.

References

[1] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller, "Think: A Software Framework for Component-based Operating System Kernels," in *USENIX Annual Technical Conference, General Track*, Monterey, California, USA, June 2002, pp. 73–86.

[2] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, H. A. Duran-Limon, T. Fitzpatrick, L. Johnston, R. S. Moreira, N. Parlavantzas, and K. B. Saikoski, "The Design and Implementation of Open ORB 2," *IEEE Distributed Systems Online*, vol. 2, no. 6, 2001.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "An Open Component Model and Its Support in Java," in *7th International Symposium on Component-Based Software Engineering (CBSE)*, ser. LNCS, vol. 3054. Edinburgh, UK: Springer, May 2004, pp. 7–22.

[4] L. DeMichiel and M. Keith, *Enterprise JavaBeans (EJB) Specification*, 3rd ed., Sun Microsystems, Inc., Santa Clara, California, U.S.A, Dec. 2005.

[5] OMG, *CORBA Component Model (CCM) Specification*, 3rd ed., Needham, MA, USA, Sept. 2002.

[6] IBM Corporation, *Service Component Architecture (SCA) Specification*, 0th ed., Nov. 2005.

[7] R. Pawlak, "Spoon: Annotation-Driven Program Transformation - The AOP Case," in *1st International Middleware Workshop on Aspect-Oriented Middleware Development (AOMD)*, ser. AICPS, vol. 118. Grenoble, France: ACM, Nov. 2005, pp. 1–6.

[8] H. Wada and J. Suzuki, "Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming," in *8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, ser. LNCS, vol. 3713. Montego Bay, Jamaica: Springer, Oct. 2005, pp. 584 – 600.

[9] M. Eichberg, T. Schäfer, and M. Mezini, "Using Annotations to Check Structural Properties of Classes," in *8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, ser. LNCS, no. 3442. Edinburgh, UK: Springer, Apr. 2005, pp. 237–252.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, Third Edition*. Santa Clara, California, USA: Addison-Westley Professional Computing, Dec. 2005.

[11] C. Walls and N. Richards, *XDoclet in Action*, ser. In Actions series. Manning Publications, Dec. 2003.

[12] Ecma International, *C# Language Specification*, 3rd ed., Geneva, Switzerland, June 2005.

[13] G. Hamilton, *JavaBeans Specification*, 1st ed., Sun Microsystems, Inc., San Antonio Road, Palo Alto, CA, Aug. 1997.

[14] E. Bruneton, T. Coupaye, and J.-B. Stefani, "Recursive and Dynamic Software Composition with Sharing," in *7th International Workshop on Component-Oriented Programming (WCOP)*, Malaga, Spain, June 2002.

[15] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[16] V. Schiavoni and V. Quéma, "*A Posteriori* Defensive Programming: An Annotation Toolkit for DoS-Resistant Component-Based Architectures," in *21st ACM Symposium on Applied Computing (SAC)*. Dijon, France: ACM, Apr. 2006.

[17] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye, "A Component Model Engineered with Components and Aspects," in *9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, ser. LNCS. Stockholm, Sweden: Springer, June 2006.

Automated Component Bridge Generator

Dominik Glaser
docufy GmbH
D-96047 Bamberg
Email: dominik@docufy.de

Gregor Fischer
Würzburg University
Institute for Informatics
D-97074 Würzburg
Email: fischer@informatik.uni-wuerzburg.de

Jürgen Wolff von Gudenberg
Würzburg University
Institute for Informatics
D-97074 Würzburg
Email: wolff@informatik.uni-wuerzburg.de

***Abstract*— This position paper describes an automatic component bridge generator for embedding COM components in the Eclipse rich client platform. While embedding of ActiveX controls (graphical COM components) is in principle possible in Eclipse, the data model of the control is not easily accessible, and if done anyway, the task is quite tedious and error-prone.**

Therefore an automated component bridge generator was developed based on a model transformation framework, that analyses information about the component and automatically generates a bridge for the component to be used in Java. Special care must be taken of resource management, event handling, type mapping and constructs in COM that are not directly available in Java like optional or out parameters.

Using the generator bridges were created within minutes that previously took several months to code. They allow numerous components like Word or Nero Burning Rom to be used, embedded and controlled from Java applications.

I. Introduction

When developing an application for the Eclipse platform, the core Java library and of course the components of the Eclipse platform itself (plug-ins) are available. Although this accumulates to quite a collection of reusable software, these are mostly rather abstract. More specific components like a spreadsheet are not easily available (yet).

On the other hand, the missing components are often available outside of Eclipse, e.g. as applications. Often applications export all or parts of their functionality as components to be embedded in other applications. Unfortunately it is quite tedious and error-prone to create bridges from the Eclipse/Java-World to the component model of the system. Therefore an automatic component bridge generator was developed. It builds upon a model transformation framework that simplifies transformations of component models. Using the framework the required transformers were implemented to generate bridges for general COM components and ActiveX controls in particular to be used from Java/Eclipse.

COM

The Component Object Model (COM) is currently the most widely used model for reusing components throughout the Windows platform. This might change once its designated successor .NET is widely established and broadly used. But for now most components on Windows are available as COM components.

The interfaces and types for a COM component are defined in a type library (as a binary standard). These are usually generated from an (M)IDL description file. An eclipse-plugin to recreate the IDL-file from the binary type library was developed in this project as a side-product.

Stubs for e.g. C/C++ can be generated from the IDL-file, so that the components of the type library can be used in those languages. Scripting languages like Visual Basic can also explore and use COM components dynamically.

Eclipse as execution-environment for COM components

The Eclipse Rich Client Platform runs on the Java Platform. Hence direct interaction with the underlying system and its components is not possible.

Although running on the Java platform, Eclipse generally provides means to embed an ActiveX control. But up to now the code to access the component model of the ActiveX control had to be manually implemented. To automate this process it was necessary to implement a library that allows interaction with the COM system. A configurable provider, that gives access to the installed components, was developed in particular. It encapsulates the native code required. The provider can of course be implemented natively using JNI, but as the SWT (which is a part of Eclipse) already contains the required functionality, a SWT provider was realized.

II. The model transformation framework

In order not to be restricted to COM as input and Java as the target for the generated bridge, a more general purpose model transformation framework was developed.

It allows to transform arbitrary models given by model providers to target models used by model consumers, as long as a chain of transformations by model transformers can be derived. All possible chains of transformation are automatically inferred from the registered model transformers, so that the user only needs to choose which chain to use. Model providers are always at the beginning of a chain of transformations. They provide a certain model. The model can be created by arbitrary means, it does not refer to a meta model. Model transformers accept a model from a given set of model classes. This model is then transformed to an instance of another model class. Model consumers finally consume a model and create some kind of output.

Transformation wizard

The different parts of the transformation framework are automatically composed for the usage in the import wizard

of Eclipse. The framework therefore first lets the user choose a model provider from the registered ones. Next, the possible chains of transformation and the matching model consumers are displayed for the user to choose from.

Once the complete transformation has been selected, each part of the chain is configured if necessary and executed. Configuration of the parts is very flexible, as the parts can provide their own wizard pages. In order to simplify creation of ”filter transformations”, general purpose SelectionWizardPages are provided.

III. Building bridges from Eclipse to COM

To be able to use COM components from Eclipse, Java wrappers have to be created for the elements of the COM type library. This is realized by first providing a model for COM components in the model transformation framework, which is then transformed and finally consumed by a Java bridge generator.

In the following sections we will show the most challenging tasks that needed to be solved to achieve this.

Communication

In order to allow communication between Java and COM, an abstract core system (comcore) was implemented. It allows instantiation of components and interaction with them through abstract classes COMUtils and COMFactory. A concrete implementation is realised based on Eclipse's SWT implementation, which already includes basic functionality to communicate with COM components. Other implementations, e.g. based on AWT are conceivable.

Type mapping

For the components to be usable from Java, a mapping of data structures and types had to be realized, so that information could be exchanged between the systems.

In COM, all ”simple” types are wrapped inside a Variant. A Variant can therefore be an integer, a real number, a date, a color, and so on. These types can be mapped to standard Java types, e.g. int, float, java.util.Calendar and a specifically created OleColor-Class.

For complex types, that are represented as classes and interfaces in COM, matching classes are created in Java. COM enumerations are matched with a subclass of a dedicated abstract class that provides type-safe constants in Java 1.4 style.

Parameters

While normal (in-) parameters to functions can be easily mapped, in COM parameters can also be declared to be out-parameters. These can be modified by the component and must be passed back to the Java environment. Because primitive types cannot be modified (as seen from the caller) when passed in as parameters in Java, these parameters are mapped to an array of the type. When an array is passed as parameter, the values within the array can be modified, and the modification is also visible to the caller.

COM also allows for optional parameters. In order to compensate for this, multiple methods are created in the generated Java code.

Events

Besides the method invocation that is initiated from the client to the server (the component), it is also possible that the server needs to invoke methods on the client. This is used when the server wants to notify the client of events.

In order for the COM server to call methods on the client-side, the client registers for these events on component creation. This is automatically done by the generated code. Also Listener-, Adapter- and Event-classes are created, so that classes can easily be written to use this functionality.

Parameter mapping is in principle done the same way as when invoking methods the other way around. However, the parameters are not directly passed to the Listener, but embedded as attributes of the Event-class.

Resource Handling

Because COM employs a cooperative memory management, users must increment a reference counter when starting to use a component, and decrement it, when the component is no longer needed.

The created wrapper classes try to manage this as far as possible by themselves. The reference is increased on construction. Also there is a dispose method to deallocate the component.

But because Java does automatic memory management, usually no explicit destruction of an object is done. Objects are destructed only when the garbage collector runs. At that time each reclaimed object's finalize method gets invoked and the component can be disposed of.

If however the Java Virtual Machine is shut-down, no destruction of the objects is done. In order to prevent memory leaks for components running outside the current process's memory space, it is necessary to employ a shut-down-hook to be able to release the components.

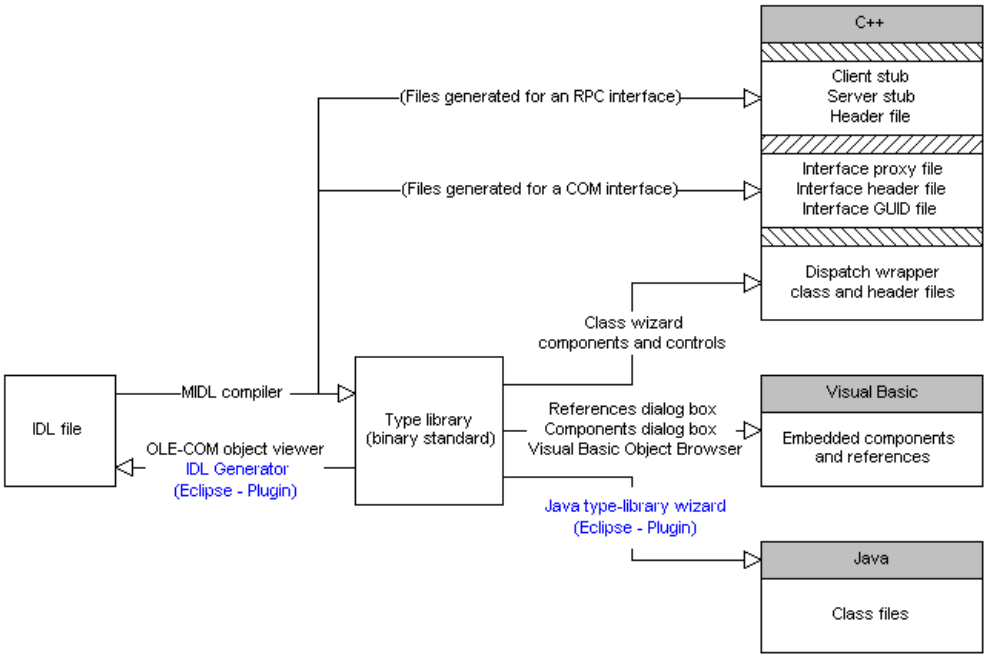
Although this technique works well, we recommend to explicitly dispose of no longer used components for efficiency reasons and for a clear separation of responsibilities.

Reading type-libraries

In order to first find COM components and then read the associated type libraries, the Windows API must be employed. This is done by a native library written in Delphi. It allows to read information about the COM components from the Windows registry and the referenced binary files.

From the gathered information a model for the type library is derived. This model can then be transformed and finally consumed in order to create a Java bridge, but e.g. also to recreate an IDL file.

The following figure shows the general usage of the type library and the newly created use cases:



Generating code

The code finally is generated using the Java Emitting Templates (JET) from Eclipse. The code generation is highly configurable using name generators. This way generated code can seamlessly be integrated in projects without breaking coding conventions.

Furthermore the generated classes include automatically generated Javadoc-comments. These are automatically extracted from the type library and the associated help-strings. This way the created classes blend in the usual coding style, and the documentation is also readily available right inside the IDE.

The generated code depends only on the comcore library and is therefore independent of Eclipse.

For each element of a component two classes are generated, an internal that manages the communication and a public class that optionally provides user defined functionality. A regeneration of the bridge only recreates the internal classes, thus the functionality is kept.

For graphical COM components (ActiveX controls) a SWT component is automatically created as a wrapper for the control. That allows seamless integration of the ActiveX controls within SWT applications.

IV. Evaluation and Case Studies

The bridge generator has been successfully used to create bridges to numerous COM-components. These include: Excel, Word, XMetaL, Internet Explorer, MSHTML, Nero Burning Rom etc., see table 1.

A bridge to the XMetaL component, that was previously implemented manually by Docufy within about 2 man-months, can now be generated within seconds with better documentation and (probably) less errors. The potential reduction of costs is therefore enormous.

V. Conclusion and Future Work

We have developed a framework to create bridges between different component models and implemented a bridge from the Java platform to the binary COM model. This bridge enables the developer to use and integrate well established, extensively tested, widely known components within a Java application as if they were native Java components.

While the development of these tools is already considered a great success, there are some points where future work can be useful:

- Embedding ActiveX-Control in AWT
  While being able to almost automatically embed ActiveX-Controls in SWT applications is already very useful, it would still be desirable to also be able to do so in AWT to reduce dependency on external libraries. This can be done, but requires natively implemented helpers, because the necessary tasks cannot be achieved with standard Java.
- Integration of .NET
  The Component Object Model has recently been superseded by the .NET framework. While currently all major components are still available as COM components and will remain so for quite some time, embedding components of a .NET architecture in Java is the next logical straight forward step.
  When specific .NET components are defined in a future version, the easiest way of transformation probably will be to wrap the .NET components in COM components. This can most easily be done by the author of the component, because here it only requires adding a special marking to the class that instructs the compiler to create the necessary type library.
  If this is not possible, e.g. because the source code is

TABLE I

CASE STUDIES: TIMES WERE MEASURED ON A PIV DUAL 3 GHZ WITH 3 GB MEMORY.

Type Library	LOC	Classes	Methods	Time
XMetaL Editor	13597	183	1281	1 sec
MS Word 11.0 Object Library	121056	1239	12484	9 sec
MS Excel 11.0 Object Library	222756	1613	22791	11 sec
MS HTML Object Library 4.0	525597	3823	62330	22 sec
Nero 1.4 Type Library	14108	291	1444	1 sec
MS Internet Explorer	12092	121	1148	1 sec

not available, a new .NET component can be created that is derived from the desired component. This newly created component can then once again be attributed to be available as COM-component, too.

REFERENCES

[1] Eclipse Corner Article: ActiveX Support In SWT. www.eclipse.org/articles/

[2] http://www.ezjcom.com/

[3] Automatische Einbindung von existierenden COM - Komponenten in Eclipse, Institut für Informatik, Universität Würzburg, Diplomarbeit, 2006

[4] http://www.nevaobject.com

[5] Microsoft Windows Platform SDK Collection for Windows Server 2003 SP1

[6] MOF Queries, Views, Transformations. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

Component Adaptation: Specification and Verification

Inès Mouakher, Arnaud Lanoix and Jeanine Souquières
LORIA – CNRS – Université Nancy 2
Campus scientifique
F-54506 Vandoeuvre-Lès-Nancy
Email: {mouakher, lanoix, souquier}@loria.fr

***Abstract*— In a component-based software development, components are considered as black boxes. They are only described by their interfaces expressing their visible behaviors. They must be connected in an appropriate way, through required and provided interfaces. To guarantee interoperability of components, we must consider each connection of a required interface with another provided interface. In the best cases, a provided interface – after some renaming – constitutes an implementation of the required interface. In the general cases, to construct a working system out of components, adapters have to be defined. They connect the required operations and attributes to the required ones. The interoperability between a required interface and provided interfaces through an adapter is guaranteed by the use of the B formal method with its underlying concept of refinement, and its powerful tool support, the B prover.**

I. Introduction

The idea underlying the paradigm of component orientation [1], [2] is to develop software systems not from scratch but by assembling pre-fabricated parts, as it is common in other engineering disciplines. As in object orientation, components are encapsulated, and their services are only accessible via interfaces and their operations. To really exploit the idea of component orientation, it must be possible to acquire components developed by third parties and assemble them in such a way that the desired behavior of the system to be implemented is achieved. A component is a unit of composition with contractually specified interfaces and explicit dependencies. An interface describes the services offered or required by a component without disclosing the component implementation. It is the only access to the informations of a component. The offered services by a component are described by a provided interface and the needed services are described by a required interface.

The success of applying the component based approach depends on the interoperability of the connected components. The interoperability can be defined as the ability of two or more entities to communicate and cooperate despite differences in their implementation language, their execution environment, or their model abstraction [3], [4]. The interoperability of two components concerns the compatibility between the required interface of one of the considered components with the provided interface of the other one. More precisely, three levels of interoperability have to be considered. The syntactic level covers static aspects of components interoperability. It concerns the interface signature: each attribute of the required interface must have a counterpart in the provided interface, but not necessarily vice versa; for each operation of the required interface, there exists an operation of the provided interface, such as their signatures are compatible. The semantic level covers the behavioral aspects of components interoperability. The protocol level deals with the allowed sequences of method calls that a component expects.

The specification of interfaces plays an important role in the verification of their compatibility. Most current interface modeling languages (IDLs), used in several component oriented platforms like JavaBeans [5], [6], CORBA [7], or COM [8], are limited for expressing signature (operation names, types, parameters) informations. They provide an insufficient information about component behaviors. Hence, one cannot insure trust in component based systems.

The availability of formal languages and tool support for specifying these interfaces is necessary in order to verify the interoperability of components. The idea to define component interfaces using B has been introduced in an earlier paper [9]. The semantics of the component services can be easily modeled by the B formalism. The use of the B refinement [10] to prove that two components are compatible at the signature and semantics levels has been explored in [11].

In this paper we focus on the generation of

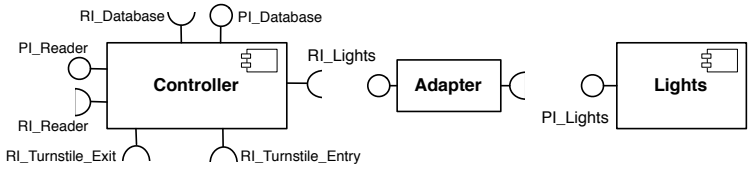


Fig. 1. A partial view of the architecture of the access control system

adapters that realize the matching between two or more existing components specified by their required and provided interfaces [12]. In fact, the need of adapters was recognized in the late nineties with a particular focus on automated adapter generation [13]. As a consequence, the expressive power of interface description techniques was limited as one had to ensure the decidability of the inclusion problem, which is necessary to perform automated interoperability checks; one could only generate adapters for specific classes of interoperability.

In our approach, we are not concerned only with specific classes of interoperability but with adapters in general. We propose to specify interfaces in terms of UML 2.0 diagrams [14]. These diagrams are then automatically transformed into B specifications [15], [16]. A model of a correct adapter is specified in B. The verification of the interoperability is automatically done by the B prover: the B model of the adapter is a refinement of the B model of the required interface using provided components. This verification process is done at the signature, semantic and protocol levels.

The rest of the paper is organized as follows. In Section II, we give an overview of our component-based development specification with the specification of the component interfaces. We then propose the definition of adapters in Section III and the verification of the interoperability between the connected components. The case study of a simple access control system serves to illustrate our proposition. We discuss related work in Section IV. The paper finishes with some concluding remarks in Section V.

II. Component-Based Development and Interface Specification

Our goal is to provide an approach for component-based software development that pays special attention to the question of how the interoperability between different components can be guaranteed. Components are specified as black boxes, so that component consumers can deploy them without knowing their internal details. Hence, component interface specifications play an important role, as interfaces are the only access points to a component. In this framework, adaptation between two components is a hard problem which has to be seen in an abstract way. We propose a methodology for specifying the required adaptation between two or more existing components by introducing a third component called Adapter. This new component is in charge, when possible, of mediating the interactions of the different components so that they can successfully interoperate.

The overall architecture of the system is expressed by a UML 2.0 composite structure diagram [14]. Such diagrams contain named rectangles corresponding to the components of the system. Components are connected by means of interfaces which may be required or provided. Required interfaces explicit context dependencies of a component and are denoted using the "socket" notation whereas provided interfaces explain which functionalities the considered component provides and are denoted using the "lollipop" notation.

Figure 1 presents a partial view of the architecture of the access control system where the access of authorized persons to a building is to be controlled. Persons have at their disposal access cards with identification information stored on it. There are two turnstiles, one at the entrance to the building, and one at the exit. At the entrance, there is also a card reader as well as a red and a green light. In the sequel, we will focus on the interaction between the Controller and the Lights components. The given requirement says that if the access is authorized, a green light is turned on, whereas, if the access is refused, a red light is turned on. More precisely, the green and the red lights cannot be turned on at the same time. The Controller component has several interfaces; one of its requested interface is related to the Lights component, namely RI_Lights. The Lights component has only one interface which is provided to a controller component, namely PI_Lights.

An intermediate component named Adapter has to be introduced: it is in charge to implement the links between the required interface of the Controller component and the provided interface of

an existing Lights component.

A. Specifying Components

For each component of the architecture, a specification of each interface has to be set up [11]. A component interface specification consists of a data model described by a class diagram with its different attributes and operations. The usage protocol of the interface is modeled by a Protocol State Machine (PSM); for each operation, its pre- and post-conditions are specified.

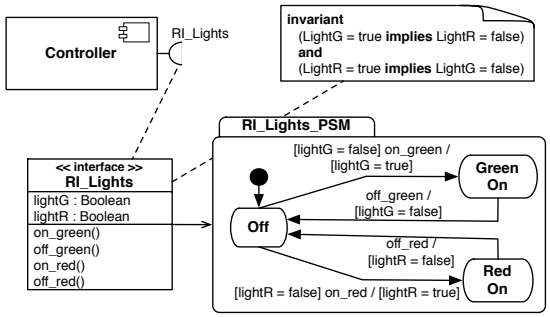


Fig. 2. The component Controller with its interface RI_Lights and its associated PSM

1) The Controller Component: With respect to its interaction with a Lights component, the Controller component requires an interface RI_Lights as presented Figure 2. A PSM is associated to this interface to specify its externally visible behavior, i.e. its usage protocol. Safety constraints on the required interface can be added by the way of an invariant expressed by an OCL annotation.

Different components corresponding to the behavior of the Lights component used in Figure 1 are available in the component library. Let us consider two of them, one called MultiLights corresponding to a component with the possibility of choosing its color and a second one called SingleLight which is a simple light.

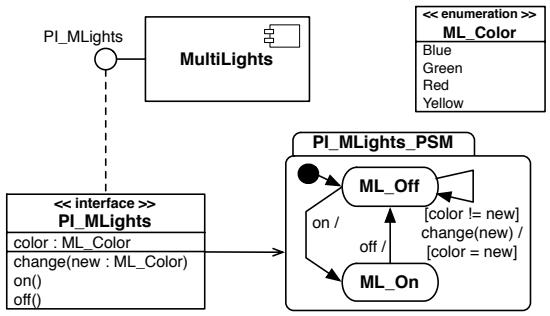


Fig. 3. The MultiLights component with its interface PI_MLights and its associated PSM

2) The MultiLights Component: The available MultiLights component offers, by the way of its provided interface called PI_MLights, the next functionalities: the light can be turned on and turned off. When the light is turned off, one can choose the light color from four predefined colors: blue, green, red and yellow. The UML specification of this component is given Figure 3, i.e. its provided interface, PI_MLights, and its associated PSM.

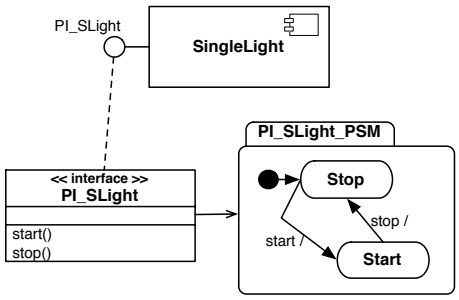


Fig. 4. The SingleLight component with its interface PI_SLight and its associated PSM

3) The SingleLight component: Another component named SingleLight is available in the component library. It corresponds to a simple light which can only be turned on and turned off. Figure 4 gives its UML specification, i.e. its provided interface PI_SLight and its associated PSM.

B. Derivation of UML Interface Specifications to B

UML 2.0 proposes expressive graphical notations to specify components and their interfaces. Nevertheless, it does not provide a suitable framework neither to support formal verifications nor to check the interoperability. The B formal method [10] supports an incremental development process, using refinement. Assembly clauses are also available [17], [18]. B methodology is based on proofs of invariance and refinement: the proof obligations are generated automatically (and often discharged) by support tools such as AtelierB [19] or B4free [20], an academic version of AtelierB.

Inspired from the derivation rules from UML 1.X class diagrams and state diagrams to B specifications [15], [16], we automatically translate the UML 2.0 interfaces and their associated PSMs into B specifications for checking their interoperability. Intuitively:

- a B model is derived from the interface,
- a set containing the "control" states of the associated PSM is added to the B model,
- each transition of the PSM is formalized by a B operation: pre- and post-conditions from a

transition become preconditions and substitutions into the B model,

- new preconditions (and substitutions) about the "control" states are incorporated into each B operation to model the PSM.

```
MODEL
  RI_Lights
SETS
  RI_Lights_STATES = {Off, GreenOn, RedOn}
VARIABLES
  lightG, lightR, lr_state
INVARIANT
  lightG ∈ BOOL ∧
  lightR ∈ BOOL ∧
  lr_state ∈ RI_Lights_STATES ∧
  (lightG = TRUE ⇒ lightR = FALSE) ∧
  (lightR = TRUE ⇒ lightG = FALSE)
INITIALISATION
  lightG, lightR, lr_state := FALSE, FALSE, Off
OPERATIONS
  on_green =
    PRE lightG = FALSE ∧ lr_state = Off
    THEN lightG := TRUE || lr_state := GreenOn
    END ;
  off_green =
    PRE lr_state = GreenOn
    THEN lightG := FALSE || lr_state := Off
    END ;
  on_red =
    PRE lightR = FALSE ∧ lr_state = Off
    THEN lightR := TRUE || lr_state := RedOn
    END ;
  off_red =
    PRE lr_state = RedOn
    THEN lightR := FALSE || lr_state := Off
    END
END
```

Fig. 5. B Models of the interface RI_Lights

Figures 5, 6 and 7 give the B model of the three component interfaces previously specified with UML.

III. Definition and Verification of Adapters

We must now prove that the controller can be connected either with the MultiLights component or with two versions of the SingleLight component. A process of proving interoperability between components using the B refinement is described in [11]. We can show that the interface PI_MLights of the MultiLights component is not a B refinement of the RI_Lights interface of the Controller component, because we have no direct matchings between their interface operations. The refinement proof fails, showing that the two components cannot be directly connected.

As specified in Figure 1, we introduce an adapter, i.e. a piece of code that takes place between the both considered components and compensates for the difference between their interfaces. Intuitively, this adapter proposes a way to connect provided operations and attributes to the required

```
MODEL
  PI_MLights
SETS
  ML_Color = {Blue, Green, Red, Yellow} ;
  PI_MLights_STATES = {ML_Off, ML_On}
VARIABLES
  color, ml_state
INVARIANT
  color ∈ ML_Color ∧
  ml_state ∈ PI_MLights_STATES
INITIALISATION
  color, ml_state := ML_Blue, ML_Off
OPERATIONS
  change(new) =
    PRE new ≠color ∧ ml_state = ML_Off
    THEN color := new
    END ;
  on =
    PRE ml_state = ML_Off
    THEN ml_state := ML_On
    END ;
  off =
    PRE ml_state = ML_On
    THEN ml_state := ML_Off
    END
END
```

Fig. 6. B Models of the interface PI_MLights

```
MODEL
  PI_SLight
SETS
  PI_SLight_STATES = {Stop, Start}
VARIABLES
  sl_state
INVARIANT
  sl_state ∈ PI_SLight_STATES
INITIALISATION
  sl_state := Stop
OPERATIONS
  start =
    PRE sl_state = Stop
    THEN sl_state := Start
    END ;
  stop =
    PRE sl_state = Start
    THEN sl_state := Stop
    END
END
```

Fig. 7. B Models of the interface PI_SLight

ones. Let us consider two components, one with a requested interface RI and the other with a provided interface PI. We define an adapter between these two components as a *new* component that *realizes* or implements the required interface RI, *using* the provided interface PI.

In order to verify the interoperability between RI and PI, we propose to specify the adapter as a B model. As shown in Figure 8, the B model Adapter_1

1) **REFINES** the B model of the required interface : the adapter is an "implementation" of the required interface and
2) **INCLUDES** the B model of the provided interface. The adapter uses "correctly" the operations of the provided interface to implement the required interface.

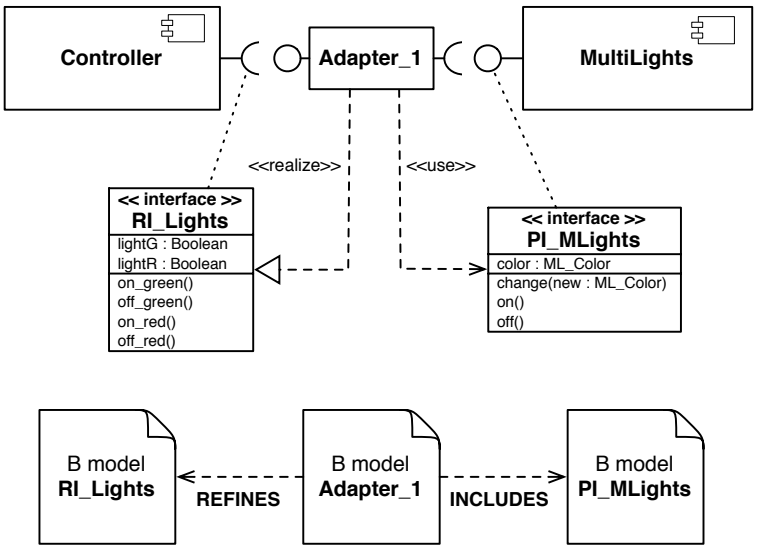


Fig. 8. An Adapter between Controller and MultiMLights components

The B specification of this adapter is composed of two main parts:

- The **INVARIANT** clause describes the links between the attributes of both required and provided interfaces (linking invariant). For each required variable, the adapter must specify how to obtain it in terms of the provided variables. The adapter must also express the links between the control states of both corresponding PSMs.
- The **OPERATIONS** clause is composed of all the operations of the required interface. The body of each operation is defined by the call of some operations of the provided interface linked together by a small part of code.

The use of the B method and its refinement mechanism allows us to verify that the proposed adapter is a "correct" refinement of the B model of the required interface RI, at the three levels of interoperability:

- the syntactic level is verified by the linking invariant concerning the attributes and by the correspondence of operations between the required interface and the adapter model,
- the semantic level is checked in terms of the B refinement: for each operation of the B adapter model, its precondition must imply, under its invariant, the precondition of the corresponding operation of the required model; the application of its substitutions must preserve the linking invariant,
- the protocol constraints expressed by PSMs have been transformed into preconditions and substitutions of the B operations. The protocol level is taken into account by behavioral constraints during the proof of the refinement.

It is to be noticed that behavioral and protocol constraints expressed in the provided interface and translated into the corresponding B specification are also taken into account. When an operation of the provided interface specification is used into the adapter specification (this is possible by the use of the B **includes** clause), its precondition is verified.

```
REFINEMENT
   Adapter_1
REFINES
   RI_Lights
INCLUDES
   PI_MLights
INVARIANT
   lightG = bool(color = Green
      ∧ ml_state = ML_On)
   ∧ lightR = bool(color = Red
      ∧ ml_state = ML_On)
   ∧ (lightG = FALSE
      ∧ lightR = FALSE ⇒
         ml_state = ML_Off)
OPERATIONS
   off_green =
      BEGIN off
      END ;
   on_green =
      IF color = Green
      THEN on
      ELSE
         LET col BE col = Green
         IN change(col) ; on
         END
      END ;
   off_red =
      BEGIN off
      END ;
   on_red =
      IF color = Red
      THEN on
      ELSE
         LET col BE col = Red
         IN change(col) ; on
         END
      END
END
```

Fig. 9. B Model of Adapter_1 between Controller and MultiLights Components

A. An Adapter for the MultiLights Component

As presented Figure 8, this adapter uses the provided interface PI_MLights implemented by the

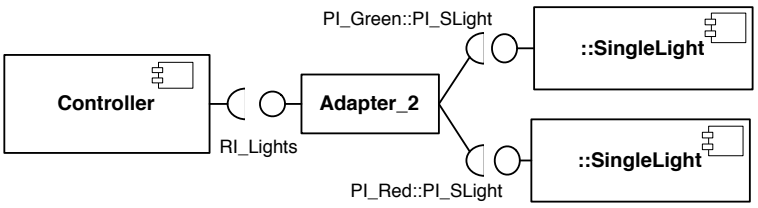


Fig. 10. Architecture of the system when using the SingleLight component

MultiLights component in order to realize the required interface RI_Lights of the Controller component. The B specification of Adapter_1 is given Figure 9.

- its invariant expresses the required interface attributes $lightG$ and $lightR$ in terms of the provided attributes $color$ and ml_state. It is to be noticed that lr_state does not correspond to an interface attribute. As previously explained, it has been introduced to express the control states,
- the **OPERATIONS** clause expresses how each required operation is implemented in terms of the provided operations. For example, the operation $on\_green()$ is defined by the choice of the suitable color before turning on the light; the sequential operation calls is expressed in B by a ";" statement and the choice, by an "if...then...else..." statement.

B. An Adapter for the SingleLight Component

Using the SingleLight component to realize the required interface RI_Lights of the Controller component is not immediate. The needed functionalities implies two colors for the lights, even if they are not on at the same time. As the SingleLight component provides only one light of one color, the adapter will use two instances of this component, namely PI_Green::PI_SLight and PI_Red::PI_SLight, to answer the required needs. The architecture of the system using two SingleLight components is presented Figure 10. It is the same as the general schema presented at the beginning of Section III, with the use of two provided interfaces.

The B specification of Adapter_2 is given Figure 11: it includes two instances PI_Green and PI_Red of the B model of PI_SLight. Once we have made the choice of using two different instances of this component, the definition of the link is immediate. The required attributes $lightG$ and $lightR$ and the operations are directly expressed in terms of the operations of the two instances of PI_SLight.

```
REFINEMENT                    OPERATIONS
  Adapter_2                   on_green =
REFINES                         BEGIN PI_Green.start
  RI_Lights                     END ;
INCLUDES                      off_green =
  PI_Green.PI_SLight,           BEGIN PI_Green.stop
  PI_Red.PI_SLight              END ;
INVARIANT                     on_red =
lightG =                        BEGIN PI_Red.start
  bool(PI_Green.sl_state = Start)  END ;
∧ lightR =                    off_red =
  bool(PI_Red.sl_state = Start)   BEGIN PI_Red.stop
                                  END
                              END
```

Fig. 11. B Model of Adapter_2 between Controller and two instances of SingleLight

C. Verification of the interoperability

We use the B4free tool [21] to verify that Adapter_1 and Adapter_2 refine the required interface RI_Lights. The verification results are as follows:

- B4free generates 14 obvious proof obligations for the B model of Adapter_1. All these proof obligations were proven automatically,
- B4free generates 4 obvious proof obligations for the B model of Adapter_2. All these proof obligations were proven automatically.

According to these results, we conclude that Adapter_1 **refines** RI_Lights using the MultiLights component and Adapter_2 **refines** RI_Lights using two instances of the SingleLight component. Consequently, each adapter we have considered implements the requested interface in terms of services provided by the corresponding component. The interoperability is verified at the signature, semantic and protocol levels.

IV. Related Work

The main drawback of component-based software engineering is the high cost of components deployment. This cost comes from the verification of the components interoperability and from the necessary definition of adapters.

In [22], [23], Zaremenski and Wing propose an interesting approach to compare two software components. It determines whether one required

component can be substituted for another. They use formal specifications to model the behavior of components and exploit the Larch prover to verify the specification matching of components.

In [24] a subset of the polyadic π-calculus is used to deal with the components interoperability, only at the protocol level. π-calculus is a very well suited language for describing component interactions. The main limitation of this approach is the low-level description of the used language and its minimalistic semantic. In [25], [26], protocols are specified using a temporal logic based approach, which leads to a rich specification for component interfaces.

Henzinger and Alfaro [27] propose an approach allowing the verification of interfaces interoperability based on automata and game theories: this approach is well suited for checking the interface compatibility at the protocol level.

Several proposals for component adaptation have already been made. Some practice-oriented studies have been devoted to analyze different issues when one is faced to the adaptation of a third-party component [28]. A formal foundation to the notions of interoperability and component adaptation was set up in [13]. Component behavior specifications are given by finite state machines which are well known and supports simple and efficient verification techniques for the protocol compatibility.

Braccalia and al [29], [30] specify an adapter as a set of correspondences between methods and parameters of the required and provided components. The adapter is formalized as a set of properties expressed in π-calculus. From this specification and from both interfaces, they generate a concrete implementable adapter.

Reussner and Schmit present adapters in the context of concurrent systems. They consider only a certain class of protocol interoperability problems and generate adapters for bridging component protocol incompatibilities, using interface described by finite parameterized state machines [31], [32], [33].

Our proposition takes benefits from object oriented notations : components are described using high-level UML 2.0 interfaces and their protocol state machines ; it also takes benefits from formal methods and their existing support tools, using existing derivation rules from UML diagrams to B specifications : we propose an adapter defined as a B specification and the interoperability verification is supported at the signature, semantic and protocol levels in the same framework using the B refinement.

V. Conclusion

To construct a working system out of components, adapters have to be defined. These adapters implement a required interface in terms of some provided interfaces. We have proposed a model of adapters expressed in the B formal method allowing to define rigorously the interoperability between components and to check it with support tools: an adapter is a correct refinement of the B model of the required interface using existing provided components. The interoperability is verified at the signature, semantic and protocol levels.

We want also to take into account more complex adapters. Generally, an adapter may use some provided interfaces offered by different components to realize some other required interfaces by others components. We are currently exploring different kinds of adapters in terms of specification matching [23]. We are working on alternative versions of compatibility and their mappings to refinement in B, in order to give patterns for the corresponding adapters in the same framework.

References

[1] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering*. Addison-Wesley, 2001.

[2] C. Szyperski, *Component Software*. ACM Press, Addison-Wesley, 1999.

[3] D. Konstantas, "Interoperation of object oriented application," in *In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition*. Prentice Hall, 1995, pp. 69–95.

[4] P. Wegner, "Interoperability," *ACM Computing Survey*, vol. 28, no. 1, pp. 285–287, 1996.

[5] *JavaBeans Specification, Version 1.01*, Sun Microsystems, 1997, http://java.sun.com/products/javabeans/docs/spec.html.

[6] *Enterprise JavaBeans Specification, Version 2.0*, Sun Microsystems, 2001, http://java.sun.com/products/ejb/docs.html.

[7] *The Common Object Request Broker: Architecture and Specification, Revision 2.2*, The Object Mangagement Group (OMG), Feb. 1998, http://cgi.omg.org/library/corbaiiop.html.

[8] *The Component Object Model Specification, Version 0.9*, Microsoft Corporation, 1995, http://www.microsoft.com/com/resources/comdocs.asp.

[9] S. Chouali and J. Souquières, "Verifying the compatibility of component interfaces using the B formal method," in *International Conference on Software Engineering Research and Practice*, 2005.

[10] J.-R. Abrial, *The B Book*. Cambridge University Press, 1996.

[11] S. Chouali, M. Heisel, and J. Souquières, "Proving Component Interoperability with B Refinement," in *International Worshop on Formal Aspects on Component Software*, H. R. Arabnia and H. Reza, Eds. CSREA Press, 2005, pp. 915–920, to appear in ENCTS 2006.

[12] G. Heineman and H. Ohlenbusch, "An evaluation of component adaptation techniques," Department of Computer Science, Worcester Polytechnic Institute, Tech. Rep. WPI-CS-TR-98-20, February 1999.

[13] D. D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors." *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 292–333, 1997.

[14] Object Management Group, "UML superstructure specification, v2.0," OMG, 2005.

[15] E. Meyer and J. Souquières, "A systematic approach to transform OMT diagrams to a B specification," in *Proceedings of the Formal Method Conference*, ser. LNCS 1708. Springer-Verlag, 1999, pp. 875–895.

[16] H. Ledang and J. Souquières, "Contributions for modelling UML state-charts in B," in *Third International Conference on Integrated Formal Methods - IFM'2002*, Turku, Finland, 2002.

[17] D. Bert, M.-L. Potet, and Y. Rouzaud, "A study on components and assembly primitives in B," in *Proceedingsof 1st Conference on the B method*, 1996, pp. 47–62.

[18] P. Bontron and M. Potet, "Automatic construction of validated B components from structured developments," in *ZB2000: Formal Specification and Development in Z and B*, ser. LNCS, J. P. Bowen, S. Dunne, A. Galloway, and S. King, Eds., vol. 1878. Springer-Verlag, 2000, pp. 127–147.

[19] Steria, *Obligations de preuve: Manuel de référence, version 3.0*.

[20] Clearsy, "B4free," Available at *http://www.b4free.com*, 2004.

[21] J.-R. Abrial and D. Cansell, "Click'n'Prove : Interactive Proofs Within Set Theory," in *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003*, ser. LNCS, D. Basin and B. Wolff, Eds., vol. 2758. Springer Verlag, 2003, pp. 1–24.

[22] A. M. Zaremski and J. M. Wing, "Signature matching: a tool for using software libraries," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 2, pp. 146–170, 1995.

[23] ——, "Specification matching of software components," *ACM Transaction on Software Engeniering Methodolology*, vol. 6, no. 4, pp. 333–369, 1997.

[24] C. Canal, L. Fuentes, E. Pimentel, J.-M. Troya, and A. Vallecillo, "Extending CORBA interfaces with protocols." *Comput. J.*, vol. 44, no. 5, pp. 448–462, 2001.

[25] J. Han, "A comprehensive interface definition framework for software components," in *The 1998 Asia Pacific software engineering conference*. IEEE Computer Society, 1998, pp. 110–117.

[26] ——, "Temporal logic based specification of component interaction protocols," in *Proceedings of the Second Workshop on Object Interoperability ECOOP'2000*. Springer-Verlag, 2000, pp. 12–16.

[27] L. Alfaro and T. A. Henzinger, "Interface automata," in *9 th Annual Aymposium on Foundations of Software Engineering, FSE*. ACM Press, 2001, pp. 109–120.

[28] D. garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard," *IEEE Software*, vol. 12, no. 6, pp. 17–26, 1999.

[29] A. Braccalia, A. Brogi, and F. Turini, "Coordinating Interaction Patterns," in *Symposium on Applied Computing (SAC'2001)*, A. Press, Ed., 2001.

[30] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," in *Journal of Systems and Software*, 2005.

[31] R. H. Reussner, "Adapting Components and Predicting Architectural Properties with Parameterised Contracts," in *Tagungsband des Arbeitstreffens der GI Fachgruppen 2.1.4 und 2.1.9, Bad Honnef*, W. Goerigk, Ed., May 2001, pp. 33–43.

[32] H. W. Schmidt and R. H. Reussner, "Generating adapters fo concurrent component protocol synchronisation," in *Proceeding of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, I. Crnkovic, S. Larsson, and J. Stafford, Eds., 2002.

[33] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reasoning on software architectures with contractually specified components," in *Component-Based Software Quality: Methods and Techniques*, A. Cechich, M. Piattini, and A. Vallecillo, Eds., 2003.

Profitability-oriented Component Specialization

Ping Zhu
School of Computing
National University of Singapore
Email: zhuping@comp.nus.edu.sg

Siau Cheng Khoo
School of Computing
National University of Singapore
Email: khoosc@comp.nus.edu.sg

***Abstract*— Partial evaluation has been applied to address the tradeoff between genericness and efficiency of off-the-shelf components. Except for a brute-force approach to creating all possible *binding-time signatures* for a component, little effort has been made in promoting the specialization of a component independent of its use context. In this paper we propose a framework pertaining to *independent component specialization*. We advocate a novel concept, named *profitability*, to capture specialization opportunities independent of how components are being deployed. We further define a specialization policy to make component specialization profitability-oriented. The conceptual profitability declaration is translated into a *profitability signature*, which is expressed in binding-time constraint form. A *profitable specialization component* is then developed, aiming to be assembled in various component-user's applications in place of the original generic component, as well as to be adapted to different specialization contexts. In addition to the merit of reusability, profitable specialization component also saves both the time and the space costs in dealing with multiple binding-time signatures, when compared with existing approaches in synthesizing those signatures. We believe that our framework would promote the use of partial evaluation in component-based software development.**

I. Introduction

Component-based software development, CBSD for short, advocates the philosophy that a software developer should reuse the generic off-the-shelf components to build complex and reliable applications [9], [14]. It has been proven to be an effective paradigm in contemporary software industry. However, the genericness of components results in degradation of system performance, which has been recognized in many areas such as operating systems and graphics. To address the tradeoff between genericness and efficiency, generic components are usually subject to specialization.

A common scenario of inefficiency can be described as: When partial input to a generic component keeps invariant for several runs, some identical computations are performed repetitively during run-time, which translates into a loss in performance. Partial evaluation [10], which is an automatic program specialization technique that specializes a program by aggressively propagating partial invariant input in earlier stage (*aka. compile-time*), has been an applicable solution to tackling this common inefficiency in recent years.

Partial evaluation has usually been performed with respect to the application into which the components are installed. We call this *main-program driven specialization*. However, it does not address the issues pertaining to *component specialization*, the emphasis of which is to specialize off-the-shelf components without taking into consideration of the latter's use contexts. To be more specific, an ideal component specialization should provide programmers means of *independent component specialization* and if possible, to deliver a new kind of specialized component which can still be adaptive to different specialization contexts.

There have been some initial investigations in this direction. Ulrik [16] validated the motivation for component specialization and raised some open questions related to this topic. Bobeff *et al.* in [2] created all possible binding-time signatures for a component and then synthesized those signatures with original generic component to produce a new component which is adaptable to various specialization contexts. But this approach admits information which is too general to be beneficial to specialization. To produce more informative binding-time signatures for a component independent of its use context, we propose a novel concept, named *profitability*, to capture the specialization opportunities of a component. We further define a specialization policy to make component specialization profitability-oriented. We design a profitability-oriented binding-time analysis to translate the conceptual profitability declaration into a *profitability signature* which guides the profitability-oriented component specialization.

The profitability signature of a component is used to generate a *profitable specialization component*, PSC for short. PSCis installed, in place of the original generic component, into component-user's application and adapted to different specialization contexts to produce a more efficient application in the sense of declared profitability. In addition to the merit of reusability, which is an essential requirement of any off-the-shelf component, PSC also saves time and space costs in dealing with multiple binding-time signatures for a single component by identifying reusable codes. This advantage distinguishes our framework from existing approaches in synthesizing multiple binding-time signatures.

We believe that the framework advocated here, which uses profitability to declare specialization opportunities independently and synthesize profitability signatures into PSC has a promising future in adopting partial evaluation to the entire process of CBSD.

The rest of this paper is organized as follows: Section II briefly introduces fundamental partial evaluation concepts which will be referred later in elaborating our framework. The limitation of conventional main-program driven specialization is also explained from CBSD point of view. Sec-

tion IV introduces the concept of profitability and define a specialization policy that govern the profitability-oriented component specialization. This is then followed by Section V which elaborates our proposal for producing PSC. Finally we summarize and conclude in Section VI.

II. Background

A. Partial Evaluation

Partial evaluation transforms program statements in two ways: Compiling away a statement whose computation is solely based on partial invariant input, and reconstructing a statement whose computation relies on varying input to form the specialized program. According to how the transformation decisions are made, partial evaluation is normally categorized into online partial evaluation and offline partial evaluation.

Online partial evaluation determines and performs the transformations in a single pass in the presence of concrete value of invariant inputs. On the contrary, offline partial evaluation is typically characterized by a preprocessing phase called *binding-time analysis*, BTA for short, in which the transformation decisions are made. The input to BTA is a *binding-time division*, which describes the binding-time values of program inputs. The binding-time domain BT_{val} normally comprises two binding-time values: `static` and `dynamic` which can be treated as abstract values of the invariant and varying parts of the input respectively from abstract interpretation point of view. BTA attempts to determine the binding-time values of the syntactic constructs at each program point and produces a two-level binding-time annotated program.

Offline partial evaluation can be further divided into compile-time specialization and run-time specialization based on when the concrete values for the `static` input are available. There exist numerous cases, *eg.* the file system operations, in which the concrete values of the invariants are not known until run time and can yet be available for extensive specialization. Compile-time specialization generates a specialized program (usually in source level) which is semantically-equivalent with source program in the condition that they share the same concrete static partial input. Run-time specialization, on the other hand, compiles or interpret the BTA result into a program (usually in binary level) which is linked with the concrete values of static input during run-time.

As for component specialization, it is preferable to adopt offline run-time specialization techniques since it is rare to establish concrete specialization values for an off-the-shelf component.

B. Creating and Synthesizing Binding-time Division

Existing partial evaluators have provided powerful support in facilitating programmers to declare binding-time division for a component.

- Tempo [12] manually declares binding-time divisions for function definitions written in C language in terms of *external specialization parameters* which include functions, global variables and data structure intended to be specialized.
- DyC [1] allows programmers to express their specialization intentions inside program codes by using a set of specialization primitives (eg. `make_static`, `make_dynamic`) and specialization policies (eg. polyvariant or monovariant specialization, different caching policies, etc.). DyC does not create binding-time divisions explicitly as Tempo does. The set of internal declarations are compiled by DyC's specific compiler into information guiding the production of final specialized code.

When a binding-time division is established, a component is ready for specialization. Tempo [6], [11] produces a *generating extension* based on the preprocessing analysis result. A generating extension of a program *P* is a program generator G_P that can be applied with different concrete values of the same static input parameters and produces the corresponding specialized code for program *P*. The generating extension is used in both compile-time specialization and run-time specialization.

C. Main-program Driven Specialization

Traditionally, partial evaluation of a program is initiated by passing specialization context to the program input. The objective of partial evaluation is then to propagate the specialization context, and perform static computation whenever possible. This approach, which requires initial specialization context, is termed as *main-program driven specialization* and does not apply directly to component specialization, due to the lack of specific use contexts associated with off-the-shelf components.

Consequently, Bobeff *et al.* choose to list down *all* applicable binding-time signatures associated with a component when they design their component specialization process. While simple in implementation, such approach does not leverage on the inherent structure of components; it also results in minimal capitalization in efficiency and reusability of specialized components.

Before proceeding to the elaboration of independent component specialization – the technique which we propose here, we highlight two terminologies which represent two categories of binding-time division:

- *Binding-time signature*: It is a binding-time division associated with an off-the-shelf component independent of component use context;
- *Specialization context*: It describes the binding-time information established at component use site.

III. Language

A component can be diversely implemented as a process, a function/procedure or a class. In this paper we begin our investigation of independent component specialization by setting a component as a C function definition which may be interrelated with other C function definitions. The terms *component* and *function definition* are used interchangeable in following sections. The concepts presented in our framework

also apply to other kinds of implementation of component. We will look into the extension in future.

The langauge used in this paper is a subset of C language, excluding the features of pointer, compound data structure, global variable. The evaluation strategy of function call is call-by value and every function call must return a value. The abstract syntax of this language is defined in Figure 1.

Abstract Syntax:

e	$\in$	**Exp**	Expression set
	::=	$const$	Constant
	|	id	Program variable
	|	$f\ (e^+)$	Non-void function call
	|	$e\ b_{op}\ e$	Binary expression
s	$\in$	**Stat**	Statement set
	::=	$id = e$	Assignment statement
	|	`int` id^+	Local-definition statement
	|	`return` e	Return statement
	|	`while` $e\ s$	Loop statement
	|	`if` e `then` s `else` s	Conditional statement
	|	$s; s$	Sequential statements
fd	$\in$	**FDef**	Function definition set
	::=	$id\ (v^+)\ \{s\}$	
p	$\in$	**P**	Program set
	::=	fd^+	

Domains:

$id,\ f$	$\in$	Identifier
$const$	$\in$	Real number
b_{op}	$\in$	$\{+, -, *, /, ==, !=, <, >, >=, <=, \&\&, \|\|\}$

Fig. 1. Syntax of C subset

IV. Profitability-oriented Analysis

Profitability is an abstract and declarative description that enables a programmer to declare points of specialization opportunities for a component independent of its use context. However, profitability is a subjective concept that is subject to the specific application domain and the programmer's specialization intentions, e.g. [7], [8]. Schultz et al[15] proposed a *specialization patterns approach*, which captures specialization opportunities arising from the usage of specific design patterns. Profitability can be detected with the help of profiling tools, such as Calpa [13]. Calpa is a tool for automatically generating the declaration for DyC. Calpa consists of two components: an instrumentation tool and a program analysis tool. The former instruments the source C program and collects some useful run-time information through a profiler. The latter generates some candidate specialization scenarios and applies a cost/benefit model on these scenarios based on information gathered from instrumentation tool to decide which are profitable scenario.

A. A Commonly-perceived Profitability

In this paper we advocate the use of profitability by considering a lightweight commonly-perceived opportunity for specialization, i.e. *the ability to specialize conditional tests away*. More specifically, the profitability for a component can be further divided into two categories:

1) *Direct profitability*: The ability to specialize a conditional test inside a function definition away. This direct profitability indicates the programmer's desire to make the binding-time value of conditional test `static`.
2) *Indirect profitability*: The ability to specialize a function call so that the (direct or indirect) profitability inside the called function definition may be materialized. This indirect profitability implies that the binding-time calling context established at the function call site should be an instance of the binding-time signature associated with corresponding function definition.

A *potential profitability point* is a program point which possesses direct or indirect profitability. There may be zero or multiple profitability points inside a function definition. We name the function definition without any conditional tests or function calls as a *plain function*. In other words a plain function does not carry any profitability.

In our framework this commonly-perceived profitability declaration is translated into a special binding-time signature expressed in binding-time constraint form. A binding-time constraint captures the relationship among parameters, and provides a concise representation of (possibly) multiple conventional binding-time signatures. The syntax of binding-time constraint is as follows:

ξ	::=	$b_1\ op\ b_2 \mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2$	// BT-Constraints
b	::=	$bt_v \mid$ `static` $\mid$ `dynamic`	// BT-Expressions
		$\mid b_1 \sqcup b_2 \mid b_1 \sqcap b_2$	
		where bt_v is a binding-time variable	
op	::=	$\sqsubset \mid \sqsubseteq \mid = \mid \sqsupset \mid \sqsupseteq$	// Relational operators

As usual, there are two constant binding-time values, `static` and `dynamic`, representing static and dynamic values respectively. They are ordered in decreasing staticness: `static` $\sqsubseteq$ `dynamic`. This ordering can be naturally extended to partial ordering over tuples of binding-time values.

A specialization process that fulfills all or part of the (direct and/or indirect) profitabilities available in a component is called a *profitable specialization*. Otherwise, it is termed as *unprofitable*. Our main thesis is that *the binding-time signatures for a component are produced to ensure the fulfilment of profitable specialization*. We name these profitability-oriented binding-time signatures as *profitability signatures*.

B. A Simple Example

Consider the following two interrelated function definitions.

```
f(x, y)
{
    if x > 0            /* profitability point 1*/
    return y+1;
    else return y-1;
}

g(x, y)
{
    if y > 0            /* profitability point 2*/
    return f(x,y);      /* profitability point3*/
    else return 0;
}
```

There are one and two potential profitability points in the function definitions f and g respectively. These are highlighted in the comments associated with the program code. The profitability signatures thus derived for these two function definitions are:

$$\begin{array}{ll} \texttt{f}: & bt_x = \texttt{static} \wedge bt_y \sqsupseteq \texttt{static} \\ \texttt{g}: & (bt_x \sqsupseteq \texttt{static} \wedge bt_y = \texttt{static}) \vee \\ & (bt_x = \texttt{static} \wedge bt_y \sqsupseteq \texttt{static}) \end{array}$$

The profitability signature of f says that as long as the binding-time value of parameter x is static, the profitability at point 1 can be fulfilled, regardless of the binding-time value of the parameter y. The profitability signature of g expresses a disjunctive condition in which the profitability at point 2 and point 3 can be fulfilled respectively.

Implementation-wise, profitability signatures are not generated by a typical forward-fashion binding-time analysis; instead, they are generated by propagating outwardly those binding-time requests at the profitability points.

The profitability signature of a plain function is encoded as `FALSE`. In other words, there is no satisfiable binding-time valuation of the parameters of a plain function.

C. Specialization Policy

To make the component specialization process strictly profitability-oriented, we set a specialization policy as follows:

> *If a specialization context for a function call f cannot achieve any profitable specialization of f, all the binding-time values of its arguments and return value will be classified as* `dynamic`*. Otherwise, the function call is specialized w.r.t the specialization context.*

Semantically, a specialization context SC for a function call f can lead to any profitable specialization of function definition f iff SC is entailed by f's profitability signature PS.

Definition 1 (Entailment Relation): Let $\mathcal{V}(BT_c)$ be the set of satisfiable binding-time valuations w.r.t the binding-time constraint BT_c. A binding-time division BTD_1 (i.e. a specialization context or a profitability signature) is said to be *entailed* by a binding-time division BTD_2 iff $\mathcal{V}(BTD_1) \subseteq \mathcal{V}(BTD_1)$.

For the following simple example:

```
g (x, y)
{
    .....
    tmp = f(x, y);
    if (x == 8)
    .......
}
```

If function f is a plain function, the call f(x,y) will be annotated as f (x : `dynamic`, y : `dynamic`), disregarding the binding-time value that x, as well as y, holds before this function call. The binding-time value of x will be restored to the value before the call during the handling of the conditional test `x==8`. The annotation of variables to `dynamic` value, which we call dyn-annotation, is similar to the introduction of "raise" expression in many existing partial evaluators.

It is worth pointing out that inside a component the fulfillment of a profitability at one profitability point may affect the fulfillment of another profitability at another profitability point. For example we have the following code:

```
g (x, y)
{
    .....
    tmp = f(x, y);
    if (tmp==8)
    .......
}
```

If the function definition f is a plain code, tmp will be annotated as `dynamic` whatever the specialization context of function call f is. Thereafter, the binding-time of tmp in the conditional test `tmp==8` will always be `dynamic`.

D. Generating Profitability Signatures

We have developed a profitability-oriented BTA to compute the profitability signature of a function definition. This analysis takes into consideration both data flow and control flow information, which is in line with the treatment found in conventional BTA. Moreover, the specialization policy also requires this novel BTA to automatically introduce dyn-annotation at various program points, when necessary. In order to handle inter-procedural function definitions satisfactorily, we require the analysis to operate over a library of interrelated functions definitions. This analysis returns:

- A global function environment F_{env} which maintains two pieces of information: A binding-time expression of every function's return value in terms of the binding-time values of its parameters; a set of the binding-time expressions describing the condition for those potential profitability points to become `static`. The latter information can be converted into a disjunctive binding-time constraint, a.k.a., profitability signature. This global environment is required for analysis of inter-procedural function definitions.
- At each program point a local binding-time environment ρ which records the binding-time expression of program variables at that point.

The detail of the analysis is omitted here for space limit.

V. Profitable Specialization Component

Before a generic component is installed into an application, it is desirable to take advantage of the knowledge of the static information available in the profitability signatures conveying profitability. In this section we demonstrate how to synthesize the generic component with its associated profitability signature to achieve independent component specialization.

A. From Component to Profitable Specialization Component

We aim to produce a kind of off-the-shelf component which synthesizes the original, generic, component and its associated profitability signature. We term this special component as *profitable specialization component*, PSC for short. The component users can thus plug this PSC, instead of the original component, into their applications. They can then generate efficient applications by specializing the applications w.r.t. the particular specialization contexts established in the applications' environments.

We envision the creation of a PSC from a component and its associated profitability signature for the following purposes:

1) It can be reused, in place of the original generic component, in multiple component applications, and
2) It can facilitate the generation of specialized applications through direct installation of the component to applications.

The first objective requires an PSC to be receptive to different user's applications. This can be hard to achieve via the existing partial evaluation technologies, since those technologies assume the availability of an *initial* specialization context. Specifically, when component specialization is performed via *profitability* declaration, the opportunity for specialization becomes less dependent on any initial specialization context. In fact, the focus of specialization, in creating an PSC, has shifted from "how to prepare a piece of code for specialization in a specific context propagated from an initial specialization context" to "how best to prepare a piece of potentially profitable code for specialization, independent of the specialization context." Consequently, this requires the PSC not to over-commit itself to any particular context.

The second objective, on the other hand, requires the PSC to be prepared in such a manner that enables full exploitation of a specific context provided by the application, when the PSC is plugged into the application.

In our framework, a component can therefore exist in three forms: Its original generic component form, its corresponding PSC which aims to support profitable specialization, and its specialized form which maximizes profitable specialization under a specific specialization context pertaining to an application.

B. Efficient Profitable Specialization Component

Profitability signature as described earlier expresses specialization opportunity in terms of a binding-time constraint, which is equivalent with a set of conventional binding-time signatures. Bobeff collected a set of *service generators*, which are produced correspondingly to a set of binding-time signatures, into a *component generator* to be reused in user's application [2]. A service generator is responsible for generating a specialized component when the concrete values for those static variables occurring in a binding-time signature are available. This has the functionality closely resemble that of generating extension in Tempo. Unfortunately, this approach cannot avoid the problems of code duplication and overlapping analysis occurring during the generation of the set of service generators. It would be much desirable that the process of generating PSC enables reuse of some specialized codes that are destined to be present in different application contexts.

During the process of generating PSC, it is instructive to perform a profitability-oriented action analysis. As found in partial evaluators such as Schism[3] or Tempo [5], [4], action analysis produces clear instructions that dictate the construction of specialized code. In essence, given a program, action analysis annotates each program points with an action. The action domain AC_{val} comprises of four action values: `eval`, `reduce`, `reconstruct`, and `identity`. The action value of each program construct is strictly determined by its binding-time value. During actual specialization, the specialization engine will simply conduct specialization on a program point based on the associated annotated action. In fact, action-annotated program is the desired product expected from run-time specialization, when no actual input value is available for conducting further specialization. Our efficient PSC will mainly be a component capturing the relevant action annotations, albeit in another form that aids reuse.

The tasks involved in creating and reusing an efficient PSC include:

- Identifying codes for *profitable and reusable* specialization;
- Constructing a specializer for those identified codes for effective specialization;
- Organizing the PSC so that it is receptive to further specialization.

C. Identifying Profitable and Reusable Specialization

Given a number $N \geq 2$, a piece of code within a function definition is said to be eligible for profitable and reusable specialization (PRS for short) if it remains profitable under at least N binding-time signatures.

- The code is a level-0 candidate for PRS if its associated function has exactly one binding-time signature.
  This means that the code, and in fact the entire function definition, can have exactly one action annotation at each of its program point, independent of the number of binding-time contexts the enclosing function definition may possibly provide.
- The code is a level-1 candidate for PRS if it has the same profitable binding-time information under at least N different binding-time signatures of the associated function definition.
  This means that the code may be action-annotated in more than one way, but a particular structure can be

constructed that enables co-existence of these action-annotated programs that enables reuse.

- The code is a level-2 candidate for PRS if its set of reducible subcode remains the same for at least N different binding-time signatures of the associated function definition. A piece of subcode is considered *reducible with respect to a specialization context/* if it is subject to either `eval` or `reduce` action under that context.
  This means that the code has more than one action tree, but there are at least N action trees which have either `eval` or `reduce` actions at the same set of sub-trees. A framework will be built with conditions attached to this set of sub-trees to facilitate generation of specialized code.
  Note that the set of level-1 candidates is a subset of the set of level-2 candidates.

D. Specializer

We have set the structure of an PSC to be a component that has been analyzed and annotated by our profitability-oriented action analysis. One of the main challenges of our construction is the ability to mitigate potential code explosion caused by overlapping action-annotations.

Our design of profitability-oriented binding-time analysis outputs at every program point a local binding-time environment recording the binding-time expression of program variables at that point in terms of the binding-time values of function parameters. This information can be further utilized to compute the action value of each syntactic construct. We associate each syntactic construct an action variable whose value is computed by using two auxiliary functions $\mathcal{A}_e$ and $\mathcal{A}_s$ which compute the action values for expression and statement respectively. (The readers can refer to [4] for the detailed implementation of these two functions.) The type signatures of these two functions are defined as follows:

$$\mathcal{A}_e :: \mathbf{Exp} \rightarrow \mathbf{BT_{exp}} \rightarrow \mathbf{AC_{val}}$$
$$\mathcal{A}_s :: \mathbf{Stat} \rightarrow \mathcal{P}(\mathbf{AC_{val}}) \rightarrow \mathbf{AC_{val}}$$

In Figure 2 we demonstrate through an example how PSC is constructed in our prototype. The figure depicts a function definition f and its corresponding PSC; the latter is an action-annotated program, in which actions associated with syntactic constructs are action variables expressed by the corresponding action functions $\mathcal{A}_e$ and $\mathcal{A}_s$.

The profitability signature for function definition f in Figure 2 to fulfill profitability is $bt_x = \texttt{static} \wedge bt_y \sqsupseteq \texttt{static}$. The known binding-time value of parameter x can be utilized to pre-compute the values of those action variables which are determined solely by the binding-time value of parameter x. To be more specific, the values of the variables $H_0, H_4, H_6, H_7, H_9, H_{11}, H_{13}$ can be frozen in the PSC. Thus we get a more efficient profitability-oriented PSC by identifying PRS codes w.r.t the profitability signature.

E. Organization of PSC

Since the objective of a PSC is to act on behalf of the original component in interacting with other codes available in an application, it can have the following two distinct usages:

1) It should support specialization activity to be conducted on the embedding application. *ie.*, it should, together with its surrounding codes, enable the application to be processed by a traditional partial evaluator, such as Tempo.
2) It should support direct execution of its code when its embedding application is executed. That is, PSC can be treated as a piece of code for execution. Usually, this entails the compilation of an PSC into a dynamically-linked library (DLL).

Currently, our research focuses on structuring PSC to support the first usage. We hope to support the second usage in the second stage of our research.

In order to subject a PSC to further specialization (concurrently with its surrounding code), we assume that compile-time partial evaluation is employed, and require PSC to be wrapped with a dispatcher such that the wrapped PSC provides the

Source Code — Local Environments

```
int f(int x, int y)                                          
{                                                            
   int l;                                    [(l, bt_l)]     
   l = 2 * x;                          [(x, bt_x); (l, bt_x)]
   if (l == 2)                               [(l, bt_x)]     
      l = l + y;                 [(y, bt_y); (l, bt_x ⊔ bt_y)]
   else l = l * y;               [(y, bt_y); (l, bt_x ⊔ bt_y)]
   return l;                          [(l, bt_x ⊔ bt_y)]     
}                                                            
```

Corresponding PSC

```
int f(int x^H0, int y^H1)
{
   int l^H2,H3
   l^H4 = (2^H5 * x^H6)^H7,H8
   if (l^H9 == 2^H10)^H11
      l^H12 = (l^H13 + y^H14)^H15,H16
   else l^H17 = (l^H18 * y^H19)^H20,H21
   return l^H22,H23
}
```

where

$\texttt{H}_0 = \mathcal{A}_e(x, bt_x)$	$\texttt{H}_1 = \mathcal{A}_e(y, bt_y)$
$\texttt{H}_2 =$ identity	$\texttt{H}_3 =$ identity
$\texttt{H}_4 = \mathcal{A}_e(l, bt_x)$	$\texttt{H}_5 = \texttt{H}_6$
$\texttt{H}_6 = \mathcal{A}_e(x, bt_x)$	$\texttt{H}_7 = \mathcal{A}_e(2 * x, bt_x)$
$\texttt{H}_8 = \mathcal{A}_s(l = 2 * x, \{\texttt{H}_4, \texttt{H}_7\})$	$\texttt{H}_9 = \mathcal{A}_e(l, bt_x)$
$\texttt{H}_{10} = \texttt{H}_9$	$\texttt{H}_{11} = \mathcal{A}_e(l == 2, bt_x)$
$\texttt{H}_{12} = \mathcal{A}_e(l, bt_x \sqcup bt_y)$	$\texttt{H}_{13} = \mathcal{A}_e(l, bt_x)$
$\texttt{H}_{14} = \mathcal{A}_e(y, bt_y)$	$\texttt{H}_{15} = \mathcal{A}_e(l + y, bt_x \sqcup bt_y)$
$\texttt{H}_{16} = \mathcal{A}_s(l = l + y, \{\texttt{H}_{12}, \texttt{H}_{15}\})$	$\texttt{H}_{17} = \mathcal{A}_e(l, bt_x \sqcup bt_y)$
$\texttt{H}_{18} = \mathcal{A}_e(l, bt_x \sqcup bt_y)$	$\texttt{H}_{19} = \mathcal{A}_e(y, bt_y)$
$\texttt{H}_{20} = \mathcal{A}_e(l * y, bt_x \sqcup bt_y)$	$\texttt{H}_{21} = \mathcal{A}_s(l = l * y, \{H_{17}, H_{20}\})$
$\texttt{H}_{22} = \mathcal{A}_e(l, bt_x \sqcup bt_y)$	$\texttt{H}_{23} = \texttt{H}_{22}$

Fig. 2. Example of PSC Construction

following two functionalities:

1) In collaboration with the execution of a traditional binding-time analysis, the wrapped PSC makes available the various binding-time signatures it supports (*ie.*, those profitable binding-time signatures). It thus eliminates the redundant time spent on re-analyzing the component for binding-time information.
   In contrast with the usual practice in binding-time analysis, the wrapped PSC does *not* attempt to generate new binding-time signatures in response to a new specialization context. Instead, it will choose the most appropriate binding-time signature to match the specialization context. This is because the wrapped PSC has known to have captured *all* profitable binding-time signatures, and we can safely deem other binding-time signatures to be *unprofitable* in leading to a profitable specialization.
2) In collaboration with the execution of action analysis, the wrapped PSC selects the relevant action-annotated meta-trees, and provides the appropriate instances of them for the existing application. This again eliminates the redundant time spent on performing action analysis on the component.

The result of the above interaction is a piece of uniformly treated application code, in which PSC is no longer present.

Of course, it is also possible to consider applying run-time specialization on the embedding application. In this case, the wrapped PSC still serves the above-mentioned functions. In addition, it also dispatches specialized code (for the component), to be combined with other specialized code to form the generating extension for the embedding application.

VI. Conclusion

In this paper, we discussed the existing research directions pertaining to component specialization, and noted the limitations of various approaches. One concern arisen from our study is the need to reconcile efficient execution with component reusability. We propose a novel framework to address this concern, under the name of *profitability-oriented specialization*.

This framework provides a refreshing view to program specialization, particularly partial evaluation, by shifting the focus of the specialization from the propagation of initial specialization context to the exploration of profitable specialization opportunity. Specifically, we propose to replace original components by their corresponding *profitable specialization components* (PSC). PSC exploits the opportunity for profitable specialization so that specialization profits can be shared across multiple applications of the component. PSC also prepares itself for further specialization when it will be installed into an application, thus avoiding redundant re-analysis of component code.

The framework is currently in its implementation stage. We intend to perform benchmarking to demonstrate its effectiveness and efficiency.

References

[1] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, New York, NY, USA, 1996. ACM Press.

[2] G. Bobeff and J. Noyé. Component specialization. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 39–50. ACM Press, 2004.

[3] C. Consel. A tour of schism: a partial evaluation system for higher-order applicative languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM Press, 1993.

[4] C. Consel and O. Danvy. From interpreting to compiling binding times. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 88–105, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[5] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Tempo: specializing systems applications and beyond. *ACM Computing Survey*, 30(3es):19–24, 1998.

[6] Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, New York, NY, USA, 1996. ACM Press.

[7] Jeffrey Dean, Craig Chambers, and David Grove. Identifying profitable specialization in object-oriented languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–96, 1994.

[8] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 93–102, New York, NY, USA, 1995. ACM Press.

[9] G. Heineman and W. Councill. *Component-Based Software Engineering C Putting the Pieces Together*. Addison-Wesley, 2001.

[10] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993.

[11] Julia L. Lawall and Gilles Muller. Faster run-time specialized code using data specialization. Technical Report Research Report RR-3833, INRIA, December 1999.

[12] A.-F. Le Meur, J.L. Lawall, and C. Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17(1):47–92, 2004.

[13] Markus Mock. *Automating Selective Dynamic Compilation*. PhD thesis, Department of Computer Science & Engineering, University of Washington, August 2002.

[14] Jean-Guy Schneider and Jun Han. Components: the past, the present, and the future. In *Proc. of Ninth International Workshop on Component-Oriented Programming*, 2004.

[15] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Specialization patterns. In *ASE '00: Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 197, Washington, DC, USA, 2000. IEEE Computer Society.

[16] Ulrik Pagh Schultz. Black-box program specialization. In *Proceedings of the Workshop on Object-Oriented Technology*, page 187, London, UK, 1999. Springer-Verlag.

Putting Components into Context

Supporting QoS-Predictions with an explicit Context Model

Steffen Becker
Software Design and Quality
University of Karlsruhe
Email: sbecker@ipd.uka.de

Jens Happe
Graduate School Trustsoft
University of Oldenburg
Email: happe@ipd.uka.de

Heiko Koziolek
Graduate School Trustsoft
University of Oldenburg
Email: koziolek@ipd.uka.de

***Abstract*— The evaluation of Quality of Service (QoS) attributes in early development stages of a software product is an active research area. For component-based systems, this yields many challenges, since a component can be deployed and used by third parties in various environments, which influence the functional and extra-functional properties of a component. Current component models do not reflect these environmental dependencies sufficiently. In this position statement, we motivate an explicit context model for software components. A context model exists for each single component and contains its connections, its containment, the allocation on hard- and software resources, the usage profile, and the perceived functional and extra-functional properties in the actual environment.** [1]

I. Introduction

One of the most cited definitions of a software component originates from the first WCOP and is stated as follows in [1]:

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

Thus, a component has to specify its context dependencies, but it remains a bit vague what is actually part of the context. It is evident that there are more relationships between a component and its context than only its provided and required interfaces. For example, the functional and extra-functional properties of the component depend on the underlying hardware. This relationship is important when predicting the behaviour and the quality of the assembled system, but it is usually not specified as an explicit context dependency. Coming back to the definition, the second important fact is that a component is composed by third parties. As a result, the component developer does not know about the context in which the produced component is placed. It is the responsibility of the system architect to assemble components to build new components or systems.

To analyse functional and extra-functional properties of a component based system, additional information is needed besides the structure of the system in terms of components and their interconnections. There are four major influencing factors perceived by the user of a component (see figure 1):

1) The implementation of the component, e.g., the selection of the used algorithms.
2) The quality of required services, e.g. calling a slow or a fast service will result in a different performance for the provided service perceived by a user.
3) The runtime environment the component is deployed on. This includes the hardware and system software like the operating system and middleware platforms.
4) The usage of the component, e.g. if the component has to serve many requests per time span it is more likely to slow down.

To denote the QoS of a component more formally, we could say that the quality of a service S can be characterised as a function taking these dependencies as input. However, the implementation of a component has to be handled differently, since it is fixed by the component developer as opposed to the other parameters, which are determined during its deployment. Hence, the function specifying the context dependencies is defined independent of the implementation:

$$q_{impl} : \mathcal{P}(S) \times DR \times UP \rightarrow Q$$

where $\mathcal{P}(S)$ is the domain of the set of external services of service S, DR specifies the deployment relationship saying which component and connector is deployed on which part of the execution environment and UP describes the usage profile. As a result, the function yields a value in the domain of the investigated quality metric Q.

From the introduced function, we can learn that functional and extra-functional properties cannot be specified within component specifications as a fixed value. This fact has to be reflected in a component model, which generally states what needs to be specified when describing a component. We propose using parametric contracts (see section II) and the explicit modelling of the component context in this position statement as a solution to this problem.

Related work in this field of research is coming from two areas. One area is concerned with the construction of component models and algorithms using information in model instances. An example for such a model is the SOFA component model [2]. Other related work comes from the analysis of the influences of the different context parts. We give one example paper per influence factor in the following. The usage profile is investigated by Hamlet et al. [3], the deployment context by Liu et al. [4] and external services by Firus et al. [5].

[1]This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

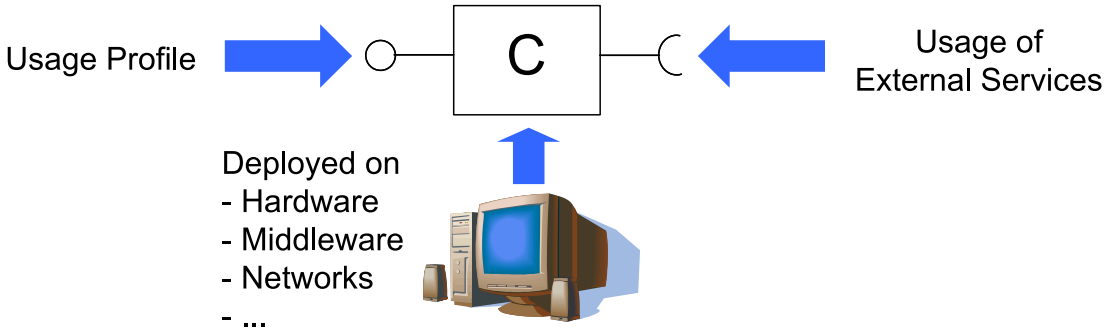


Fig. 1. Influences on quality

The contribution of this paper is an explicit model of context for each component usage in order to support QoS predictions. Contextual influences on components are illustrated by the means of examples. In each example, we demonstrate how the additional information specified in the context model can be used for QoS predictions.

The position statement is structured as follows. After this introduction, we give some foundations on parametric contracts in section II. Section III highlights several context influences by means of examples. Some model attributes of the context are presented in section IV. The open issues are discussed in section V. Section VI summarises this position statement.

II. Parametric Contracts

Additional information on the inner component structure is needed, to predict the QoS of a component that is embedded into a concrete context. This information has to be specified in a way such that it uses the influences of external services, the execution environment, and the usage profile (see Fig. 1) as input, to derive the QoS attributes perceived by users of the component. Parametric contracts [6] allow us to create a component QoS specification that is independent of those influences and can be evaluated when the environment of a component is known.

Parametric contracts characterise the intra-component dependencies of provided and required interfaces with so-called *service effect specifications*. A service effect specification models how a provided service calls the services specified in the required interfaces. Thus, it is an abstraction of the provided service's control flow. A service effect specification can be a signature list or a set of call sequences, depending on whether the execution order is important or not. Call sequences can be described by any kind of language specification.

Figure 2 illustrates parametric contracts for QoS attributes. In the service effect specification shown there as a finite state machine, transitions represent calls to external services that are specified in the required interfaces. States represent internal component code.

Both can be associated with QoS annotations, which describe the relations of the context model attributes and the perceived functional and extra-functional properties in a parametric manner. For example, the execution time of a service

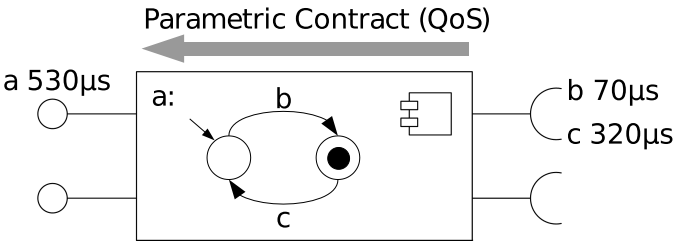


Fig. 2. Parametric Component Contract.

is specified by a number of abstract work units executed on a computational resource. The actual computational resource is described in the component's context. The specification of the resource contains information on how many work units the resource can process in a given time span. With this information, the execution time of the service on that resource is determined.

Parametric contracts also allow to model the influence of external services. For example, the component shown in figure 2 provides a single service called `a`, which requires two services `b` and `c`. The execution time of `a` can be calculated from the execution times of `b`, `c`, and the execution time of the internal component code. The computation requires the service effect specification of `a` depicted in the component. In this case, the only variable parameter of the component specification are the external services.

III. Context Influences

Since QoS attributes of a component are strongly influenced by the environment the component is deployed in, the actual delivered QoS can only be determined knowing all influencing factors. We identified three aspects defined during system design that frame the context model: Assembly, hierarchy, and allocation.

A. Assembly - Horizontal Composition

An assembly specifies which components are used within a system and how they communicate. Within the assembly, the required interfaces of a component are connected to provided interfaces of another component. That way it is determined which concrete external services are called by a component.

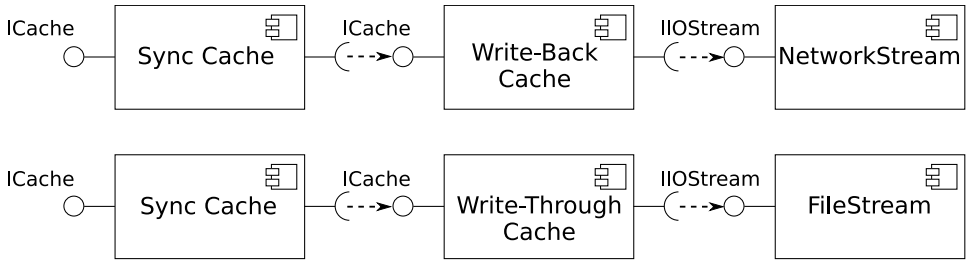


Fig. 3. Component assembly.

The assembly thus determines one of the influencing factors shown in figure 1.

A component can be used multiple times within a single assembly. Figure 3 illustrates this with a simple example. Three different types of components exist in the assembly shown there. On the right hand side, we have two I/O components that either manage the access to a file or network connection. Two different kinds of caching components that implement different caching strategies are shown in the middle. The `SyncCache` component on the left-hand side allows multiple tasks to access the caches concurrently without producing an incorrect state of the connected single-threaded caches.

The same component (`SyncCache`) is inserted at two different places within the assembly. Both representations of the component are connected differently. Thus, users or other components that call the services provided by the different component representations will experience different QoS on the provided interfaces of the respective component representations. This is caused by the different caching strategies and I/O devices used by the `SyncCache` components. Modelling the component context explicitly allows us to hold the information on the diverse connections and the resulting quality attributes without changing the component specification.

B. Hierarchy - Vertical Composition

Besides the assembly, another important part of the context is the hierarchy in which a component is used. In figure 4, a composite component (`BillingManager`) is depicted which has been designed to create bills and store each one in a single PDF (Portable Document Format) file. The component is additionally supposed to write a summary of all the created bills as PDF file. Hence, the component `PDFCreator` is used in two different places. Notice however, that this kind of usage is usually unknown to the creator of the outer composite component. For her, the inner component (`BillCreator`) is a black box. She does not know the internal details and, hence, the usage of the inner `PDFCreator` is hidden.

In this case, the `PDFCreator` component is used in different contexts on different hierarchy levels. Note, that this only makes sense if the underlying component model supports hierarchical components at all. Considering parametric contracts, both components might offer different characteristics (QoS, functions offered, etc.). Additionally, they are *used* differently in their contexts. The `PDFCreator` of the inner component produces bills with less pages than the summary PDF file created by the outer `PDFCreator`.

C. Allocation

An explicit context model is especially advantageous to model the allocation of components on hardware and software resources. Figure 5 depicts a system that uses replicated components to fullfil requests. In our example, server I is assumed to be slow and server II is assumed to be fast. Hence, the workload is not distributed equally, but 30% of the requests are directed to server I and 70% are directed to server II.

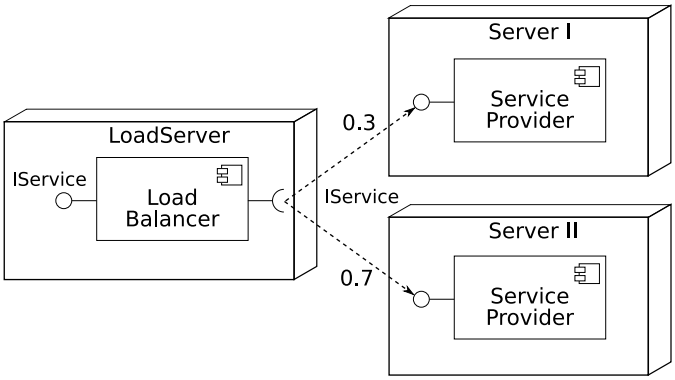


Fig. 5. Component allocation.

Here, we see several context influences. We have two copies of the same component allocated on different machines and, thus, in different contexts. The workload of each replicated component is different because of the distribution strategy. The processing power available to both replicated components is varying with the underlying hardware systems. However, both components are connected with an identical logical link going from the required interface of the workload balancer to the provided service of the replicated component. But again, each of these logical connections is most likely using a different physical communication channel, i.e., different network links.

IV. An Explicit Context Model

In the previous section, we identified different input factors of the provided QoS of the *same* component in various contexts. In order to cope with these factors, we encourage the explicit modelling of the context when using components. Table I summarizes the attributes of our context model.

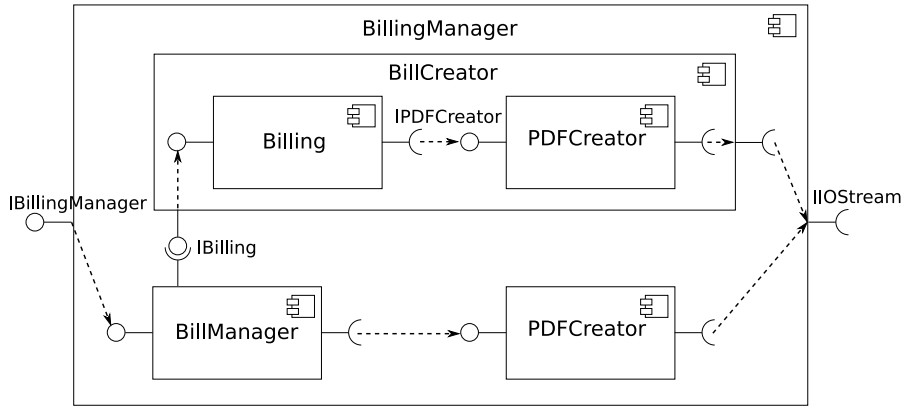


Fig. 4. Component hierarchy.

	Composition	**Allocation**	**Usage Profile**
Specified	Connection Containment	Deployed-on relation Execution environment - Concurrency, security, ... - Container properties - Component configuration	System usage: - Call probability - Call parameter - Workload
Computed	*Functional* Results of parametric contracts	*Non-Functional* QoS-Attributes	Inner Component usage

TABLE I
PROPERTIES OF THE CONTEXT

We arrange the tabular according to two dimensions. One is dividing the attributes into ones that have to be specified during the design process and such that can be computed or predicted using the former ones. The other dimension divides the properties into those defined by system composition, those determined by the allocation to an execution environment, and those determined by the actual usage characteristics of the components.

During component and/or system composition, assembly and delegation connectors are used to describe the communication channels of the components used within the architecture. By doing so, the QoS attributes of the external services of the components can be determined and their influence on the QoS attributes of the component under consideration can be derived. Furthermore, the actual available services can be computed by using parametric contracts [7]. This is especially important if not all required interfaces of a compenent are connected. In this case, parametric contracts allow us to determine the provided services that are not affected by the unbound interfaces.

Attributes specified in the allocation dimension contains information on the actual hardware (CPU speed, cache sizes, available memory, available bandwidth, ...) and on system software (details on the used middleware, virtual machines, container configurations, ...). Using this information, it is possible to estimate QoS properties, like the actual execution time of given code segments on the specified runtime environment.

The specification of the usage profile contains probabilities for calling specific services, probability distributions on the actual parameter characteristics, or the request arrival rate. From this information, the usage profile of the components connected to the required interfaces of the components with system boundary interfaces can be computed. However, it depends on the capabilities of the analysis method, which information has to be specified and which QoS metrics can be derived.

Note, that our context model differs from existing approaches in context-aware computing (e.g., [8]). There, the context is used to describe dynamic aspects of a system that vary during runtime, like location and user awareness, whereas our approach is concerned with static aspects that are fixed during design time, like the deployment environment.

Deployment descriptors known from component frameworks, like J2EE, contain information similar to our context model. They specify the connections of all components within an architecture and do not contain information about the execution environment explicitly. Opposed to this, our context model is specified individually for each component. Furthermore, deployment descriptors describe the architecture with its components and connections as a flat structure, whereas context models allow a hierarchical composition of components. In contrast to deployment descriptors, our context model contains information about the functional and extra-functional properties of components, such as computed parameteric contracts and QoS attributes that depend on the environment.

V. Open Issues

The explicit identification and modelling of a component's context allows us to describe the dependencies of functional and extra-functional properties on the usage context of a component. However, some questions are still open.

a) State of Component Protocols:: Parametric contracts model dependencies between component interfaces with protocols which, therefore, have a state. In some cases, this leads to difficulties when analysing the interoperability of components communicating via an interface. Assume an interface provided by component `A` is accessed by components `B` and `C`. If `B` changes the state of the interfaces by calling a service, does component `C` see the changes or does it have its own view on `A`? This question cannot be answered in general. In some cases, components share the state, e.g. when using the Singleton pattern, in other cases they don't. To solve this issue, additional information in the component model is required. Ports and interfaces with cardinalities seem to be a promising concept.

b) Identification of the Relevant QoS Parameters:: To achieve accurate QoS predictions, the parameters influencing the attributes of interest need to be identified. A lot of work has already been done in this context, in UML for example by the definition of the UML SPT profile [9]. However, the existing work needs to be reviewed, to be extended and the identified parameters needed to be specified within our component model. Furthermore, means to analyse and derive the desired performance metrics from the input values have to be found and/or developed.

c) Implementation:: The concept of a component's context model needs to be implemented within the Palladio component model. In our case, this means that we have to define a transformation from the model of a component architecture to an implementation, simulation, or analytical model.

VI. Conclusion

In this position statement, we motivate the explicit modelling of the context of software components in component models. With various examples we demonstrate that this information can be used by algorithms based on parametric contracts to predict functional- and extra-functional properties of the components. We identify initial attributes which are part of the context. Finally, open issues with models for component contexts are highlighted.

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] F. Plasil and S. Visnovsky, "Behavior Protocols for Software Components," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1056–1076, 2002.

[3] D. Hamlet, D. Mason, and D. Woit, *Component-Based Software Development: Case Studies*, ser. Series on Component-Based Software Development. World Scientific Publishing Company, March 2004, vol. 1, ch. Properties of Software Systems Synthesized from Components, pp. 129–159.

[4] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen, "Designing a Test Suite for Empirically-based Middleware Performance Prediction," in *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, ser. Conferences in Research and Practice in Information Technology, J. Noble and J. Potter, Eds. Sydney, Australia: ACS, 2002.

[5] V. Firus, S. Becker, and J. Happe, "Parametric Performance Contracts for QML-specified Software Components," in *Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA)*, ser. Electronic Notes in Theoretical Computer Science. ETAPS, 2005.

[6] R. H. Reussner, S. Becker, and V. Firus, "Component Composition with Parametric Contracts," in *Tagungsband der Net.ObjectDays 2004*, 2004, pp. 155–169.

[7] R. H. Reussner, "The Use of Parameterised Contracts for Architecting Systems with Software Components," in *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, W. Weck, J. Bosch, and C. Szyperski, Eds., June 2001.

[8] A. Lopes and J. L. Fiadeiro, "Context-Awareness in Software Architectures," in *Software Architecture, 2nd European Workshop, EWSA 2005, Pisa, Italy, June 13-14, 2005, Proceedings*, ser. Lecture Notes in Computer Science, R. Morrison and F. Oquendo, Eds., vol. 3527. Springer-Verlag, Berlin, Germany, 2005, pp. 146–161.

[9] Object Management Group (OMG), "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms," http://www.omg.org/cgi-bin/doc?ptc/2005-05-02, May 2005.

An Architectural Component-Based model to solve the Heterogeneous Interoperability of Component-Oriented Middleware Platforms

Francisco Domínguez-Mateos
Languages and Systems Department
Rey Juan Carlos University
Email: francisco.dominguez@urjc.es

Raquel Hijón-Neira
Languages and Systems Department
Rey Juan Carlos University
Email: raquel.hijon@urjc.es

***Abstract*— This work has two main aims. In the first place, we expose a solution to the components heterogeneous interoperability running into different Component-Oriented Middleware Platforms. In order to achieve this, we propose an architectural component-based model using design patterns, abstract machines and reflection. As a result of this model development, new systems named middlebuses have arisen. The main advantage is its high extensibility, since expanding the system to interoperate with a new middleware platform is as easy as to create and add a new component. Moreover, this component is built by implementing only four interfaces. All things considered, a middlebus is a software system that behaves as a network communication bus but in the middleware level. This is to say, that the interconnections of heterogeneous middleware platforms allow them to interoperate with each other in a bidirectional and transparent way.**

I. Introduction

A heterogeneous system is the one that is made of many unrelated parts; including interfaces, software, hardware, operating systems, programming languages, and network protocols.

We can find lots of solutions to heterogeneity problems at all levels of computer technology. For instance, the internet is an example of success at network and transport protocol layers of heterogeneous communication solution, the key to this success is standardisation.

In one hand, Component-Oriented Middleware Platforms arise as a way to solve the operating systems, programming languages, platform, and networking heterogeneity of inter-processes component communication. On the other hand, the boom of middleware systems has transferred the problem of heterogeneity to this level. In this moment, two different middleware platforms such as CORBA and Java RMI can not interoperate. In this case, none of the existing platforms have won the standardization's war.

Due to the discovery of middleware heterogeneous problem [5], many techniques have been proposed. In previous works [3] we have analyze twenty one different technologies. The results of this study are:

- A taxonomy definition inspired on other heterogeneous communications technical solutions, which were already working properly in the realm of networking communications such as: adapters, bridges, tunnels, switches and buses.
- Identification of requirements needed to build heterogeneous interoperable platforms solutions.

In other research on abstract machines [2], we discovered how virtual machines and reflection can support interoperability since there is an isomorphic structure and behaviour between abstract machines and middleware platforms.
The above mentioned works steered to a first Object Oriented Reflective Abstract Machine Architectural Model not based on components. This architectural model specifies the creation of software designs to implement systems that allow heterogeneous interoperability among applications with most of the requirements identified in [3] and described bellow.

With the idea of getting more easily extensible and modular systems here we expose a second version that is an Architectural Component-Based Model. Besides, extending the system to interoperate with a new middleware platform is as easy as creating and adding a new component by implementing two interfaces.

The next sections of this paper expose more deeply all this work. Firstly, section two and three explain which the requirements of a middlebus are and why we have chosen a reflective abstract machine as the base of our work. Secondly, section four shows the middlebus behaviour. In the next section, a first architectural model and a component-based model approach are described and finally, in a conclusion section, we argue about extensibility improvement achieved and future research lines opened.

II. System requirements

The architectural model patter design should allow the creation of architectonical instances that enable the development of systems with the following requirements:

- Heterogeneity, which allows heterogeneous language interoperability, different operating systems, and heterogeneous middleware platforms.
- Transparency, this is to say that elements from a specific platform can see elements from other external platforms as if they were in the same one.
- Not intrusive technology, there is no need to do any changes neither to the existing applications nor to the new ones designed for a specific platform.
- Extensibility, the system could be easily extended with future communication mechanisms. Therefore, it would be possible to build a new platform extension in other different platform.
- Automated adaptation, proxies or wrappers are created automatically only if they are needed, without any human intervention or configuration.
- Uniformity, external objects do not differ from internal ones.
- Interaction, we can interact with other applications or extend the model with additional services for monitoring, configuring, scheduling, etc. What is more, the programmer does not need to learn any other different platform from the one he is used to work with.

III. Isomorphism between abstract machines and middleware platforms

To begin with, and as we stated above in [2], we researched some of the most outstanding commercial implementations of abstract machines, namely: the Smalltalk virtual machine, Java virtual machine and the .NET underlying virtual machine, which consists in the CLI and LI definitions. In addition, some others virtual machines are: Carbayonia from the research project Oviedo3, and Nitro from the research team LABTOO, which were deeply surveyed. In all of them, the same basic structural features appeared. Above all, the most interesting issues in our work are: firstly, memory areas to locate items and to reference them have been identified. Secondly, techniques to represent the data that are going to be stored and executed have also been identified. Finally, there are invocations facilities allowing the objects to interact with each other, that is to say, allowing message sending.

In the second place, in [3] we studied deeply some others most outstanding middleware distributed object platforms such as: CORBA, DCOM and Web Services. In this case, we discovered some common facility services, from which we are interested in: location facility, introspection and dynamic invocation.

Finally, we are ready to talk about the resemblance between virtual machines and middleware platforms, since:

- Location facility in middleware is equivalent to location in memory storages in abstracts machines.
- Introspection describes objects in middleware as virtual machines memory structures do.
- Dynamic invocation allows message sending from objects in the middleware as invocations facilities from virtual machines do.

Actually, these three common features define the necessary high conceptual interface to implement and give live to a runtime object system. We may realise that in the back-office of each object middleware there is a runtime object system working. Therefore, we thought that by using virtual machines or an abstract machine model at design level we would solve the interaction heterogeneous problem easily.

IV. Behavioural description

In this section, we show a high level insight of the architectural model; and in the following, a software engineering overview through a Model Architectural view based on the Unified Development Process [4]. We will discuss both: a structural diagram and a behavioural diagram.

Figure 1 shows the systems we can build from our architectural model. We call these kinds of systems middleware bus, middlebus, or simply bus; this name comes from its behaviour, which resembles the one of a bus in other communication layers. The bus is named LIIBUS. In the illustration there are some middleware connected to the bus, such as: DCOM, RMI Java, CORBA, RPC and SOAP.

The objects of each platform will see other platform objects as if they were in the same platform. And what is more important, each one of them can communicate with the others without any intrusive adaptations or bridges.

Once a new platform is installed on the bus, it is ready to be used by objects of all other platforms and also objects of this new platform can use objects of other platforms. This means that there is a bidirectional communication. The foundations of this bus are object oriented reflective abstract machines.

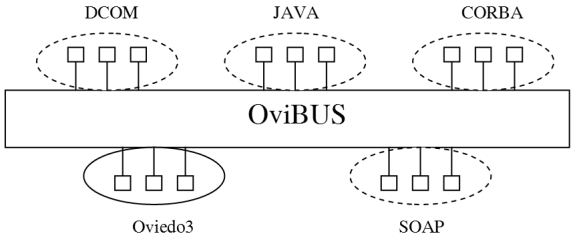


Fig. 1. The OVIBUS platform for interconnecting different components. The dotted circles represent system that can be plugged to the OVIBUS system. In the figure, Oviedo3 represents the core of the system.

With the complete LIIBUS implementation, for instance, an expert programmer on Web Services can invoke DCOM objects without knowing anything about DCOM programming. Furthermore, there can be made all the combinations desired. Moreover, this system allows communication many to many.

A. Basic Abstract Machine Architecture

The key concept to understand this Architectural model is Abstract Machine. We suppose that we are extending a existing Basic Abstract Machine, and before we extend it, it is necessary to get an insight of it. In the figure 2 it can be seen a static architectural view of the basic abstract machine used as the lower stage to construct our middle bus architectural model.

The two main abstract classes are MAIntance and MAClass, the former is used to handle the instances and the latter is used to handle the classes into the abstract machine. For example, if the machine is required to manipulate numbers, two classes are needed: on one hand by using the MAInstanceNumber class it is possible the manipulation of instances of the number class, on the other hand by using the MAClassNumber class it is posible the use of the number class itself. As it can be seen in the figure 2 if the machine users require the use of the string class the same operation must be done by using their required classes, which would be MAInstanceString and MAClassString that inherit from MAInstance and MAClass respectively.

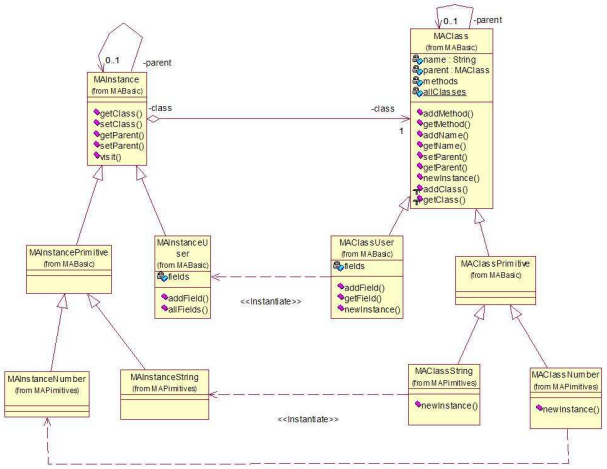


Fig. 2. Basic Abstract Machine Static Architectural view

Therefore, if any new class has to be added, it can be done by inheriting in the same way from MAInstance and MAClass.

V. Object-Based Architectural model

In this section we present a first approach to a Software Architectural Model, in this case, it is an Object-Based one [1].

A. Classes Hierarchy

The next class diagram, figure 3, shows the main classes involved in the architectural model. Thus, MAClass, MAInstance, MaInstanceExternal and MAClassExternal are part of the more extended model of the basic abstract machine showed above, in figure 2.

This abstract machine model is the lower stage used in our solution to the heterogeneous interoperability problem.

The more significant classes here are MAExternalSystem and MAExternalDispatcher. The first one is in charge of wrapping communications from internal objects of the abstract machine to the external platforms. The second one is in charge of dispatching external invocations from external platforms to internal objects. For each platform we need to inherit from these abstract classes and implement them by wrapping or adapting the information to the external system.

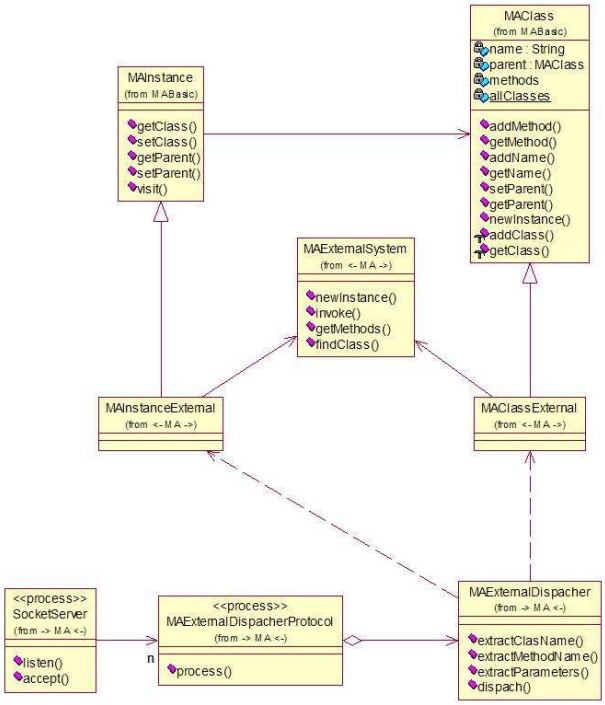


Fig. 3. Architectural model class diagram

The common types system is the one the abstract machine already uses implicitly. Therefore, there is not an explicit invocation engine, due to the abstract machine itself is an implicit invocation engine.

B. Generic Collaboration

The above class diagram shows a static architectural view, with the most important classes, relationships and the hierarchy they define. Now we know the function of each one of these elements. In this section, we will see all those classes working together in a collaboration diagram representing a behavioural architectural view. In figure 4 there are two external systems interoperating through the architectural model of the system, each of them don't know this system is between them. Therefore, each external system does not know that it is interoperating with a different and heterogeneous system. Thus, communication from the external system point of view is happening with another external system although of the same

kind. This means that, for instance, a CORBA system sees other external systems as if they were CORBA systems.

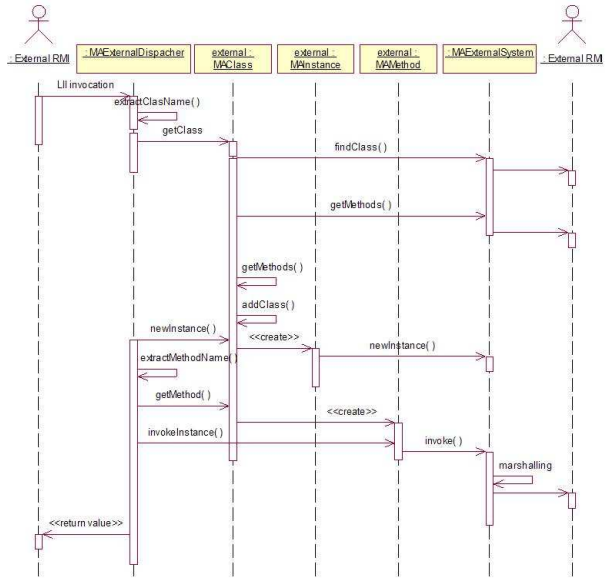


Fig. 4. Architectural model sequence diagram

First, the External RMI system on the left starts an invocation. Next, the MAExternalDispatcher class intercepts the invocation with all the information about the sender, parameters, return type, exceptions, etc. Then, it extracts all the information from the invocation and throws appropriated messages to the internal abstract machine objects. The first message is to itself, it allows identifying the name of the class to which one of its objects has been addressed by the invocation. Once this is done, it gets the class through the message getClass, and this internally calls to finClass of MAExternalSystem. This last message interacts with the external real recipient of the message who gets the right information. Following to that, the message getMethods interacts with the external and real message recipient to find the description of all methods of that class and finally wrap the external class recipient of the invocation.

Once the wrapper is finished, this new class is added to the pool of classes of the abstract machine using the message addClass. From now on, the recipient class is wrapped and available for all objects into the abstract machine and outside it.

Afterwards, we need to build an instance of this class and that is what newInstance from the class MAExternalClass stands for. This last call is redirected to MAExternalSystem newInstance method, which interacts again with the real external system and builds the real instance that is wrapped again through MAInstance.

Lastly, we are in a situation to build an invocation addressed to a wrapped instance belonging to a wrapped class, and that is what invokeInstance does; again, this message is redirected to MAExternalSystem invoke method, which again makes this call to the actual external system.

C. Implementation and validation

Once the architectural model is defined based on a basic abstract machine, we need to rebuild a new architectural model for an actual virtual machine. To do our prototype implementation, we chose Carbayonia from the Oviedo3 project working on Windows, which is a virtual machine containing more than two hundred and fifty C++ classes including both, a graphical user interface system and a structural reflection system. We implemented communication from Carbayonia to DCOM objects that mean we had to inherit from MAExternalSystem and implement a wrapper for DCOM Automation Objects. In other direction we implemented communication from Web Services to Carbayonia, in this case we had to inherit from MAExternalDispatcher to dispatch invocations from Web Services invocations through the GET/POST protocol instead of the more widely used SOAP protocol. Finally, and as a result of this work the system allows to do invocations from Web Services to DCOM objects.

In the Java code below we can see a remote dynamic web services invocation to the CheckSpellin() method of the DCOM object named Word.Application, which is the ubiquitous text processor from Microsoft.

```
String u="http://fdomingu.escet.urjc.es:8888/WORD/APPLICATION";
// we create the service
Service service=new Service();
// we create a call for this service
Call call=(Call) service.createCall();
// we define the destiny URL
Call.setTargetEndpointAddress(new java.net.URL(u));
// we define the operation of the service to invoke
Call.setOperationName("CheckSpelling");
// we indicate the paramenters
Call.addParameter("sHola",XMLType.XSD_STRING,ParameterMode.IN);
// we define the return value type
Call.setReturnType(XMLType.XSD_BOOL);
String sHola=new String("Hola");
// we do the invocation by passing it an object vector
// initialized as parameters
Boolean ret=(Boolean) call.invoke(new Object[]{sHola});
```

VI. Component-Based Architectural model

The above Object-Based Architectural model was fine, but if there was a need to add a new platform we had to create the appropriated classes for the new platform and to recompile the complete machine. Therefore, we realise that extensibility was a poor property on this kind of systems. Finally, to improve this feature we thought that using components could be useful. In this way, we extended the class hierarchy as is shown in the next section.

A. Architectural Component view

Installing a new platform to the middlebus without stopping the system improves the extensibility. In order to achieve this, we defined two classes, namely: ExternalSystemProxyComponent and ExternalDispatcherProxyComponent. Those are the proxies for the component platform desired in the

implementation and they inherit from MAExternalSystem and MAExternalDispatches.
Position statements should clearly state how they relate to the workshop theme, what particular problems they address, and what solutions they envisage, and why the statement is expected to be relevant to both this workshop and the community.

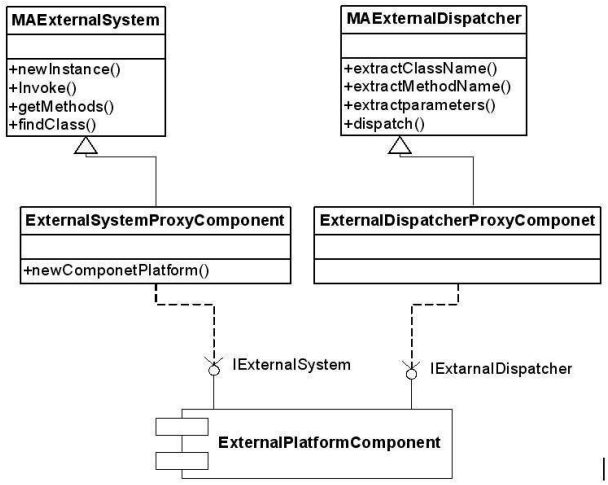


Fig. 5. Architectural model class diagram extension allowing component connections

As it is shown in the figure 5, for adding a new platform to the middlebus now, all we have to do is to build a component implementing two interfaces IExternalSystem and IExternalDispatcher. Moreover, methods of this interface are the same as methods of MAExternalSystem and MAExternalDispatcher.

Each instance of ExternalSystemProxyComponent and ExternalDispatcherProxyComponent wraps communication with a different component that allows communications with an external middleware platform. Both of this classes have a method named newComponentPlatform(). This method is in charge of initialize the adequate new component platform proxy.

B. Implementation and validation

For an implementation of this new design we used the same prototype with the DCOM component platform to allow the extensibility. Now ExternalSystemProxyComponent and ExternaDispatcherPRoxyComponent are two DCOM clients, an instance of each one is created for each platform added to the system. We created only a component named WebServicesProxyComponet, this component allows bidirectional communication from internal Carbayonia object to web services.

The validation code was the same that in the first case. No modification of the code was necessary. From the user point of view or the external middleware platforms nothing has changed. But now it is very easy to extend the middlebus with a new middleware platform to interoperate with.

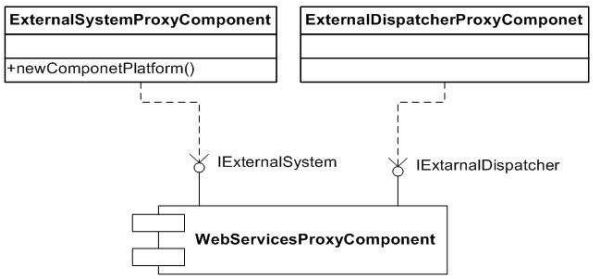


Fig. 6. Actual Architectural model class diagram with a Web Services external component

VII. Conclusions

Model-driven Architectural development, with its use of generators, can provide a suitable solution to architectural views and technical component execution platforms. In order to achieve this, a comprehensible mapping on component-oriented middleware platform is necessary and important. Our proposal can do this without any use of generators at all. A system architectural designer does not need to worry any more about other Platform Specific Architecture. Besides, the designer will just care about functional and QoS requirements of the system to be build, in only one particular middleware architecture or in only one particular virtual machine where the middlebus is been deployed. Moreover, it is allowed to add new middleware components platforms without having to stop the system. Further works, will allow the virtual machine programmer to extend the platform directly by programming into the virtual machine.

References

[1] Francisco Dominguez Mateos. *LIIBUS: Arquitectura de Sistemas Interoperables entre Middlewares Heterogneos usando Máquinas Abstractas Reflectivas Orientadas a Objetos*, Universidad de Oviedo, 2005.
[2] Francisco Domnguez. *Diseño e Implantación de la Máquina Abstracta MA y la extensión reflectiva de Carbayonia*, Servitec. ISBN: 84-689-3500-X, 2005.
[3] Francisco Domnguez Mateos, Manuel Rubio. *Investigacin en Sistemas Distribuidos y su Heterogeneidad*, Servitec. ISBN: 84-689-3501-8, 2005.
[4] Ivar Jacobson, Grady Booch, James Rumbaugh. *El Proceso Unificado de Desarrollo de Software* Addison Wesley. ISBN: 84-7829-036-2, 2000.
[5] Tanenbaum Andrew, Van Steen Maarten. *Distributed Systems Principles and Paradigms*, Prentice Hall. ISBN: 0-13-088893-1, 2002.

Automated Deployment of Component Architectures with Versioned Components

Leonel Aguilar Gayard[1], Paulo Astério de Castro Guerra[2],
Ana Elisa de Campos Lobo[1] and Cecília Mary Fischer Rubira[1]
Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6176 – 13083-970 – Campinas – SP – Brazil
{leonel.gayard,alobo,cmrubira}@ic.unicamp.br[1], asterio@acm.org[2]

***Abstract*— In the deployment of component-based applications, concrete configurations only behave as expected if they are deployed together with their right components in their chosen versions. The assembling of an application from its components is therefore a crucial step in the deployment phase and should be automated in order to avoid human mistakes. However, this task is generally performed by instantiation code written in the deployment phase in an *ad hoc* manner. In this paper, we propose CosmosLoader, a tool that automates the assembling of an application at runtime by means of customized class loaders. CosmosLoader automatically instantiates components and connects them to form an application at runtime, eliminating the writing of instantiation code by developers.**

I. Introduction

The software architecture of a system is the structure or structures of the system, which comprises software elements, the externally visible properties of these elements and the relationships among them. It represents the decisions that should be made first in its design. The software architecture of a system is represented by its architectural components and architectural connectors [1]. A connector is an architectural building block used to model interactions among components and rules governing those interactions [2].

In constrast, a software component is a unit of composition with provided and required interfaces and explicit context dependencies, which can be deployed independently [3]. Context dependencies specify the deployment environment required by a software component. A concrete component is a binary piece of software that can be instantiated at run time as a part of an executable system. In general, the deployment of component-based applications consists in installing implementations of components, satisfying their context dependencies and assembling them to compose a concrete configuration [3].

Deployed components are retrieved and loaded at run time to compose an application: retrieving a component consists in finding its implementation binaries, while loading a component consists in loading these implementation binaries in memory, usually through low-level mechanisms of the underlying programming platform. These activities may differ according to the component model in use. For instance, in the Enterprise JavaBeans model [4], components are retrieved through an EJB container, which is also responsible for loading them.

An important issue in composing a concrete configuration is to consider the version of its components: as components evolve, different versions of the same components may behave differently. Moreover, architectural components may present variation points [5], which means they have the ability to connect themselves to generate a generic architecture. This architecture is generic enough to allow different software products to be obtained by small variations in their compositions, such as the choice of their subcomponents. These factors make the deployment process more complex to be executed.

Ideally, the meta-information about the components and their versions used to form a concrete configuration should be explicitly declared either in the deployment phase or during the start-up of the application, in order to automate its installation. Also, the platform on which the application runs should have some means to know which components are being loaded and what are their versions in order to guarantee that the deployed configuration is the correct one.

In this paper, we present *CosmosLoader*, a tool which supports the deployment process in the COSMOS component model by automating some deployment tasks, more specifically, the choice of the versions of the components and the installation of components to compose the concrete configuration. *CosmosLoader* reads an XML description of the components of a concrete configuration, instantiating and connecting its components accordingly at runtime. By correctly describing the configuration in the XML description file, developers can guarantee that a running application is composed with the correct components in the correct versions.

This paper is organized as follows. Section II provides background information on the COSMOS component model. Section III compares the activity of deployment in EJB and COSMOS, and exposes issues that arise in deployment. Section IV describes our proposed solution, *CosmosLoader*. Section V analyses the generality of our solution and its application in other languages. Section VI compares the proposed approach with related work. The last section presents some concluding remarks and directions for future work.

II. The COSMOS component model

COSMOS [6] is a component model for designing and implementing flexible software components in Java. It materializes concepts from software architecture and component-based development (CBD), such as components, connectors, and provided and required interfaces in constructs available in

object-oriented languages, such as Java and C#, thus providing a direct mapping between a system's architecture and its implementation.

In COSMOS, each component is conceived as a Java package containing two subpackages: the specification package, `spec` and the implementation package, `impl`. The former contains two subpackages as well, which contain specifications for the component's provided and required interfaces (respectively, `spec.prov` and `spec.req`). The latter contains the component's implementation classes.

In addition to the interfaces provided by the component, the `spec.prov` package also contains the `IManager` interface, which performs configuration activities related to the connections of components in the architecture. The connection of components is done in a programmatic way through methods defined in interface `IManager`, that describe the component interfaces and create connections between components. The `impl` package contains a mandatory class `ComponentFactory`, with only the method `createInstance`, which is responsible for instantiating a component.

COSMOS specifies that components should not be connected directly. Instead, they must be connected by means of a connector, in order to avoid dependencies between component interfaces, which would lead to high coupling between components. A COSMOS connector is a simpler component with no specification packages: it provides and requires the interfaces of the components it connects. The implementation of the connector is responsible for resolving mismatches between the components it connects.

COSMOS supports the definition of atomic and composite components. A component is considered a composite if it instantiates other components, which are then considered its parts. As such, a subcomponent is known only by the composite which instantiated it, and is not visible to other components in the configuration. This recursive definition of composition is fundamental to a component model [3].

III. Deployment of component-based applications

Therefore, the deployment process consists of writing meta-information about the concrete configuration, which is used to correctly assemble the application from its components. The deployment of components takes different forms depending on to the component model used. Suppose the concrete configuration depicted in Figure 1.

We describe the deployment of EJB and COSMOS component models of this architecture. Suppose there are EJB implementations of components and connectors A, B, AB, XA, BX and X. The steps for deployment of an EJB implementation are as follow [4]:

1) The Java classes which constitute each EJB component are packaged in WAR files, one per EJB.
2) A *deployment descriptor* is written. It is an XML file that contains all deployment information. (Many EJB containers offer tools to automate the writing of deployment descriptors.)
3) The JNDI[1] names for each EJB are written in the descriptor. Every EJB is known to the EJB container by a JNDI name. However, this name is only known to the container, other EJBs know it by another JNDI name.

   As an example, suppose EJB AB is registered within the container with the JNDI name "AB". EJB A requests a reference to AB, not by asking a look-up to "AB", but instead to a URL that may not match the same name, say, "java://my-ab". Both names "AB" and "java://my-ab" are registered in the container, which is then responsible for translating the latter into the former and retrieving the correct EJB.
4) Context dependencies, such as external libraries are also packaged in WAR files.
5) The deployment descriptor and the WAR files are copied to the EJB container.
6) The container should also be configured to allow Java clients to access its EJBs. If a client program is also to be run in the container (i.e., a web servlet that accesses the EJBs), it should be deployed as well.

The deployment of a COSMOS implementation of the same components is as follows:

1) The class files for each component are packaged as separate JAR files, one for each component.
2) The developer writes a class whose purpose is to instantiate the components and make the connections between them according to the architecture. This is done by calls to the `ComponentFactory` class and to methods `getProvidedInterface` and `setRequiredInterface` in interface `IManager` in each component, as described in [6]. The instantiation code usually resides in a public class with a static method `main`.
3) The instantiation class is copied to the host machine as well.
4) The application class path is set. This is usually done by setting the environment variable CLASSPATH to contain all the component files and the instantiation class.
5) The class path is also set to contain the context dependencies. These are also usually packaged as JAR files.
6) The Java virtual machine is invoked with the instantiation class as the application's main class.

If the application is web-based, and a web container such as Tomcat [7] is used, then the steps above should be changed according to the container's specification. For instance, servlet containers do not acquire the class path from an environment variable, but instead look for classes in predefined directories; the instantiation code is written in a start-up servlet, instead of a main class.

In both component models, developers are responsible for managing the versions of the components and the connections between them. Two major errors may arise in the deployment

[1] Java Naming Directory Interface

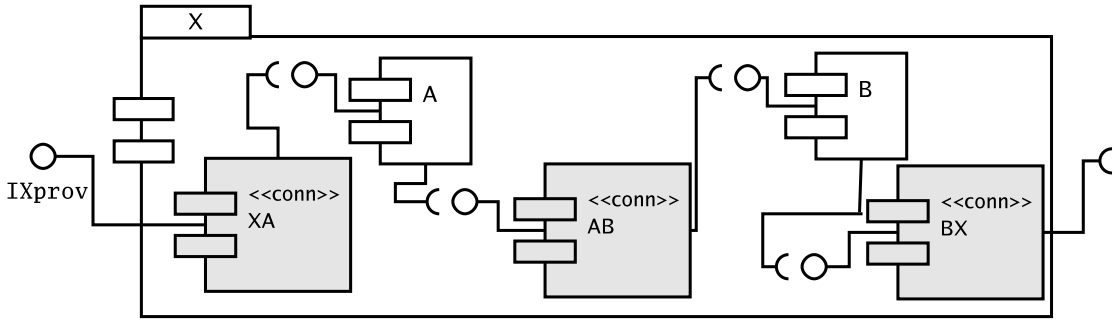


Fig. 1. Component X is an application composed with components A and B and connectors AB, XA and XB.

activity: (1) the wrong versions of components may be deployed, if the wrong files are copied to host machine, or if the class path contains the wrong versions of the components; and (2) the components may be connected erroneously, if, in EJB, the wrong JNDI names are provided, or, if there are errors in the COSMOS instantiation code. Errors in deployment can cause a mismatch between a running system and its configuration.

IV. *CosmosLoader*: A TOOL FOR AUTOMATED CONFIGURATION LOADING

CosmosLoader is a tool to automate the task of creating a running system from a set of COSMOS components (see Section II). Given a description of a concrete configuration, *CosmosLoader* is able to load the appropriate versions of components and connectors, and connect them as described in the architecture. This relieves the developer from the tasks of writing instantiation code and managing the component files and class path so that only the correct versions of the application components are loaded. Instead, the developer only has to describe the configuration and provide access to a component repository: *CosmosLoader* finds and instantiates the correct versions of components and connects them to form the concrete configuration.

A. The CosmosLoader class loader hierarchy

The three main classes in *CosmosLoader* are the `ComponentLoader`, the `ExternalLoader` and the `InterfaceLoader`; all of them are (indirect) subclasses of `java.lang.ClassLoader` and follow the delegation hierarchy shown in Figure 2. Their purposes are as follows:

- **ComponentLoader**: loads classes and implementation resources (e.g. properties files) pertaining to a component. Each instance of a component has an instance of `ComponentLoader` associated to it.
- **ExternalLoader**: loads classes and implementation resources from external libraries on which the components depend. Each instance of `ComponentLoader` has an instance of `ExternalLoader` to which it delegates the loading of resources from external libraries.
- **InterfaceLoader**: loads Java interfaces corresponding to component provided and required interfaces, as described in Section II. There is only one instance of the `InterfaceLoader`, shared among all the `ExternalLoader`s.

Different instances of class loaders allow duplication of classes accross the system and isolation of classes inside components: in Java, two classes can have the same fully qualified name, if they are loaded by distinct class loaders; in fact, if a class is loaded by two different class loaders, it will be considered as two distinct classes. Therefore, classes in different instances of the same component are considered different classes.

This is an important feature in the implementation of components. As an example, suppose that a class in a component uses class variables (marked with the keyword `static` in Java). Class variables are a common programming feature used, for instance, in the patterns *Singleton* and *Factory* [8]. Without the class separation provided by class loaders, a class would be unique among two instances of a component, and there can be a race condition for the use of the class variable.

Moreover, using different class loaders for each component and external library has the additional benefit of allowing different versions of the same class (and therefore, different versions of components and libraries) to exist simultaneously in a system.

Because components are connected by transfering their interfaces (see Section II), the interface types should be unique accross the system. This is achieved by forcing all interface classes to be loaded by the same class loader. In *CosmosLoader*, the `InterfaceLoader` is unique and loads all the interfaces.

B. Configuration description

A concrete system is considered an instance of a composite component. The instantiation of a composite component requires a configuration of other concrete instances to fulfill the roles defined for its subcomponents. Such a concrete instance can result from the instantiation of an elementary component or, recursively, other composite components.

While the roles of the subcomponents of a system (or composite component) are defined by its internal software architecture, the configuration of a concrete system depends on its specific version. Different concrete systems result from the instantiation of a same system with different concrete subcomponents. In the design of CosmosLoader, a version of

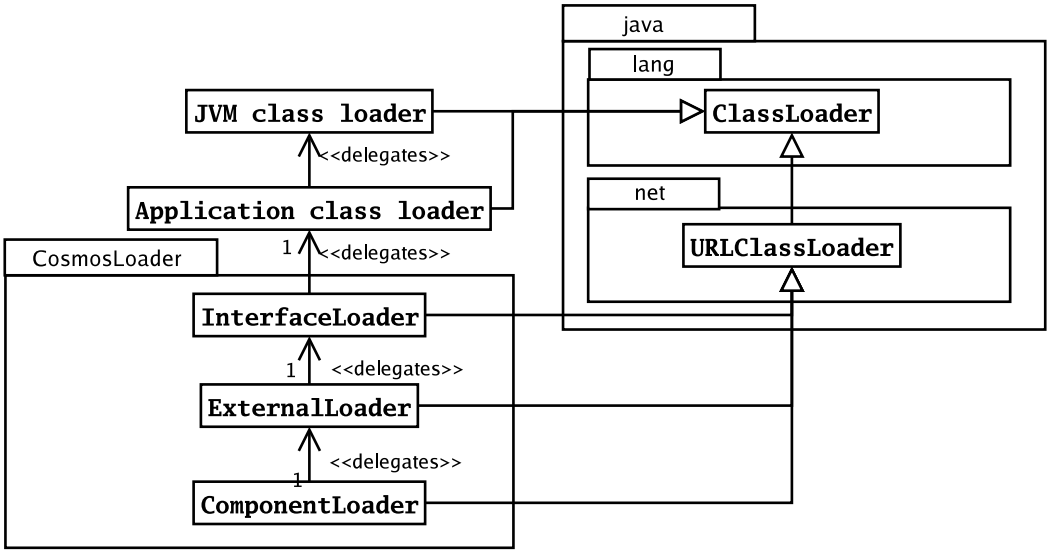


Fig. 2. The hierarchy of class loaders in *CosmosLoader*.

a concrete system (or composite component) is represented by an XML file specifying the set of component versions to be instantiated as its subcomponents. These XML files are stored at the component repository, together with the various versions of the elementary components. A configuration description can be either a single XML file describing every subcomponent in the architecture, or it can contain links to other XML files which describe other composite components. This construction is possible because of the tree structure of XML files and reflects the recursive property of composition.

C. A component repository

CosmosLoader loads component implementations from a *component repository*. The three class loaders in *CosmosLoader* load files from the repository by means of URLs. URLs contain information about the component in which to find the class and its version. This URL mechanism allows for more sophisticated repositories, such as remote repositories: in an intranet, it is possible to place the repository in a web server and access it by well-known protocols such as HTTP or FTP.

Initially, we have designed a basic repository based on directories on the file system. All versions of a component implementation are available on the file system, according to a simple directory hierarchy: there is a folder for each component; in each component folder, there is a folder for each version of the component implementation and which contains the directory structure for the component.

Consider the configuration shown in Figure 1. All components in that configuration are available in the repository as directories, as described before. As an example, suppose the version of component `a` is 1.0.1; then its class `ComponentFactory` would be found in URL *file:///usr/repo/a/1.0.1/a/impl/ComponentFactory.class* (Suppose the repository resides in directory `/usr/repo`).

D. CosmosLoader Implementation

The requirements to construct CosmosLoader were elicited from our practice experience at a medium-size Brazilian software company specialized in software solutions for financial organizations. Component-based development is used as their development paradigm, and their major products are based on COSMOS components.

In this context, composition is extensively used as a form of variability [5]: according to the needs of different customers, a component may or may not be included in a configuration, in order to make a functionality available on that application or not. Applications should not be deployed as single units, but should instead be assembled on site from their components, and only the components in that configuration should be deployed. If the deployment is as described in Section III, then a different set of component files should be copied and a different instantiation code should be written. *CosmosLoader* will allow configurations to be materialized without the need for customized instantiation code.

Furthermore, *CosmosLoader* can be extended with authorization mechanisms so that applications in a customer site can load their components at run time from a remote repository located in the developer site, so that components need not be deployed to the customer site.

CosmosLoader is still in its initial development stages. A prototype has been developed and tested with small concrete configurations.

V. On the generality of the solution

The simple structure of components makes the COSMOS model suitable to any platform that supports the concepts of interfaces and separation of classes in packages or namespaces. The implementation of COSMOS components in object-oriented languages other than Java, such as C#, is straightforward.

On the other hand, the instantiation and connection of components performed by *CosmosLoader* depends on the ability to load classes dinamically, in order to load the correct components, and on meta-programming capabilities of the platform, in order to find the `ComponentFactory` class and invoke its method `createInstance`, as described in Section II. The platform should also provide a mechanism to isolate classes inside components, in order to prevent problems such as those described in section IV. In Java, the class loader mechanism provides a natural way of isolating classes.

As an example, the .NET platform [9] provides dynamic class loading and meta-programming through classes in the `System.Reflection` namespace, but the underlying infrastructure for class loading behaves quite differently from the one specified for a Java Virtual Machine. Therefore, a C# implementation of *CosmosLoader* targeting the .NET platform is feasible, but would require a different design in order to allow several class loaders with different scopes and thus allow the same class to be loaded many times, one for each component in the system.

VI. Related work

Class loader customization. Class loader customization is a common technique for dynamically changing the behavior of a system at run time. Several applications use customized class loaders or hierarchies of class loaders in order to provide better encapsulation of components. Among these, we can cite Eclipse platform (`http://www.eclipse.org`), which builds a class loader hierarchy to support its plug-in architecture, and the Tomcat container (`http://tomcat.apache.org`), which uses class loaders to separate deployed web applications at class level, thus increasing each application security. Like *CosmosLoader*, both Eclipse and Tomcat use customized class loaders to enforce the separation of software parts, although not in the context of Component-Based Development.

Hall [10] describes customized class loaders to support deployment of different versions of software modules (which can be components, plug-ins or a similar modularization concept) across a system. Differently from *CosmosLoader*, Hall's Module Loader by itself does not support software composition, and must be embedded in component frameworks to do so.

Component instantiation frameworks. Fractal[11] is a component model similar to COSMOS. JULIA is a framework that provides support for implementing Fractal components in Java. Similar to *CosmosLoader*, JULIA reads an application description and instantiates a concrete configuration accordingly. The JULIA configuration file is different from the CosmosLoader configuration as the developer needs to indicate what classes will implement the component; therefore, the configuration file is described at class level, rather than at component level. This class-centric approach makes the distinction between the roles of component developer and system integrator unclear, as the integrator must to some degree understand the implementations of components in order to combine them in a concrete configuration.

VII. Conclusions

We have presented *CosmosLoader*, a tool to automate the instantiation and connection of COSMOS components and aid in the deployment of component-based applications. This tool reads the description of an architecture from a configuration file and instantiates the components of the application accordingly, even in the case where several versions of the same component exist. Its main functionality is based on customized class loaders that are able to load the components that compose the application and are described in the configuration file.

CosmosLoader automates three important tasks in deployment of COSMOS applications: the copy of the concrete components that form the application, the configuration of the application class path, and the selection of the appropriate version of each component. The use of *CosmosLoader* also reduces the effort needed in the deployment phase as it replaces the code that instantiates and connects the components which form the application.

Future work includes, but is not restricted to addressing the following issues: *Integration with a component repository* [12]: The integration with the component repository will use its search mechanism to find a given version of a component in the repository, in order to load it. *Integration with Bellatrix* [13]. Bellatrix is an integrated environment that materializes the elements of a software architecture based on COSMOS model. Integration between Bellatrix and CosmosLoader will allow a software configuration built by Bellatrix to be deployed by CosmosLoader.

Acknowledgements

L. A. Gayard is supported by CAPES/Brazil under grant 01P-05603/2006. C. Rubira is partially supported by CNPq/Brazil under grant 351592/97-0. The authors wish to thank AUTBANK - Projetos e Consultoria for their support in our research in component-based development, and Fernando Castor Filho for his contributions to this article. FINEP/Brazil, under grant 1843/04 of CompGov (a project for Shared Library of Components for E-Government), partially supports the work of Paulo Asterio de Castro Guerra and Cecília Rubira.

References

[1] L. Bass, P. Clements, and R. Kazman., *Software Architecture in Practice*, 2nd ed., ser. SEI Series in Software Engineering. Addison-Wesley, 2003.

[2] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[3] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[4] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, and B. Stearns, *The J2EE tutorial.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] J. Bosch, "Software variability management," *ICSE 2004*, 2004.

[6] M. C. da Silva Júnior, P. A. de Castro Guerra, and C. M. F. Rubira, "A java component model for evolving software systems." in *ASE*. IEEE Computer Society, 2003, pp. 327–330.

[7] The Apache Software Foundation, “Apache tomcat,” abril 2006, http://tomcat.apache.org/.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlisside, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

[9] Microsoft Corporation, “Microsoft .net homepage,” agosto 2005, http://www.microsoft.com/net/.

[10] R. S. Hall, “A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks.” in *Component Deployment*, ser. Lecture Notes in Computer Science, W. Emmerich and A. L. Wolf, Eds., vol. 3083. Springer, 2004, pp. 81–96.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, “An Open Component Model and Its Support in Java,” in *Component-Based Software Engineering: 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25*, 2004.

[12] L. P. Tizzei, H. S. Pinho, P. A. C. Guerra, and C. M. F. Rubira, “Um repositório de componentes com suporte à evolução centrada na arquitetura de software,” in *5o Workshop de Desenvolvimento Baseado Em Componentes (WDBC’2005)*, Juiz de Fora, Minas Gerais - Brazil, 2005.

[13] R. T. Tomita, F. C. Filho, P. A. C. Guerra, and C. M. F. Rubira, “Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes,” in *4o Workshop de Desenvolvimento Baseado Em Componentes (WDBC’2004)*, João Pessoa, Paraíba - Brazil, 2004.

Describing Framework Static Structure: promoting interfaces with UML annotations

Sérgio Lopes, Adriano Tavares, João Monteiro, Carlos Silva
Department of Industrial Electronics
University of Minho, Campus de Azurém
4800-058 Guimarães, PORTUGAL
Email: <sergio.lopes,adriano.tavares,joao.monteiro,carlos.silva>@dei.uminho.pt

***Abstract*—Frameworks are an important form of reuse that can help to significantly decrease the time and cost of application development. Although widely known, there are still some difficulties associated with framework reuse, which are critical to its success. In this paper, we focus on the issues regarding the framework reuse process, and more specifically, on the framework architecture description.**

This paper discusses our position on the subject. It enables a component-oriented approach to framework reuse by emphasizing the description of black-box variation-points, and introducing *call-points*. We define the UML-FD profile for UML 2.0, which extends UML to support these and other concepts, dedicated to describe the static structure of application frameworks.

I. INTRODUCTION

Reuse has been a major concern for software engineers, in their quest for easier application development with reduced time-to-market and cost. Frameworks are an important form of reuse that can help to get closer to this long-time pursued goal. However, several problems associated with frameworks have been identified [2], starting from the framework development and ending at its maintenance. This work is concerned with the difficulties inherent to framework reuse from the perspective of the application developer.

Object-oriented frameworks [1] are widely known, but building applications by reusing them poses problems that software developers have to struggle with. Frameworks capture a specific domain's commonalities and variabilities, by implementing common elements and providing an architecture that localizes variability at *variation-points*. In contrast with passive forms of reusable software (typically, procedural libraries or traditional class libraries), frameworks are active and exhibit predefined behaviour, which imposes some control flow among its components. Consequently, they are often complex and hard to understand, what can make reusing them a difficult and time-consuming assignment [1].

These problems must be alleviated in order to make framework reuse an alternative way of building applications that is decisively attractive. In fact, it is not always guaranteed to be advantageous comparatively to application development by "reinventing the wheel". The typical framework slow learning curve is one major issue because it results in a delayed productivity payoff, which can arrive unrewardingly late or even be unacceptable. Therefore, diminishing these problems is a decisive factor for the success of framework-based reuse approach. It has been consensually recognized the need to effectively communicate frameworks and provide appropriate tool support, in order to minimize the effort and time required to build applications. The present paper focuses on difficulties in communicating the framework to the re-user, and our proposal to lessen them.

Although communicating the architecture is a framework developer responsibility, the notation chosen is decisive to the following phases of the reuse process to be carried out by application developers. It is widely accepted that visual notations have crucial advantages over textual languages in the communication of software. Being a *de-facto* standard object-oriented design notation in industry, UML [21] is a beneficial choice for describing frameworks, but it is a general scope and extensible language, not specifically tuned for this purpose. It has been previously demonstrated [4] the need to explicitly represent frameworks variation-points, not supported by UML, in order provide effective framework description. Actually, applying a subset of UML to object-oriented frameworks reuse has been addressed before, with a few dedicated extensions being proposed [4], [8], [9]. However, a few limitations still endure and we investigate how to tackle them, in order to provide a more complete support for the framework reuse needs. We propose the UML-FD profile that integrates a different perspective, namely, promoting a component-oriented approach to application frameworks.

This paper describes on going research, and starts by discussing some issues regarding framework description in following section. In section III, we introduce our perspective and discuss the foundation of our proposal. In section IV, the requirements for object-oriented notations supporting framework reuse are outlined. The UML-FD profile, defining the proposed UML extensions, is described in section V. In section VI, the related work is reviewed and our contribution is explained. Finally, we present the concluding remarks.

II. ABOUT FRAMEWORK REUSE

A framework can be reused in many different ways that require different kinds and amounts of information, which may be constrained by business interests. The support that tools can offer is also affected, but this is not a matter for this paper.

We discriminate two fundamental forms of object-oriented framework reuse: unanticipated reuse and anticipated reuse. The differences of them are given below, in a discussion that describes their relationships with framework documentation aspects.

A. Anticipated Reuse vs. Unanticipated Reuse

We make the distinction between these two forms of reuse because the information needs, the activities and the results of each one are quite different.

Anticipated reuse takes place when the particular needs of a re-user are fulfilled by the functionalities of the framework. In more concrete terms, the framework provides enough *variation-points* (also named *hot-spots* [3] or *hooks* [17]) with enough flexibility, to cope with a re-user objectives. Or, the other way around, the re-user goals can be achieved by a subset of all the possible variation-points' adaptations. The framework adaptation is realized by providing application specific components for variation-points that observe the respective constraints. With this kind of activity, typically, the re-user does not have to worry about possible erroneous interactions between the framework components.

Unanticipated reuse happens when the re-user wants to add some functionality that is not provided by the framework components, or only to make a slight change to some feature. Usually, these kind of goals cannot be achieved solely by adapting the framework's variation-points. Most probably, it will be necessary to make adaptations outside the set of pre-defined variation-points. By doing it, the re-user can more easily introduce erroneous interactions between application specific components and the framework. Furthermore, these flaws can be difficult to correct, as we see next.

B. Description Information and Business Rules

Telling apart the two forms of reuse above is important because they have a strong impact on both the kind and quantity of necessary information about the framework, which in turn is a subject of business concerns.

Anticipated reuse is easier to document because the anticipated variability is localized at variation-points. Therefore, it is sufficient to provide detailed design documentation only about them. In particular, describing the purpose of each variation-point, how to adapt it, and their semantic restrictions which guarantee that the adaptation will work correctly. Tools can also be built to provide specific assistance for filling the variation-points. In contrast, unanticipated reuse can occur at almost any part of the framework. Hence, besides the variation-point description, it requires detailed design (and possibly implementation) information about the complete framework. Notice this is generic software documentation, because it is not possible to provide specific reuse information. The same applies to tools, which cannot provide any special development support for unanticipated reuse.

Communicating a precise and deep understanding of the framework to the re-user is essential to enable the assessment of viability that is necessary to achieve specific goals, and the development of adaptations that do not violate the framework architecture. Naturally, the description should be independent of implementation details not important for design, which might limit the framework generality, or lead to complications caused by framework evolution (see [19] for this kind of problems). These issues apply to both kinds of reuse, but are much harder to manage when support for unanticipated reuse is intended. Unanticipated reuse requires complete information about the framework, in order to enable the re-user to develop unpredicted adaptations, which correctly interact with the framework, and/or change parts of it while maintaining behaviour consistent with untouched parts.

Furthermore, the problem with detailed architecture description, necessary for supporting unanticipated reuse, is that it may collide with business interests. There are a few free open-source frameworks but, on the other hand, there are commercial frameworks provided by vendors. Development of a framework is a long and costly process that requires high expertise in the target domain. Thus, revealing the architecture details is not usually considered good business because it may give advantages to competitors in the same market, and framework vendors may be suppliers of dedicated support tools. For these reasons, if a framework is not open and there is no detailed documentation about it, it may prove to be very difficult to achieve a successful unanticipated adaptation.

C. Description Techniques and Reuse Possibilities

The spectrum of approaches for framework documentation can be classified according to two categories: informal prescriptive techniques, and formal descriptive techniques.

Prescriptive techniques [14], [15], [16], [17] describe how to use the framework, normally using natural language, or other informal means of documentation. They provide valuable guidance but only to the limited adaptation possibilities described. It is not possible to predict all the ways of adapting a framework, in fact, not even is feasible to describe a large number of them. Therefore, these techniques are more suited to support anticipated reuse. They are also oriented towards less skilled users, or to enable experienced users a quicker first application build.

Descriptive techniques describe the framework architecture, usually using formal or semi-formal visual languages, like UML [4], [8], [9]. They do not dictate or elicit any particular way of reusing it; instead, they try to communicate the framework architecture to the user. They do not provide significant guidance for adapting a framework; it is up to the re-user to figure out how to adapt it, in order to meet her/his specific requirements. Consequently, these techniques are appropriate to support unanticipated reuse, and are more oriented towards experienced users, who need detailed information more than guidance.

III. Our Approach

Before we get to the proposed solution to communicate frameworks, we explain our point of view about framework reuse, which is the foundation for it. First, the technique chosen to describe frameworks, and then the perspective on reuse technology, are presented.

We follow the same line as the UML techniques cited above, i.e., investigating how to augment the reuse flexibility that descriptive techniques provide, with as much guidance as possible. In agreement with the exposed in the previous section, adopting a descriptive technique to describe a framework in detail supports unanticipated reuse. Augmenting a general descriptive technique with support for explicit variation-point description enables to provide guidance for anticipated reuse. Variation-points are the typical key concept for organizing this kind of documentation for frameworks, but they are not enough, as it will be argued in the next section. No special requirements apply to the description of a framework for unanticipated reuse, it is much like describing any other piece of software. Therefore, the requirements for OO notations for framework description (presented below) reflect only the part of reuse that is anticipated, because it is the one that requires a dedicated approach.

White-box frameworks [5], more than object-oriented, are class-oriented because their adaptation is frequently based on inheritance (sub-classing framework classes), which is a mechanism that describes class hierarchies. We favour a predominantly black-box approach in which the framework is reused by calling its interfaces and providing components that implementing the interfaces it requires. This approach emphasizes use relationships instead of inheritance, and thus is suitable to support the representation of object interactions. In turn, this also facilitates the specification of restrictions on clients.

When using a framework, an application developer is reusing both a design and its implementation. Therefore, we choose not to abstract the variation-point description to the design level, as opposed to [4]. In fact, we consider that the framework should be described with as much precision as possible (without neglecting what was stated in the previous section). Design variations-points defined by inheritance can be refactored into use relationships, with interfaces to be implemented by application specific components [4]. This can be done using the Strategy design-pattern [6], or other suitable design patterns based on separation meta-patterns [3]. This polymorphism and forwarding technique separates the interfaces from implementations, making the design more decoupled and flexible than with inheritance, and it is the base for a black-box approach to reuse.

Moreover, if framework classes provide separate computational and compositional interfaces, it enables a decomposition of the framework instantiation process into two different reuse activities. In the literature, the process of reusing a framework to build an application is usually denominated framework instantiation [1]; we subdivide it in two activities or two phases – framework adaptation, and application instantiation – which we explain next.

The adaptation phase (also designated as 'framework instantiation' in [4]) consists in providing application specific adaptations that define the behaviour of variation-points. The application developer learns the details about the framework architecture from the annotated UML diagrams, and extends it with the application specific components.

Once all components necessary for an application are available, the final application can be defined. This is accomplished in the application instantiation phase, by creating instances of framework and application components, configuring and interconnecting them to form the final executing application. The components provide a compositional interface including methods whose names typically start by 'set', 'add', 'remove', etc, that enable run-time configuration.

A complete discussion of the reasons behind the separation between framework adaptation and application instantiation can be found in [22], which discusses our perspective on tool support for these activities.

To conclude, frameworks designed this way enable the application definition by creating and configuring its run-time units individually, like components. We consider them component-oriented frameworks, because they can be adapted by composing components, although we do not consider any standard component model. We provide support for reusing them, but we also provide specific constructs for white-box reuse that can be useful for describing "gray-box" frameworks.

IV. Framework Description Requirements

Considering the discussion in the previous section, we present below, what we consider to be, the main requirements for describing frameworks in order to facilitate its reuse.

Based on our experience in adapting and implementing frameworks, and on the revision of previously proposed solutions for framework design, we have elicited a requirements list for design languages to describe OO frameworks. First, we present it, and then we discuss each one of the requirements:

1) Domain and purpose of the framework and its specific features;
2) Framework static structure with explicit variation-point identification;
   a. Support for white-box, black-box and client reuse;
   b. Define different types of variation-points with clear semantics;
   c. Variation-point syntax enabling the definition of semantic restrictions on the adaptation;
3) Framework dynamic behaviour with explicit support for variation-points;
   a. Define causal obligations for variation-point adaptation;
   b. Explicit differentiation of variation-points messages in behavioural compositions;
4) Guidance for framework adaptation process, with support for optional variation-points;
5) Guidance for the application instantiation process.

The complete framework documentation should include the identification of its target domain, as much as possible defining the boundaries of that domain, and stating which problem the framework solves in that domain. It should also provide a functional view of the features provided by the framework.

The description of framework static structure identifies the components that compose its design and their relationships. It

gives a static view of the objects' collaborations. It should explicitly distinguish the variation-points from the framework core, in order to assist the framework user in identifying more easily the parts that need to be provided, or adapted, to create applications. This a form of endowing descriptive techniques with some guidance for the framework adaptation phase.

White-box and black-box variation-points have been consensually recognized as forms of adapting frameworks, and their explicit identification and description has been supported by graphical notations dedicated to framework reuse ([4], [8], [9]). Nevertheless, frameworks do not always rely exclusively in the Hollywood Principle ("don't call us we'll call you", or inverted control mechanism based on Template Method [6]) to communicate with application components. Sometimes they provide services to be called by clients, as has been recognized in [18]. We have developed a framework for measurement systems, inspired by [13], that combines predominant 'inverted' flow of control with pieces of non-inverted control flow. This example experience suggests broadening the framework variety to frameworks that have a neither purely called neither purely calling architecture. The interaction between clients and framework through use relationships may vary from single method invocation to complex protocols that impose obligations on the clients. We share this view with [10], which also emphasizes that use relationships, as a basis of behaviour composition, play an important role in framework integration. Therefore, we introduce the notion of call-points as parts of the framework interface, anticipated for client use, that play a key role in the framework operation. We believe call-points are a concept that reflects an important variety of reuse needs and, for this reason, we widen the explicit identification of framework reuse points to support them.

Variation-points should be classified according to different types, more refined than white-box and black-box, providing additional semantics which are helpful for guiding the framework adaptation. Their semantics should be made as precise as possible, with clear description of the abstract possibilities it opens and abstract restrictions it imposes. Their syntax should support the representation of additional semantic adaptation restrictions that may be useful to specify limits to the set of possible application instantiations.

The description of dynamic behaviour gives a view of the dynamic aspects of the framework design that clarifies the objects' responsibilities, their context dependencies, and how they can be combined. By representing explicitly the run-time collaborations between objects, it reveals the framework architecture. How much of this information is provided depends on the factors considered in the previous section. This information is fundamental to comprehend the framework and, once more, it is vital for opening the door to the flexibility of unanticipated reuse. Furthermore, it also enables the description of causal obligations for variation-points and call-points. These behavioural restrictions should be documented, if they exist, and the corresponding messages in object interactions should be explicitly differentiated from the framework core messages.

The adaptation process should be guided by a description that helps to reduce the complexity of the task, especially for medium and large-scale frameworks. Some variation-points may be optional, and others may require the adaptation of another variation-point. These dependencies should be described in order to provide more guidance and facilitate the job of the application developer.

Finally comes the instantiation process, which should also be guided some how. A framework may be adapted to build an application, or to be integrated into a larger project. These processes should be described, if not in abstract, at least with partial, or complete, concrete examples.

V. Static Structure Description

Although the set of requirements in the previous section cover all aspects of framework documentation, this paper deals only with the description of static structure, i.e. corresponding to requirement 2.

UML 2.0 is a convenient choice for describing frameworks due to its widespread use. It provides structural diagrams, which depict the static features of the model, and behavioural diagrams that describe the dynamic aspects of the model. UML structural diagrams include the class, object, package, component, composite structure, and deployment diagrams. To describe the frameworks' static structure several of these available diagrams can be used for different purposes. In our opinion, two diagrams are rather useful: the class diagram for explicit identification and characterization of variation-points and call-points, and composite structure diagrams as a complement to elucidate its architecture.

We introduce the UML Profile for Framework Description (UML-FD), which extends UML with dedicated concepts supporting a few different variation-points and call-points. Naturally, the proposed annotations address the aforementioned requirements 2-a through 2-b, and therefore we do not discuss them further. The UML-FD profile is defined for UML 2.0, i.e. it augments the current version of UML making use of its improved extensibility mechanism.

Fig. 1 defines the profile abstract syntax and its integration with the UML 2.0 meta-model. All the UML meta-classes extended by UML-FD belong to the `Classes::Kernel` language unit. Tables I to III describe the semantics of each individual extension in a compact tabular form (similar to the presentation of UML 2.0 standard stereotypes). Each table groups variations-points according to white-box variation-points, black-box variation-points, and call-points, providing a clear separation between these different reuse categories.

White-box variation-points are supported because they can be useful to describe white-box frameworks, which is classically the first form that every framework assumes. The *application class* annotation is not a variation-point at all, but instead, it can be used to discriminate framework classes from application specific ones. Both *extensible class* and *non-overridable method* follow the Open-Closed Principle of object-oriented design. *Variable method*s are usually abstract methods of abstract classes. The three white-box variation-points can be can be directly implemented by subclasses,

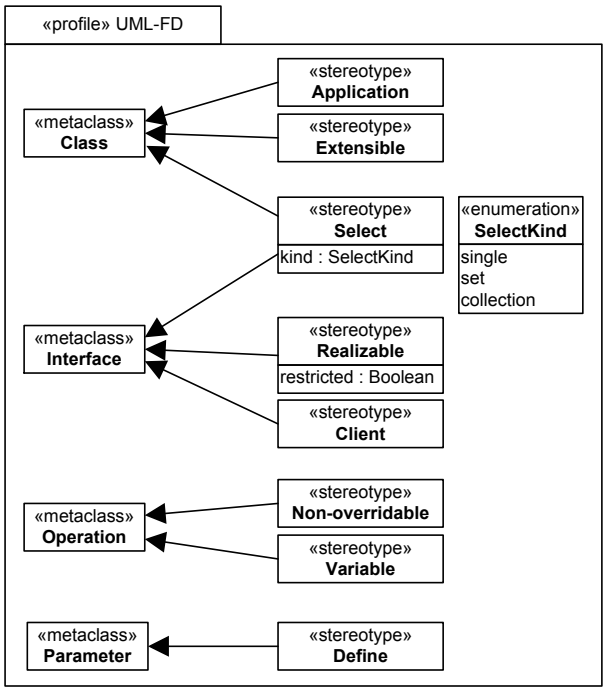


Fig. 1. The UML-FD Profile for UML 2.0.

however that is not recommended. They can also be combined in both concrete and abstract classes.

Black-box variation-points and call-points are the recommended option for reusing a framework. The *realizable interface* is the black-box reuse variation-point with inverted control flow, to be implemented during the framework adaptation phase. All three white-box variation-points can (and should) be converted to a realizable interface, as described in section III. *Select class* and *define parameter* are black-box variation-points which are defined during the application instantiation phase (according to the approach described in section III).

In the next subsections, we discuss in more detail the main contributions of the UML-FD profile to the description of framework static structure.

A. Client Interface

A *client interface* identifies a framework call-point. A call-point is defined by a bidirectional association between one client interface and a framework class that provides one corresponding *service interface*. The association end connected to the framework class identifies (has the name of) a service interface, or *control method*, to be used by the client interface.

The example of Figure 2 illustrates a call-point that is an implementation of the Observer design pattern [6]. The call-point is identified by three elements: the `Observer` interface; the association connecting `Observer` and `Figure` with an association end for `Figure` whose role name is `subject`; and, the part of `Figure` class interface defined by the `Subject` interface. The `Observer` interface is the client interface that must be implemented by client components using the call-point. The `subject` role name of the association end identifies the name of the service interface to be used by the `Observer` client interface. The `Subject` interface is the service interface that defines the `Figure` method(s) to be called by `Observer` client(s).

The client interface construction is a kind of localized role modelling at implementation level, in which client interfaces represent role-types to be integrated by client classes, and service interfaces represent role-types assigned to core framework classes. It enables the modelling of multiple collaborations, through disjunctive groups of semantically related methods (role types), on the same framework interface. Each collaboration is specified by one association that identifies the framework interface methods to be called (service interface) and connects to the respective interface required on clients (client interface). This solution is described only under the perspective of the static structure description: roles, repre-

TABLE I
WHITE-BOX VARIATION POINTS

Applies to	Stereotype	Semantics
Class, Interface	«Application»	An **application class**, or application interface, is part of the application, as opposed to classes which belong to a framework.
Class	«Extensible»	An **extensible class** can have new methods added.
Operation	«Variable»	A **variable method** is a method to be implemented by application classes (implementation variation).
Operation	«Non-overridable»	A **non-overridable method** can be extended but cannot be overridden, *i.e.,* any overriding method must always invoke it.

TABLE II
BLACK-BOX VARIATION POINTS

Applies to	Stereotype	Semantics
Interface	«Realizable»	A **realizable interface** is an abstract type for which application classes can be defined. It has a property named **restricted** that if true forbids sub-typing (classes implementing it, cannot have a different interface).
Class, Interface	«Select»	A **select class** limits the variation-point to the concrete sub-components provided by the framework. It has a property named **kind**, whose value can be **single**, **set**, or **collection**.
Parameter	«Define»	A **define parameter**, is a parameterized variability that defines an important characteristic of the framework (e.g., in opposition to ordinary attributes or parameters related to component interconnection). Constraints on the valid values may be defined as supported by UML (e.g., Enumeration).

TABLE III
CALL-POINTS

Applies to	Stereotype	Semantics
Interface	«client»	A **client interface** is an interface to be implemented by application classes that interact with a framework by calling services of its components (use relationship). Any (*call-back*) methods it has define client constraints, namely static behaviour obligations.
Operation	«control»	A **control method** is a method to be called by clients to externally control some special framework function or trigger some event.

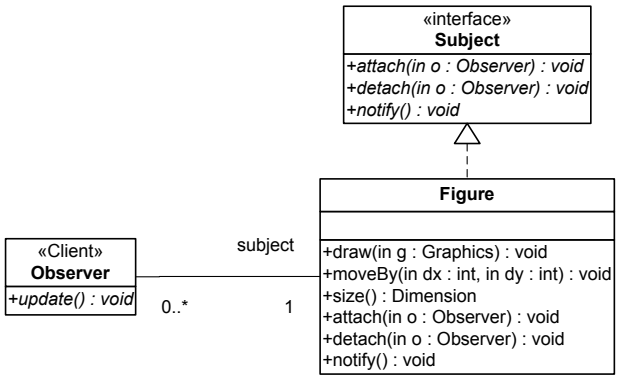


Fig. 2. Example of a Client Interface.

sented by interfaces, describe type information only (or static behaviour). However, client interfaces can be empty, which supports the specification of dynamic behaviour obligations on frameworks clients, independent from structural properties.

The idea behind it is to take advantage of some useful properties of the role modelling technique while avoiding some of its intrinsic verbosity and complexity (see next section properties discussion). Namely, it enables the definition of client restrictions without over constraining client implementation structure. It also provides more structure and semantic information about framework call-points and its relationships with clients. The framework description is kept succinct because client interfaces express role-types which are confined to call-points. This technique avoids the overweight and complexity of the coexistence of reusable role models with respective implementations, by keeping the description at implementation level and within a single paradigm.

B. Control Method

The control method identifies another kind of framework call-point. It is a simpler construction for using framework services that involves a single component method, because it requires no separate service interface for the core framework component. In addition, it is not intended to be used with a client interface, although it might (as defined above). As an example, we have used it to model the trigger function for a real-time embedded framework, shown in Figure 3. The `Sensor` application component invokes the `update()` control method, to stimulate the `Trigger` framework component. Obviously, the framework description does not include the application class, which is included in the figure only for illustrative purposes.

Control methods are applicable more generally to event-driven frameworks that depend on externally fed events. We believe, control methods can also be used to model frameworks that enable easier composition with other frameworks, by providing externally regulated control-flow. They can be used to synchronize the control-flow of such frameworks.

VI. Related Work

The framework description problem has been addressed from informal textual language approaches [7], to formally

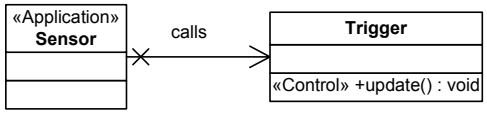


Fig. 3. Example of a Control Method.

defined visual notations that extend UML [8].

Informal textual techniques are usually prescriptive. One first example is the *cookbook* [14] for the Model-View-Controller framework, useful for implementing graphical user interfaces. A similar work is found in [15], where little more structured of a set of Alexandrian-based *patterns* helps to reuse the Hot-Draw framework. Both describe the framework purpose and how to use it. They consist of non-uniform narrated descriptions with minimal structure, and examples solving problems about how to use the framework. This kind of technique was improved by *hooks* [17], which are more uniform, formal and structured adaptation receipts. Hooks define a classification of adaptation methods and kinds of support provided. However, its typology does not provide a clear separation of adaptation activities involved, and they may focus on different framework functionalities with different levels of detail. As discussed in section II.C, prescriptive techniques are focused on the framework intended use and therefore do not offer support for unanticipated reuse.

Although discussing how our approach can be combined with software tool support is outside the scope of this paper, we still look at tool-based solutions, but we concentrate on framework communication and we overlook the facilities for automating framework adaptation. By using software tools, it was possible to improve the cookbook technique to *electronic books*. *Active cookbooks* [20] are a prominent example, which provides interactive receipt descriptions that explain how to use the framework design to solve problems. However, it lacks flexibility because the user has to follow the dictated steps. Evolutions of the electronic book approach are *Smartbooks* [11] and *Specialization Patterns* [12]. Smartbooks are based on a hierarchical interactive hypertext interface through which the desired framework functionalities are chosen. From it, a task plan is generated which guides the adaptation. Specialization patterns are described by a dedicated notation, which lacks tool support. The specialization patterns for a specific framework are embedded in a tool, which handles them providing support for building applications. None of these approaches provides the explicit representation of variation-points within the framework design. Although formalized somehow (to enable tool processing), it is still the framework designer who prescribes its adaptation options.

A few works have been devoted to descriptive visual approaches for documenting frameworks. UML has been the obvious target, being extended with concepts dedicated to framework documentation [4], [8], [9]. These works have similarities – all provide UML extensions to identify variation-points – and parts that are complementary: [4] focus more on variation-points identification and characterization, while [8] provides stronger support for expressing framework syntax and semantics, and [9] introduces selection of black-box com-

ponents and parameterization. The role modelling technique [10] is a complementary technique that tackles object relationships, which are fundamental for framework integration and composition. The requirements on clients calling framework services are explicitly represented but, on the other hand, this approach disregards the explicit identification of variation-points. frameworks may require different instantiation mechanisms. Catalysis [23] also applies UML to support reuse. However, it defines model frameworks as collaborations of abstract types, which are reused through parameter substitution. It does not address the reuse of (code) frameworks, and consequently it does not provide dedicated annotations for explicit representation of its variation-points. Catalysis defines a software development method based on the concepts of model frameworks and components.

Our research also explores UML as visual descriptive techniques for describing frameworks. It builds on previous work, but we provide a wider and more complete coverage of the different reuse needs. While keeping the support for white-box variation-points, a clear and precise definition of black-box variation-points is provided. We introduce UML extensions for explicit expression of use relationships with constraints on clients, to facilitate the reuse and composition of called frameworks [18] and black-box [5] frameworks. We do that by introducing call-points, which borrow inspiration from concepts of the cited role modelling technique. By putting a special emphasis on use relationships, or object relationships, we enable a black-box approach to framework reuse.

VII. Conclusion

Software engineering has pursued for decades the ambition of increased reuse and software quality. Frameworks are an important alternative, which offers high reuse potential, but still have a few problems to be tackled. Addressing these difficulties, namely by employing graphical notations and providing appropriate tool support, it is critical to its success as an option for application development, and for the reuse goal in general.

Some important factors that influence the support that is provided for framework reuse were discussed. An explanation of our perspective on framework reuse was given. A requirements list for object-oriented design notations providing specific support for framework reuse was elaborated and discussed. The UML-FD profile for UML 2.0 was defined, offering a wider coverage of needs for describing framework static structure description. The role modelling technique was analysed in more detail, because it is the background for part of our work.

We have provided a clear separation of different reuse options: white-box, black-box and call-points. Although the proposed notation supports white-box variations-points, we encourage a component-oriented approach by emphasizing back-box variation-points and call-points. For that purpose, we also define how framework interfaces must be described in order to enable a black-box application development.

We have introduced the concepts of client-interfaces and control methods that expand the spectrum of reuse concepts, by including specific points for calling framework services. These concepts are important in the context of black-box reuse and framework integration or composition.

We believe the presented work contributes to facilitate the communication of frameworks. Hence, it also helps to decrease the difficulties and complexity associated with the framework reuse-based development, making it a more attractive, easy and rewarding alternative to develop applications.

Acknowledgements

The authors would like to thank the anonymous reviewers for their contribution to this work.

References

[1] M. Fayad, and D. Schmidt, “Object-oriented Application Frameworks,” in *Communications of the ACM*, vol. 40, no. 10, ACM Press, Oct. 1997, pp. 32–38.

[2] J. Bosch, P. Molin, M. Mattsson, PO Bengtsson and M. Fayad, “Object-oriented frameworks — problems & experiences,” in *Building Application Frameworks — Object-Oriented Foundations of Framework Design*, M. E. Fayad, D. C. Schmidt and R. E. Johnson, Ed. New York, NY: Wiley & Sons, 1999, pp. 55–82.

[3] W. Pree, “Meta Patterns — a means for capturing the essentials of reusable object-oriented design,” in *Object-Oriented Programming*, ECOOP ’94, Tokoro, Mario & Pareschi, Ed. Remo: Springer-Verlag, 1994, pp. 150–162.

[4] M. Fontoura, W. Pree and B. Rumpe, “UML-F: A Modeling Language for Object-Oriented Frameworks," in *Proc.of the European Conference on Object-Oriented Programming (ECOOP’00)*, LNCS 1850, 2000, pp. 63–84.

[5] R. E. Johnson and B. Foote, “Designing reusable classes,” *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, Jun. 1988.

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

[7] L. Murray, D. Carrington and P. Strooper, “An approach to specifying software frameworks,” in *Proc. of the 27th Conference on Australasian Computer Science*, Dunedin, New Zealand, 2004, pp. 185–192.

[8] N. Bouassida, H. Ben-Abdallah, F. Gargouri and A. Ben Hamadou, “Formalizing the framework design language F-UML,” in *Proc. of the 1st IEEE International Conference on Software Engineering Formal Methods*, 2003, pp. 164–172.

[9] T. Oliveira, P. Alencar and D. Cowan, “Towards a declarative approach to framework instantiation,” in *Proc. of the Workshop on Declarative Metaprogramming to Support Software Development of the 17th IEEE International Conference on Automated Software Engineering*, Edinburgh, Sept. 2002, pp. 5–8.

[10] D. Riehle, and T. Gross, “Role model based framework design and integration,” in *Proc. of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’98)*, Vancouver, Canada, 1998, pp. 117–133.

[11] A. Ortigosa and M. Campo, “Smartbooks: a step beyond active-cookbooks to aid in framework instantiation,” in *Technology of Object-Oriented Languages and Systems*, 25, IEEE Press, June 1999.

[12] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa and J. Viljamaa, “Annotating reusable software architectures with Specialization Patterns,” in *Proc. of the Working IEEE/IFIP Conference on Software Architecture*, August 2001.

[13] J. Bosch, “Measurement systems framework”, in *Domain-specific Application Frameworks*, M. E. Fayad, D. C. Schmidt and R. E. Johnson, Ed. New York, NY: Wiley & Sons, 2000, pp. 177–205.

[14] G. Krasner and S. Pope, “A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80”, in *Journal of Object-Oriented Programming*, 1(3), 1988.

[15] R. Johnson, “Documenting Frameworks using Patterns,” in *Proceedings of the Conference on Object-Oriented Programming Systems,*

Languages and Applications (OOPSLA'92), Vancouver, Canada, 1992, pp. 63–78.

[16] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[17] G. Froehlich, H. Hoover, L. Liu and P. Sorenson, "Hooking into object-oriented application frameworks", in *Proceedings of the 1997 International Conference on Software Engineering*, Boston, MA, 1997.

[18] S. Sparks, K. Benner and C. Faris, "Managing object-oriented framework reuse," *IEEE Computer*, pp. 53–61, Sep. 1996.

[19] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt, "Reuse contracts: managing the evolution of reusable assets," in *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications*, San Jose, CA, October 1996, pp. 268–285.

[20] W. Pree, G. Pomberger, A. Schappert and P. Sommerlad, "Active guidance of framework development," *Sofware — Concepts and Tools*, Springer-Verlang, 1995.

[21] Object Management Group (2005, July 4th). *Unified Modeling Language: Superstructure* (version 2.0) [Online] Available: http://www.uml.org.

[22] S. Lopes, C. Silva, A. Tavares and J. Monteiro, "Application development by reusing object-oriented frameworks," in *Proceedings of the IEEE International Conference EUROCON 2005*, Belgrade – Serbia & Montenegro, November 2005.

[23] D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: the Catalysis approach*, Addison-Wesley, 1999.

Interactive Component Assembly with SuperGlue

Sean McDirmid
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
sean.mcdirmid@epfl.ch

I. Introduction

Programming is a very powerful way of interacting with a computer because the user is not constrained by the narrow scope of an application's functionality. Unfortunately, even for experienced programmers, writing a program is often too difficult and time consuming to be performed casually because most programming languages focus on low-level constructs to express arbitrary computations. However, programming need not be difficult if a language focuses on high-level constructs to express component assemblies rather than arbitrary computations. If such a language is supported with the ability to conveniently repeat or continuously execute the program as it is being written, then users can use the language to effectively perform *interactive component assembly* to automate a wide variety of tasks such as repetitive data processing.

A language's support for interactive component assembly depends on two features. First, the language must support **declarative connectors** that encapsulate low-level details about how components communicate, which substantially reduces the complexity of (component assembly) glue code. An effective declarative connector is often a standardized interface with multiple procedures that are called by component implementations, but not by glue code, according to a well-defined protocol. In this way, glue code does not have to worry about the control-flow details of how components communicate. Examples of declarative connectors include pipes and signals, which support the expression of inter-component data-flow relationships. In contrast, procedures and events, which are the core constructs of many programming languages, are not declarative and expose glue code to many details about how components communicate.

Second, a language's support for interactive component assembly also depends on how well an environment for the language can support **tinkering**. A development process that includes careful planning and deliberation is too slow to be viable in an interactive component assembly context. Instead, a user should continually or repeatedly execute a program while changing its code. Such tinkering merges the editing and testing parts of program development so a user can more quickly converge on a desired program. The levels of tinkering that we consider in this paper are as follows:

- **Level-0**: the status quo for most languages where editing and testing are mostly separated and tinkering is not supported very well.
- **Level-1**: a program can be edited and re-executed very rapidly. First-level tinkering is supported in shell-based environments such as Bourne Shell [2].
- **Level-2**: a program continuously executes where any edits to the program's code immediately changes the behavior of this execution, which is referred to by Tanimoto as **liveness** [16]. Second-level tinkering is supported in many visual-programming environments such as Forms/3 [3] and Quartz Composer [11].
- **Level-3**: the code and execution of a program are the same so users edit a program's execution directly, which is referred to by Shneiderman as **directness** [14]. Third-level tinkering is supported by many tools that allow user interfaces to be laid out directly as they will appear at run-time. Third-level tinkering is also supported by interactive applications that support WYSIWYG document editing.

The tinkering levels described here are only reference points and many programming environments do not fit perfectly into one of these levels. For example, Eclipse [9] supports Java development beyond zero-level and first-level tinkering with interactive compilation and limited hot code replacement.

For a programming language to be effective in interactive component assembly, it must also provide good support for abstraction along with support for declarative connectors and tinkering. Designing a language for interactive component assembly is challenging if this language is to support abstraction very well. Declarative connectors sacrifice flexibility for ease of programming: without the ability to specify control-flow constructions such as loops or event handlers, it is difficult for glue code to abstract over **space** in the form of array-like compound data or abstract over **time** in the form of mutable state. Designing declarative connectors that can abstract over space and time is challenging. Second, as the level of tinkering supported by a language increases, code necessarily becomes more concrete with respect to the executing program, which further exacerbates the problem of abstraction in assembly code.

Interactive component assembly can be seen as an extreme form of component-oriented programming that heavily emphasizes high-level connectors, predefined architectures, third-party reuse, and rapid integration. All of these attributes also have their uses in conventional software development contexts where large programs undergo substantial amounts of planning. This paper describes how interactive component assembly can be supported in SuperGlue [12], which is a declarative connection-based language that focuses on com-

ponent assembly. In the rest of this paper, we describe existing interactive component assembly approaches (Section II), SuperGlue (Section III), related work (Section IV), and our conclusions (Section V).

II. Background

Interactive component assembly is based on the old idea of interacting with a computer by writing a small program rather than through a pre-built application. In a general sense, all applications are somewhat programmable through interactive manipulation; e.g., a document in Word is like a program that customizes Word components to format the document's data. However, such programmability is very limited and provides for almost no abstraction. Although today most users interact with computers through "interactive" graphical applications such as Word, many users lament that interaction was richer when programs could be assembled from commands on the command line.

A. Shell Programming

Bourne Shell [2] is an example of an environment that supports the command line assembly of components, as commands, with a few keystrokes to perform various text-processing tasks. For example, in Bourne Shell a user can type "`cat notes.text | grep XXX | wc -l`" to output the number of lines that contains the string "`XXX`" in the file `notes.txt`. Bourne Shell supports interactive component assembly with text streaming *pipes* as declarative connectors and a convenient shell interface to support tinkering. Commands in Bourne Shell import and export pipes that can then be connected together via the pipe operator (`|`) or connected to (`>`) or from (`<`) a file. The shell supports program tinkering with features such as immediate execution (just type enter), command completion, and a history-based mechanism for quickly changing and re-executing a command. The shell also provides a user console that can be connected to or from pipes, which is especially useful for tinkering because it allows the user to directly specify a command's input or view its output.

Pipes can be generalized into data-flow connectors that focus on the communication of data while hiding flow-control from glue code. Most programming languages can support pipes at least as user-defined abstractions while many scripting languages such as Perl provide concise syntax for using pipes. However, the stream-based nature of pipes limit their effectiveness in the construction of programs. For example, although pipes can be used to filter (`grep`) or organize (`sort`) small flat lists of data, tree-like data or data in very large tables (databases), which cannot be practically iterated over, are much more difficult to deal with. Additionally, pipes are not very useful in the construction of interactive programs where, in contrast to batch programs, component communication is often reactive because mutable state is involved. These limitations can be dealt with to some extent; e.g., through a second language for expressing database queries or using cron for reasoning about time intervals. However, these work arounds are limited and make writing programs more complicated.

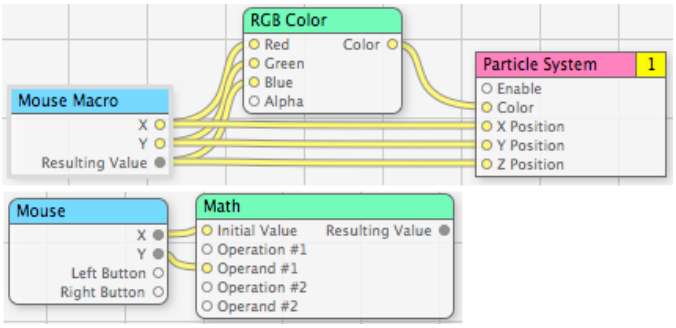


Fig. 1. An example of a program in Quartz Composer (top) and the composition of the Mouse Macro patch (bottom).

The approach of programming through an interactive shell is still popular: shells are supported for many scripting and functional languages, such as Ruby, Scheme, and ML, while new system shells, like Bourne Shell, are still being developed; e.g., see Microsoft's new Windows PowerShell. On the other hand, shell programming is not very scalable to programs that require multiple lines of code. Although shell variables can solve this problem to some extent, they do not naturally fit into a shell environment and naming is an issue. As a result, larger programs in Bourne Shell are best expressed in their own file where tinkering is less viable. Finally, interactive shells only support first-level tinkering: a line of code cannot be edited while it is being executed. For many batch programs, this is not a big issue as the programs are not interactive and terminate quickly. However, stronger second-level tinkering is more desirable for interactive programs or batch programs that process a lot of data.

B. Visual Programming

Beyond shell programming, interactive component assembly is also the foundation of many visual programming languages. As one example, we consider Apple's Quartz Composer [11], which is used to express media compositions such as animations. Components in Quartz Composer are known as *patches* whose connectors are live data-flow values that can change over time. Both components and their connectors have visual representations as exemplified by the program in Figure 1. The `Mouse Macro` patch outputs the current horizontal (`X`) and vertical (`Y`) values of the mouse pointer, as well as the sum modulo (`(X + Y) % 1.0`) of these values (`Resulting Value`). These three values are then used to compute the RGB color and three-dimensional position of a particle system. The resulting program creates a particle system whose position and color changes as the mouse pointer is moved around the computer screen.

As an interactive component assembly language, Quartz Composer improves on Bourne Shell in two ways. First, unlike Bourne Shell's pipes, the live data-flow values that connect patches together can effectively abstract over time (mutable state); e.g., the program in Figure 1 deals with a continuously moving mouse pointer. Such time abstraction is especially important in component assemblies that involve graphical user

interfaces or integrate continuously changing data. Second, Quartz Composer supports second-level tinkering: a program can be edited in a visual buffer while the program is running, where changes to the program immediately change its execution. For the media-oriented programs that Quartz Composer supports, second-level tinkering is very useful. For example, the alpha channel of a computed color can be tweaked while the animation in Figure 1 is running to find a value that visually works nice. In contrast, if only first-level tinkering were supported, tweaking the alpha channel would be much more tedious as we would have to restart the animation after every change.

According to the ability to express abstraction, Quartz Composer and many other visual programming languages suffer from a big problem: given that components and their connectors have concrete visual representations, abstracting over space is either impossible or requires awkward constructs that are difficult to use. In Quartz Composer, multiplexor-like components are used to iterate over arrays of data while replicator-like components are used to instantiate the same patch assemblies multiple times. The resulting code is a "visual spaghetti" of numerous patches and even more numerous overlapping patch connectors. The problem with visual programming languages in general is that their programs are less compact than programs in textual languages. In contrast, text is more effective at abstraction with its ability to express many kinds of space-saving constructions such as loops, conditionals, and variables.

It is also possible to support second-level tinkering in a declarative text-based language. For example, XAMLPad [13] for WinFx supports the editing of XML-based user interface code while providing continuous previews of the user interface being constructed. Although XAML code is declarative and not by itself very expressive, data binding in XAML code can to some extent be used to abstract over time and space.

III. SuperGlue

We are currently enhancing SuperGlue [12] to support interactive component assembly. Unlike Quartz Composer and like XAMLPad, SuperGlue is a declarative textual language. However, like Quartz Composer and unlike XAMLPad, components in SuperGlue are connected together through live data-flow values that we refer to as *signals*. Unique to SuperGlue, signal connections are expressed as rules that can refer to universally-quantified variables and are organized according to object-oriented types. SuperGlue's support for rules and object-oriented typing is an elegant way for abstracting over space to deal with unbounded-size data structures such as lists, tables, or trees. As an example, the following SuperGlue code views the folders of a mailbox in a user-interface tree control:

```
let folderView = TreeView;
let mailbox = Mailbox;
folderView.root = mailbox.root_folder;
var node : folderView.Node;
var fldr : mailbox.Folder;
if (node = fldr)
  node.children = fldr.sub_folders;
```

The first two lines of this code respectively create a user-interface tree-control (**folderView**) and an email **mailbox**. On the third line, the root node signal of the folder view tree is then connected to the root folder signal of the mailbox. The fourth and fifth lines declare two universally-quantified variables: *node*, which abstracts over all nodes in the folder view tree, and *fldr*, which abstracts over all folders of the mailbox. The last two lines form a rule that connects the children signal of any node object to the sub_folders signal of any folder object when the node object is connected to the folder object.

By using signals, rules, and objects, space and time abstraction integrates together completely in SuperGlue. As an example, the following SuperGlue code views the messages of a selected email folder in a user-interface table control:

```
let messageView = TableView;
if (folderView.selection.size == 1 &&
    folderView.selection[0] = fldr)
  messageView.rows = fldr.messages;
```

The rule in this code connects the rows of a message view table to the messages of an email folder if some node is uniquely selected in the folder view tree and this node is connected to the email folder. The number of nodes selected in the folder view tree can be changed by the user, causing the rule's first antecedent to change from true and false at run-time. As what node is selected in the folder view tree is changed by the user, how and if the rule's second antecedent can change at run-time, which changes what messages are viewed in the message view table. Finally, as messages are added to or removed from the selected email folder, the message view table automatically inserts or deletes rows for these messages. See our conference paper [12] for a more detailed description of how SuperGlue combines signals, rules, and objects.

SuperGlue also supports SQL-like *arrays* that can be filtered and mapped with declarative code. As an example, the following code filters a list of messages according to whether the sender is **myBoss** and maps the subjects of the resulting messages to a user-interface **urgentList** list:

```
urgentList.input =
  fldr.messages(sender = myBoss).subject;
```

For the purpose of describing what this code does, we could rewrite the code in an SQL-like syntax as follows:

```
urgentList.input = SELECT subject
  IN fldr.messages WHERE sender = myBoss
```

Arrays in SuperGlue are supported completely with time abstraction; e.g., when a message whose sender is **myBoss** is added to (or is deleted from) from the email folder, the message's subject will immediately be added to (or removed from) the **urgentList**. Arrays in SuperGlue can also be implemented by components in a way that avoids iteration when being filtered. For example, an IMAP-based implementation of email components can implement sender-based filtering in such a way that avoids transmitting all messages in a potentially huge mailbox across the networks.

A. Interactive Component Assembly in SuperGlue

Our work up until now has focused on simplifying component assembly in SuperGlue with signals as declarative connectors that are supported by rules and objects. In this paper, we propose enhancing SuperGlue with support for second-level tinkering so that it can be used as an interactive component assembly environment. We argue that SuperGlue's programming model is especially suited to second-level tinkering for the following two reasons:

- SuperGlue programs are declarative, meaning they are not exposed directly to program control-flow details. As a result, repairing a program's execution to reflect an edit does not involve "unwinding" the effects of computations that have already been executed. Support for true second-level tinkering is very difficult for imperative or functional languages because of computations that must be unwound when an edit occurs. Typically, such languages can at best can only support limited forms of hot-code replacement that ignore all past computations.
- Edits to a SuperGlue program can be reified as changes in the program's mutable state, which are easily communicated as changes in signal values. Adding, changing, or removing a rule to or from a running program causes the signals that the rule connects to change in value. The values of signals can already change according to changes in program state, where editing can simply be another way of changing program state.

Given SuperGlue's accommodating programming model, support for second-level tinkering is largely a tool problem: can we build an environment, which includes an editor (front-end) and interpreter (back-end), that enables the editing of a SuperGlue program while it is running? Enhancing the interpreter to support second-level tinkering is straight-forward: code that manages signal changes need only be aware of edit-based changes. Support for detecting edit-based changes, however, is not so straight-forward in SuperGlue. Because visual languages often support only a language-restricted form of program editing, detecting and reporting changes in a program is very easy. However, language restricted editing is not very suitable in text-based languages because they do not support intermediate edits very well [17]. For this reason, we have decided that only language-directed editing is appropriate, where a programmer can make arbitrary edits to the text of a program, possibly causing it to become syntactically or semantically incorrect.

B. Requisite Technology

We have almost completed our first prototype of an environment that supports interactive component assembly in SuperGlue. This environment is designed around novel presentation compilation technology that supports the error recovery needed to make sense of edits that cause the program to become structurally incorrect. Our presentation compiler is based around the following three technologies:

- An enhanced form of **precedence parsing**, which allows the structure of a parse tree to be easily repaired when an edit is made. With precedence parsing, most edits are processed by splitting and fusing parse tree nodes so that a new tree structure can be computed quickly without throwing away any old but still useful state. Because precedence parsing does not enforce a grammar, a parse tree for a program can always be computed even when the program's text does not conform to the program's intended grammar. Although this property means that grammar compliance must be checked separately, it is desirable in an interactive component assembly context where syntactic errors commonly occur. For expressiveness, our incremental precedence parsing algorithm has also been enhanced to handle brace matching in a way that supports effective error recovery.
- A form of **data-flow processing** to incrementally process a program's parse tree to perform type checking and repair the interpreter's run-time data structures. Each node in the parse tree refers to an input, which is pushed from its parent or predecessor, and an output, which can be pulled by its parent or predecessor. A node is scheduled for "cleaning" when its input has changed, when the nodes that it can push to have changed, or when the outputs that it can pull have changed. Using data-flow processing, type checking can be expressed in a completely incremental way. Error recovery is also supported by using a node's old correct output when a newly computed output is structurally incorrect.
- Enhanced symbol tables that record both uses as well as definitions. As a result, symbol additions, removals, and name changes can all be processed incrementally.

With this technology, we are able to repair a SuperGlue program's execution as the program's text is being edited. When the code of a user-interface program is edited, the user-interface being built is updated in lock-step with the edit. This is done in the editor's own thread so that the next keystroke will not even appear until the user-interface has been updated! Edits are often processed fast enough for two reasons. First, our compiler technology is completely incremental, where a single edit (one key stroke) can often be processed in constant time. Second, our error recovery mechanisms prevent a program's execution from changing unless new valid constructions are produced by the edit. On the other hand, keystroke lag can be noticeable because everything currently executes in the editor's user-interface thread. For example, keystroke lag is noticeable when an editor-based state change occurs concurrently with state changes caused by other sources such as timers.

Similar to interactive editing in Quartz Composer, SuperGlue can be enhanced to support second-level tinkering in developing component assemblies. Although our incremental compiler technology is currently very new and still being debugged, our environment is already in a demo-able state. As a result, we are optimistic that SuperGlue will be an effective environment for interactive component assembly.

IV. Related Work

The signals that SuperGlue supports are similar to signals in functional-reactive programming languages such as Fran [7], Yampa [8], and FatherTime [4]. However, signals in SuperGlue support the assembly of components implemented in an imperative language (in our case Java), while signals in Haskell-based Fran and Yampa (but not Scheme-based FatherTime) support an explicit notion of time that cannot interface with imperative code. Like SuperGlue, signals in FatherTime can interface with imperative code [10]. However, FatherTime's support for Scheme's recursion and higher-order functions make it difficult to support it with second-level tinkering.

Many component assembly languages focus on procedures as component connectors, which as already mentioned are too low-level to support interactive component assembly. ArchJava [1] is a component assembly language that supports custom-defined connectors. However, one of the main features of ArchJava is its support for managed implicit sharing between components, which prevents it from supporting second-level tinkering as inter-component aliases are difficult to unwind in response to an edit. In contrast, all inter-component communication in SuperGlue occurs through explicit signal connections and no aliasing can occur between components.

Research in visual programming that is related to SuperGlue includes Forms/3 [3], which is a Turing complete language for building spreadsheets. Forms/3 supports second-level tinkering and some amount of third-level tinkering; e.g., shapes can be drawn by the user directly rather than inferred from an abstract visual representation. However, Forms/3 is designed as a language for expressing computations and not component assembles. Prograph [5] is a visual language that is based on the data-flow paradigm in the form of pipes and as a result, like Bourne Shell, is limited in its support for interactive component assembly. Cocoa [15] is a visual language that, like SuperGlue, is based on rules. Unlike SuperGlue, Cocoa cannot yet support second-level tinkering as programs need to be restarted when rules are changed via editing.

V. Conclusions and Future Work

This paper has presented the problem of interactive component assembly and how SuperGlue can solve this problem with signals, rules, objects, and an interactive programming environment. Given these constructs, SuperGlue can be used to rapidly build user interfaces or automate repetitive data processing tasks through a very casual programming interface. We are currently constructing our third prototype of SuperGlue, which will be our first prototype that supports the interactive editing needed to support second-level tinkering.

Besides finishing our prototype, in the future, we want to explore how SuperGlue can support some amount of third-level tinkering, where programmers edit program executions directly. Third-level tinkering is limited in expressiveness because it cannot support abstraction very well. However, the ease-of-use benefits of concreteness cannot be ignored and users should be given options between editing either an abstract textual or a concrete visual representation of a program. Changes between both representation should be "linked" so the user can switch between both modes of editing as needed.

References

[1] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *Proc. of ECOOP*, Lecture Notes in Computer Science. Springer, 2003.

[2] S. R. Bourne. An introduction to the UNIX shell. In *Bell System Technical Journal*, July 1978.

[3] M. Burnett, J. Atwood, R. Walpole, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. In *Journal of Functional Programming*, pages 155–206, Mar. 2001.

[4] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. To appear in ESOP, 2006.

[5] P. Cox, F. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditions. In *IEEE Workshop on Visual Languages*, 1989.

[6] J. Edwards. Subtext: Uncovering the simplicity of program. In *Proc. of OOPSLA Onward*, 2005.

[7] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of ICFP*, volume 32 (8) of *SIGPLAN Notices*, pages 263–273. ACM, 1997.

[8] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2002.

[9] IBM. *The Eclipse Project*. http://www.eclipse.org/.

[10] D. Ignatoff, G. H. Cooper, and S. Krishnammurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. To appear in FLOPS, 2006.

[11] P.-O. Latour. *Quartz Composer*. Apple Computer, 2005. http://developer.apple.com/graphicsimaging/quartz/-quartzcomposer.html.

[12] S. McDirmid and W. C. Hsieh. SuperGlue: Component programming with object-oriented signals. To appear in ECOOP, 2006.

[13] Microsoft. *The WinFX SDK*. http://msdn.microsoft.com/winfx/.

[14] B. Shneiderman. Direct manipulation: a step beyond programming. In *IEEE Computer*, pages 57–69, Aug. 1983.

[15] D. Smith, A. Cypher, and J. Spohrer. Kidsim: programming agents without a programming language. *Communications of the ACM*, pages 54–67, 1994.

[16] S. Tanimoto. VIVA: A visual language for image processing. In *Journal of Visual Langauges and Computing*, pages 127–139, June 1990.

[17] T. Teitelbaum and T. W. Reps. The Cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

Active Documents

Taking advantage of component-orientation beyond pure reuse

Markus Reitz*
University of Kaiserslautern
Software Technology Group
P.O. Box 3049, 67653 Kaiserslautern
Germany
reitz@informatik.uni-kl.de

***Abstract*— Since its introduction, the component-oriented paradigm has exerted strong influence on the development process of software. Second generation component technologies like .NET or Java provide a plethora of general-purpose components ready to be reused, putting development efforts on a firm footing of well-tested building blocks. Albeit being a very successful approach in software development, customers usually do not benefit from component-orientation, as it is in general invisible in the user domain. By adjusting and augmenting component-oriented concepts with respect to the requirements of end-users, ACTIVE DOCUMENTS pave the way for next-generation software systems which are profoundly user-adaptive and feature-personalisable. This paper introduces the overall concepts of ACTIVE DOCUMENTS, describes their application in the COMDECO[1] project, sketches the added value and discusses some technical key features of general-purpose ACTIVE DOCUMENT systems.**

I. INTRODUCTION

Component-orientation has induced a metamorphosis of the often monolithic and vendor-locked software development process. By glueing together building blocks which are made available by steadily growing component repositories, developers are able to build reliable software for fast-emerging markets in shorter periods of time. Unfortunately, Cox's vision of *component markets* [1] which are similarly structured as their counterparts in the hardware sector has not been achieved yet. Nevertheless, *thinking in components* is one of the most important design principles these days. The reuse aspect of component technology is quite well-understood, but it represents a facet that is mainly related to software development.

End-users at most marginally encounter the consequences of reusability utilised by developers. This is not as bad as it may seem, because other features are of greater relevance to end-users. Software systems should be

- adaptable to slight changes and variations of requirements (*Adaptability*),
- personalisable according to specific user preferences, usage patterns and usage habits (*Personalisability*),
- flexible to cope with changing requirements during lifetime[2] (*Flexibility*), and
- prepared to handle increasing complexity, changing models of abstraction and usage contexts (*Openness*).

Moreover, configuration and tailoring should be based on easy composition operations, that can be carried out without the need to have an expert at hand (*Simplicity*).

The rest of this paper is structured as follows. The remainder of this section discusses general properties of ACTIVE DOCUMENT systems. Section II gives an overview of concepts and techniques that partially form the foundation for ACTIVE DOCUMENT technology. The usage of ACTIVE DOCUMENTS in the context of derivative contracts is sketched in Section III. A brief overview of technical key concepts as realised by the general-purpose ACTIVE DOCUMENT framework *Omnia* is given in Section IV. Finally, Section V sketches further scenarios which may benefit from utilising ACTIVE DOCUMENT techniques.

A. The Document Metaphor

Software developers and end-users are usually entirely different entities. Being confronted with composition mismatches, writing glue code or wrapper code is one out of a multitude of possible solutions developers can make use of to resolve occuring problems. In contrast to that, an end-user often does not have the necessary technical expertise and would therefore be overstrained in the same situation. Transferring component-orientation to the end-user domain induces the necessity for adaptations of the conceptual model. *End-user oriented frameworks* have to be based on well-known metaphors that provide an (ideally) intuitive interaction interface which (partially) hides the complexity of component-oriented systems.

A document is an archetypical concept the majority of end-users is familiar with. Books, newspapers, or tax declarations are representatives of a metaphor that plays an important role in the everyday life of human beings. ACTIVE DOCUMENTS augment the originally static information representation with state of the art software technology concepts, resulting in a

*Supported by the cluster of excellence *Dependable Adaptive Systems and Mathematical Modeling* (DASMOD) of Rhineland-Palatinate, Germany.

[1]Composable Derivative Contracts, a subproject of DASMOD (`http://www.dasmod.de`).

[2]Currently, although only a feature subset provided by the latest software version is needed, a customer has to pay for all features.

hyperdocument. Being computer-based, dynamicity replaces fixed representations, pushing away boundaries and limitations of traditional media.

B. Properties of ACTIVE DOCUMENTS

Normally, users think in terms of applications when creating documents. For example, a letter is written using a word processing application and illustrations are created with the help of a graphic tool. However, this distinction is artificial and cumbersome, especially when creating mixed media documents, e.g. a text having embedded illustrations. A task-oriented point of view focusing on the document instead of the applications required for its creation is more suitable. Centered around the document metaphor, ACTIVE DOCUMENTS support a task-oriented style of human-computer interaction.

Built upon component-oriented principles, the power of any ACTIVE DOCUMENT system stems from the contents of its component repository that may be used to create new or to enhance already existing documents. Specific repository configurations support specific areas of application. An ACTIVE DOCUMENT system is easily adapted to the user requirements just by adjusting the component repository. Each added or removed component influences the whole system[3], i.e. application boundaries tend to diminish, eventually vanishing completely. Being component-oriented, structural as well as semantical constraints may be easily checked and enforced by the supporting runtime system. Composition operations being usable by non-experts in conjunction with fault-tolerant and partially guided composition mechanisms are provided (*Explorative Composition*).

Active Documents possess a state that is subject to change in case of interactions between the document's components. Moreover, state changes are triggered by user interactions, i.e. an ACTIVE DOCUMENT's state may vary in case of different users (*User-dependent Statefulness*).

II. RELATED WORK

At least three mainstream component technologies are currently competing for the favour of software developers. CORBA, a language and vendor neutral conceptual umbrella framework, puts the focus on general-purpose component systems by advocating the *CORBA Component Model* [3]. (Enterprise) Java Beans [4], favourised by SUN, are the premier choice for "write once, run everywhere" component-orientation in the Java world. The .NET platform [5] in conjunction with the Java opponent C# [6] aims at an unification of already available technologies like OLE or COM, hand in hand with the introduction of supplemental features. However, all these technologies focus on requirements of developers. Support for *end-user compatible composition* is at most rudimentary. Visual composition techniques are directly provided or based on the respective component model. Unfortunately, the available level of abstraction is too low, requiring technical expertise an end-user cannot be expected to possess in general.

[3]This is partially similar to Raskin's ideas of *humane interfaces* [2].

Due to end-user compatible interaction patterns, ACTIVE DOCUMENTS are often visually composable, but it is important to notice that the underlying supporting framework is not fundamentally based on visual composition principles. In fact, there may exist non-visual representations of ACTIVE DOCUMENTS[4].

Starting with Microsoft's *Object Linking and Embedding* (OLE), the concept of a *compound document*, a kind of container which is able to conglomerate (theoretically) arbitrary media types, was introduced. Based on pure geometric constraints, the end-user is able to autonomously arrange desired elements. Using the concept of *in-place editing*, parts of applications capable to provide appropriate editing facilities are superimposed on the "master" application, letting application boundaries partially diminish. Apple's OpenDoc [7] project aimed at the development of a multi-platform framework providing the common technological infrastructure for compound document systems, but failed. With the introduction of Mac OS X, OpenDoc became a deprecated technology and any further support was dropped. GNOME Bonobo [8] is one of the few remaining multi-platform compound document frameworks. Alas, work has slowed down and feature extensions are rare. However, due to the limited expressiveness taking only geometric properties into account, complex composition operations cannot be supervised by a compound document system.

ACTIVE DOCUMENT technology is an evolutionary approach which combines, inter alia, the principles of compound documents, general-purpose component-orientation and rule-based programming. The notion of an ACTIVE DOCUMENT first appeared in the context of the EU-funded Easycomp project [9], initially targeting the web engineering domain. *Minerva* [10] focused on the development of a framework supporting user-centered document systems in the context of electronic learning materials (e-Learning). Albeit being an important first step, *Minerva* does not provide the necessary techniques for general-purpose ACTIVE DOCUMENTS. The underlying component model is implicitly tied to the requirements of e-Learning applications, therefore it is usually not suitable for use in other contexts.

III. DERIVATIVE CONTRACTS AS ACTIVE DOCUMENTS

Financial engineers work with documents and produce documents when designing and valuating *derivative contracts*. A *termsheet* listing the properties of a specific product is one example.

A. Derivative Contracts

Any derivative contract may be expressed in terms of its payoff and the conditions describing the circumstances that allow to exercise it[5].

Example III.1 *Suppose an investor believes that in two years starting from now on, the market price for palladium will*

[4]The XML based representation of a derivative contract in the COMDECO project is an example for a non-visual representation.

[5]For a general introduction to derivative contracts refer to e.g. [11].

significantly increase. Nevertheless, he wants to invest with caution, i.e. if the price falls in contrary to expectations, losses should be limited. A derivative contract that reflects this estimation could be

> *"The holder of this derivative contract has the duty to buy one ounce of palladium on June 10th 2008 for the market price valid on June 10th 2006. If the purchase price is above the then current market price, the holder receives a consolation of 5 €."*

By immediately selling the goods for the then valid market price, the contract is converted into monetary units, i.e. abstracting away physical goods, the given contract guarantees a payoff of

$$\max(S^{€}_{\mathrm{Pd}}(t_1) - S^{€}_{\mathrm{Pd}}(t_0), 5)$$

at contract's maturity, with $S^{€}_{\mathrm{Pd}}(t)$ denoting the market price for palladium at time t where t_0, t_1 represent the dates as defined by the contract.

In order to be able to trade a contract, a financial engineer has to determine its market price, i.e. the contract's monetary value for $t_0 \leq t < t_1$[6]. It is common practice to use spreadsheet-based prototypes when determining, among other things, a contract's *fair price* [12]. These prototypes use appropriate valuation algorithms and models typically implemented as Microsoft Excel spreadsheets. These "development environments" are usually augmented with glue code in form of *Visual Basic for Applications* (VBA) macros and additional C/C++ *dynamic link libraries* (DLL) as the need arises. Because financial engineers are usually not software developers, solutions provided by software technology, e.g. design patterns [13], are only slowly adopted. Inflexible prototypes dominate and cause tremendous problems. Additionally, spreadsheet-oriented designs do not scale up well in case of increasing complexity. Although financial engineers and software engineers work in quite different domains, the principal problems are very similar.

B. ComDeCo's Objectives

Derivative contracts represent a large and higher-than-average growing segment of local and world wide financial markets. The almost arbitrary flexibility[7] in conjunction with the pressure of permanently shortening product life and time to market cycles creates the need for efficient methods of design and valuation. ComDeCo [14] aims at providing tools and techniques to handle the complexity of current and future derivative contracts. In contrast to well-established description techniques based on plain mathematical formulae used by financial engineers for decades, ComDeCo offers an enhanced termsheet representation using a special variant of an Active Document - software technology meets financial engineering. Utilising Active Documents, any contract is represented by a component-oriented software program which is subject to composition by the non-expert end-user.

[6]For $t = t_1$, the contract's value is already known in advance, of course.

[7]"With derivatives you can have almost any payoff pattern you want. If you can draw it on paper, or describe it in words, someone can design a derivative that gives you that payoff." (Fischer Black, 1995)

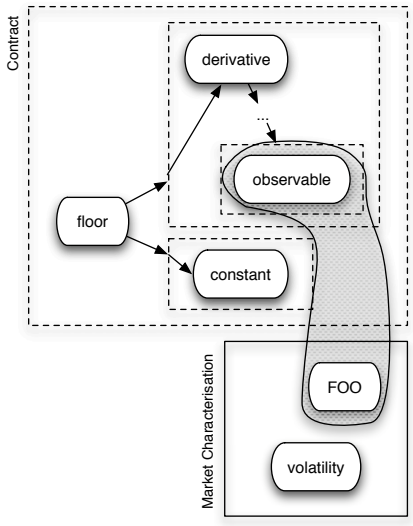


Fig. 1. The schematic of a *plain vanilla european call* modeled by an Active Document (boxes and geometric objects of grey colour represent environments). Combining Active Documents representing a contract and a market characterisation allows for fair price calculations.

Using a *hypertermsheet*, a simplified and intuitive design process supported by the runtime system (see Figure 2) replaces error-prone low-level programming attempts. The explorative composition style is based on easy to understand drag and drop gestures. Design attempts for a set of "standard" contracts, e.g. a *plain vanilla european call*, are automatically detectable and wizard-based *visual constructors* are provided to further simplify the design process.

C. Hierarchical Active Documents

The fact that a derivative contract may be described using a functional programming language [15] indicates the applicability of hierarchically structured description techniques. ComDeCo uses *hierarchical* Active Documents to represent any kind of derivative contract (see Figure 1). Being hierarchical, a XML-based representation is easily inferred, resulting in a standards-based exchange format. The runtime system provides services necessary to instantiate a hierarchical Active Document. The usage of XML offers several advantages.

1) Besides explorative and interactive composition techniques, a programming language like approach may be used for contract design.
2) Standard XML technologies like *XSL* or *XML Schema* allow for transformations, checks or manipulations performed by the framework or third party tools.
3) As XML is a human-readable information representation, users are not locked in proprietary binary formats which require complex reverse engineering efforts in case of migration.

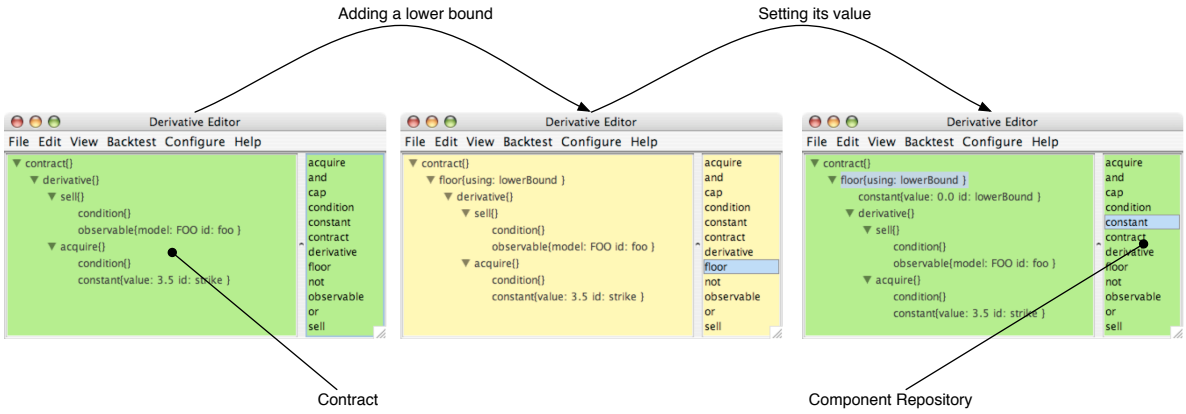


Fig. 2. Starting from a contract based on the underlying *FOO* without a stop loss limit, a *floor* component is added, creating a lower bound protection. Consistent states, i.e. states in which all constraints are fulfilled, are indicated by a green background. Transitional states caused by constraint violations because of incomplete document parts are signaled by a yellow background.

D. Modular composition constraints

COMDECO currently provides a set of ”standard” components which can be used to construct derivative contracts. The simplest contract is represented by a *derivative* component which encapsulates further components such as *sell* or *acquire*.

Example III.2 *A derivative contract which guarantees the right to buy the underlying* FOO[8] *for a fixed price of three monetary units at timestep 0 and the right to sell the underlying at timestep 42 for the then current marketprice is written as:*

```
<contract>
  <derivative>
    <sell>
      <condition>
        <at timestep="42"/>
      </condition>
      <observable id="foo" model="FOO"/>
    </sell>
    <acquire>
      <condition>
        <at timestep="0"/>
      </condition>
      <constant id="strike" value="3"/>
    </acquire>
  </derivative>
</contract>
```

It is obvious that a limited set of components utilisable for the construction of contracts is not desirable. Varying and extensible sets of building blocks have to be supported to cope with the flexibility of derivative contracts. Composition is based on the *decorator pattern* [16] which allows for evolutionary enhancements of already existing contracts.

Example III.3 *To create a* plain vanilla european call *having a payoff of*

$$P(t) = \begin{cases} \max(FOO(t) - 3, 0) & t = T \\ \textbf{unknown} & \textbf{otherwise} \end{cases}$$

the simple contract of the previous example is augmented by an additional floor *component, resulting in*

[8]For example, *FOO* could be a stock or a bond traded on a stock exchange.

```
<contract>
  <floor using="lowerBound">
    <constant id="lowerBound" value="0"/>
    <!— same as before —>
  </floor>
</contract>
```

Components to be used in an ACTIVE DOCUMENT provide *modular composition constraints* which are merged by the runtime system managing the component repository resulting in a *dynamic document specification*. The set of constraints varies depending on the repository’s contents.

1) Structural Constraints: Structural constraints control the overall architecture by restricting the set of possible relations between different entities constituting an ACTIVE DOCUMENT. In case of hierarchical ACTIVE DOCUMENTS, a combination of several standard schema techniques such as *Relax NG* [17], *XML Schema* [18], and *Schematron* [19] may be used. When constraining general-purpose ACTIVE DOCUMENTS, a graph-based specification restricts the set of possible entity configurations.

Example III.4 *A* cap *component only influences a* floor *component or a* derivative *component and its bound may only be represented by a* constant *component or a* observable *component. These constraints are expressed by the following code fragment exploiting the hierarchical structure.*

```
...
<group name="ConstantOrObservable">
  <choice>
    <element maxOccurs="1" minOccurs="1"
             name="observable" type="observable"/>
    <element maxOccurs="1" minOccurs="1"
             name="constant" type="constant"/>
  </choice>
</group>
...
<complexType name="cap">
  <choice>
    <sequence>
      <group maxOccurs="1" minOccurs="1"
             ref="ConstantOrObservable"/>
      <choice>
        <element maxOccurs="1" minOccurs="1"
                 name="derivative" type="derivative"/>
        <element maxOccurs="1" minOccurs="1"
                 name="floor" type="floor"/>
```

```
      </choice>
    </sequence>
    <sequence>
      <choice>
        <element maxOccurs="1" minOccurs="1"
                 name="derivative" type="derivative"/>
        <element maxOccurs="1" minOccurs="1"
                 name="floor" type="floor"/>
      </choice>
      <group maxOccurs="1" minOccurs="1"
             ref="ConstantOrObservable"/>
    </sequence>
  </choice>
  <attribute name="using" type="xs:string"
             use="required"/>
</complexType>
...
```

2) Semantical Constraints: Specifying conditions on composition parameters, an entity's neighbourhood, its current state etc., the set of valid ACTIVE DOCUMENTS is further limited, reducing possibilities of semantical composition mismatches. Semantical constraints are specified using a declarative style, thus separating logic and data implementation, further reducing coupling.

Example III.5 *For the lower bound F and the upper bound C of a* floor *and a* cap *component being related to each other, $F < C$ should always hold. Otherwise, the corresponding payoff formula would be $P = \min(\max(X, F), C)$, which is independent of the value of X in case of $F \geq C$.*

First Order Logic is currently applied for the specification of semantical constraints. As an ACTIVE DOCUMENT consists of components which have a state of their own, a forward-chaining rule engine that makes use of the RETE algorithm [20] is suitable for the semantical constraint checking layer in the *Omnia* framework.

E. Separation of concerns

Potentially varying characteristics like market volatility or the current market value of an underlying *FOO* are not part of the ACTIVE DOCUMENT representing the contract. This ACTIVE DOCUMENT just specifies the existance of such a relation, but the concrete occurence is deferred until valuation. For that reason, when performing price calculations, at least one additional document has to be considered: the market specification[9]. *Observables* represent hooks, that have to be associated with real or simulated market data. In case of the exemplary contract of Figure 1, the fact that the value of the contract is dependent on the performance of the underlying *FOO* is expressed by observable *foo*. By establishing communication links between the observable and its counterpart in the market specification, the pricing engine is able to gather the necessary input parameters to perform the valuation. Letting corresponding entities communicate with each other partially merges the two independent documents.

[9]Note that modeling the market specification as an ACTIVE DOCUMENT is not the only option. Alternatively, the ACTIVE DOCUMENT representing the derivative contract could be valuated by combining market specification data gathered from an appropriate web service, for example.

F. Componentised valuation engine

When solving the pricing problem, there are, roughly speaking, three methodological categories to choose from:

1) Closed-form solutions provide an analytical expression that is used to determine the corresponding contract's market price. However, only a small fraction of the set of imaginable contracts may be priced by hitherto discovered closed-form solutions.
2) Monte Carlo simulation provides an almost general-purpose framework for the pricing of derivative contracts. Unfortunately, algorithms are often computationally intensive.
3) Tree-based algorithms represent a financial engineer's swiss army knife, because any kind of derivative contract may be priced using these algorithms. Nevertheless, the memory footprint of such solutions may impose practical restrictions in multi-dimensional settings.

Since the derivation of the first closed-form solution by Black & Scholes [21] in 1973, their number has grown steadily, but pricing arbitrary derivative contracts using closed-form solutions remains wishful thinking. Monte Carlo simulations and tree-based algorithms provide an almost general-purpose framework, but both are not a panacea: their proper applicability depends on certain properties of the contract to be valuated. Using closed-form solutions when appropriate, apply tree-based algorithms otherwise and offer Monte Carlo simulations as an alternative is therefore the preferred strategy when solving the pricing problem.

Besides that, pricing has to be flexible with respect to valuation models, because the de facto standard *Black & Scholes* is not applicable in case of arbitrary complex contracts. Models and valuation strategies are realised as components, stored in a queryable repository, being plugable on demand.

Operating on pure mathematical descriptions, the determination of the best-fitting algorithm and valuation model is tricky. The ambiguity of *mathematical assembler* consisting of low-level operators like *max* or *min* may cause problems, making global analysation techniques necessary.

Example III.6 *Let a payoff function be given by*

$$P(T) = 3 \cdot S(T) \cdot \prod_{i=1}^{n} \left\lfloor \frac{\min(S(t_i), B)}{B} \right\rfloor$$

Although making use of the $\min$ *operator, the corresponding contract does not consist of any caps. Instead, it is a barrier option.*

High-level concepts, e.g. *floor* or *cap*, help to disambiguate and simplify the design, classification and analysation phase. However, these abstractions cannot completely prevent *spaghetti contracts*[10]. Their construction is just significantly complicated, so end-users should not be able to create them by accident.

[10]The analogon to *spaghetti code* in computer programming, i.e. hard to understand contract specifications.

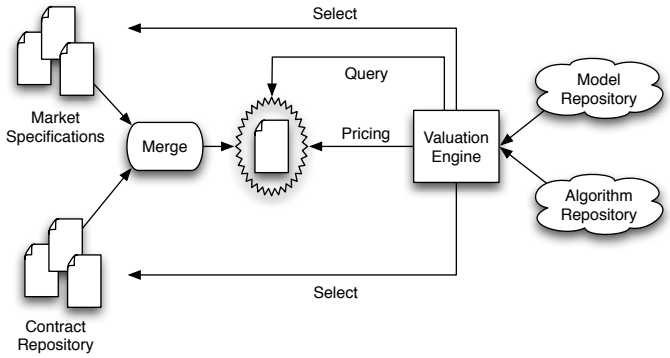


Fig. 3. A conceptual overview of COMDECO's valuation engine. ACTIVE DOCUMENTS provide query capabilities that are utilised for classification purposes. These capabilities may also be used for queries like "List all contracts that have asian-style smoothing and a knockout barrier of 50 €." which are useful for portfolio management tasks.

Since ACTIVE DOCUMENTS are queryable entities, additional information further improving the valuation process can easily be retrieved. The best-fitting approach can be selected with no or little user interaction depending on the information gained by issuing appropriate queries.

G. Further advantages

Besides offering an optimised design and valuation approach that shows up better scalability with respect to reduced turnaround times, ACTIVE DOCUMENTS pave the way for new kinds of applications. Normally, customers of derivative contracts rely on financial engineers for contract design and valuation. Individual contracts imply an increased administration effort, making OTC[11] a viable option only in case of large financial investments. With COMDECO, a non-expert customer is able to autonomously design derivative contracts according to individual needs. An online broker may provide customers access to his contract and component repository, valuate user-designed contracts instantaneously and add them to the customer's portfolio upon acceptance in real time. Even small and medium non-expert investors are able to profit from the benefits, although they do not belong to COMDECO's primary audience. Further advantages beyond the scope of this paper are briefly discussed in [22].

IV. BEHIND THE SCENES

Having sketched a possible usage scenario of ACTIVE DOCUMENTS in the domain of financial engineering, this section gives a brief overview of the technical and conceptual key aspects of the ACTIVE DOCUMENT framework *Omnia* which is developed in parallel during the ongoing COMDECO project.

Any ACTIVE DOCUMENT consists of combinations of the two principal entities *component* and *environment*. In contrast to the majority of current document systems, the document itself is a piece of software that is composed by the user. By providing an intuitive composition environment, non-expert end-users are able to design software according to personal requirements.

Environments are able to control the messages being exchanged between components and components provide services to their neighbourhood (a component's neighbourhood is the set of components that may send service requests to the component or receive responses from it). An arbitrary number of components and other environments may be embedded, letting an environment act like a container.

An important aspect is the communication model which specifies the modus operandi of message exchange between involved entities. Message propagation is controlled by environments that receive messages from their neighbourhood and forward these to the embedded entities. *Omnia* distinguishes two kinds of messages, namely

1) **Intra-environmental messages** which are exchanged between entities embedded in the same environment, i.e. intra-environmental messages never leave the environment they were initially sent into.
2) **Inter-environmental messages** which cross environmental boundaries, e.g. if an intra-environmental message is received by an environment and propagated to its embedded entities, the initially intra-environmental message becomes an inter-environmental one.

The communication model restricts an environment's sphere of action concerning message propagation control. Only inter-environmental messages may be filtered (e.g. blocked) or altered by an environment. Intra-environmental messages are transparently propagated to all embedded entities. This design decision has an almost straightforward analogon in real life. Think of components as the employees of a firm, interacting with each other by means of communication. The employees are spread over several rooms, represented by environments. Assuming not so large rooms, every employee is able to communicate directly with its room mates, whereas communication with employees in other rooms is physically restricted. However, this analogy is not exhaustive, because a component may be embedded into more than one environment and environments may be nested arbitrarily.

A. Partial anonymous communication

Using message multicasting mechanisms, components send messages to their neighbourhood and may receive associated responses. Neither determining the number of entities in a component's neighbourhood nor checking for entities which are able to react to a certain message is possible. In contrast to message sending, message reception is transparent, i.e. receiving entities may determine the initiator of received messages or responses, so distinguishing different senders is an option. From a component's point of view, its neighbourhood is a black box which consumes and produces messages. This *partial anonymity* results in loosely coupled entities, tremendously limiting the set of assumptions a component designer may act on. A component

[11] Over The Counter, i.e. financial products individually tailored according to customer requirements.

- can neither expect a message to be processed by its neighbourhood nor to receive a response for a message being sent and
- can never expect message monotony, i.e. the component's neighbourhood reaction / response pattern triggered by a message being sent may change over time.

B. Dynamic service interfaces

Environments act as containers, structuring both: message propagation and the overall system architecture. From the technical point of view, an ACTIVE DOCUMENT is created by defining the nesting structure of environments, followed by component injections into appropriate neighbourhoods. The embedding relation may change during runtime, i.e. entities are able to join and leave environments arbitrarily. An environment's provided interface is the union of the provided interfaces of its embedded entities. The provided interface of a component is specified in relation to the services the component expects from its neighbourhood. As the neighbourhood may change dynamically, the provided services for a set of components are determined by calculating the transitive closure of the requires / provides specifications of all involved components.

Example IV.1 *The components A, B, C and D are members of a common neighbourhood, each providing services* $\pi^1 \dots \pi^n$*. The following provides / requires specifications hold (an empty right side indicates that the corresponding service is provided independently from any other available neighbourhood service):*

$$\begin{array}{lcl} \pi^1_A & \leftarrow & \pi^1_B \\ \pi^1_B & \leftarrow & \\ \pi^2_B & \leftarrow & \pi^1_A \\ \pi^1_C & \leftarrow & \pi^1_A, \pi^1_D \\ \pi^1_D & \leftarrow & \end{array}$$

The provided services resulting from this specification are $\{\pi^1_A, \pi^1_B, \pi^2_B, \pi^1_C, \pi^1_D\}$*. If component B leaves the common neighbourhood, this changes to* $\{\pi^1_D\}$*.*

C. Assembling & Composing components

The discussion of COMDECO has already introduced two categories of constraints each component may contribute to:

1) Structural constraints
2) Semantical constraints

These and other component characteristics are subsumed in the component's "binary" representation which merges the distinct parts into a structured stream[12].

- **Java bytecode** specifies a component's behaviour. Because of *contextual polymorphism*, there may exist a one to many mapping between messages and behaviour implementations depending on the environment from which messages are received.
- **Requires & provides specifications** explicitly define a component's context dependencies.
- **Composition constraints** specify structural and semantical conditions that have to be met when being composed.

Example IV.2 *The* cap *component in* COMDECO *uses the following assembly specification which is used by* Omnia*'s component compiler to generate the final "binary" component.*

```
<component>
  <messages>
    <message id="getCurrentValue"
             returntype="java.lang.Double"/>
    <message id="getUpperBound"
             returntype="java.lang.Double"/>
  </messages>
  <implementation class="CapFinancialComponentFeature">
    <mapping>
      <map id="getCurrentValue"
           ishandledby="getCurrentValue"/>
      <map id="getUpperBound"
           ishandledby="getUpperBound"/>
    </mapping>
  </implementation>
  <constraints>
    <universe>
      <constraint name="universerule"/>
    </universe>
    <world>
      <constraint name="worldrule"/>
    </world>
  </constraints>
</component>
```

Note that due to implementation specific considerations, the structural specification is *not* part of the assembly specification, but is provided by special method invocations triggered by the runtime system during composition. The mechanisms to determine a component's required and provided services are currently under careful consideration. A combination of static descriptions as part of the assembly specification and a dynamic analysation phase during composition using bytecode inspection techniques is favourised at the moment.

Both semantical as well as structural constraints may be distinguished in local and global specification categories, i.e. constraints may express conditions for the whole system or just for a component's neighbourhood. Technically, local constraints are tied to environments, whereas global constraints are bound to the ACTIVE DOCUMENT. Local constraints are subsumed in the *world section* whereas global constraints are listed in the *universe section* of the assembly specification.

D. Special kinds of environments

In context of COMDECO, a hierarchical ACTIVE DOCUMENT is used to model any kind of derivative contract. Due to the fact that a contract is representable by a *n-ary tree datastructure*, environments being used to construct the corresponding ACTIVE DOCUMENT representation have to obey to additional structural constraints.

- At most one component and an arbitrary number of environments are allowed as embedded entities (*Single component per environment constraint*).
- An environment lets pass a message to its embeddings only if the message-initiating entity is a member of the same neighbourhood as the environment (*Local messages only constraint*).

[12]Instead of using a directory abstraction as in the case of Mac OS X applications, all necessary information is stored in a single file, simplifying cross-platform component deployment.

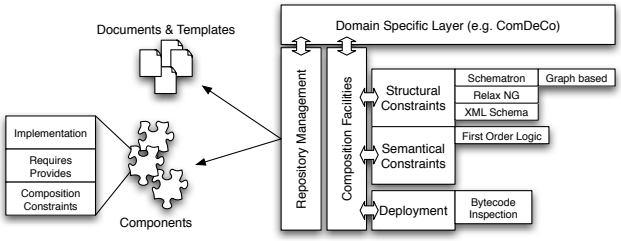


Fig. 4. Overview of the *Omnia* framework which provides the necessary mechanisms to support the development of general-purpose ACTIVE DOCUMENT systems.

Because this kind of environment is essential in the context of derivative contract design, *Omnia* provides a default implementation. Further ready to use specialisations currently exist, e.g. *fallback environments* that allow for inheritance-like messaging mechanisms and *directory-based environments* which allow for rapid prototyping hierarchical environment structures by making use of the filesystem hierarchy[13].

V. CONCLUSIONS

A distinguishing property of ACTIVE DOCUMENTS in comparison to compound document technologies is the capability to express and enforce structural as well as semantical constraints on the involved entities. The well-known and intuitive document metaphor in conjunction with an infrastructure allowing for explorative and dynamic composition operations guarantees easy composition. In comparison with developer-centric component models, user-oriented technologies have to provide a high degree of runtime fault tolerance. In contrast to software developers, ordinary end-users are usually unable to provide appropriate glue code in case of composition mismatches. A high degree of loose coupling is achieved by the component model of ACTIVE DOCUMENTS which minimises potential composition mismatches by explicitly specifying valid compositions. Albeit not being a silver bullet for all problems raised by the component-oriented paradigm, ACTIVE DOCUMENTS offer enhancements and solutions for difficulties showing up when adapting the component-oriented paradigm to the end-user domain. Next generation software based on the concept of an ACTIVE DOCUMENT allows for autonomous adaptations according to individual preferences.

COMDECO's already available proof of concept *contract design tool* demonstrates that the component model's expressiveness is sufficient for the currently investigated domain of financial engineering.

For the e-Learning domain, *Minerva* has already proven conceptual practicability of ACTIVE DOCUMENTS and the results of COMDECO may be used in this context, too. This emphasises the *vanishing application boundaries property* of ACTIVE DOCUMENTS. In fact, there is no distinct *Development Environment for Derivative Contracts*, just an ACTIVE DOCUMENT system which may be used for this purpose or (by adjusting the component repository and the set of constraints) may be turned into an e-Learning environment that is used for in-house training of financial engineers, for example.

Financial engineering and e-Learning are two domains these concepts can be applied to successfully, but almost any application operating on some kind of document on the conceptual level, e.g. IDEs for software development, is a candidate for the transformation into an ACTIVE DOCUMENT.

[13]This kind of environment is currently used to implement the *market specification* and to support debugging within COMDECO.

ACKNOWLEDGEMENT

The author would like to thank the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

[1] Cox, Brad J. and Novobilski, Andrew J.; Object-Oriented Programming - An evolutionary approach. Second Edition. Addison-Wesley Publishing Company Incorporated. 1991
[2] Raskin, J.; The humane interface: new directions for designing interactive systems. ACM Press. Addison-Wesley Pearson Education. 2004
[3] Object Management Group; CORBA Component Model Specification Version 4.0. 2006.
[4] Sun Microsystems Inc.; Enterprise JavaBeans Specification, Version 2.1. 2003.
[5] European Computer Manufacturers Association; ECMA 335: Common Language Infrastructure. Third Edition. 2005.
[6] European Computer Manufacturers Association; ECMA 334: C# Language Specification. Third Edition. 2005.
[7] Apple Computers Inc.; Inside Macintosh: OpenDoc Programmer's Guide. Addison Wesley Publishing Company Incorporated. 1996
[8] McCance, S.; Overview of the GNOME Platform. The GNOME Project. 2006.
[9] EASYCOMP (IST Project 1999-14191). Easy Composition in Future Generation Component Systems.
[10] Reitz, M. and Stenzel, C.; Minerva: A component-based framework for Active Documents. Proceedings of the Software Composition Workshop (SC 2004). Electronic Notes in Theoretical Computer Science 114. Elsevier. 2005
[11] Korn, R. and Korn, E.; Option Pricing and Portfolio Optimization. AMS. Rhode Island. 2001
[12] Dalton, S.; Excel Add-In Development in C/C++: Applications in Finance. Wiley. 2005
[13] Joshi, M.; C++ Design Patterns and Derivatives Pricing. Cambridge University Press. 2005
[14] Reitz, M. and Nögel, U.; Derivative Contracts as Active Documents - Component-Orientation meets Financial Modeling. Proceedings of the 7th WSEAS International Conference on Mathematics and Computers in Business and Economics. 2006
[15] Peyton Jones, S. L. and Eber, J-M.; How to write a financial contract. In: Fun of Programming. Palgrave Macmillan. 2003
[16] Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J.; Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1997
[17] ISO/IEC 19757-2: Document Schema Definition Languages (DSDL) - Part 2: Regular-grammar-based validation - RELAX NG. International Organization for Standardization (ISO)
[18] XML Schema - W3C Recommendation. World Wide Web Consortium (W3C)
[19] ISO/IEC 19757-3: Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron. International Organization for Standardization (ISO)
[20] Forgy, C.; Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. Artificial Intelligence, 19, 17–37. 1982
[21] Black, F. and Scholes, M.; The pricing of options and corporate liabilities. Journal of Political Economy, 81, 637–659. 1973
[22] Reitz, M. and Nögel, U.; Composable Component-Oriented Contracts - How Software Technology may influence Financial Engineering, WSEAS Transactions on Information Science and Applications, Issue 9, Volume 3, 1756–1763. 2006

Component based method for enterprise application design

Emmanuel Renaux
Trigone Laboratory
University of Lille, France
emmanuel.renaux@univ-lille1.fr

Eric Lefebvre
École de technologie supérieure
Montréal, Canada
lefebvre@ele.etsmtl.ca

Abstract

Component support has actually been enhanced with version 2.0 of the Unified Modeling Language and component appears as the best reusable unit of software, whereas more and more pre-built components are made available. However, reuse of components to build system remains a difficult task. Components are mostly identified in the late phases of the system development cycle without considering the end-users' requirements specified in the early phases. The effort required to develop or re-use components which satisfy the requirements is still significant, so that a lot of developers generally prefer to develop a system from scratch, while being largely influenced by technological concerns.

This article presents a Model-Driven Engineering method based on the early identification of business components. Setting up the component identification during use case modeling, transforming requirements into logical components, enriches analysis using UML diagrams. Business archetype concept and component paradigm are jointly used to structure the component-based process. This innovative method demonstrates that 1) components must be identified right from the use case model at the requirement stage of the development cycle and 2) a mechanism must be set up to ensure their traceability along the next stages. Building Platform Independent Models from the use case model, using a set of four Business Archetypes, maintains the consistency between components and requirements, ensures their traceability and facilitates their transformation into Platform-Specific Models, and then into code. In the whole, the proposed method should bring another significant progress to Model-Driven Engineering.

Keywords: Component, traceability, requirements, information systems, engineering process.

1. Introduction

The component idea is today omnipresent in software engineering mainly for development concerns. Support of component during analysis has been enhanced with version 2 of the Unified Modeling Language (UML) [25] but its use is not guided by a clear method. Component appears as the best reusable unit of software, whereas more and more pre-built binary components are made available. Component-Based Development (CBD) [6] has lately been extended to integrate the concepts of service and of Service-Oriented Architecture and then to allow a wide exchange of pre-built off-the-shelf components. In the mean time, Model-Driven Engineering (MDE), following the Model-Driven Architecture (MDA) specified by the Object Management Group (OMG), makes modeling an activity of software production instead of documentation. MDE clearly separates the business models independent of any technology from the system models dependent of them. Both CBD and MDE follow a convergent way, since MDE needs CBD to facilitate reusability during Platform Independent Model (PIM) building, whereas CBD needs MDE to facilitate the interoperability between technological platforms. However CBD and MDE are today not well integrated, so that reuse of components to build a system remains a difficult task. Moreover, decomposing a system into really reusable components is still non-trivial.

The main idea presented in this paper is the design of a system with a better integration of the component paradigm and model driven mechanism in a Unified Process-like [2] approach. We will particularly demonstrate that, because of this lack of integration, the traceability between requirements and components designed during deployment task is far from obvious in current software development processes.

In this paper, we propose to enrich the UML notation with the logical component concept by extending its meta-model and we describe the method to ensure traceability. The extension mainly consists in adding

the logical component and business archetype concepts [23]. Thus, it allows to map business processes as described by the use cases with logical components of the analysis model. We finally discuss the benefits of our approach, according to our current works. A simple case study, the "hotel room management" system illustrates our proposal.

2. The component dimension in current engineering processes

Based on the use case model, the analysis task consists in finding and defining system entities. Design and implementation tasks deal with technological concerns (see Figure 1). The use case approach allows to specify user requirements and to provide an artefact that is understood by each stakeholder. Nowadays, software engineers master analysis of information systems by applying approaches like the Unified Process (UP) [2]. The UML [1] is commonly used to build most of the models. Each model corresponds to a view on the whole system specifications, which is the responsibility of a particular competency [3]. However, working with engineers on actual projects in insurance, health care and banking companies, we observed that there is no structural guide to map use cases with deployable components. Moreover, designers and developers are guided by technology concerns, and therefore often change decisions previously made by business analysts about final users' requirements. Different stakeholders in a project team guided by a current engineering process build several distinct models using UML. Each stakeholder has his own knowledge and responsibilities. Thus, existing tools provide an appropriate view of the system according to stakeholders' different concerns.

The use case model contains the system requirements grouped in coarse-grained functions, i.e. the use cases. Sets of scenarios detail use cases. Dynamic UML sequence diagrams realize use cases and model each scenario of use. A sequence diagram shows interactions between architecture elements to process and realize requirements. Static UML class diagrams, an artefact of the analysis task, define the types of the architecture elements and their relationships. Package diagrams regroup classes to reduce system complexity and dependencies. The design model is the transformation of the analysis model taking into account technological concerns, applying design patterns, proposing new decomposition depending on these new concerns, and so on. A main goal of the design task is to discover UML deployment components and model them in a component diagram. This diagram is a representation of the binary components coded during the development task. Then, deployed components are tested to finally check that all requirements have been realized.

The goal of this simplified description of a UP-like process is not to be exhaustive but to demonstrate that traceability between each of these models is not implicitly supported and therefore not guaranteed. Different models provide mismatched system decompositions. Use case model and sequence diagrams are organized by use cases. In analysis classes diagram, software engineers take care of object-oriented concerns, and of technological interests in design model and component diagrams. Testers come back to the use case paradigm to check that requirements have been completed. An experienced architect or a competent project team leader is the only one with a global perspective of the work. He is responsible to check the consistency between several views, using a matrix mechanism linking requirements with architecture elements of all views. This reduces quality and readability of the system.

Since the technological targets are component based, we claim that component must be the master piece of software and be present all along the development process. According to these issues and context, we list, in the next section, the fundamental features to be identified in a component-based system. The concept of component first appeared in research on middleware [8] [9] to deploy part of software that can be reused in another technological context. Interoperability was the initial purpose of component research. Some formalisms and notations have been elaborated to describe component-based systems. However, design approaches did not appear immediately. Thus, engineers adapted object oriented methods to deal with finding and defining system components. Currently, there exist component-based approaches [4] [6]. They are an adaptation of the UP [2]. Generally, they use and adapt UML and apply common development life cycle that drives developers from use cases to UML deployment and component diagrams. However these models remain relatively independent. Finally, these approaches do not cover a complete MDA process and do not obtain its benefits. UML2 only enriches the class concept with some component concepts like Provided interfaces that represent and categorize operations or services provided by a component and Required interfaces that represent operations and services requested by a component to successfully complete its functions.

The aim of a component-based method is to define the component boundaries, their interfaces and finally their connections to realize all the user requirements. According to these features, we claim that UML diagrams must be enriched with the concept of *logical component*. In the next part, we present in an iterative

way, our method to implement this proposal. We demonstrate that it should be the best unit of software at a conceptual level.

3. Method to transform requirements into logical components

The Information System (IS) engineering major issue is to define the best way to decompose a system, and thus to reduce and to manage its growing complexity. As the complexity of IS due to the large business domains and to the mismatched decompositions depending on the different views in the project life cycle, component based processes require a huge effort to maintain traceability. Logical component concept aims to reduce this effort by partitioning system models and by explicitly representing dependencies between its parts. Our proposal is to add the logical component concept in each view, by encapsulating a group of view constituents, as a membrane. Since each view addresses a specific concern, constituents are of different types, and their traceability is difficult to maintain. As the logical component semantic is the same in the different views, a logical component acts as a pivot to support traceability and should not be transformed.

Figure 1 is the use case diagram of our case study "Hotel room management". The system allows to reserve rooms, to check-in, and check-out. The accountancy concerns are processed. When a customer leaves the hotel and checks out, he pays to an employee who validates the payment.

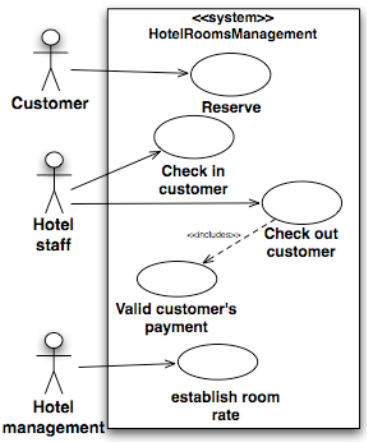


Figure 1 - Use case diagram of the "Hotel room management" system

We named our proposal CUP as *Component Unified Process* [24] because, it adapts the Unified Process by extending it with the logical component concept. We propose to split the system model in four views. The different views of our process are presented with examples from the case study. The CUP meta-model of the method links these four views. Any constituent depends on each other. The four views are the framework of the method and structure its presentation.

One of the first activities in a design process is the identification and the specification of the system functions. The habits are to use a UML use case diagram to do this. Anyone can understand this diagram, without any technical knowledge. It plays the role of a boundary object between each stakeholder. CUP introduces in the **use case view** the concept of logical component. It is an innovation because it allows an early identification of the component of the system. **Component design view** is a white box view representing the internal static structure of the logical component, its provided and required interfaces. The **interaction view** shows dynamically the interactions between each internal object of a component. This view is close of a UML sequence diagram. Moreover, CUP introduces external object interactions which specify that an object inside a component needs a service outside of it. This view specifies the dynamic of a component in an independent way. It shows interactions between parts and through interfaces of a component to complete use-case scenarios. The whole system is represented in the **assembly view** as an assembly of connected logical components. This black box view only shows provided and required interfaces. It does not consider their realization.

CUP is an iterative process. Thus, we present two iterations. The first one proposes a straight application of the method. It is a quasi-systematic way to proceed. The resulting model consists in decomposing the system into primitive logical components. The second iteration enriches the first one by detailing the features. It proposes a simple mechanism to support analysis. The system is represented by composite logical components which are a merge of the primitive ones.

3.1 Primitive logical components design

Use case view

A logical component is a membrane, which contains a subset of model elements. Requirements are modeled by a set of use cases. Our proposal is to identify logical components when building use cases. With the use case view of the system, we first propose to encapsulate each use case in a logical component. In that sense, a use case will be realized by elements enclosed within this logical component in each view of the system. We name this concept a *Primitive component*.

Component design view

Object oriented analysis consists in providing an abstraction of the real world. Object idea is twofold. Firstly, it has properties which represent its state. Secondly, it has operations which implement its behavior. A basic way to represent the real world is to group entities which have common state and behavior within an object. This is a wrong way to do, because objects are too small to be re-used and are too dependent of their context. We propose to use archetypes [23] to define a better way to create a more re-usable piece of software. Archetypes contribute to solve the issue of what is the best reusable module in the software. We propose the use of four business archetypes [23], one business archetype, which represents the dynamics of the business process whereas the three others represent the statics of the business entities involved in the process.

The Party, Place, or Thing (PPT) archetype depends on the subjacent entity. It allows to characterize an object as an entity which has properties and operations referring to business data and to business behaviour. It is used in one or several business processes. The archetype *description* models a record of data attached to a *PPT*, as, for instance, the Category of a room (see Figure 2).

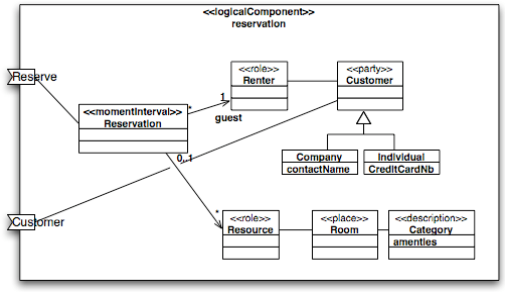


Figure 2 - Static diagram of Reservation and Check-in use cases

The *moment-interval* archetype refers to a business process. It is a session-long life object. It allows to model the purpose of a business process, or a use case as the Reservation, in our example.

The *role* archetype allows to link a moment-interval which has a short life with a specific purpose to the business entities represented by a *PPT* archetype. With a *role* archetype, we can represent the way a *PPT* is involved in one or several contexts of use. There is a strong link between the use case definition and the business process one. As we know that different companies share similar business processes and that a business process can be modelled by the four archetypes, we can link requirements with analysis artefacts in a readable way. Links are established by identifying roles of objects, from a really simple and straightforward object oriented analysis. We then maintain traceability, with the *moment-interval* archetype.

Interaction view

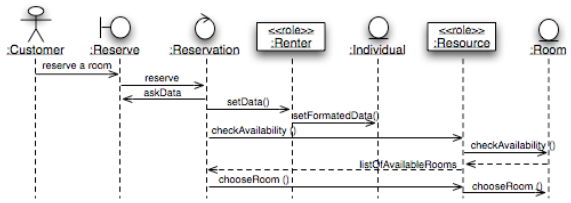


Figure 3 - Interaction view of the primitive

The interaction view (see Figure 3) partially models collaboration. Handling this view to find collaborating objects should help to discover components and to establish their boundaries. The view uses graphical UML analysis stereotypes [12] to represent instances of archetypes and traditional classes. **Entity**: *PPT* archetypes can be represented by an *entity* analysis stereotype. It is a business object containing business data and behaviour. **Control**: a *moment-interval* represents behaviour and data contained in an object which exists only during a transaction, we use a *control* analysis stereotype to represent it. **Boundary**: port is an interaction with another component, as an actor link is a relationship with a human or a computer system. We therefore use a *boundary* analysis stereotype to represent a port or an actor interaction.

Assembly view

An assembly view gives a black box point of view of the logical component (see Figure 4) and connections with other ones.

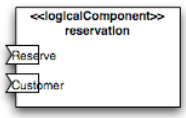


Figure 4 - Assembly view of the primitive logical component "Reservation"

Identification of primitive logical components in the use case view corresponds to Jacobson proposal of the use case module. Jacobson defines a use case module [10] to deal with that issue. "*If we could keep use case and its realization separate, and maintain that separation, we would get a system really simpler to understand, to change, and to maintain*". Unfortunately, he demonstrated that tangling (a component contains code that realizes several use cases) and scattering (a use case is realized by the code of several connected com-

ponents) phenomena do not allow splitting a system in use case modules only.

3.2 Composite Logical Components discovery

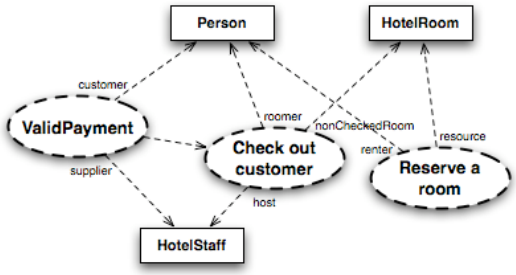


Figure 5 - UML collaborations

As we saw before, the use case is not the best module of software reuse. Dependence between objects is dramatically hard to manage. If we keep separate objects in different logical components, the complexity remains identical. UML 1.4 [1] specifications define a collaboration as a set of roles played by objects and their interactions [26] (see Figure 5). Each interaction of an object enrolled in a collaboration is represented by a link, which is characteristic of its role in this interaction. For instance, a *Person* object is enrolled in the *ValidPayment* collaboration, and has *customer* role within it. The same object *Person* is enrolled in *Check-out customer* collaboration, and has a *host* role with other responsibilities. As the use case concept, collaboration has a useful goal. Thus, a use case can be transformed into a set of one or several collaborations which share the same role. This is a crucial logical link between the use case model and the analysis model which the process must save.

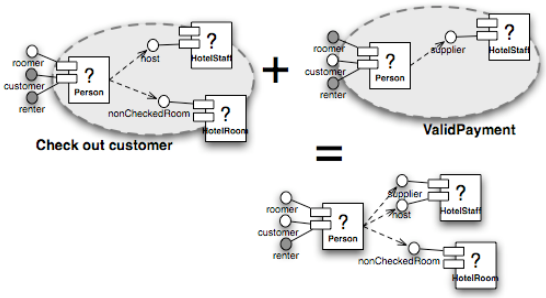


Figure 6 - Design frameworks composition

Logical interfaces, i.e. responsibilities, are added to an object, for each collaboration within which it is olled. To specify and then implement the whole object, it is mandatory to know all its responsibilities [4]. Named design framework in Catalysis method, a composition mechanism allows to define a complete specification of objects (see Figure 6). This mechanism achieves to explicit precisely how a use case is realized in the analysis task. It shows why it is difficult to keep traceability manually, because of the many choices to do.

We apply this composition mechanism to propose a heuristic to discover logical component boundaries. In our *Hotel room management* example, we group use cases as follows. Use cases are realized by collaborations which enroll objects of the system. The goal is to minimize dependencies between components. To achieve it, we do not authorize the enrollment of an object in collaborations which realize a use case encapsulated in another logical component. Sometimes, it is not possible and we add required interfaces to define interactions between components as actors interactions. The use case view (see Figure 7) shows two logical components that encapsulate sets of use cases. The interactions represented in are crossing logical component boundaries through interfaces between components. The second iteration of the process illustrates and explains the composition mechanism that creates composite logical components, in each view.

Use case view

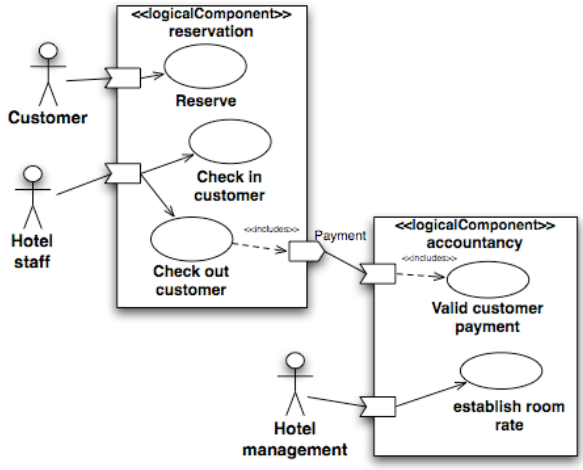


Figure 7 - Use case view of "Hotel room management" system

Regarding subjacent object collaborations realizing uses cases, some of the use cases can be grouped in the same component (see Figure 7). This kind of logical component is called *composite*. We choose this configuration containing two components, one for reservation and occupation of the room and one for accountancy concerns. Designers are free to choose the best selection, the one that reduces interactions and dependencies through component boundary, or simply the one that takes into account SI constraints. Figure 7 results

from some choice made by the designer. Boundaries of primitive logical components are not commonly kept in order to enhance the model readability.

Component design view

The goal of a UML collaboration [1] between a set of objects is to identify the best reusable piece of software, not the objects themselves. The use of archetypes can answer the issue: how to design collaborations? We propose to use archetypes to represent in a class diagram the associations between objects in one collaboration represented by a *moment-interval* (see Figure 8). A collaboration is typically designed with sequence diagrams. But sequence diagrams, which represent use case scenarios, detail interactions between objects. Moreover, the class diagram is no longer linked with the sequence diagrams that allow to define roles of objects.

A composite logical component applies the composition mechanism. As in a use case view, primitive logical components boundaries are not kept. But in this view, the designer has to choose the *PPT* objects. In the example (see Figure 8), the designer chooses to group the Reservation, Check-in and Checkout use cases. As views have to be consistent, the designer in charge of the component design view, groups the corresponding *moment-intervals*. The three *moment-intervals* define the roles of the *PPT* objects. As in the composition of frameworks, each role must be the responsibility of one entity object. In this view, the designer has also to define the interfaces of the *PPT* objects. Another decision has been to keep the Room entity outside of the component to enhance its reusability. This decision has been made in the assembly view presented in the next section.

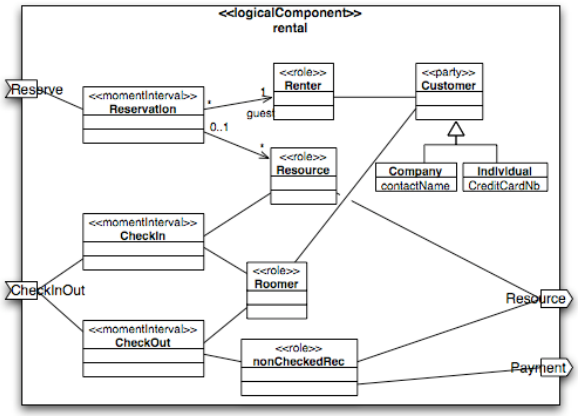


Figure 8 - Component Design view of the "Rental" logical component

In a collaboration, a set of objects interacts for one goal, as specified in the requirements. It is really difficult to reuse a part of a collaboration, i.e., a subset of the participating objects. Because they are linked by the collaboration, they have responsibilities corresponding to the role they have in this collaboration. Then, the analyst's goal is to define the best boundary. Archetypes provide a way to explicit collaboration in an object-oriented vision. They explicit interactions between objects enrolled in one or more collaborations.

Thanks to the archetypes, whatever the design choices are, traceability is maintained. The *moment-interval* archetypes always represent use cases and their related business process.

Assembly view

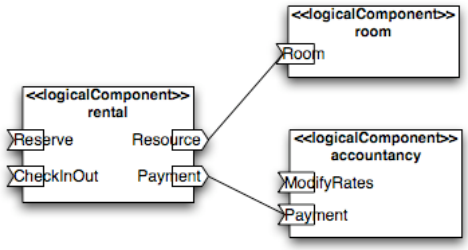


Figure 9 – Assembly view

In this view, an architect decides to isolate room entities. Then the Rental component becomes independent of what is rented and is reusable in another context. Actually, the main concern of this view is the reusability of the logical components.

Interaction view

As we saw before (see Figure 7), a use case view allows to decompose a system in logical components, grouping a set of use cases. Standard relationships between use cases, *extends* and *includes* are kept. The difference with a traditional use case diagram is that these relationships can cross component boundaries through interfaces of the logical components. They explicitly show the functional dependencies between components. It is obvious that there are interfaces in other views, which drive the designer to discover these new interfaces.

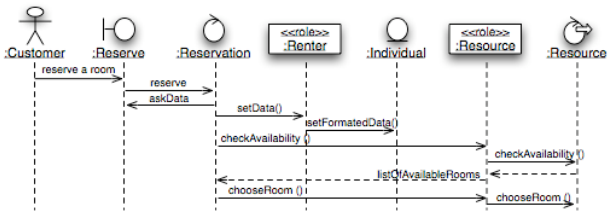


Figure 10 - Interaction view

In the interaction view, we add the externalControl stereotype for the *Resource* object. This analysis stereotype represents the component required interfaces. It shows the dependencies of the component with its context, while keeping autonomous the component.

4. Implementation and evaluation

4.1 Method implementation using MDE mechanisms

CUP is specified by an extension of the UML meta-model divided in four parts (see Figure 11). The concept of logical component has been added, with its required and provided interfaces, and the externalControl role. As the notation used in the method is described in a meta-model, its implementation can be facilitated by a modeling tool such as ModX [27], or by the IBM Eclipse Modeling Framework [28].

Object constraint language rules [1] have been defined to constrain and guide the design. They define consistency relationships between each model element. Once the PIM has been built (➔**1,2**), these tools can generate code according to some generation rules (➔**3,4**). As the properties of our component model are independent of any technological aspect, this enables the translation of a CUP component into several target platforms. In order to define mapping rules, we just need to associate a CUP concept (described by a meta class) with a platform specific concept. Secondly, at each model level, the design can induce modifications at another model level. The tool will inform the designer to carry out these modifications in order to restore the design consistency of the complete model.

The CUP approach has been experimented in actual projects in large business domains, i.e. insurance, banking, health care. It has been proven that applying such division early in the development process, and maintaining consistency with a tool is more efficient. Then traceability is more maintainable. The impact of an evolution is more quickly detected. The efforts are more easily evaluated. Some components have been reused in different projects. It is a practical and efficient way to validate the method and its benefits about requirements traceability.

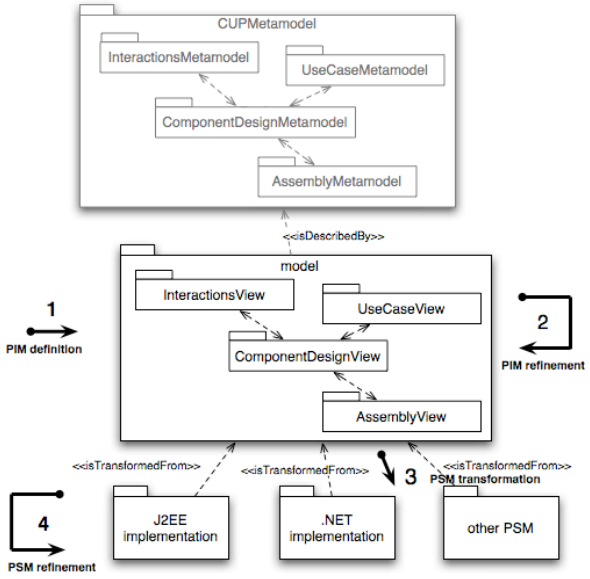


Figure 11 - Model Driven Engineering process

4.2 Related works

Components are often introduced too late in the process, according to technological concerns as in the popular UP process [12]. We propose with CUP to deal with component boundaries very in the process, during the requirements phase to ensure better traceability. Then, all stakeholders can participate in the component identification of their information systems. Benefits of a model-driven approach, such as CUP, are that, with design methods, analysts and designers focus on functional concerns, leaving aside the technological ones. Then the produced PIM is more re-usable, and could be more easily transformed into a Platform Specific Model, then into code.

Existing component-based methods often come with a complexity of use. Catalysis [4] is one of the most complete method, but is really difficult to implement. CUP is formalized by a meta-model, and its related constraints, and then MDE tools [27] can guide designers with more flexible processes and tools.

Finally, the concept of logical component guarantees a more focused and more efficient work, and as it is omnipresent, it guarantees consistency between views, from requirements to deployment. Regarding UML 2.0 [25] components, that are enriched classes, CUP ones, are more as a framework, i.e. a set of classes as [5]. Then reusing these components provides a better return on investment, because it includes the reuse of all the models of all the component views and

allows to connect these components without any technical concern.

5. Conclusion

In this paper, we propose an innovative method to ensure requirements traceability in the project development cycle. First, a use case based solution is used to express requirements. CUP allows an early identification of logical components in the use case view to be decided by each stakeholder of the project. The introduction of archetypes increases the quality of the system model, by checking the functional division of the system. Then, collaborations that realize use cases are explicitly designed with *moment-interval* archetypes in the component design view. Analysis based on archetypes, helps to find and to consolidate boundaries of logical components. Finding the required and provided interfaces completes the PIM building. Traceability has been ensured by the subjacent meta-model which links the different views. Then generative tools [27] can exploit the results thanks to MDE technology.

Dealing with component identification early in the process makes easier the component re-use. For instance, an existing component answering one or more use cases can be early detected. The re-use of business components will be favored by our approach and allows to build more efficiently future information systems.

References

[1] OMG. UML1.4 – Unified Modeling Language. Object Management Group, September 2001.

[2] Grady BOOCH, Ivar JACOBSON, and James RUMBAUGH. RUP Software Engineering. 1997.

[3] Kruchten P., « Architectural BluePrints -- The "4+1" view Model of Software Architecture », IEEE Software 12, pages 42-50,1995.

[4] Desmond Francis D'SOUZA and Alan Cameron WILLS. Objects, Components, and Frameworks with UML - The Catalysis Approach. Addison-Wesley, 1998.

[5] John CHEESMAN and John DANIELS. UML Components - A Simple Process for Specifying Component-Based Software. Addison-Wesley, 2001.

[6] Peter HERZUM and Oliver SIMS. Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise. Wiley Computer Pub., 2000.

[7] Hassine, I., Rieu, D., Bounaas, F., and Seghrnouchni, O. Symphony: a Conceptual Model based on Business Component. Revue ISI 7, 4, HermSs, 2002.

[8] OMG, « CORBA 3.0 New Components Chapters », OMG ptc/2001-11-03, Object Management Group, Novembre, 2001.

[9] Sun, E.J.B. Home Page, 2001.

[10] Jacobson I., « Basic Use Case Modeling », ROAD 1, 1994.

[11] Jacobson I., « Use Cases and Aspects - Working Seamlessly Together », Journal of Object Technology, vol. 2, num. 4, ETH Zurich, pages 7-28, July-August 2003.

[12] Jacobson I., Booch G., Rumbaugh J., 97, « The Unified Software Development Process », Addison-Wesley, 1997.

[13] Mellor, Stephen and Balcer, Marc, Executable UML, Addison-Wesley, 2002

[14] Kruchten, Philippe, The Rational Unified Process-An Introduction, 2nd edition, Addison-Wesley, 2000.

[15] Gamma, Eric et al, Design Patterns, Addison-Wesley, 1995.

[16] IBM Corporation, Business Systems Planning, Information Systems Planning Guide, Publication No. GE20-0527.

[17] Kerner, David, Business Information Characterization Study, Data Base 10, No.4, pp.10-17, Spring 1979.

[18] Carlson, W.M., Business Information Analysis and Integration Technique (BIAIT)-The new horizon, Data Base Vol.10, No.4, pp. 3-9 Spring 1979.

[19] Lefebvre, Éric, Améliorer les méthodes de planification informatique: une approche pluraliste, Thèse de doctorat, Université Grenoble II, avril 1996.

[20] Fowler, Martin, Analysis Patterns, Addison-Wesley, 1996.

[21] Taylor, David, Business Engineering with Object Technology, Wiley, 1995.

[22] Jacobson, I., et al., Object-oriented Software Engineering: a Use-Case Driven Approach, Addison-Wesley, 1992.

[23] Coad, Peter, Lefebvre, Eric and De Luca Jeff, Java Modeling in Color with UML, Prentice Hall, 1999.

[24] Renaux E., Caron O., Geib J.M., SEA - IASTED, Marina del Rey - Los Angeles Nov. 2003, The CUP Project - Component Unified Process

[25] UML 2.0 Infrastructure : OMG doc. ad/00-09-01

[26] Cariou E., Beugnard A., Jézéquel J.M., « An Architecture and a Process for Implementing Distributed Collaborations, EDOC'2002, 2002.

[27] Le Pallec X., Renaux E., Olavo Moura c., ModX - a graphical tool for MOF metamodels, ECMDA-FA'2005, Open Source and Academic Tools, November 7-10th, Nuremberg, Germany

[28] IBM, Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf