# Correctness of Model-based Software Composition (CMC)

**- Proceedings -**

ECOOP 2003 Workshop #11
Darmstadt, Germany -- July 22, 2003
`http://ssel.vub.ac.be/workshops/ECOOP2003/`

In Association with the
17th European Conference on Object-Oriented Programming
`http://www.ecoop.org`

Ragnhild Van Der Straeten, Vrije Universiteit Brussel, Belgium
Andreas Speck, Intershop Research, Germany
Elke Pulvermüller, Universität Karlsruhe, Germany
Matthias Clauß, Solutionline CSS GmbH, Germany
Andreas Pleuss, University of Munich, Germany
(Eds.)

Universität Karlsruhe
Fakultät für Informatik / Institut für Programmstrukturen und Datenorganisation (IPD)
Adenauerring 20a
76131 Karlsruhe, Germany

Universität Karlsruhe
Fakultät für Informatik
Interner Bericht (Internal Report)
Technical Report No. 2003-13
July 2003

# Preface

This proceedings contains the contributions to the Workshop on Correctness of Model-based Software Composition, held in conjunction with the 17th European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany on July 22, 2003.
While most events concentrate on realisations of composition on the technological level this workshop aims at closing the gap of ensuring the intended composition result supported by the usage of models.

Two important problems in composition are first how to model the different assets (such as components, features or aspects) and second the composition of assets such that consistency and correctness is guaranteed. The first problem has been addressed in the Workshop on Model-based Software Reuse (ECOOP 2002). The latter problem occurs when dealing with, e.g., component interoperability, aspect weaving, feature interaction and (on a more abstract level) traceability between different views or models.
One approach to deal with the composition problem is to use models allowing to model the composition. This allows checking the interoperability of the different assets to compose, the correctness of the configuration of assets and predicting properties of the assembled system (especially compliance with user requirements). In case of problem detection suitable resolution algorithms can be applied.

10 reviewed contributions give an overview about current research directions in correctness of model-based software compositions.
Results from the discussions during the workshop may be found in the ECOOP 2003 workshop reader to be published by Springer LNCS.

The web page of the workshop as well as the contributions of this proceedings may be found at URL:
http://ssel.vub.ac.be/workshops/ECOOP2003/

Affiliated to previous ECOOP conferences a related workshop about feature interaction (ECOOP 2001) and an additional about model-based software reuse (ECOOP 2002) have been held. Their contributions are published as technical report No. 2001-14 and as technical report No. 2002-4, respectively, at the Universität Karlsruhe, Fakultät für Informatik.
URLs:
http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/
http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=/ira/2001/14

http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ECOOP2002/
http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=/ira/2002/4

We would like to thank the program committee for their support as well as the authors and participants for their engaged contributions.

The Workshop Organisers
Ragnhild Van Der Straeten, Andreas Speck, Elke Pulvermüller, Matthias Clauß, Andreas Pleuss

## Program Committee

Liping Zhao, University of Manchester, UK
Jilles Van Gurp, University of Groningen, Netherlands
Shmuel Tyszberowicz, Tel Aviv University, Israel
Wilhelm Rossak, Friedrich-Schiller-University Jena, Germany
Silva Robak, University of Zielona Gora, Poland
Kim Mens, Université catholique de Louvain (UCL), Belgium
Shmuel Katz, The Technion Haifa, Israel
Mario Jeckle, DaimlerChrysler Research, Ulm, Germany
Heinrich Hussmann, Dresden University of Technology, Germany
Dirk Heuzeroth, Universitaet Karlsruhe, IPD, Germany
Maurice Glandrup, University of Twente, The Netherlands
Bogdan Franczyk, University of Leipzig, Germany
Kai Boellert, Zurich-Group, Germany
Marie Aimar, Université Bordeaux1, France

## Organisation

Ragnhild Van Der Straeten
Vrije Universiteit Brussel, Belgium
System and Software Engineering Lab
Email: rvdstrae@vub.ac.be

Andreas Speck
Intershop Research, Germany
Email: a.speck@intershop.com

Elke Pulvermüller
Universität Karlsruhe, IPD, Germany
Email: pulvermueller@acm.org
WWW: http://www.info.uni-karlsruhe.de/~pulvermu/

Matthias Clauß
Solutionline CSS GmbH, Germany
Email: matthias.clauss@gmx.de

Andreas Pleuss
University of Munich, Germany
Institute for Computer Science - Media Informatics Group
Email: andreas.pleuss@gmx.de

# Table of Contents

**Composition**

Verification of Software Composed From Components ........ 1
*Martin Rösch (General Objects, LTD, Germany)*

Architecture of an XML-based Aspect Weaver ........ 9
*Eduardo Kessler Piveta (Centro Universitário Luterano de Palmas / Universidade Federal de Santa Catarina, Brazil) and*
*Luiz Carlos Zancanella (Universidade Federal de Santa Catarina, Brazil)*

Preserving Real Concurrency ........ 15
*James Ivers (Carnegie Mellon University, USA) and*
*Kurt Wallnau (Carnegie Mellon University, USA)*

How to lock a model from a miscomposition of objects? ........ 23
*Marie Beurton-Aimar (Université Bordeaux 1, France),*
*S. Pérès (Université Bordeaux 2, France),*
*N. Parisey (Université Bordeaux 2, France) and*
*J.P. Mazat (Université Bordeaux 2, France)*

Composing Dynamic Hyperslices ........ 29
*Ruzanna Chitchyan (Lancaster University, UK) and*
*Ian Sommerville (Lancaster University, UK)*

**Correctness**

Correctness of Model-based Component Composition without State Explosion ........ 37
*Paul C. Attie (Northeastern University, USA),*
*David H. Lorenz (Northeastern University, USA)*

Consistency Validation of UML Diagrams ........ 45
*Boris Litvak (Tel-Aviv University, Israel),*
*Shmuel Tyszberowicz (Tel-Aviv University / Academic College of Tel-Aviv Yaffo, Israel) and*
*Amiram Yehudai (Tel-Aviv University / Academic College of Tel-Aviv Yaffo, Israel)*

Use Case-Oriented Software Architecture ........ 55
*Eliezer Kantorowitz (Technion Haifa, Israel) and*
*Alexander Lyakas (Technion Haifa, Israel) and*
*Arthur Myasqobsky (Technion Haifa, Israel)*

Observing Component Behaviors with TLA ........ 65
*Nicolas Rivierre (France Télécom R&D, France) and*
*Thierry Coupaye (France Télécom R&D, France)*

Validation of Business Process Models ........ 75
*Andreas Speck (Intershop Research AG, Germany),*
*Elke Pulvermüller (Universitaet Karlsruhe, Germany) and*
*Dirk Heuzeroth (Universitaet Karlsruhe, Germany)*

# Verification of Software Composed From Components

Martin Rösch, General Objects LTD
Email: mr@generalobjects.com
Tel. +49(177)773-9000

*Software components are much more similar to mechanical components than many software developers believe. Software components can be planned, verified, produced and assembled just like mechanical components. In fact, the only difference is that they are not visible. This paper describes how components can be verified individually and interacting in a composition. It reports about actual project experience where all the techniques described have proven to work technically though imposing some cultural challenges for the software developers involved. The paper presents techniques for describing components, guidelines for modelling, tools for verifying them, and finally a look at the human side of things.*

## Background

This paper is based on actual project experience which has been gathered since 1994 in about a dozen customer projects. The largest project involved over 100 developers, building 17 components, based on 4,000 documented requirements and 11,000 automated test.

## Basics

The following techniques are the basis for the work presented in this paper.

- Technique: Complete and automated verification of individual UML models
- Technique: Automated requirements tracing to UML classes (and vice versa)

*Basic techniques (Objects 9000)*

## Verifying Component Software

Building on the above mentioned basics it is technically possible to verify components – individually and interacting in a composition. The following techniques (as well as the basic techniques) have all been proven to work in real projects. So there is nothing speculative about them, even though some may sound unfamiliar or outright impossible at first glance.

- Technique: Modelling a component, together with its interfaces, as 1 UML model
- Technique: Automated identification of requirements that have impact on neighbours
- Technique: Faking the behaviour of neighbouring components

*Techniques for verifying component software (Components 9000)*

With these techniques it is possible to describe and verify each component individually and interacting with other components, *using* views provided by other components (imports) and *offering* views on the inner workings of a component (exports) to be used by others.

Because these techniques enable automated, thus reliable requirements tracing, developers of a component can identify those of their own requirements that are important to other groups of component builders who use their component (exports). But developers can also determine how their own component depends on requirements that must be fulfilled by others (imports).

## Verification of Individual UML Models

This technique, a part of Objects 9000, has laid the groundwork for everything else presented in this paper. It combines terminology from ISO 9000:2000[ISO9000] (e.g. verification, validation) with object-orientation and allows the verification of individual UML models.

The verification process of Objects 9000 starts with the users' requirements. Each of these requirements must be documented, together with tests. Then the UML model is constructed.

In order to automate the verification of the UML model, it is implemented using a simple software architecture: 1 address space, no database, no user interface and no network. Then, the tests are programmed just the same, as scenarios: first a *situation* is set up, then an *action* is performed and finally the actual results of each test are compared to *expected results*.

If all tests of a requirement produce their expected results, the requirement is considered to be *fulfilled*. And if all requirements for an UML model are fulfilled, the model itself is regarded as *verified*.

Here is a short overview of ISO 9000 terminology[ISO9000] used for Objects 9000:

**verification:** confirmation, through the provision of →*objective evidence*, that →*specified requirements* have been fulfilled (page 30)

**objective evidence:** data supporting the existence or verity of something (p30)

**specified requirement:** a specified →*requirement* is one which is stated, for example, in a document (p19)

**requirement:** need or expectation that is stated, generally implied or obligatory (p19)

**conformity:** fulfilment of a →*requirement* (p26)

**nonconformity:** non-fulfilment of a →*requirement*. (p26, the German term is „Fehler")

*ISO 9000 terminology used in Objects 9000*

**validation:** confirmation, through the provision of →*objective evidence*, that the →*requirements* for a specific intended use or application have been fulfilled. (p31)

**defect:** non-fulfillment of a →*requirement* related to an intended or specific use. (p26)

*ISO 9000 terminology NOT USED*

It is vital to note two important differences between "verification" and "validation":

1. verification demands that *specified* requirements are fulfilled, whereas validation extends this to *generally implied or obligatory* requirements – even if they never were written down by anybody.
2. verification does not refer to *a specific intended use or application*, whereas validation does – without demanding the intended use or application to be specified in writing.

Actually the difference between the two is so great, that the text of the norm ISO 9000 contains a warning against validation (p26): *"The distinction between the concepts defect and nonconformity is important as it has legal connotations, particularly those associated with product liability issues. Consequently the term 'defect' should be used with extreme caution."*

Objects 9000 was first applied in 1994 in Stuttgart at the Gebäudeversicherung Baden-Württemberg: Since then it has been used in a number of projects, among others at Sparkassen Informatik in Duisburg and Credit Suisse in Zurich. The first commercial tool supporting it (with minor extensions), is Rational Suite (RequisitePro, Rose and TestManager) from IBM Rational[IBM].

### *Limitations*

1. In order for Objects 9000 to work properly, the tests for each requirement must be *formulated in a positive way*, like “in situation a, if action b is performed, then the result must be c”. We have not found a way to automate *negatively* formulated tests like e.g. “the outcome of operation op1 should never be 100”. This reformulation from negatively formulated requirements and test into positively testable ones is where the “ingenious” mind of software engineers is challenged – much the same way as it was when industrial engineering was born, at the Ford motor works about 100 years ago.
2. While a UML model can be verified against a set of requirements, we have not found a way to make sure that *the requirements themselves are correct*. This must remain the responsibility of those who specify the requirements, not of those who write the software.

### *How the Other Techniques Build Upon Objects 9000*

**Automated requirements tracing:** The automated requirements trace uses information from the execution of the tests, and it records information about which test called which operation.

**Modelling a component in 1 UML model:** By mapping 1 component to 1 UML model, the verification technique described above can be applied to components.

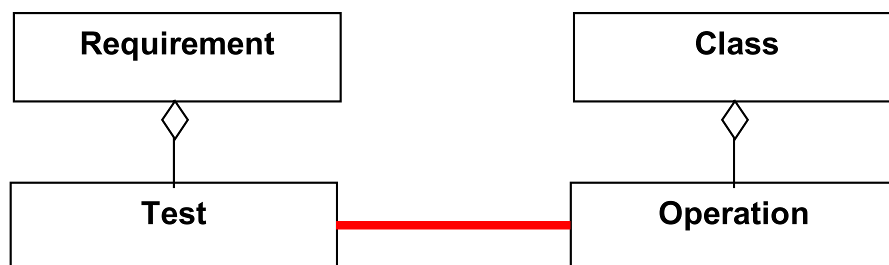
**Automated identification of requirements that impact other components:** The imported classes from other components are verified as an integral part of a component’s UML model.

**Faking the behaviour of neighbouring components:** By faking the *implementations* of neighbours, a component’s UML model can be tested unchanged, alone or in a composition.

## Automated Requirements Tracing

The automated execution of the tests in Objects 9000 creates an interesting by-product: a reliable trace between the requirements and the verified UML model. This is possible because

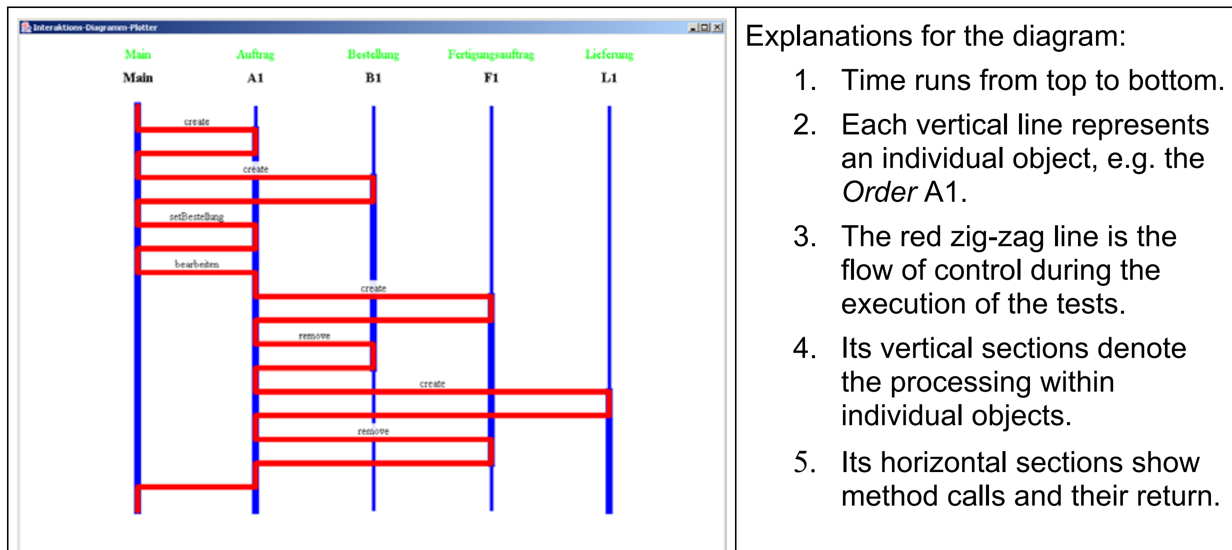
a) each test belongs to exactly 1 requirement
b) each operation belongs to exactly 1 class of the UML model, and
c) each test is performed by (directly and indirectly) invoking operations on objects




*Part of the internal UML-Model of Objects 9000*

In this picture the internal structure of Objects 9000 is shown as an UML model itself: The left half of this model shows *Requirements* and their associated *Tests*, while the right hand side shows the *Classes* and *Operations* that are part of the UML model of the software to be built. The big red link in the middle is maintained automatically while the tests are executed.

A trace of each test can also be shown visually, as shown in the following diagram:




*Automatically generated sequence chart for one test, showing objects, operations & time*

This trace will be needed for the technique which identifies the impact of the requirements of a component on its neighbouring components.

## Modelling a Component in 1 UML Model

When modelling a complete system, composed of multiple components, each component is represented by 1 UML model.

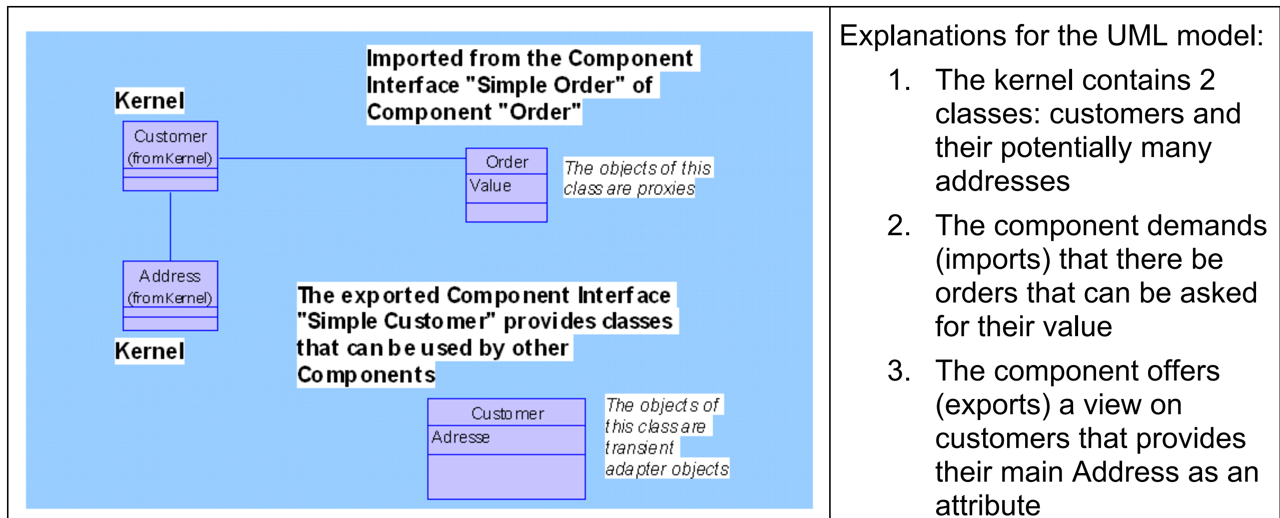
The UML model for a component contains three types of classes:

1. *Kernel classes*. The objects of these classes do the real work of the component. Their structure and their level of granularity represent exactly what and how the builders of this component think about its functionality. Example: the objects of a *Customer* class may be associated with one or more *Address* objects (street address, P.O. Box address, delivery address, etc.).
2. *Exported classes* that this component makes available to other components. The exported classes may expose the kernel classes as-is, but in most situations so far they have provided some sort of simplified view upon the core classes. Example: An exported class (e.g. class *Customer* of component *Customer*) may pick one of the potentially many *Address* objects of its corresponding *Customer* object and make it available as an attribute of itself. This way of hiding the inner complexity of the *Customer* component makes life easier for other components and their builders.
3. *Imported classes*. These classes are exported by other components and used by this one. The objects of an imported class usually are accessed via generated (CORBA style) proxy objects that pass on method calls to the exporting components' objects. Example: The marketing department may have an *Order* component which needs to access a *Customer* object's mailing address. To this end it imports the *Customer* class that is exported by the *Customer* component.

When compared to a component of an automobile (e.g. its motor), the similarities are obvious:

1. A motor has its internal structure, most of which is irrelevant to other components
2. A motor has exported features, e.g. the gear (Zahnrad) that transmits the power
3. A motor has imported features, e.g. the electrical input for the starter (Anlasser).
4. If a motor is to be tested standalone, the electrical input must be simulated

Example: The following diagram shows an UML model with 2 kernel classes (left, *Customer* and *Address*), one imported class (*Order*) and one exported class (*Customer*)[1]:




*UML model for a Customer component*

## Identifying Requirements That Impact Other Components

When this UML model is verified by its builders they will also, as a by-product of the verification, know which of their requirements, directly or indirectly (via other objects) use the *Value* attribute of the imported class *Order*.

These requirements can then be communicated to the builders of the component that exports the *Order* class. Or, if the other group already has published requirements for the *Order* class, they can be compared to the local requirements for class *Order,* to see how they match.

## Faking the Behaviour of Neighbouring Components

If the builders of the *Customer* component want to verify their component standalone, they need an implementation of the imported class *Order*. This implementation may have a reduced functionality compared to its corresponding kernel class of component *Order* because its only purpose is to take part in the verification of other components that have imported it.

Example: The *Value* attribute of the imported class *Order* in component *Customer* may be set directly, whereas “at home” (in its exporting component *Order*), the value of an order is always computed by the operation *Compute Value* of the kernel class *Order* (see next page).

If both implementations of Order (the reduced *Export* and the full-fledged *Home* implementation) fulfil the requirements of the Customer component, they can be used interchangeably. The verification of the *Customer* component should show the same, expected result, no matter whether it is verified standalone or interacting in a composition with the *Order* component.

[1] Components as well as their kernel, import and export sections establish name spaces, so the name *Customer* can be used multiple times (for the component itself, for a class in the kernel and for an exported class).

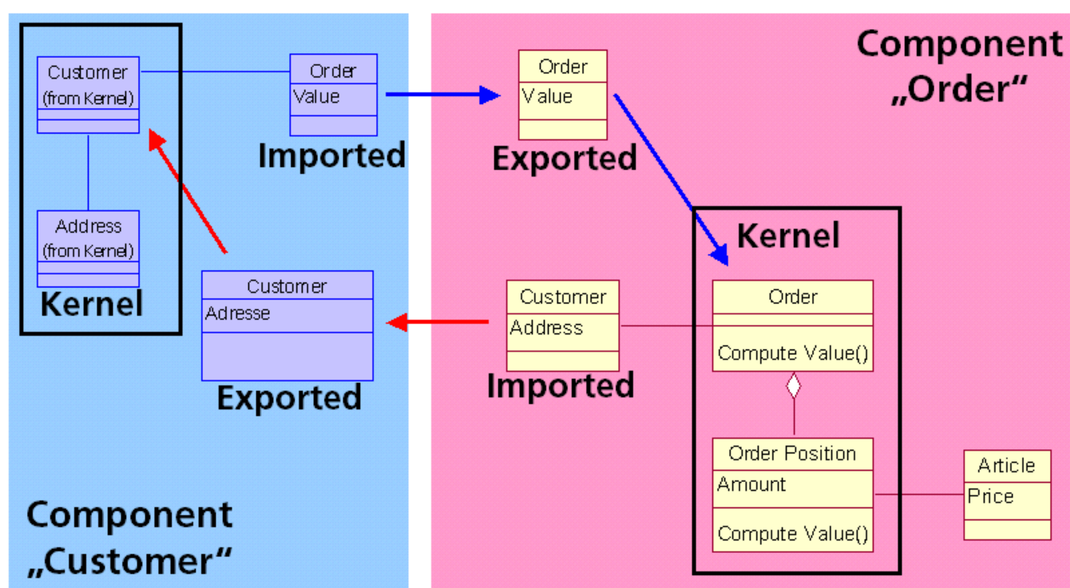
## Verification of Interacting Components (Many UML Models)

The preceding paragraphs have described the techniques that have been used for verifying software composed from multiple components.

On the basis of Objects 9000 for verifying individual UML models (which produces a trace from the requirements to the UML model) the internal structure of components can be mapped to 1 UML model per component. This makes it easy to verify components.

In addition, the requirements tracing capabilities of Objects 9000 are used to identify those requirements that directly or indirectly depend on imported classes. These requirements are then communicated between the different groups of component builders.

The following diagram shows how a second component, called *Order* can be combined with the *Customer* component from the previous page:




*Two interacting components*

An interesting aspect of this collaboration is, that verification only needs to take place *within* each component, using the imported classes as proxies for the remote components' behaviour. As yet it has *not* been necessary to verify *across* component boundaries, even though we expected this to happen when we started to develop this technique back in 1998.

For the communication between the builders of the different components it has been sufficient to communicate on the basis of requirements (accompanied by their tests) both of which are written in natural language – and translated to a programming language for verification.

## How Much Precision Can Be Achieved?

Even though natural language seems to be quite imprecise we have *not found* this to be a problem in practice. We think this is due to the following factors:

a) The tests for each requirement are written as scenarios in natural language. This helps humans to understand them easily, quickly and unambiguously. It also helps to detect special cases (which normally lead to new requirements). This activity already reveals (and corrects) most errors, omissions and misunderstandings in the requirements.

b) When building the UML model (again using UML graphics and natural language) the modeller normally also finds a couple of errors in the requirements. If the modeller introduced new errors into the model, they usually are detected in the next step.

c) In order to verify the UML model, the model operations and the tests are coded in a programming language. If the execution of these tests *exposes* errors in the requirements or the UML model, these can be traced back easily because code and model elements correspond 1:1.

   Coding seldom *leads to* new errors because these pieces of code are very small, and if it does, they are practically always detected when running the tests: in nine years and many thousand requirements and tests we have found only one coding error that was not caught by the tests. (We think, it would have been detected by a Code Inspection)

Our experience has shown that this way of building UML models leads to models that can be verified with very little effort. This is especially true for iterative development and in the maintenance phase of a software system. Because the tests are performed automatically they can ensure that no unintentional changes (i.e. errors) creep in.

However, the gap between verification and validation remains, as outlined above.

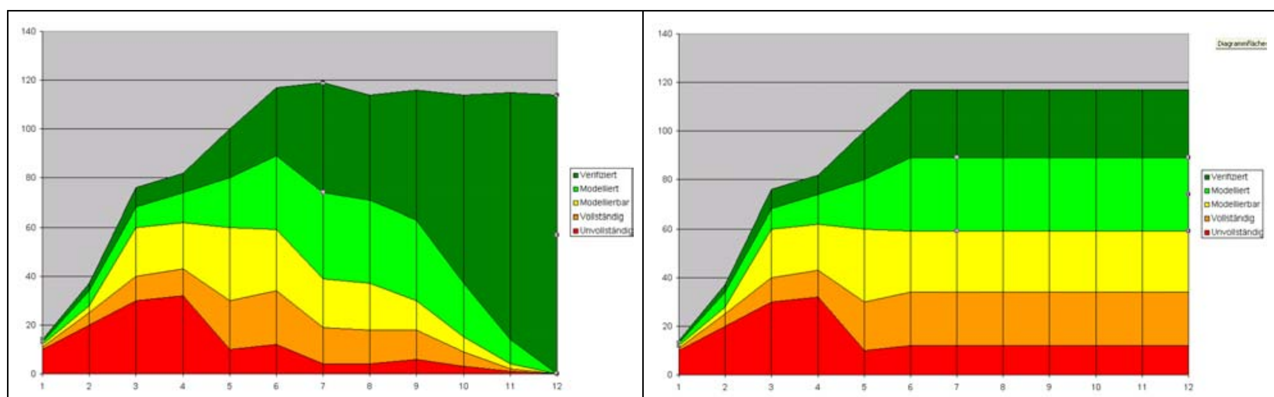
After the verification of the UML model, the next step is to write the production software. This is a very interesting topic[MDA][GEN], but I will avoid it here for the sake of brevity.

The technique presented here has been named Components 9000.

## Lessons Learned About Modelling Components

a) Software components should be treated much like mechanical components: just like these a software component encapsulates the knowledge of its builders and makes it available for others without requiring them to learn all the component's inner details.

*b)* There may be situations when the designated builders of a component simply don't have the knowledge they would need to build the component. Situations like that often go undetected for a long time, and can lead to huge problems later-on. With requirements statistics these situations are detected early and easily, just by looking at the diagram of a component's requirements over time. The following picture shows statistics for two components (these statistics are automatic by-products, too).




*A project managers dream* *A dead project*

*These two diagrams show the number of requirements over time. They visualize the amount and the quality of the knowledge contained in a component's requirements: dark green = verified, light green = modelled, yellow = ready to be modelled, orange = complete with tests, red = incomplete.*

Even without knowing anything about these 2 components one can easily see that team 1 did a perfect job, whereas team 2 practically stopped working after six months.

These statistics allow further, interesting analyses like e.g. the *Value Earned* in a project and the rate of *Value Creation*, but this is beyond the scope of this paper[VAL].

## State of Development of Components 9000

The IT company for which Components 9000 was developed to maturity and applied in full breadth, sadly has been sold to SAP by its parent company, and the project was abandoned.

So this report is not the glorious success story that I would have liked it to be. It is rather an experience report that informs you about what went well and what did not.

Organizations interested in getting the benefits of Components 9000 should plan about 3 months' time to set up the tool environment.

## Human concerns

The techniques presented in this paper have polarized many audiences, because they fundamentally change the ways for building software.

a) They make knowledge explicitly visible. Requirements document the knowledge flowing into the software development process, enabling non-IT experts to review it.

b) The requirements of verified UML models also preserve this knowledge for the organization when software engineers change projects.

c) The verification of UML models, be it standalone or for components, helps to make sure that software does exactly what it is supposed to do – and nothing else.

Many experienced software developers and managers tend to be enthusiastic about these techniques' potential for achieving clarity and efficiency through precision and automation.

On the other hand, a considerable number of software professionals was not so enthusiastic: some voiced concerns that these precision techniques might take the fun out of writing software. Others just couldn't imagine these techniques to be applicable to software at all.

## Summary

The techniques presented in this paper make it possible to verify software components, individually and interacting in a composition. Under the names of *Objects 9000* and *Components 9000* all of them have been tried and tested successfully in large projects.

None of the techniques presented here is really new. They have all been copied from early industrial techniques that were introduced into industrial engineering about 100 years ago.

We found that all of these techniques worked well, technically. But we also noticed that they created quite a lot of anxiety and psychological resistance, much like they did when they were first introduced in the 19th century.

Their biggest advantages – to explicitly make visible the knowledge that is automated by software – and to preserve it reliably over time – are feared by some and loved by others.

---

[ISO9000] Deutsche Norm „Qualitätsmanagementsysteme, Grundlagen und Begriffe (ISO 9000:2000)" / „Quality management systems – Fundamentals and vocabulary", trilingual version EN ISO 9000:2000, December 2000. Beuth Verlag GmbH, 10772 Berlin, Ref. Nr. DIN EN ISO 9000:2000-12, 60 pages, supersedes the norm EN ISO 8402:1995

[IBM] More information: www.rational.com, the academic program SEED which makes these tools available to universities can be found at http://www-3.ibm.com/software/info/university/

[MDA] Model Driven Architecture (MDA) is a current standardization effort of the Object Management Group (OMG): www.omg.org/mda/

[GEN] The website of General Objects contains further information on how to generate software following the principles set forth by MDA: www.generalobjects.com/begriffe/mda/

[VAL] Earned Value Management (EVM) is an established tool for project management. General information can be found at www.pmi.org (use their search function). For the application of EVM to software projects see www.acq.osd.mil/pm/paperpres/fleming1.pdf and www.generalobjects.com/begriffe/evm/

# Architecture of an XML-based Aspect Weaver

Eduardo Kessler Piveta[1,2] and Luiz Carlos Zancanella[2]

[1] Centro Universitário Luterano de Palmas, Laboratório de Banco de Dados e Engenharia de Software (LBDES), 1501 Sul SN,
77054-970 Palmas - TO, Brazil
{piveta}@ulbra-to.br
http://www.inf.ufsc.br/ kessler

[2] Universidade Federal de Santa Catarina, Laboratório de Segurança em Computação (LabSEC), Campus Universitário,
88040-900 Florianópolis - SC, Brazil http://www.inf.ufsc.br/ zancanel

**Abstract.** This paper proposes an architecture to enable the development of an XML-based aspect weaver, providing ways to manipulate programs using and XML representation of source code information. The main advantages on using this approach when implementing an aspect weaver is that there are several tools to support XML documents manipulation and the use of XML provides a standard way to represent and manipulate source code.

## 1 Introduction

Design processes usually consider a system as a set of small units, that are implemented using the abstractions and composition mechanisms provided by a programming language in order to produce the desired system [Becker and Geihs, 1998].

The programming language coordinates well with the design if the abstractions defined by the design process are available in a clear way in the language. In [Kiczales et al., 1997], the authors claim that the abstraction mechanisms provided by the most used languages (such as procedures, functions, objects, classes) are not usually enough to implement all the design issues that arise in a software project.

There are several systems' characteristics that do not fit well into traditional abstraction and composition mechanisms, affecting the system's semantics and/or performance. As an example we could cite: exception handling, real-time constraints, distribution and concurrency control.

If these properties are implemented using object-oriented approaches, their code is usually spreaded over several classes, making the code harder to understand, maintain or extend. This decreases the modularity of the system, making difficult to separate such properties from the system's basic classes.

These situations increase the software complexity and the dependency among components, distracting the user about what is the main responsibility of a software module. If the user wants to know how these properties affect the system's classes, he must search into several classes looking for references to that property.

Aspect-oriented programming is an approach that allows the separation of these properties that crosscut the systems basic functionality, in a natural and clean way, using abstraction and composition mechanisms to produce executable code.

An implementation based on the aspect-oriented programming paradigm is usually composed of:

- a component language to program components (i.e. classes);
- one or more aspect languages to program aspects;
- an aspect weaver to compose the programs written in these languages;
- programs written in the components language;
- one or more programs written in the aspect language.

This paper proposes a architecture to enable the development of an XML-based aspect weaver. This weaver is composed in four main modules described in section 3. The idea is to provide mechanisms to manipulate the programs using XML documents representing the source code information.

## 2 Aspect-Oriented Programming

In [Kiczales et al., 1997] is defined that a system property that should be implemented could be seen as an aspect or as a component:

- the property is implemented as a component if it could be encapsulated in a functional module (class, method, procedure). For instance, we could see components representing users, persons, accounts and points as components.
- aspects are not usually derived from functional decomposition, they generally affect several modules systematically. As example, we could cite: concurrency control in processes scheduling, exception handling polices, session tracking and real time constraints.

The main goal of aspect-oriented programming is to provide mechanisms to separate components from aspects, components from components and aspects from aspects, using abstraction and composition mechanisms to produce the overall system, extending others approaches (object oriented, structured and functional) that do not offer good abstractions to deal with crosscutting concerns [Kiczales et al., 1997].

## 3 Aspect Weaver

An aspect weaver has the main task to process aspect and component programs in order to generate the specified behavior. To make it possible, it's essential the concept of join-points: the elements of the component language semantics that aspect programs coordinate with. Examples of join-points are method's calling, constructor's calling and field's read/write operations.

To design and construct an aspect-oriented system the developer must know how concerns are described in both languages (aspect and component) as well as common features in them. Although these languages have different abstraction and composition's mechanisms, they should use some common terms that would allow the weaver to compose the different programs' types.

An aspect weaver traverse the aspect programs and gathers information about the join-points used in them. Afterwards, the weaver finds the shared points between the languages, weaving the code to develop what is defined in the languages [Böllert, 1998].

For instance, an aspect weaver could be implemented using a pre-processor that scans the component program's parsing tree and inserts the sentences specified in the aspect programs. This process could be done both in compile time and in runtime.

## 4 The Architecture

The architecture described in this section has the goal to describe mechanisms to develop an XML-based aspect weaver, in order to improve reuse of the parsers and weaver's implementation. The use of XML to represent the programs' source code allow the use of all the tools available to XML documents manipulation and defines standard mechanisms to exchange information about programs.

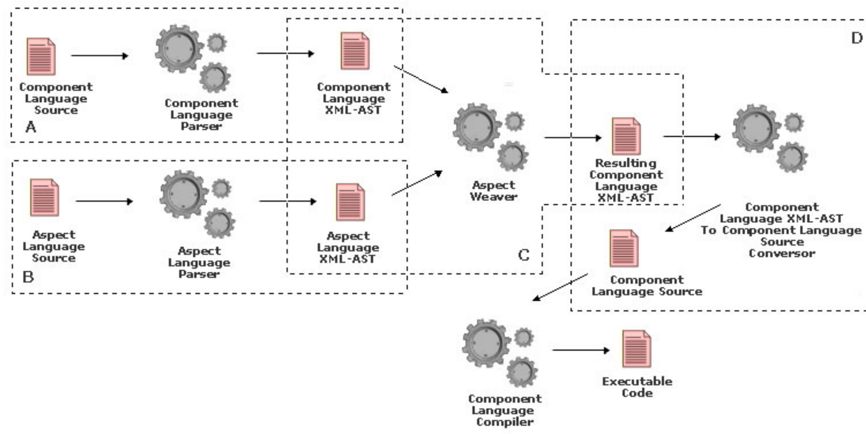



**Fig. 1.** Aspect Weaver Architecture

It's comprised into four main modules, as can be visualized in Figure 2:

- a converter from component language source code to an XML representation of the programs implemented using it;

- a converter from the aspect language source code to an XML representation of the programs implemented using it;
- an aspect weaver that receives the aspect and component programs and generates an XML tree representing them, generating a resulting component language XML tree;
- a converter that translates the resulting XML tree to component language source code.

### 4.1 Converter from Component Source Code to XML

The main responsibility of this module is to generate an XML tree to represent the component language programs' source code (Figure 2 - A). The most important element in this module is the *Component Language Parser*. It receives a set of programs written in the components language and generates as output an XML tree corresponding to these programs.

Implementing this module require the execution of the following tasks:

**Define an XML representation to component programs** The component programs could be represented using a DTD or an XML Schema. This task could be automated if the developer has a BFN grammar of the component language.

**Implementation of a component language parser** Using the component language grammar the developer could create a parser using one of the available parsers generator (such as: *Javacc* and *Lex and Yacc*) or implement the parser from scratch. Semantic actions should be specified in order to generate correctly the XML representation of the program.

### 4.2 Converter from Aspect Source Code to XML

This module aims to produce an XML tree representing the aspect programs (Figure 2 - B). It is composed by an aspect parser, that receives the aspect programs and generates the XML tree. To implement this module, the developer should perform the following activities:

**Define an XML representation for aspects** A DTD or Schema must be defined to represent the aspect programs. These representations could be generated using the aspect language grammar.

**Develop a parser to the aspect language** A parser that identifies aspect programs and generates an XML representation of the aspect code should be implemented in the same way that the component language parser. Existing parsers could be modified/extended to perform this task.

### 4.3 The Aspect Weaver

The aspect weaver should be able to recognize and manipulate XML trees representing component and aspect code (Figure 2 - C). It receives the aspects

and components XML trees and generates a unique tree (using the component language representation) applying the semantics described in both languages.

To produce this resulting tree, the aspect weaver should traverse the components XML tree, modifying it according to the aspects semantics. This semantics is usually described as a set of join-points and a set of actions. These actions should be triggered every time the component code reaches the conditions specified in the join-points.

The weaver should be able to add state and behavior to the components tree as well as to modify a method's body. These transformations in the XML tree could be done using existing techniques to manipulate XML documents, such as: DOM, XSL, SAX. The developer could also use XML-binding tools (such as JAXB) to help the generation of code to manipulate the XML tree in a more natural way.

The code generated by the binding tool is similar to a meta-object protocol. The program generated could add methods, fields, modifiers and obtain meta-information. After the program is modified the XML tree can be easily generated by binding methods.

### 4.4 Converter from XML to Component Language Source

This module is responsible convert the XML tree representing the entire system and generate code in the component language. After that the resulting classes could be compiled to produce executable code. All the techniques used in the weaver implementation could be used here to generate the component languages source code, such as: XSL, DOM etc.

## 5 Discussion and Future Work

The use of XML to represent aspect and component code could benefit the entire system because the programs could be manipulated by existing XML tools. Another advantage is that XML documents could be used to represent meta-information about the programs.

The aspect weaver could be implemented in a way that the code manipulating component languages is separated from the code that manipulates aspect languages. The advantage is that individual modules could be replaced without modifying the whole weaver. The only module affected is the aspect weaver central module (Figure 1 - C). More than one aspect or component language could be used simultaneously using the structure.

The architecture described here could be used to implement a weaver into existing programming languages, such as AspectJ and Java. In Figure 2, we could see a schema of how the architecture could be applied to work with these languages. In this example, the component language is Java and the Aspect Language is AspectJ. The developer should implement the parsers, the weaver and the conversor from XML to source code in order to apply the architecture described in this paper.

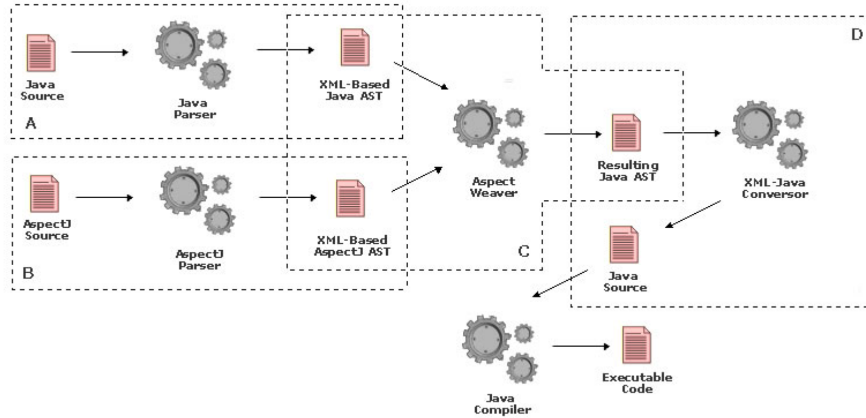



**Fig. 2.** Implementing the architecture using Java and AspectJ

## References

[Becker and Geihs, 1998] Becker, C. and Geihs, K. (1998). Quality of service - aspects of distributed programs. In *Int'l Workshop on Aspect Oriented Programming (ICSE 1998)*.

[Böllert, 1998] Böllert, K. (1998). Aspect-oriented programming case study: System management application. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

[Lopes, 1997] Lopes, C. V. (1997). *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University.

[Mendhekar et al., 1997] Mendhekar, A., Kiczales, G., and Lamping, J. (1997). RG: A case-study for aspect-oriented programming. Technical Report SPL-97-009, Palo Alto Research Center.

# Preserving Real Concurrency

James Ivers and Kurt Wallnau

Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213, USA

**Abstract.** We are concerned with reasoning about the behavior of assemblies of components from models of components and their patterns of interaction. Behavioral models of assemblies are "composed" from behavioral models of components; the (composed) assembly model is then mapped to one or more reasoning frameworks, each suitable for a particular kind of analysis. Information relevant to each reasoning framework must be faithfully rendered in the assembly model, and it must be preserved under interpretation to the reasoning framework. Our concern in this paper is the faithful rendering of concurrency. Our approach makes use of information provided by components and extracted from static assembly topologies to faithfully model real concurrency. The result is more effective analysis.

## 1 Introduction

Our work is concerned with predicting the behavior of assemblies of components from the specifications of components and from their patterns of interaction. Behavioral models of assemblies are "composed" from behavioral models of components; the (composed) assembly model is then mapped to (interpreted in) one or more reasoning frameworks, each of which is based on its own computational model, and each of which is used to predict some directly or indirectly observable behavior of an executing assembly. Information relevant to each reasoning framework must be faithfully rendered in the (composed) assembly model, and it must be preserved under interpretation to the reasoning framework.

Our concern in this paper is with the correct rendering of implementation concurrency in assembly models that are mapped to temporal logic model checkers [1]. In particular, we are concerned that the composed model neither over- nor under-approximates concurrency. Over- and under-approximation are the inclusion of more or fewer processes (respectively) in a model than exist in the corresponding implementation. Over-approximation leads to spurious counterexamples that take time and resources to eliminate and contributes to statespace explosion, as it results in additional interleavings that must be evaluated. Under-approximation is more insidious, since it leads to missed counterexamples and, potentially, to runtime errors. Composing models that are faithful to real concurrency is not trivial for the kinds of reactive systems that we are addressing. These systems are concurrent and distributed but not massively so; in particular, components may exhibit internal concurrency, may execute behavior on the caller's thread of control, and may (and typically) do both.

In this paper we outline our approach to composing assembly models that are faithful to real concurrency. We require that some aspects of the internal concurrency structure of components be exposed; we specify these aspects through component *reactions*. Information gleaned from a static topology of an assembly allows us to compose reactions in a way that is faithful to real concurrency.

The rest of this paper is structured as follows. Section 2 provides some background on our approach to predictable assembly and states our composition problem in a bit more detail. Section 3 outlines our component specification and assembly composition approach. Section 4 briefly touches on some related work, and Section 5 summarizes.

## 2 Background

Our approach to predictable assembly is to construct prediction-enabled component technologies (PECTs). The central idea of a PECT is that a construction model of a system, specified in terms of an assembly of components [2], can be mapped (via syntactic interpretation) to one or more analytic models, each corresponding to a particular computational model and reasoning technique. A construction model describes a component-based system in terms of these characteristics:

- A *component* has interfaces, specified as *pins*, that accept stimulus from its environment (*sink* pins), and can initiate stimulus on its environment (*source* pins). Pins can produce and consume data, and are specified with a signature similar to conventional APIs. Pins are specified as supporting either synchronous (call/return) or asynchronous (message based) interaction.
- An *assembly* is a set of components composed in a particular *runtime environment*. The runtime environment provides connectors to connect the source pins of one component to the sink pins of another. Connected pins must be conformant: their signatures and interaction modes must match. The assembly must also obey other connector-imposed constraints (1:1, 1:N, etc.).

Because components are composed with different combinations of other components to form different assemblies, their exact behavior is not always knowable based only on their own specifications. For example, assume that a component has two sink pins, and that the behavior of each modifies some shared (internal) variable. This component may behave differently depending on the context in which it is composed. There is no possibility violating mutual exclusion if both pins execute on the same thread. However, if each pin can execute on a different thread, then mutual exclusion violations are possible unless the calling threads are coordinated. Each of these cases calls for a different composed model of behavior, which can only be determined when the component's context is known.

## 3 Approach

Our solution to systematically achieving correct composition with respect to order dependent properties of concurrent systems relies on using the following process, the steps of which are elaborated in the following sections:

- Allocate component behavior to potentially concurrent units called reactions.
- Model each reaction.
- Compose reactions in a way that preserves implementation concurrency.

### 3.1 Allocate Behavior to Reactions

We begin by examining a component's implementation, which has some intrinsic concurrency and some undetermined concurrency. Intrinsic concurrency comes from threads created and managed by the component. Certain behavior of the component only executes in these threads. Undetermined concurrency comes from units of behavior that are not allocated to threads by the component, such as function calls. The real concurrency of such units of behavior will depend on how the component is composed with other components in its environment.

When modeling a component in a reactive system, we focus on how it responds to stimulation of its sink pins. A natural starting point would be to produce one model for each sink pin that shows how the component responds when stimulated on that sink. However, this approach could model more concurrency than is implemented in the component. Instead, we use our understanding of the component's potential concurrency to allocate the behavior for handling each sink pin into potentially concurrent units called reactions.

A *reaction* is a model of a collection of behavior that always executes within the same thread of control. A reaction describes the relationship between a collection of sink and source pins by defining how the source pins are stimulated in response to stimulations of the sink pins. For example, a particular reaction could model a thread of a component that retrieves messages from a queue (sink pin stimuli), performs some computation based on the type of message received, and sends messages (source pin stimuli) based on the results of the computation.

Behavior is allocated to reactions following prescribed rules based on a component's intrinsic and undetermined concurrency:

- The behavior of each sink pin is allocated to exactly one reaction.
- All sink pins that are handled by the same thread of the component must be allocated to the same reaction. Each such reaction is called a threaded reaction and represents a unit of intrinsic concurrency in the component.
- Each sink pin that is not handled by a thread of the component is allocated to a reaction of its own. Each such reaction is called a non-threaded reaction and represents a unit of undetermined concurrency in the component.

Recall that a reaction is a *potentially* concurrent unit. Allocation of each sink pin of undetermined concurrency to a separate reaction is a pessimistic

decision. While two such pins may always be stimulated by the same thread in a particular assembly and so could be allocated to a single reaction, this cannot be determined from the implementation of the component. Such decisions are made later in our process, when the context is known and reactions are composed.

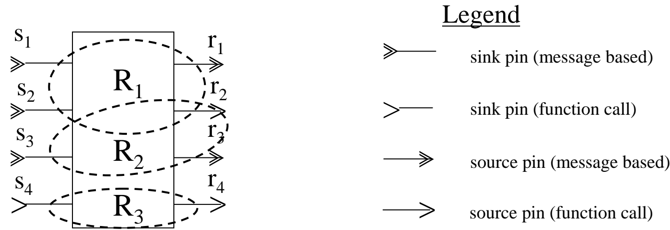



**Fig. 1.** Allocation of component behavior to reactions.

Figure 1 is a graphical representation of a component and the allocation of its behaviors to reactions. The dashed ovals represent an approximation of how the behavior of the component is allocated to reactions $R_1$ through $R_3$.

This component has two threads (not shown graphically). The first handles sinks $s_1$ and $s_2$ and may stimulate sources $r_1$ and $r_2$; therefore, all behavior associated with these two sinks, including the circumstances under which this thread will stimulate these sources, is gathered into a single threaded reaction, $R_1$. The second thread handles only sink $s_3$ and may stimulate sources $r_2$ and $r_3$; all behavior associated with $s_3$ is allocated to the threaded reaction $R_2$.

The last sink, $s_4$, is a bit different. It is not handled by one of the component's threads; instead, it executes on the thread(s) of any clients interacting with that sink. The implementation handling that pin is still part of the component (e.g., a function call exported by the component), but the execution of the implementation handling that pin is not on one of the component's threads. Consequently, the behavior associated with this sink pin is allocated to its own non-threaded reaction, $R_3$, and its true concurrency is not known until the component is composed with other components.

### 3.2 Model Reactions

The sequential processing performed in each reaction is modeled using a state machine based description. We use a variation of UML statecharts extended with an action language based on a subset of C, much like the action language of xUML [3].[1]

The level of detail found in each reaction varies. Due to model checking limitations (notably the statespace explosion problem), reactions are usually abstractions of the behaviors they represent. As our goal is an understanding of

[1] Modeling reactions with statecharts is a modification to earlier work in which reactions were modeled in CSP [4]. We adopted statecharts because CSP was found to be too difficult for expected users.

the potential sequences of pin stimulations and states during system execution, we concentrate on the control structures within reactions, and how they influence how a component interacts with its environment via its pins.

Typically, each reaction begins in an initial state in which it will accept input (stimulation) on any of its sink pins. What happens next is dependent on the component, but typically each reaction reaches a point where it concludes the processing for that sink pin stimulation and returns to a state in which it will again accept input on its sink pins.

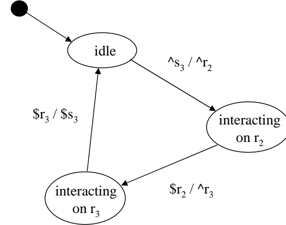



**Fig. 2.** Statechart for reaction $R_2$.

Figure 2 shows a reaction model for $R_2$ from Figure 1. The behavior of this reaction is rather simple. The reaction begins in the *idle* state, waiting for its sink pin to be stimulated ($\hat{}s_3$ is the event that represents a stimulation of $s_3$). Stimulation of $s_3$ initiates an interaction, during which the reaction in turns stimulates source pins $r_2$ and $r_3$. However, between stimulating $r_2$ and $r_3$, the reaction waits for interaction on $r_2$ to complete by waiting for the $\$r_2$ event, which indicates completion of the interaction on $r_2$. After stimulating $r_2$ and $r_3$, the reaction concludes the interaction on $s_3$ by generating the $\$s_3$ event and returns to the *idle* state where it waits for the next sink pin stimulation.

### 3.3 Compose Reactions

When components are composed in a particular topology, we have sufficient context to correctly model the real concurrency of a concrete system. To produce this model, we compose reactions using the following rules:

1. Recursively eliminate all reactions that are not used in the assembly. That is, if no sink pins of the reaction are connected to source pins of the environment or other components that are used in the assembly, then the reaction will never be stimulated, and does not need to be included in the composition.
2. Determine which reactions stimulate each non-threaded reaction. A reaction stimulates another if one or more of its source pins are connected to one or more of the other reaction's sink pins.
   (a) For each reaction that stimulates a non-threaded reaction, make a copy of the non-threaded reaction and compose it sequentially with the stimulating reaction. The result is that each stimulating reaction grows larger by

incorporating the behavior of the non-threaded reaction. As part of this composition, any source pins stimulated by the non-threaded reaction should now be stimulated by the reaction with which it is composed.

(b) When finished, eliminate the original non-threaded reaction.

3. Combine the remaining reactions, each of which corresponds to a thread in the implementation, using parallel composition.
4. Add statecharts defining the interaction (connector) semantics used in the assembly. For example, two components communicating via message passing would use statecharts defining the semantics of message passing in their runtime environment (e.g., a FIFO message queue that blocks when full).

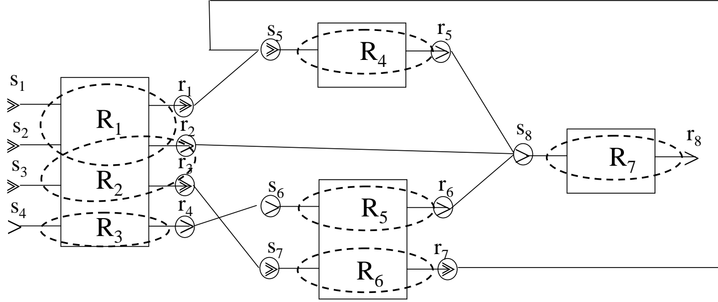



**Fig. 3.** An assembly of components with component behavior allocated to reactions.

Figure 3 shows a sample assembly and the allocation of behavior to reactions. Applying the above composition rules, we get the following results at each step:

– No reactions are eliminated in step 1, as we assert (but do not show) that each sink pin of the left-most component is connected to the environment.
– The three non-threaded reactions in this example are $R_3$, $R_5$, and $R_7$. Rule 2 is applied to each of these reactions in turn.
  - $R_3$ is stimulated only by the environment. Therefore a copy of $R_3$ is sequentially composed with the environment. $R_3$ is then eliminated.
  - $R_5$ was stimulated only by $R_3$. However, after the above step, $R_3$ was eliminated, and $R_5$ is now stimulated by the environment due to the sequential composition of $R_3$ with the environment. Consequently, a copy of $R_5$ is sequentially composed with the environment, and $R_5$ is then eliminated.
  - $R_7$ is now stimulated only by $R_1$, $R_2$, $R_4$, and the environment (via $R_5$ and the above step). A copy of $R_7$ is sequentially composed with each of these models, and $R_7$ is then eliminated.
– After the preceding steps (and ignoring the environment, which is a separate topic), we are left with only four reactions–$R_1$, $R_2$, $R_4$, and $R_6$. These reactions are now composed in parallel, and each now correctly corresponds to a real thread of execution.

– Finally, the resulting model is composed with models of the connectors, which supply the correct interaction semantics for each type of communication used in the system (e.g., function calls vs. message queues).

## 4 Related Work

There has been some interest in composing heterogeneous model fragments, e.g., various diagram types in UML [5]. Our concern is composing homogeneous fragments, e.g., executable UML statecharts with the particular end-goal of using the composed models as input to multiple reasoning frameworks. Liang et. al. adopt model composition of Petri nets, and likewise preserve information about real concurrency, although their interest is restricted to RMA schedulability and therefore leads to a coarser treatment of concurrency [6]. Sora et. al. also conclude that the internal flow structure of components must be made explicit for faithful composition, but concurrency is not addressed per se [7]. There are numerous examples of using process algebras to define the semantics of composition languages [8, 9]; however, these efforts adopt simplifying assumptions about real concurrency such that a faithful rendering of concurrency is not a concern. While we have explained how we preserve real concurrency during model composition, we have not addressed a related problem–preserving real concurrency during the interpretation to the input language of a model checker. Work in faithful model translation such as that in [10] is expected to be of great use.

## 5 Conclusion

Reasoning about the concurrent behavior of arbitrary assemblies, and reasoning from the perspective of multiple computational models, imposes new requirements on the techniques used to specify and compose behavioral models. This is true whether our concern is with e.g., model checking, as in this paper, or with fault tolerance, reliability, or timing analysis.

The approach outlined has drawbacks, and several open questions remain. One drawback is that the explicit modeling of connection mechanisms introduces artificial concurrency under our current interpretations to model checkers. We minimize excess interleavings by combining connection models where appropriate (e.g., when involving shared resources such as message queues), but a more accurate approach may yet be defined. Pragmatically, it is not clear how readily end-users will adopt this approach to specifying potential component concurrency via reactions.

However, there are clearly benefits to this type of approach. By preserving the actual concurrency of a system in its model, we gain confidence that concurrency errors will not be missed and that counter-examples that are reported are indeed relevant (because they, by definition, represent interleavings that are possible in the implementation). We may even open up an opportunity to exploit knowledge of the implementation's scheduling policy. For example, if we know that a thread is only pre-emptible at certain points or that a lower priority

thread can never pre-empt a higher priority thread, this knowledge can be used to eliminate classes of model process interleavings that are not possible in the corresponding implementation.

A last point worth noting is that our emphasis on model composition reflects the distinction between compositional reasoning and reasoning about compositions. The benefits of compositional (and the stronger modular) reasoning are simple: divide and conquer. As noted elsewhere, however [2], the criteria that must be satisfied for compositionality and modularity of reasoning are often too strong to achieve in practical settings. It is in these circumstances that compositional modeling becomes necessary. Nonetheless, an important element of predictable assembly is, and will continue to be, expanding the frontiers of compositional and modular reasoning.

## References


1. Edmund M. Clarke, J., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
2. Wallnau, K.C.: Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, Sofware Engineering Institute, Carnegie Mellon University (2003)
3. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison Wesley (2002)
4. Ivers, J., Sinha, N., Wallnau, K.: A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, Sofware Engineering Institute, Carnegie Mellon University (2002)
5. van der Straeten, R.: Semantic Links and Co-evolution in Object-oriented Software Development. In: Proceedings ASE 2002. 17th IEEE International Conference on Automated Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society (2002)
6. Liang, P., Arevalo, G., Ducasse, S., Lanza, M., Schaerli, N., Wuyts, R., Nierstrasz, O.: Applying RMA for Scheduling Field Device Components. In: ECOOP 2002 Workshop Reader. (2002)
7. Sora, I., Verbaeten, P., Berbers, Y.: A Description Language for Composable Components. In: Proceedings of Fundamental Approaches to Software Engineering 6th International Conference, FASE 2003. Number 2621 in LNCS, Warsaw, Poland, Springer-Verlag (2003)
8. Giannakopoulou, D.: Model Checking for Concurrent Software Architectures. PhD thesis, Imperial College of Science, Technology, and Medicine, University of London, England (1999)
9. Achermann, F., Lumpe, M., Schneider, J., Nierstrasz, O.: Piccola – a Small Composition Language. In Bowman, H., Derrick, J., eds.: Formal Methods for Distributed Processing–A Survey of Object-Oriented Approaches. Cambridge University Press, Cambridge, UK (2002) 403–426
10. Katz, S., Grumberg, O.: A Framework for Translating Models and Specifications. In: Proceedings of IFM2002 (International Conference on Integrated Formal Methods). Number 2335 in LNCS, Springer-Verlag (2002) 145–164

# How to lock a model from a miscomposition of objects ?

M. Beurton-Aimar[1], S. Pérès[2], N. Parisey[2], J.P. Mazat[2].

[1]Laboratoire Bordelais de de Recherche en Informatique - Universit Bordeaux 1.
[2]Laboratoire physiologie mitochondriale - Universit Bordeaux 2.

## Abstract

Biological systems are known for their complexity and nowadays a challenge of bioInformatics is to design them by computer programs. For a long time, computing procedures have only been calculus procedures and data are often represented in simple matrix without any semantics. Recent works try to use object oriented models and so to improve the design of such biological systems. Such models were described using specific XML-like languages. We have studied a model currently used for modelling metabolic pathways into the E-cell project and translated in SBML language. E-cell is an ambitious project which focuses on *"in silico"* cell modelling. But, although object oriented model was chosen, we have shown that a final miscomposition of a few main classes completely locked the design and so forbade the re-using and extension of modules. We have proposed a new architecture which integrates the original classes and part of the old organization. Then, some new classes have been added to explicit hidden concepts (a part of causes of miscomposition) and the model root was changed. We have tested this architecture within a specific intra-cellular organelle, the "mitochondria" and we worked on design of several pathways belonging to this organelle.

## 1 Introduction

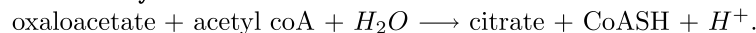
BioInformatics is a large research area and although public attention has focused on comparison of genes, there is also another domain : simulation of biological process where many problems have been addressed by the computer science for a long time. Usual computing programs are based on mathematics equations. The interface of such programs allows users to insert their own formula and/or matrix of data, lists of parameters and so on, to simulate different biological functions or metabolic pathways. The main purpose of these programs is to search the parameter values of steady states but not really to simulate complex connexions between several pathways. Biologists themselves must aggregate the results to conclude what happen in the core of the organism (or organelle).

Some new international projects try to design the biological systems using object oriented methods. So, these complex systems can be divided into small parts and recombined to form complete biological mechanism. Moreover, object

oriented systems allow to represent data semantics from abstraction and reduce complexity of some process descriptions. E-cell and Virtual cell are such projects. People implied in the E-cell project have defined an XML-like language : SBML, to design their model. So, we can consider that the grammar of SBML gives the architecture of the object model. To simplify our explanation, we will present the model through the SBML composition of TAGS.
After a small description of metabolic pathways principles (the biological view), we will explain how to design a simple metabolic pathway with SBML and then we will show the composition problems induced by the SBML organization. In a second part, we will propose modifications to unlock the model and addition of some new concepts missing inside SBML. Finally, we will describe a new object oriented architecture more suitable to the design of complex biological systems.

## 2 At the beginning, many good ideas

Biological pathways are generally described by a set of equations. One equation is the representation of one reaction triggered by an enzyme meeting one or several reactants fig.(1). The result of this reaction is a product which can become in turn an incoming reactant for a new reaction. The reaction can be weighted by some parameters (kinetics parameters) or by the presence of inhibitors/activators(ligands).
One of our studied example is the TCA cycle which is characterized by a set of 8 equations. The two first equations are :

1. `citrate synthetase` :
   
   oxaloacetate + acetyl coA + $H_2O \longrightarrow$ citrate + CoASH + $H^+$.
2. `aconitase` : citrate $\longrightarrow$ + $H_2O \longrightarrow$ isocitrate.

where `citrate synthetase` and `aconitase` are the two enzymes in the reactions and the others metabolites are reactants or products whether they are input or ouput elements.
Like biological systems work at the steady state ; mathematical design often uses differential equation to calculate the parameter values which lead to this steady state. A metabolic pathway is a chain of reactions (equations) identified by their role inside the organism. Each computing program has its own data structure and biologists need to change the data format when they change their computing procedures. So, due to the time consumed to copy data again and the risk of mistakes, they have no guarantee that data (and results) have the same interpretation through different programs. Obviously, comparison among results and re-using of design efforts can not be automatically performed.

Projects like E-cell [3] have built an object oriented design of metabolic pathways. Their goal is to reduce the description load, to share data between procedures due to modularity of modules and to build tools to easily design different versions of biological system parts.

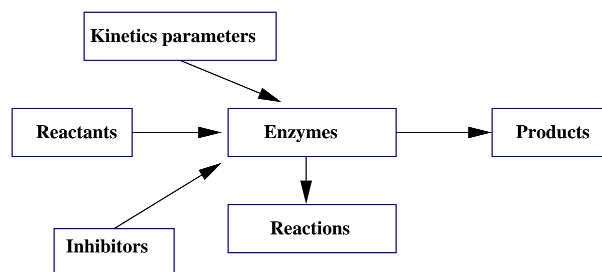



**Fig. 1.** A metabolic reaction

The need to describe data (structures and value) is addressed by XML-like languages. XML (Markup eXtended Language) allows to put into a text file all the information about the data. It is a structured language with marks (`tags`) defining concepts or object classes. SBML [2] (System Biology Markup Language) is an augmented version of XML including `tags` for biology and specifically for describing metabolic process, kinetic equations ...
The SBML grammar get as the root frame the TAG `model` and all the components are associated to this frame. A `model` is defined by a name, a set of species which participates to the reactions and a set of mathematic formula (with its parameters). As shown in fig.2, this organization easily matches with classic calculus software like *Gepasi*. In order to use these programs, only a list of parameters, reactions and species must be defined.

## 3 Where is the problem ?

Firstly, it appears that the model described in SBML is a great advance from several points. Now we can shared dictionaries of elements belonging to the studied organism and simply address them to build a new pathway. SBML is programming language independent and the whole data structure is in a text file, external to the program. Data structure gives the semantic and allows to manipulate an abstract level.
Problems appeared when we have tried to translate our own metabolic pathways in SBML. Independently to the E-cell model, we have begun to find our useful objects and some of them do not appear as TAG in SBML. The most important is the enzyme object. We could not understand how such an important element, the master of the reaction could not be explicitly represented. The answer is that the enzyme was present but not as an element of the reaction. They put all the concept inside the object `reaction` itself. So, it is not possible to separate data directly attached to a specific enzyme (static) and data attached to a specific situation of the reaction (depending on concentration, a few ad-equation with the environment ...). Finally, in SBML a `model` is a version of a particular

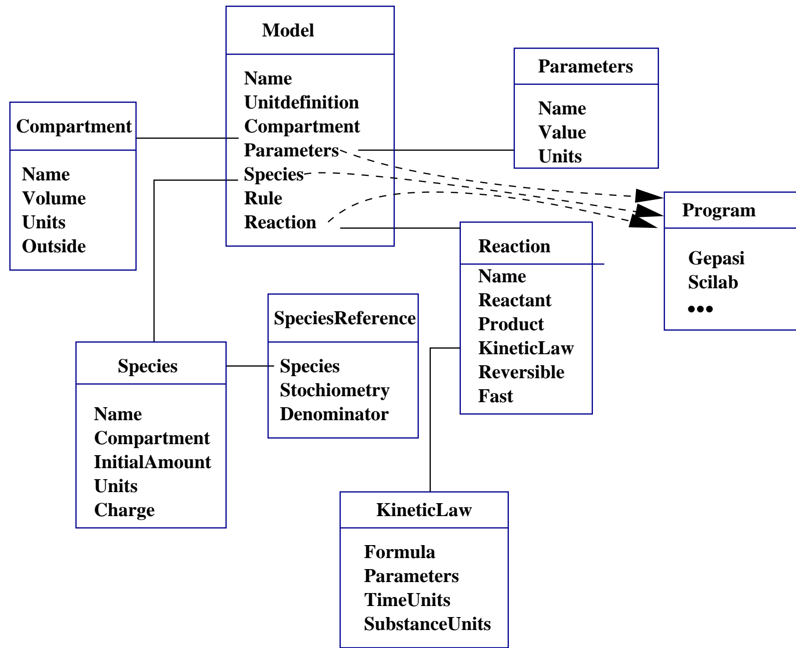



**Fig. 2.** SBML model

pathway.
The main lacks of this design are :

- It is not possible to re-use enzyme data to build a new version (new biological hypotheses) of the reaction : different composition of reactants/inhibitors/activators.
- The same biological function can be realized by different enzymes following disponibility of the enzyme or context species (human, yeast . . . ). It is not possible to express this situation without re-writing the model.
- In a organism, metabolic pathways can have several "version" depending on metabolic pathway object. Several composition could be defined. Since the model is the root of the arborescence, a composition is not possible.

In fact, only a part of the object architecture is used to re-make the "old known-how" : one model with a list of equations and parameters (see fig) and the whole architecture is task oriented.

## 4 Model re-arranging

Our goal is not to throw away SBML model but to modify what we need and to add some important things. The new architecture fig.3 addresses two main problems : to improve semantics and to allows composition of treatments. With our

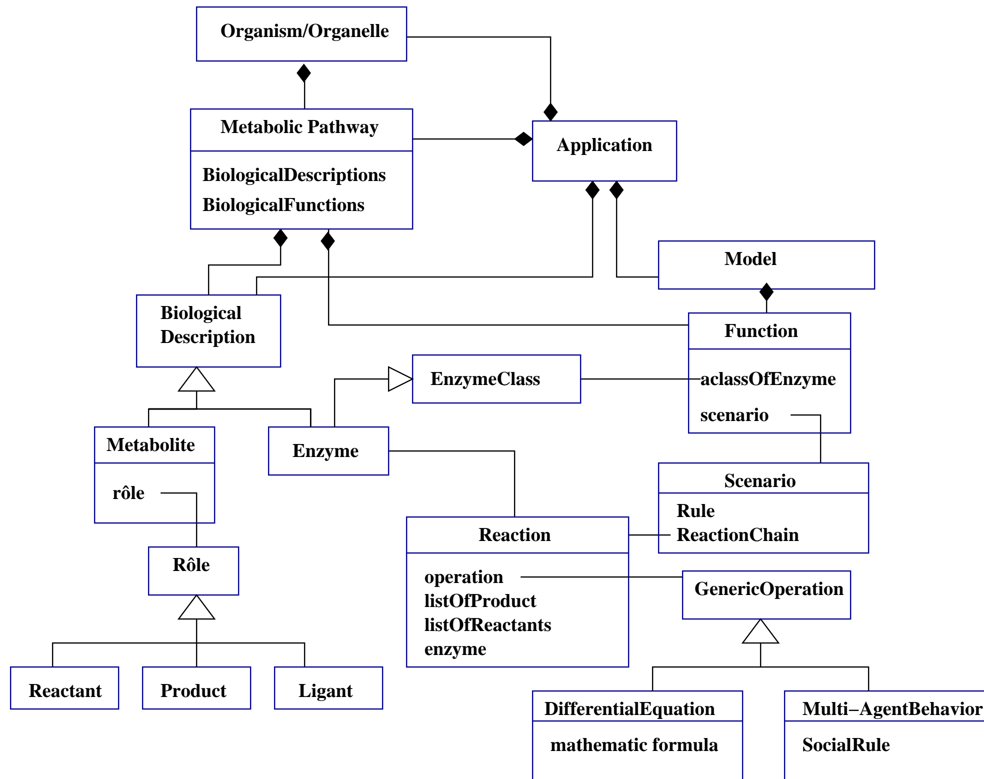



**Fig. 3.** the whole object oriented model

solution, the separation of concerns : data design and polymorphism of treatments, is possible. Our model has a different root : the object `Application` which links the two model parts. Now, the description of several instance of `Metabolic pathways` can be take into account and `Organism` is explicitly represented as a composition of description of several `Metabolic pathway` objects. The `Biological function` has two roles : a static description of some generic behaviors (generic reaction chain) and definition of specific procedures to compute the reaction chain. The `Scenario` addresses the definition of computing procedures and `Enzyme class` addresses the static description of functionalities. The `Scenario` object allows to define several version of the same function, for example taking into account more or less reactions in the chain. The `Generic operation` allows an implementation of heterogeneous systems.
So, we can expect to share and to communicate some informations to another programs which are designed with original SBML. We also plan to propose these modification to the SBML consortium for an extended version of the language. This model is used to describe some parts of mitochondrial metabolism [1]

## 5 Conclusion

The great complexity of biological systems impose to the design of adaptive systems. The hierarchy of components and tree structures are the first elements that take into account complexity of compositional behaviors or organization. Separation between data and treatments are the second principle that must be respected and Object Oriented Methods are useful tools in order to reach this goal. The use of marked-up language like SBML to make visible the description of the whole knowledge and put it into text file without any link with programming language is an evident way to allow exchange and multiplicity of treatments. But as we have shown, it is easy to lock a model even if the base objects are right and main concepts present. Composition and error in this domain can compromise the results obtained by the benefits of object oriented design.

## References


1. M. Beurton-Aimar, S. Ludinard, B. Korzeniewski, JP. Mazat and C. Nazaret. *Virtual Mitochondria : Metabolic Modelling and Control International Journal on Molecular and Cellular Biology(Kluwer Academic Publishers), vol 29:1-2, p227-232, 2002.*
2. *M. Hucka1, A. Finney1, H. M. Sauro1, et al.* The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models Bioinformatics, 2002.
3. Masaru Tomita1, Kenta Hashimoto1, Kouichi Takahashi1, Thomas Simon Shimizu1,3, Yuri Matsuzaki1, Fumihiko Miyoshi1, Kanako Saito1, Sakura Tanida1, Katsuyuki Yugi1, J. Craig Venter2 and Clyde A. Hutchison III2 *E-CELL: software environment for whole-cell simulation BioInformatics . Bioinformatics, Volume 15, Number 1, 72-84 (1999*

# Composing Dynamic Hyperslices

Ruzanna Chitchyan, Ian Sommerville

Computing Department, Lancaster University, Lancaster, LA1 4YR, UK
{rouza | is}@comp.lancs.ac.uk

**Abstract.** This paper describes a composition mechanism for the dynamic hyperslices model outlined in [1]. This mechanism composes primary concerns, directly aligned with requirements and designs (decomposed in accordance with the Hyperspaces approach), maintaining each concern unchanged as a first class entity all through its lifetime. Consequently this mechanism allows for dynamic change and re-configureability of the resultant systems.

## 1. Introduction

Recent work in the field of Software Engineering has resulted in a set of approaches collectively termed Aspect-Oriented (AO) Techniques [2]. AO introduces yet another level of decomposition: separation of crosscutting properties in software, and brings to light the notion of *concern*. A *primary concern* – a matter of interest in a software system that cannot be decomposed into smaller meaningful parts, can now be identified as the atomic unit of any software artefact[1]. It has been shown that it is the inappropriate separation of these concerns [3] that results in monolithic designs [4] and code, which are difficult to understand, maintain and reuse.

Some research has already been carried out on modelling concerns [5], mapping them to design [6] and implementation units [7].

Our work on the Dynamic Hyperslices model [1] follows this line of research, aiming to preserve the unchanged primary concerns as first class entities all through the life cycle of software that concerns form part of – from concern modelling to software run-time. This path leads to flexibility in concern manipulation, reuse and maintenance, as well as dynamic change and re-configureability of the resultant systems. However it shifts the complexity of development and maintenance into concern composition. In view of the increasing importance of composition, we have proposed [1] to distinguish it as a separate *developer-related* concern (i.e. a concern which arises due to specific development-related activities carried out by software developer).

In the present paper we provide some detail on the composition mechanism of our Dynamic Hyperslices model: section 2 describes the background work upon which the Dynamic Hyperslices model is based, section 3 briefly outlines the model followed by the outline of the composition mechanism in section 4 and analysis of possible change scenarios and correctness of composition in section 5. We conclude with brief summary and future work in section 6.

---

[1] Conceptual concern, not programming language expressions, such as variable declaration etc.

## 2. Background

As software becomes ever more closely integrated in our everyday life, on one hand costs of interruptions for maintenance and change of some systems become disproportionately high (e.g. safety critical systems), on the other hand demand for higher adaptability of systems grows (e.g. mobile and disappearing computing). We suggest that dynamically composeable systems could provide a solution for these and other similar problems by providing for dynamic changeability and context-sensitive re-configureability. One such approach – A Model for Dynamic Hyperslices – is discussed below.
In developing the Dynamic Hyperslices model we draw on the concern decomposition mechanism of the Hyperspaces approach, message interception and manipulation ideas of the Composition Filters approach and component integration mechanism of Connectors. The present section provides a brief description of these technologies.

### 2.1 Hyperspaces

This approach [8], [9] proposes to use a set of modules each of which address a single concern (called hyperslice). Hyperslices can overlap, i.e. a given unit in a hyperslice can appear, possibly in a different form, in other hyperslices and dimensions of concerns[2]. All the concerns of importance are modelled as hyperslices, which are then composed into hypermodules (i.e. compound hyperslices with a composition rule specifying how the hyperslices must be composed) or to a complete system. At the composition stage issues such as overlapping are resolved via composition rules.
Composition is based on commonality of concepts across units: different units describing the same concept are composed into a single unit describing that concept more fully. To compose one needs to match units in different hyperslices that describe the same concepts, reconcile their differences and integrate the units to produce a unified whole. Composition rules specify the relationships between composed hyperslices.
In HyperJ [10] (a composition tool developed for OO instantiation of Hyperspaces) composition-related concerns are not treated as first class entities. Although hyper-module composition is specified in a separate composition file, it is only a transitory unit. Consequently, when the elementary concerns are composed, they get contaminated with properties of the composite concern[3].
In the Dynamic Hyperslices approach we differ in composition from HyperJ (see section 3) but we maintain the decomposition principles of Hyperspaces. We also clearly define two types of concerns: user and developer-related, both of which are to be treated as first class entities all through the software development and maintenance process.

---

[2] Dimensions of concerns are ways of decomposition, such as for instance per object classes, per viewpoints, per features etc. This concepts stem from the multi-dimensional decomposition approach, for which Hyperspaces approach is an instance.

[3] More discussion available in [1]

### 2.2 Composition Filters

The Composition Filters (CF) model [11] extends the Object-Oriented model in a modular and orthogonal way. Since behaviour in the OO model is implemented by exchanging messages between objects, the CF model proposes to use a set of *input and output filters* for message interception and manipulation. By wrapping these filters around the objects, CFs are able to manipulate object behaviour without directly invading object implementation.
The CF model is very well suited to implementing concerns that lend themselves to modelling through message-coordination and introduction of actions executed before or after executing a method (e.g. intercept message, put record of message arrival in Log file, execute message).
Our Dynamic Hyperslices utilise the message interception and manipulation capabilities of Composition Filters. Filters form part of our *composition connectors* (see section 3).

### 2.3 Architectural Connectors

The concept of connectors originates from the area of software architecture [12] [13]. Connectors were proposed to facilitate component integration by catering for specific features of interactions among components in a system. The current work in this area argues for giving connectors a first class entity status because they contribute towards the better understandability of system architecture [14] through localising information about interactions of components in a system; capturing the design decisions and rules of interactions amongst components; handling incompatibilities between components and so on. In [15] the idea of connectors as run-time entities is discussed.
Unlike the previous work on connectors, the composition connectors in our model are connectors for hyperslices (sections 3&4), i.e. not (necessarily) for complete object classes or (OO) components. Our connectors don't simply match provided/required services, or specify roles for connected components, but rely on dynamically updateable composition strategy to build up functionality of coarser-grain components (e.g. object classes) from primary hyperslices[4], as well as carry out the communication between the member hyperslices at run time.

## 3 Brief Outline of the model for Dynamic Hyperslices

The model for Dynamic Hyperslices [1] intends to provide a composition mechanism that will allow all the primary concerns, decomposed in accordance with the Hyperspaces approach, to endure in the composite concerns after composition.
The model:

[4] Thus, the composition strategy in the connectors can be perceived as a kind of "merger algorithm" for producing higher order artifacts. Here the "merger" is performed through run-time message manipulation within connectors, without physically merging the hyperslices.

- Uses the Hyperspaces decomposition approach in separating concerns into single-minded hyperslices (or primary concerns).
- Requires that an additional dimension for Composition concerns is specified in each Hyperspace-type decomposition. This additional dimension contains connector-concerns. At the composition stage the connector concerns are used to compose other concerns.
- Utilises a composition connector to integrate any primary/composite concerns. Consequently, any interaction between other concerns will be channelled through a set of connectors.
- Provides connectors with capability to reflect upon their immediately connected concerns, while still keeping these internals hidden from all other connectors and hyperslices.

Figure 1 below provides a high-level diagram for the model. In this model composition concerns are first class entities (depicted as ovals) which also retain hyper-slice integration information. All *user-related* concerns used in the system are retained unchanged at runtime (depicted as squares with solid borders) and all interactions between the concerns are resolved through their connectors (depicted as solid arrows). This model is aimed to be open and extensible, as concerns can be added/updated/removed by simply adding/updating them and their respective connectors.

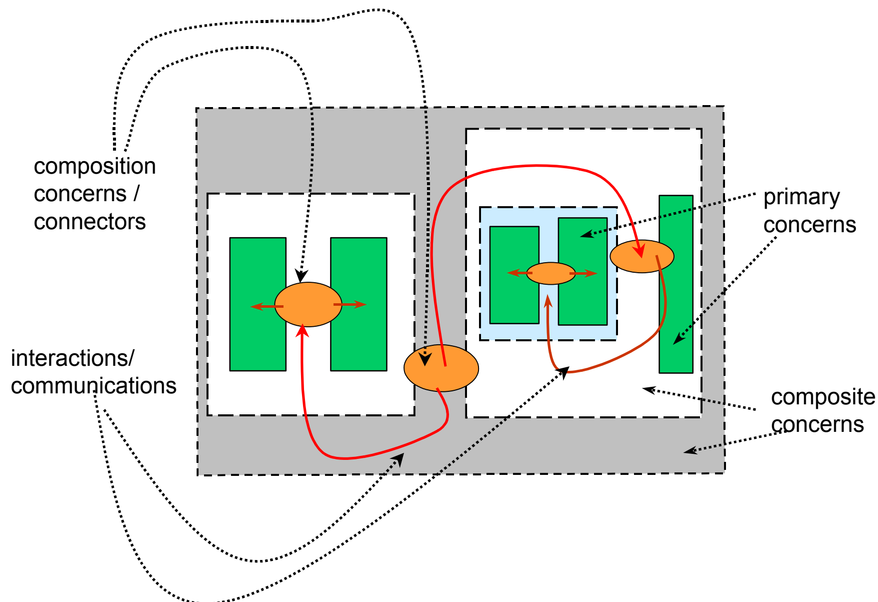



**Fig. 1.** An outline of the Dynamic Hyperslices model

## 4 Connectors

Our connectors decouple hyperslices in a hypermodule and promote reuse of individual concerns and their compositions. Since at any level each of the primary concerns remains intact, changing requirements can be easily mapped onto a composite hyperslice by modifying the out-of-date primary concern. In the process of primary component change only its immediate connectors will be affected, other parts of the model will automatically adjust to changes, if necessary. Similarly a composite part of a hyperslice can be updated/replaced with change introduction localised in its immediate connectors. This locality attribute of our architecture arises due to structuring it around dynamically updateable composition strategies and hiding all (levels of) hyperslices, except for the directly participating ones, from composition.
A connector structure is depicted bellow in Figure 2. It consists of the following elements:

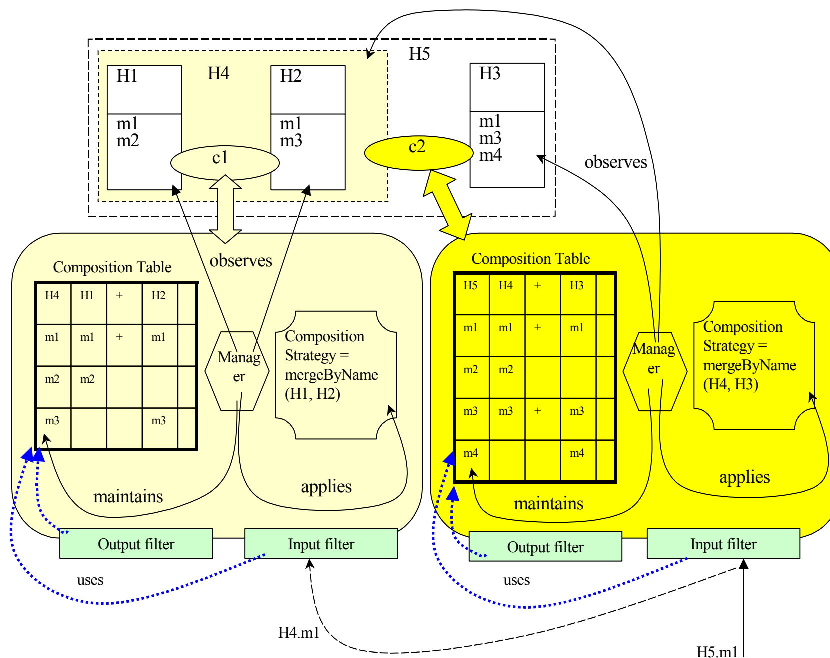



**Fig. 2.** Elements of a Composition Connector

*Sets of input/output filters*: these are used to intercept incoming/outgoing messages sent to the hyperslice and manipulate the message in accordance with filter specification. Filters are similar to those defined in the Composition Filters approach in that they can manipulate or substitute the target and selector of the intercepted message. Thus, for instance, a certain filter type (lets call it dispatch filters as in CF) will be able to re-direct the intercepted message to a different hyperslice or put it through for execution to the initially intended target.

*Composition table* contains the records of the subjects that constitute the composed hyperslice. The first column in the table displays the hyperslice public interface. Each of the following columns contains the references to the hyperslices immediately connected by the present connector (top row of the table) and the elements of the connected hyperslices that contribute to the corresponding unit of interface in the resultant hyperslice (all but top row in the table). The composition operators between the column elements represent the operations required to compose the individual hyperslice elements into that of composed hyperslice. The operators are applied in accordance with the Composition Strategy specification.
*Composition Strategy* is the specification of how exactly should the constituent hyperslices be composed. For instance the *mergeByName* strategy is used in the example shown in Figure 2, stating that all elements with same names in constituent hyperslices should be merged into one element of the resultant hyperslice.
*Connector Manager*: as suggested by its name, this element is responsible for overall "management" of the connector. It uses the Composition Strategy and the public interfaces of the contributing hyperslices to fill in the Composition Table. It also keeps the other elements under observation and updates the composition table content when any of the contributing constituents gets updated.

## 5 Consistency Preservation and Correctness Checks

### 5.1 Change Scenarios and Consistency Preservation

The following set of scenario analyses illustrates how our composition model will deal with some possible changes in the constituent hyperslices. All scenarios are based on the case illustrated in Figure 2.
Scenario 1: *The implementation of one/several methods in constituent hyperslices is changed, but their interface is maintained.* No change is required to any part of any connector, the updated method will be used when a call is directed to it.
Scenario 2: *The interface of a method has changed - method m1 in H1 has been renamed to met1 - but met1 is still to be part of m1 in H4, i.e. the composition has not changed*. In order to maintain the consistency of the Composition Strategy (since it did not change) the c1 Connector Manager adds a new clause to Composition Strategy, indicating that *met1* is an equal name for *m1*. Then it updates the Composition Table in the connector. The second row of the Composition table will change from [m1|m1|+|m1] to [m1|met1|+|m1], indicating that the m1 in public interface of H4 consists of the merge of methods *met1* in H1 and *m1* in H2. No further change is required.
Scenario 3: *Method m2 is deleted from H1*. The Connector Manager of c1 removes *m2* from its composition table. The Connector Manager of c2 detects that the interface of H4 has changed and updates its composition table by first removing H4.m2, then since H5.m2 has no constituent parts, it is also removed. Thus *m2* is removed from the interface of H5.
It should be noted that subtractive changes to the hyperslices' interfaces will be guarded by use counters. Before a subtractive change use counters of respective

hyperslices will be checked to verify that the items marked for deletion are not currently in use. If they are – the change will be postponed till use counter is reduced to 0. Principles of pertinence from [16] could also be beneficial here.
Scenario 4: *Method m7 is added to H1.* The Connector Manager of c1 adds *m7* to its composition table, then c2 Connector Manager updates its composition table with *m7*, thus *m7* appears in the interface of H5.
In all the above scenarios changes introduced to the primary hyperslice are either localised within the immediate connectors of these hyperslices, or automatically propagated to the coarser granularity connectors by respective Connector Managers. Consistency preservation is also a task of the respective Connector Managers, supported by a set of composition rules.

### 5.2 Factors Facilitating Correctness Checking

Several properties of this model facilitate correctness checking of compound hyperslices yet allowing for good changeability (discussed above) of its constituents:

- *Reduced complexity of checking*: smaller single-minded concerns reduce the complexity of the specification, design and implementation, and verification tasks;
- *Checking individual concerns*: since the composition does not affect the concerns in any way, each concern can be checked and tested against its own specification initially, independently of its composition context. This helps to assure correctness of constituents in the composite.
- *Incremental correctness check*: since the larger modules are built by incrementally composing individual concerns, incremental verification of composition is also supported, easing the testing of larger composite units.
- *Direct mapping of change*: as traceability of artefacts at different levels is preserved (due to use of the Hyperspaces decomposition), change in any of the requirements will be directly reflected in its design/implementation concern, and will be easier to trace and validate.

All of the above factors could facilitate correctness checks in our model; however the precise techniques for checking need to be developed.

## 6 Summary & Future Work

We have observed that simplification of software development through new software decomposition techniques (such as those provided by Aspect-Oriented paradigm) tends to move complexity into the composition process. Consequently, we are working to produce a model to separate composition concerns themselves into first class entities and simplify the composition process. In this paper we have discussed our model for Dynamic Hyperslices, which closely follows the spirit of AO, and outlined its composition approach.
There are a number of open issues in our model that need to be addressed, for instance we are working on providing a clear structure for *Connector Manger* elements, refining the composition mechanism, defining techniques for verifying correctness of composition, and developing a meta-model for the Dynamic Hyperslices Model.

Some of our future work will include implementation of a system that realises this model, investigating ways of incorporating domain-specific knowledge into the composition process.

## References


1. Chitchyan, R., I. Sommerville, and A. Rashid. A Model for Dynamic Hyperspaces. in Workshop on Software engineering Properties of Languages for Aspect Technologies: SPLAT (held with AOSD 2003). 2003.
2. Elrad, T., R. Filman, and A. Bader, Theme Section on Aspect-Oriented Programming. Communications of ACM, 2001. **44**(10).
3. Tarr, P.L., et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. in Proc. 21st International Conference on Software Engineering (ICSE 1999). 1999: ACM.
4. Clarke, S., et al., Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code, in Proc. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). 1999. p. 325-339.
5. Sutton, S.M. and I. Rouvellou. Modeling of Software Concerns in Cosmos. in International Conference on Aspect-Oriented Software Development. 2002: ACM.
6. Clarke, S., Composition of Object-Oriented Software Design Models, in School of Computer Applications. 2001, Dublin City University.
7. Clarke, S. and R.J. Walker, Mapping Composition Patterns to AspectJ and Hyper/J, in Workshop on Advanced Separation of Concerns (ECOOP 2001). 2001.
8. Ossher, H. and P. Tarr, Multi-Dimensional Separation of Concerns using Hyperspaces. 1999(21452).
9. Hyperspaces. 2003, IBM Research; http://www.research.ibm.com/hyperspace/.
10. Tarr, P.L. and H. Ossher, Hyper/J user and Installation Manual. 2000: IBM Research.
11. Bergmans, L. The Composition Filters Object Model. in RICOT symposium: Enabling Objects for Industry. 1994.
12. Garlan, D. and M. Shaw. An Introduction to Software Architecture. in Advances in Software Engineering and Knowledge Engineering. 1993. New Jersey: World Scientific Publishing Company.
13. Allen, R. and D. Garlan, A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology (TOSEM), 1997. **Volume 6**(Issue 3): p. 213 - 249.
14. Balek, D., Connectors in Software Architectures (PhD Thesis), in Faculty of Mathematics and Physics. 2002, Charles University: Prague.
15. Ducasse, S. and T. Richner. Executable Connectors: Towards Reusable Design Elements. in ESEC '97. 1997: LNCS.
16. Benatallah, B. and Z. Tari. Dealing with Version Pertinence to Design an Efficient Schema Evolution Framework. in International Database Engineering and Applications Symposium (IDEAS). 1998. Cardiff, Wales, UK: IEEE Computer Society.

# Correctness of Model-based Component Composition without State Explosion*

Paul C. Attie David H. Lorenz

Northeastern University
College of Computer & Information Science
Boston, Massachusetts 02115 USA
{attie,lorenz}@ccs.neu.edu

**Abstract.** We present a methodology for designing component-based systems and verifying their temporal behavior properties. Our verification method is mostly automatic, and is not susceptible to the well-known *state-explosion* problem, which has hitherto severely limited the practical applicability of automatic verification methods. Our method specifies the externally visible behavior of each component $C$ as several *behavioral interface automaton* (BIA), one for each of the other components which $C$ interacts directly with. A BIA is a finite-state automaton whose transitions can be labeled with method calls. For each pair of directly interacting components, we compute the product of the BIA. These "pair machines" are then verified mechanically. The verified "pair properties" are then combined deductively to deduce global properties. Since the pair-machines are the product of only two components, they are small, and so their mechanical verification, e.g., by model checking, does not run up against state-explosion. The use of several BIA per component enables a clean separation between interfaces, so that the interactions of a component $C$ with several other components are cleanly separated, and can be inspected in isolation. This in itself promotes the understandability of a design. Our method also enhances extensibility. If a component is modified, only the pairs in which that component is involved are affected. The rest of the system is undisturbed. To our knowledge, our method is the first approach to behavioral compatibility that does not suffer from state-explosion.

## 1 Introduction

Software components [17] are supposed to make software less fragile and more reliable. In practice, however, part of the fragility is merely shifted from the component artifacts to the connectors and the composition process. When the composition is unreliable, component systems are just as fragile and unreliable as monolithic software. Improving the theoretical and practical foundation of third-party composition techniques [21] is thus essential to improving overall component software reliability.

In this paper, we make initial steps toward a new component model which supports behavioral interoperability and is based on the use of temporal logic and automata to

* This work was supported in part by the National Science Foundation (NSF) under Grant No. CCR-0204432, and by the Institute for Complex Scientific Software (www.icss.neu.edu) at Northeastern University.

specify and reason about concurrent component systems. Unlike other temporal logic and automata-based methods for software components, our work avoids using exhaustive state-space enumeration, which quickly runs up against the *state-explosion* problem where the number of global states of a system is exponential in the number of its components.

We present formal analysis and synthesis techniques that address issues of behavioral compatibility amongst components, and enable reasoning about the global behavior (including temporal behavior, i.e., safety and liveness) of an assembly of components. By avoiding state-explosion, our technique is not restricted to small, unrealistic applications.

### 1.1 Component interoperability

Components are "units of independent production, acquisition, and deployment" [17]. In *component-based software engineering* (CBSE) [10], software development is decoupled from assembly and deployment. Third party composition (assembly) is the activity of connecting components, which originate from different third party component providers in binary format and without their source code. During assembly, the application (or component) is assembled from other (compiled) components. The activity takes place after the compilation of the components and before the deployment of the application (which might be itself a compound component).

For two components, which were independently developed, to be deployed and work together, third-party composition must allow the flexibility of assembling even dissimilar, heterogeneous, precompiled third-party components. In achieving this flexibility, a delicate balance is preserved between prohibiting the connecting of incompatible components (avoiding false positive), while permitting the connecting of "almost compatible" components through adaptation (avoiding false negative). This is achieved during assembly through introspection, compatibility checks, and adaptability.

### 1.2 Interface compatibility

Parnas's principles [16] of information hiding for modules emphasize the separation of interface from implementation: components providing different implementations of the same interface can be swapped without having a functional affect on clients; two components need to agree on the interface in order to communicate. This works well in object-oriented programming where the design is centralized, but is not practical in component-based designs [14]. Agreement beforehand is possible only if third-party component providers were coordinated.

Extending Parnas's principles to component-based programming (CBP), the component clients (i.e., other components) must be provided with composition information and nothing more. Even agreement on the interface is no longer an accepted level of exported information. Components gathered from third parties are unlikely and cannot be expected to agree on interfaces beforehand. For third-party composition to work, components need to agree on how to agree rather then agree on the interface.

Indeed, CBP builder environments typically apply two mechanisms to overcome this difficulty and support third-party composition [13]. First, to check for interface

compatibility, builders use *introspection*. Introspection is a means for discovering the component interface. Second, builders support *adaptability* by generating adapters to overcome differences in the interface. Adapters are a means of fixing small mismatches when the interfaces are not syntactically identical.

### 1.3 Behavioral compatibility

The goal of work in behavioral compatibility for components is to develop support in CBP for *behavioral introspection* and *behavioral adaptability* that can be scaled up for constructing large complex component systems. While there is progress in addressing behavioral introspection and adaptability [22, 20, 23, 19, 18] there is no progress in dealing with the state explosion problem. The main focus of this work is in addressing the latter in a manner that can be applied to event-based components.

Currently, the introspector reveals only the interface, and adapters are used in an ad-hoc manner relying on names and types only. There are emerging proposals for handling richer interface mechanisms that express contractible constraints on the interface, e.g., the order in which the functions should be called, or the result of a sequence of calls. These methods typically rely on defining finite-state "behavioral" automata that express state changes. When two components are connected, the two automata can be tested for compatibility by producing their automata-theoretic product. This fails to provide a practical foundation for software growth, because of sate explosion; computation of the product of $K$ behavioral automata, each with $O(N)$ states, generates a product automaton of size $O(N^K)$. We address the challenge of avoiding state explosion. Elsewhere [3, 4] we present a pairwise design of an elevator system which, when scaled up to 200 floors, requires an upper bound of only $1,166,400$ states, instead of the $10^{180}$ that an approach which computes the product of all components would require. This is well within the reach of current model checkers.

## 2 Formal methods for components and composition correctness

Our interest is in large systems of concurrently executing components. A crucial aspect of the correctness of such systems is their temporal behavior. Behavioral properties can be classified as follows [12]: (1) *Safety properties*: "nothing bad happens" — for example, when an elevator is moving up, it does not attempt to move down without stopping first, and (2) *Liveness properties*: "progress occurs in the system" — for example, if a button inside an elevator is pressed, then the elevator eventually arrives at the corresponding floor. The required behavioral properties are given by a *specification*, which precisely documents what the system must achieve. *Formal methods* are those that provide a rigorous mathematical guarantee that a large software system conforms to a specification. Formal methods can be roughly classified as (1) *Proof-theoretic*: a suitable deductive system is used, and correctness proofs are built manually, or using a theorem prover, and (2) *Model-theoretic*: a model of the run-time behavior of the software is built, and this model is checked (usually mechanically) for the required properties. In our work, we emphasize model-theoretic methods, due to their greater potential for automation.

| | Interface compatibility | Automaton (BIA) compatibility | Behavioral compatibility |
|---|---|---|---|
| Export | interface | interface + automaton | complete code |
| Reuse | black box | adjustable | white box |
| Encapsulation | highest | adjustable | lowest |
| Interoperability | unsafe | adjustable | safe |
| time complexity | linear | polynomial for finite state | undecidable |
| Assembly properties | none | provable from pair properties | complete but impractical |
| Assembly behavior | none | synthesizable from pairwise behavior | complete but impractical |

**Table 1.** The interoperability space for components

## 3 The interoperability space for components

A *behavioral interface automaton* (BIA) of a component expresses some aspects of that components run-time (i.e., temporal) behavior. Depending on how much information about temporal behavior is included in the automaton, there is a spectrum of state information ranging from a maximal BIA for the component (which includes every transition the component makes, even internal ones), to a trivial automaton consisting of a single state. Thus, any BIA for a component can be regarded as a homomorphic image of the maximal automaton. This spectrum refines the traditional white-box/black-box spectrum of component reuse, ranging from exporting the complete source code (maximal automaton) of the component—white-box, and exporting just the interface (trivial automaton)—black box. Table 1 displays this spectrum.

In practice, it is unrealistic to expect the programmer to provide the maximal BIA, just as precisely specified semantics are rarely part of programming practices. As long as the most important behavioral properties (e.g., the safety-critical ones) can be expressed and established, a homomorphic image of the maximal automaton (which omits some information on the components behavior) is sufficient (Table 1 middle column).

The BIA can be provided by the component designer and verified by the compiler (just like typed interfaces are) using techniques such as abstraction mappings and model checking. Verification is necessary to ensure the correctness of the BIA, i.e., that it is truly a homomorphic image of the maximal automaton. Alternatively, the component compiler can generate a BIA from the code, using, for example, abstract interpretation or machine learning [15]. In this case, the BIA will be correct by construction. We assume the first option for third party components, and will explore the second option for components assembled in our builder.

## 4 Avoiding state-explosion by pairwise composition

In [1, 2], we present a method for the synthesis of finite-state concurrent programs from specifications expressed in the branching-time propositional temporal logic CTL [8].

This method avoids exhaustive state-space search. Rather than deal with the behavior of the program as a whole, the method instead generates the interactions between processes *one pair at a time*. Thus, for every pair of processes that interact, a *pair-machine* is constructed that gives their interaction. Since the pair-machines are small ($O(N^2)$), they can be built using exhaustive methods. A *pair-program* can then be extracted from the pair-machine. This extraction operation takes every transition of the pair-machine and realizes it as a piece of code in the pair-program [2, 9]. The final synthesized program is generated by a syntactic composition of all the pair-programs. This composition has a conjunctive nature: a process $P_i$ can execute a transition if and only if that transition is permitted by *every* pair-program in which $P_i$ participates. Thus, two pair-programs which have no processes in common do not interact directly. Two pair-programs that do have a process $P_i$ in common will interact via $P_i$: the pair-programs in effect must synchronize whenever $P_i$ makes a transition, so that the transition is executed in both pair-programs simultaneously. For example, if $P_1 \parallel P_2$, $P_2 \parallel P_3$, and $P_1 \parallel P_3$ are three pair-programs which all implement a two-process mutual exclusion algorithm, then they can be composed as discussed above into a single program which implements three-process mutual exclusion. In this program, when $P_1$ wishes to access the critical section, it must be permitted to do so by both $P_2$ (as per the $P_1 \parallel P_2$ pair-program) and by $P_3$ (as per the $P_1 \parallel P_3$ pair-program).

Due to the complexity of the synthesis and verification problems for finite-state concurrent programs, any efficient synthesis method is necessarily *incomplete*: it may fail to produce a program that satisfies a given specification even though such a program exists. In the synthesis method of [1, 2], the incompleteness takes the form of two technical assumptions that the pair-programs must satisfy in order for the synthesized program to be correct. One technical assumption requires that after a process $P_i$ executes, either it can execute again (i.e., is enabled), or it does not block any other process. This prevents deadlock. The other technical assumption requires that a process cannot forever block another process if the second process must make progress in order to satisfy a liveness property in the specification. This guarantees liveness.

We refer the reader to [1, 2] for examples of synthesis of solutions to the following problems, all for an arbitrarily large number of processes: $n$-process mutual exclusion, dining philosophers, drinking philosophers [5], $k$-out-of-$n$ mutual exclusion, and two-phase commit.

### 4.1 Applying pairwise composition to component assembly

To apply the pairwise method to components, we must be able to define the pairwise interaction amongst components. We do this by extending the component model so that each component $C$ is accompanied by several BIA [7, 20], one for each of the other components that $C$ interacts directly with. The BIA provides information about the externally observable temporal behavior of the component. For example, such an automaton could provide information on the order in which a component makes certain method calls to other components.

Given two components and their BIA, we construct the pair-machine for their interaction by simply taking the automata-theoretic product of the BIA. We can then model

check the pair-machine for the desired behavioral compatibility among the two components. If successful, we can then use this pair-machine as input to the pairwise method, as discussed above.

### 4.2 Discussion

The pairwise architecture enables a clean separation between interfaces. In the usual approach, a component has a single interface, through which it interacts with all other components. Thus, different interactions with different components are all mediated through the same interface. This results in an "entangling" of the run-time behaviors of various components, and makes reasoning (both mechanical and manual) about the temporal behavior of a system difficult. By contrast, our architecture "disentangles" the interactions of the components, so that the interaction of two components is mediated by a pair of interfaces, one in each component, that are designed expressly for only that purpose, and which are not involved in any other interaction. Thus, our architecture provides a clean separation of the run-time interaction behaviors of the various component-pairs. This simplifies both mechanical and manual reasoning, and results in a design and verification methodology that scales up.

Our architecture also facilitates extensibility: if a new component is added to the system, all that is required is to design new interfaces for interaction with that component. The interfaces between all pre-existing pairs of components need not be modified. Furthermore, all verification already performed of the behavior of pre-existing component-pairs does not need to be redone. Thus, both design and verification are extensible in our methodology. We can also apply our approach at varying degrees of granularity, depending on how much functionality is built into each component.

Vanderperren and Wydaeghe [22, 20, 23, 19, 18] have developed a component composition tool (PascoWire) for JavaBeans that employs automata-theoretic techniques to verify behavioral automata. They acknowledge that the practicality of their method is limited by state-explosion. Incorporating our technique with their system is an avenue for future work.

DeAlfaro and Henzinger [7] have defined a notion of interface automaton, and have developed a method for mechanically verifying temporal behavior properties of component-based systems expressed in their formalism. Unfortunately, their method computes the automata-theoretic product of all of the interface automata in the system, and is thus subject to state-explosion.

## 5 Conclusion

We have presented a methodology for designing components so that they can be composed in a pairwise manner, and their temporal behavior properties verified without state-explosion. Our method specifies the externally visible behavior of each component $C$ as several behavioral interface automaton, one for each of the other components which $C$ interacts directly with. Finite-state automata are widely used as a specification formalism, and so our work in compatible with the mainstream of component-based software engineering.

Ensuring the correct behavior of large complex systems is the key challenge of software engineering. Due to the ineffectiveness of testing, formal verification has been regarded as a possible approach, but has been problematic due to the expense of carrying out large proofs by hand, or with the aid of theorem provers. One proposed approach to making formal methods economical is that of automatic model checking [6]: the state space of the system is mechanically generated and then exhaustively explored to verify the desired behavioral properties. Unfortunately, the number of global states is exponential in the number of components. This state-explosion problem is the main impediment to the successful application of automatic methods such as model-checking and reachability analysis. Our approach is a promising direction in overcoming state-explosion. In addition to the elevator problem, the pairwise approach has been applied successfully to the two-phase commit problem [1] and the dining and drinking philosophers problems [2].

Large scale component-based systems are widely acknowledged as a promising approach to constructing large-scale complex software systems. A key requirement of an successful methodology for assembling such systems is to ensure the behavioral compatibility of the components with each other. This paper presents a first step towards a practical method for achieving this.

## References


1. P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR'99: 10th International Conference on Concurrency Theory*, number 1664 in LNCS. Springer-Verlag, Aug. 1999.
2. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, Jan. 1998.
3. P. C. Attie and D. H. Lorenz. Establishing behavioral compatibility of software components without state explosion. Technical Report NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Mar. 2003. `http://www.ccs.neu.edu/home/lorenz/papers/reports/NU-CCIS-03-02.html`.
4. O. Aytar, P. C. Attie, and D. H. Lorenz. An implementation of an elevator system in the ioa language and toolset. Technical Report NU-CCIS-03-04, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Mar. 2003. `http://www.ccs.neu.edu/home/lorenz/papers/reports/NU-CCIS-03-04.html`.
5. K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Reading, Mass., 1988.
6. E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.
7. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2001.
8. E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, Mass., 1990.
9. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
10. G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

11. ICSE 2001. *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, May 12-19 2001. IEEE Computer Society.
12. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, Mar. 1977.
13. D. H. Lorenz and P. Petkovic. Design-time assembly of runtime containment components. In Q. Li, D. Firesmith, R. Riehle, G. Pour, and B. Meyer, editors, *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems*, pages 195–204, Santa Barbara, CA, July 30-Aug. 4 2000. TOOLS 34 USA Conference, IEEE Computer Society.
14. D. H. Lorenz and J. Vlissides. Designing components versus objects: A transformational approach. In ICSE 2001 [11], pages 253–262.
15. E. Mäkinen and T. Systä. MAS - an interactive synthesizer to support behavioral modeling in uml. In ICSE 2001 [11], pages 15–24.
16. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communication of the ACM*, 15(12):1059–1062, 1972.
17. C. Szyperski. *Component-Oriented Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
18. W. Vanderperren. A pattern based approach to separate tangled concerns in component based development. In Y. Coady, editor, *Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 71–75, Enschede, The Netherlands, Apr. 2002.
19. W. Vanderperren and B. Wydaeghe. Separating concerns in a high-level component-based context. In EasyComp Workshop at ETAPS 2002, April 2002.
20. W. Vanderperren and B. Wydaeghe. Towards a new component composition process. In Proceedings of ECBS 2001, April 2001.
21. K. C. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Software Engineering. Addison-Wesley, 2001.
22. B. Wydaeghe. PACOSUITE: Component composition based on composition patterns and usage scenarios. PhD Thesis.
23. B. Wydaeghe and W. Vandeperren. Visual component composition using composition patterns. In Proceedings of Tools 2001, July 2001.

# Consistency Validation of UML Diagrams

Boris Litvak[1**], Shmuel Tyszberowicz[2] and Amiram Yehudai[2]

[1] Tel-Aviv University,
boris.litvak@lycos.com
[2] Tel-Aviv University and the Academic College of Tel-Aviv Yaffo,
{tyshbe,amiramy}@post.tau.ac.il

**Abstract.** UML provides several kinds of diagrams to model the behavior and structure of a system under development. A consistency problem may arise due to the fact that some aspects of the model may be described by more than one diagram. Hence, the consistency of the system description should be checked before implementing the system.
This paper describes an algorithmic approach to a consistency check between UML Sequence and State diagrams. The algorithm we provide handles also complex State diagrams, e.g. diagrams that include forks, joins, and concurrent composite states. We present BVUML, a tool we have implemented in order to automate the validation process.

## 1 Introduction

All the UML diagrams used in a software project must form a consistent view of the project. An example of a relatively simple inconsistency is mentioning a method name for an object in a Sequence diagram, which does not appear in the method names of the class of this object.

Inconsistencies can be more complex than that. Let us introduce some formal definitions to make the explanation easier.

- *Leg:* A basic component of a Sequence diagram or a State diagram, depicted in UML as an arrow with an optional guard and an optional action. In UML, a *Leg* is called a *simple transition* in a State diagram, whereas in a Sequence diagram it is called a *message.*
- *Transition:* Consists of a trigger and a nonnegative number of legs.

** *Supported by the Deutsch Institute*

- *Sequence (diagram) transition:* A transition in a Sequence diagram that consists of an incoming leg (message) and all the outgoing legs (messages), if any, triggered by it.
- *State (diagram) transition:* A transition in a State diagram that includes the trigger and all the legs on the path between two persistent states, with no persistent states in between.
- *Current object state:* The conjunction of all the active states in a State diagram, that defines the behavior of the object for any incoming event. At the beginning of the object's lifetime, its *current object state* includes only all the initial State diagram states. The *current object state* is updated with each state transition. It can include multiple simple states, if the diagram contains concurrent composite or fork states.

The *Sequence transition* must match at least one *State transition* that emanates from a *current object state*. Otherwise the diagrams are inconsistent.

The problem of finding such inconsistencies becomes very complex in many cases. For example, the *current object state* may consist of many simple states due to transitions into concurrent composite, fork, and join states. These simple (non-composite persistent) states may be in different levels of State diagram hierarchy, as will be explained later. States that change the control behavior, such as choice states and history states, make it harder to locate the *current object state* after a given transition. Many additional elements, such as transition guards (conditions), on entry/ on exit actions, transitions that pass hierarchical levels in the State diagram, etc, were not mentioned in this introduction.

The rest of the paper is organized as follows: In Section 2 we present the proposed solution by means of an example. Section 3 describes the algorithm used to solve the consistency problem. We conclude with a short reference to related work in Section 4.

## 2 A Simple Example

The example describes the interaction between the components of a cellular phone, and the phone user. The example is taken from [5].

One use case of the cellular system is dialing a number. In one of the scenarios of this use case the user has entered a phone number. Just before he presses the send button, there is an incoming call. The cellular phone allows the user to answer an incoming call even after he has entered several digits. When the user presses the send

button, this means that the `Dialer` object either has to make a call or to receive an incoming call. The decision is made according to the `Dialer` state, whether there was an incoming call or not.

We want to present how our tool, BVUML, finds the possible inconsistencies. To illustrate the inconsistency search process, we use a Sequence diagram that is inconsistent with respect to the State diagram. The Sequence and State diagrams for the `Dialer` are specified in Figures 1 and 2, respectively. The Sequence diagram designer intended to make a call, even though the pressing of the `Send` button should- according to the State diagram of the Dialer- answer an incoming call. The run of the `Dialer` will detect the inconsistency at the first `send` message (message #6 in the Sequence diagram).

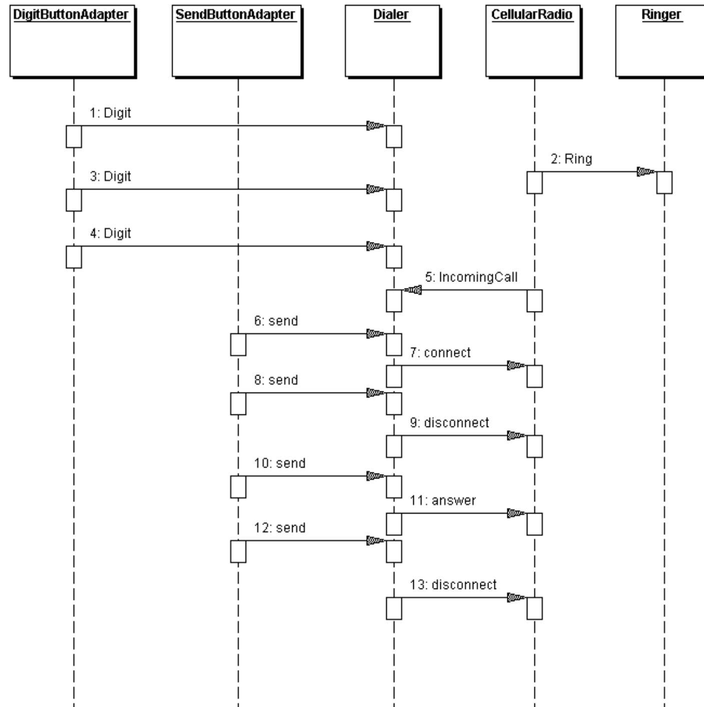



**Fig. 1.** Dialer Sequence Diagram

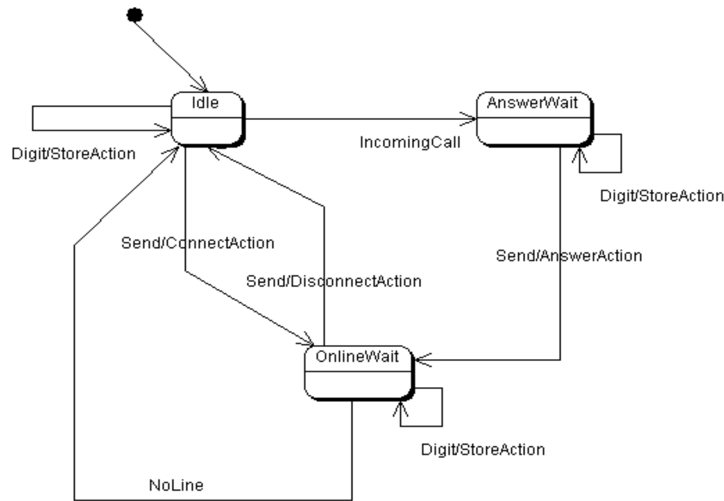



**Fig. 2.** Dialer State Diagram

### 2.1 BVUML top level results

In the following section we shortly describe the output of BVUML on the State and Sequence diagrams shown above.

A line which starts with “Trying < State diagram leg ”, ends with “... failure” or “... success”. This specifies the status of a particular transition search, i.e. given a sequence transition and the current states, the status of whether it is legal to traverse this leg or not, is displayed. There may be many failures of this kind in a successful run, as shown next.

```
| Running (001) Trigger=Digit Legs=[] :
|   Running top with currentStates[top.InitialState] :
|   Running top.Idle (externally) :
|      Running  Trigger=IncomingCall Leg: {}
          to top.AnswerWait.  ... failure
|      Running  Trigger=Send Leg:
                {action=ConnectAction} to top.OnLineWait.
                                          ... failure
|      Running  Trigger=Digit Leg: {} to top.Idle.
                                          ... success
| Running (002) Trigger=Digit Legs=[] :
|   Running top with currentStates[top.Idle] :
|   Running top.Idle (externally) :
|      Running  Trigger=IncomingCall Leg: {}
                      to top.AnswerWait.  ... failure
```

```
|      Running  Trigger=Send Leg: {action=ConnectAction}
                      to top.OnLineWait.  ... failure
|      Running  Trigger=Digit Leg: {} to top.Idle.
                      ... success
| Running (003) Trigger=Digit Legs=[] :
|   Running top with currentStates[top.Idle] :
|   Running top.Idle (externally) :
|      Running  Trigger=IncomingCall Leg: {}
                      to top.AnswerWait.  ... failure
|           Running  Trigger=Send Leg:
                      {action=ConnectAction}
                      to top.OnLineWait.  ... failure
|           Running  Trigger=Digit Leg: {} to top.Idle.
                                          ... success
| Running (005) Trigger=IncomingCall Legs=[] :
|   Running top with currentStates[top.Idle] :
|   Running top.Idle (externally) :
|      Running  Trigger=IncomingCall Leg: {}
                      to top.AnswerWait.  ... success
|      Running  Trigger=Send Leg: {action=ConnectAction}
                      to top.OnLineWait.  ... failure
|      Running  Trigger=Digit Leg: {} to top.Idle.
                                          ... failure
| Running (006) Trigger=Send Legs=[Leg(007):
                {action=ConnectAction}] :
|   Running top with currentStates[top.AnswerWait] :
|   Running top.AnswerWait (externally) :
|      Running  Trigger=Send Leg: {action=AnswerAction}
                      to top.OnLineWait.  ... failure
|      Running  Trigger=Digit Leg: {} to top.AnswerWait.
                                          ... failure
```

Let us look again at the last transition (number 006) of the Sequence diagram. The `ConnectAction` is included in the Sequence diagram (though ArgoUml does not show actions), but the only transition for the trigger `Send` in the State diagram includes `AnswerAction` as its action. Thus the Sequence diagram is not built correctly, and the following error message is produced:

```
Run Failed - Sequence Diagram is not suitable to the State Diagram
```

Much more complex examples can be verified using our approach. An example of a complex State diagram that BVUML verified is provided in Figure 3. This is part of a Navy Command and Control System. This system tracks objects that appear in a warfare arena. Due to space limitation we cannot elaborate on the full example. The interested reader can find the full example in [4]. Following is an example of two consecutive transitions as given by BVUML:

```
| Running (1.7.0) Trigger=FinishedGlobalRadetUpdate
     Legs=[] :
|   Running top with
```

```
      currentStates[top.Stage2.ContactIntelligence,
                    top.Stage2.EndRadetUpdate] :
|   Running top.Stage2.ContactIntelligence (externally) :
|     Running  Trigger=FinishedRequest Leg: {}
               to top.Stage2.Answered.  ... failure
|   Running top.Stage2 (externally) :
|   Running top.Stage2.EndRadetUpdate (externally) :
|     Running  Trigger=FinishedRadetUpdate Leg: {}
               to top.Join1.  ... failure
|     Running  Trigger=FinishedGlobalRadetUpdate Leg: {}
               to top.Join2. :
|       ... success - join count incremented
| Running (1.7.1) Trigger=FinishedRequest Legs=[Leg(1.7.2):
               {action=SendConfirmation,
                guard=AnswerConfirmed},
               Leg(1.7.3):
                {action=WriteLog, guard=WriteLog}] :
|     Running top with
               currentStates[top.Stage2.ContactIntelligence] :
|     Running top.Stage2.ContactIntelligence (externally) :
|       Running  Trigger=FinishedRequest Leg: {}
               to top.Stage2.Answered. :
|       Trying Leg:
               {action=SendConfirmation,
                guard=AnswerConfirmed}
               to top.Stage2.WillingToAnswer  ... success.
|       Trying Leg: {guard=NoAnswer}
               to top.Stage2.EndIntUpdate  ... failure.
|       Trying Leg: {guard=NoLogWrite}
               to top.Stage2.EndIntUpdate  ... failure.
|       Trying Leg: {action=WriteLog, guard=WriteLog}
               to top.Stage2.EndIntUpdate  ... success.
| Running (1.7.4) Trigger=FinishedIntelligenceUpdate
               Legs=[Leg(1.7.6):
               {action=OpenTransactionAction}] :
|   Running top with
               currentStates[top.Stage2.EndIntUpdate] :
|   Running top.Stage2.EndIntUpdate (externally) :
|     Running  Trigger=FinishedIntelligenceUpdate Leg: {}
               to top.Join2. :
|       ... join is saturated.   // See Section 5.2.
|       Trying Leg: {} to top.Stage3  ... success.
```

## 3 Solution Description

This section describes the algorithm that we use in BVUML. The inputs to the algorithm are:

- The software project model description file. This file contains all the information about the UML model.
- The name of the Sequence diagram to be checked.

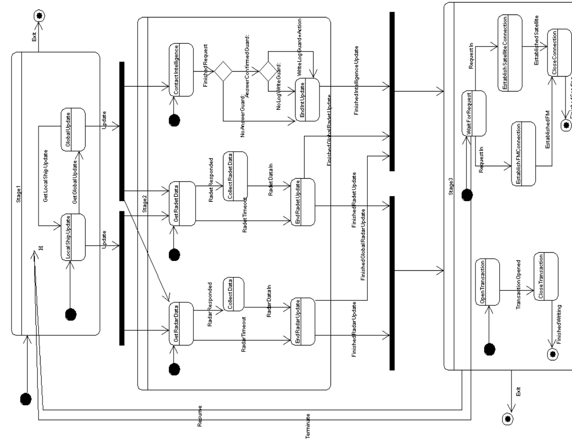


**Fig. 3.** Target Update State Diagram

– The name of the object in the Sequence diagram one would like to check.

Each run of BVUML validates the consistency on a specific object in a Sequence diagram. BVUML should be activated several times for different objects of the same Sequence diagram. In this way one can ensure that some interactions between the objects in the Sequence diagram are possible, according to their corresponding State diagrams. One can also check that some runs are impossible according to the appropriate State diagrams. It is especially vital when one wants to ensure that some action always happens before another.

### 3.1 The main idea of the solution

The BVUML output visualizes a hybrid Sequence-state diagram, i.e, states are associated with the Sequence diagram object lifeline. From the input to BVUML we can infer which State diagram we want to check and which object in the Sequence diagram we want to check. A message in the Sequence diagram is interpreted in one of two ways in the State diagram:

1. As a trigger, when the message is an incoming message to the checked object of the Sequence diagram.
2. As an action, when the message is an outgoing message from the checked object of the Sequence diagram.

A message can change the *current object state* of the State diagram. Notice that the *current object state* changes according to the previous transitions.
Our algorithm iterates over the Sequence diagram transitions of a given object. Each sequence transition is compared with the set of state transitions that start from the *current object state*.

In order to find whether a Sequence diagram and a State diagram of an object $obj_i$, which is included in the Sequence diagram, are consistent, the following steps are taken.

The Sequence diagram transitions are traversed by the algorithm. For each transition, it checks whether there is any suitable State diagram transition, i.e. a transition with the same trigger, the same actions with identical guards, and the same action ordering as in the Sequence diagram transition. The suitable state transitions are searched from the *current object state*. If there is a suitable state transition, the iteration continues; otherwise an error message is issued. This process of iteration and comparison is called a *run* of a Sequence diagram over the State diagram. This run can be done for each object in a Sequence diagram. The State diagram that is a part of the run is the one associated with the class of the object.

Notice that — due to the BFS search structure of the algorithm, several state transitions can be executed per each sequence transition, from the relevant states in the *current object state*. In this way concurrency is supported in a State diagram.

## 4 Related Work

Several projects and researchers have tackled related problems. Due to lack of space we only briefly mention some of them. A more detailed discussion can be found in [4].

- HUGO [7] uses model checking techniques to verify Collaboration and State diagrams. HUGO currently does not support some UML constructs, such as: internal transitions, choice states, time events, etc..
- Prosasim UML simulator [3] runs the application and follows, in real-time, the execution inside the UML model context. Simulation is based on native source code. State and Activity diagrams

are supported. Prosasim presents the behavior of the UML model graphically.
– PecSee project [6] automatically generates Statechart designs from Sequence diagrams, taking into account information present in the Class diagrams.
– LSC, Live Sequence Charts [1], looks for inconsistency among State and Sequence diagrams by means of logic and introduces state-marks into the Sequence diagrams.
– IIOSS Model Debugging Facility (MDF) [2] is a software development environment that offers a tool whose multiple facets include: a UML editor (MEF), a model simulator, an interface builder, a file converter, and an object-oriented database.

## References


1. W. Damm and D. Harel. Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.
2. The IIOSS project. Integrated Inter-Exchangeable Object-Modeling and Simulation system. http://www.iioss.org/index-e.html.
3. Using Prosa professional UML case envirinment together with Microsoft Visual Studio 6.0 and application wizard. http://www.insoft.fi/eng/pr2Proc.htm.
4. B. Litvak. Consistency validation of UML diagrams. Master's thesis, Tel-Aviv Universtity, Israel, 2003. In preparation; See also http://www.anglefire.com/linux/borisl/thesis.ps.
5. R. Martin. UML tutorial. http://www.objectmentor.com/courses/resources/articles/umlsequencediagrams.pdf. Cf. Race conditions chapter.
6. J. Schumann and J. Whittle. Automatic synthesis of UML designs from requirements in an iterative process. In *The Workshop for Precise Modeling and Deduction for Object-oriented Software Development (PMD '01)*, Siena, Italy, 2001.
7. S. M. T. Schafer, A Knapp. Model checking UML state machines and collaborations. *Proceedings Workshop software model checking*, 55(3), 2001.

# Use Case-Oriented Software Architecture

Eliezer Kantorowitz, Alexander Lyakas, and Artur Myasqobsky

Computer Science Dept.
Technion—Israel Institute of Technology
32000 Haifa, Israel
{kantor|lyakasal|artiom}@cs.technion.ac.il
http://www.cs.technion.ac.il/~kantor/

**Abstract.** One of the challenges of software development regards ensuring that the code implements the specifications precisely (code verification). This study focuses on a framework designed to enable a "direct" manual translation of sufficiently detailed natural language use case specifications into code, whose equivalence with the specifications should be easy to establish. The experimental framework *Simple Interfacing+* ($\mathcal{SI}+$) provides components for writing applications with use case-oriented architectures. Software produced with $\mathcal{SI}+$ centers around use case components, that implement the different use cases of the application. It is therefore relatively easy to trace the specification to which each piece of code belongs. This is useful when it is required to modify the code, as we readily know the use case specification, that the modified code must satisfy. The user interface is produced by an $\mathcal{SI}+$ component, named *use case displayer*, which implements an environment, where the user can execute the various use cases. $\mathcal{SI}+$ has been used to implement small information systems. The code based on the high level $\mathcal{SI}+$ abstractions was sufficiently close to the natural language use case specifications to facilitate the verification. The implementation of more complex applications with $\mathcal{SI}+$ is planned to obtain a more complete evaluation.

## 1 Introduction

This study is concerned with the design of a framework for construction of use case-oriented interactive information systems (IS). A use case is a single kind of usage of an application by its user(s) [1]. Also, [1] states, that "the set of all use case descriptions specifies the complete functionality of the system", i.e., with use cases it is possible to provide a complete specification of the system.

We are especially interested in use case specifications, as they facilitate the manufacturing of systems with high usability, i.e., systems that enable the users to accomplish their work in a pleasant and

efficient way. This is achieved by employing usability considerations in the design of the use cases [2]. We consider therefore software development processes, where the system is specified by its use cases. An example of such a process is the Unified Software Development Process (USDP) [3], which was designed in connection with the quite popular Unified Modelling Language (UML) [4].

The USDP process begins with requirements elicitation and continues with the specification of the application's functionality by its use cases. First the use cases of the system and their users (actors in UML terminology) are identified and specified by UML use case diagrams. Each use case is thereafter designed and described in full detail from the user's point of view, i.e., without implementation details. The use cases are designed to have good usability properties, for example by employing the approach of [2]. The resulting use cases are described in a sufficiently detailed natural language and include detailed drawings of the graphical user interface. The natural language specification may say such things as "hitting the OK button initiates the computation of ...".

The use cases are specified by a natural language in order to enable validation by domain experts and usability experts, who may not be familiar with formal specification languages. Sometimes a prototype of the user interfaces is prepared from the use case specification [5, 6], enabling a more intensive validation. In the USDP process, the validated natural language use cases are then translated into formal UML diagrams. The USDP process continues with the analysis of the formal use cases specification, system design, implementation and testing. Since this USDP process is based on use cases that have been designed and validated for their usability, the produced application is expected to have the same good usability.

The USDP testing phase includes verification, i.e., checking that the code implements the use case specifications precisely. This is usually done by testing each one of the different use cases with sets of test cases, that ideally cover all the different kinds of possible scenarios (black box testing).

The verification may also be accomplished with formal methods. This requires that the natural-language use case specification is translated into a formal specification. In this case, an additional verification of the correctness of the translation from the natural

language into the formal specification language is required. Using formal methods, e.g., [7], to show that the code implements the formal specification is thus not a trivial task.

The high costs of verification by either testing or by formal methods may be avoided in situations, where it is possible to produce the code automatically from the specifications. One example is a state charts specification of reactive systems [8], that may be translated automatically to code [9]. Another powerful and interesting approach is the *Play-In/Play-Out* approach of [10]. Here the system is specified by a set of scenarios, entered directly via a GUI (play-in). Later, the specification may be tested and debugged by operating the GUI and checking the system reactions, again via the GUI (play-out). In some cases, play-out may actually serve as the final implementation. This promising approach focuses on reactive systems. Its suitability for information systems has not yet been investigated.

We are not aware of a general method for automatic generation of the code of information systems from their specifications. We employ therefore a less ambitious verification costs reduction approach, that was introduced in [11]. In this approach the sufficiently detailed natural-language use case specification is translated directly by programmers into high-level code, which implements the use cases. The verification effort in this approach is reduced to showing the equivalence between the natural-language use case specification and the code. A framework that enables such a direct manual coding of the specifications is called a *specification-oriented* framework.

The approach was tested [11] with an experimental framework called $\mathcal{SI}$. In one experiment 2.5 $\mathcal{SI}$-based Java code statements were required on the average for each English statement in the use case specification. This was considerably less than the 7 Java statements required when coding in the traditional way, i.e., using Java's Swing and JDBC packages. The $\mathcal{SI}$-based Java code was thus considerably shorter, which indicates its high level of abstraction. The $\mathcal{SI}$-based JAVA code had furthermore a higher level of resemblance to the English language use case specification, which facilitated the establishing of their equivalence.

In this paper we report the results obtained with a redesigned version of $\mathcal{SI}$ called $\mathcal{SI}$+. The new improved $\mathcal{SI}$+ framework exploits experiences gained from using $\mathcal{SI}$.

## 2 The Model

The $\mathcal{SI}+$ framework assumes that an interactive information system application has

- A database that represents the state of the domain of the application
- A GUI that facilitates the communication between the human user and the application

The use case specifications of such an information system must therefore have instructions for input and output through the GUI as well as instructions for database manipulations. The $\mathcal{SI}+$ was therefore designed to have high level abstract classes for the coding of these two kinds of instructions. $\mathcal{SI}+$ is not supporting use case instructions beyond these two kinds. The developer must either find ready frameworks for the implementation of the use case instructions, that are not supported by $\mathcal{SI}+$, or develop the code for their implementation.

An application developed with the $\mathcal{SI}+$ will have a separate software component (object) for each one of its use cases. Such a use case component may incorporate a number of different sub-use cases (objects, again), which may be reused in different use cases. A use case component contains the code obtained by the manual translation of the natural language use case specification into Java. The use case component also contains an abstract specification of the GUI that serves the use case. This GUI specification lists the employed controls (buttons, menus, list boxes etc.) and their purposes. The geometrical shape of the controls, their color and further properties, are not specified in the use case component, but in a separate *interaction style* component [12]. Each pane (rectangular screen area) of the GUI may employ a different interaction style, and each interaction style component may be reused in different use cases and in different applications. Having the GUI details hidden in separate interaction style components makes the code of the use case short and easy to check.

Figure 1 presents the code, that constructs the GUI of the *Translate* use case in a simple Hebrew-English dictionary system. The programmer instantiates panes (which are interaction style components)

and adds GUI controls or another panes to them. When adding a control to a pane, the programmer gives the control a symbolic name, specifies its instantiation parameters (defaults are provided for all controls), and provides the name of the method that will be invoked when the control is operated. Controls are created by *control factories* (see Factory design pattern in [13]), that configure the controls correctly according to the instantiation arguments. The GUI of the *Translate* use case is presented on Figure 2.

```
SIPane word_to_translate_pane = new SIPaneOneRow(null, this);
word_to_translate_pane.addControl(SIDefaultFactory.TEXTFIELD,
                                  "word_to_translate", 20, null);
word_to_translate_pane.addControl(SIDefaultFactory.BUTTON, null,
                                  "Translate!", "translate_pressed");

SIPane translate_pane = new SIPaneOneColumn("Translate:", this);
translate_pane.addSIPane(word_to_translate_pane);
translate_pane.addControl(SIDefaultFactory.LISTBOX, "translations",
                          null, null);

SIPane main_pane = new SIPaneOneColumn(null, this);
main_pane.addSIPane(translate_pane);
main_pane.addControl(SIDefaultFactory.BUTTON, null,
                     "Add Translation", "add_trans_pressed");

setMainSIPane(main_pane);
```

**Fig. 1.** Constructing the GUI of the *Translate* use case

The use cases are executed by a component called use case displayer. The displayer organizes the GUI's of the different use cases in a single desktop. The geometrical layout of these GUI's is computed by the displayer in accordance with interaction styles specified. Figure 3 demonstrates one of the displayers available in $\mathcal{SI}$+. On this figure the displayer is used to visualize the use cases of MP3 Manager application. Currently the user executes three use cases—*Manage MP3 List*, *Play MP3 Files* and *Edit ID3 Tag*—and views online help for the *Edit ID3 Tag* use case.

A further component of an $\mathcal{SI}$+-based application is an of-the-shelf database management system. In the experimental system we employ relational database systems. These mature, widely available sys-

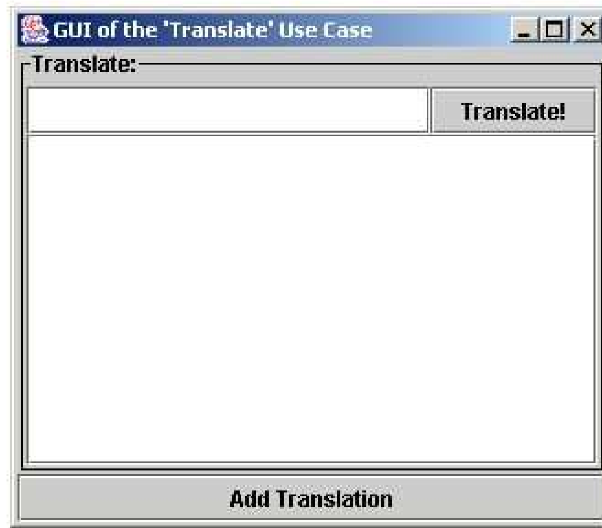



**Fig. 2.** The GUI of the *Translate* use case

tems come with a powerful structured query language (SQL) and a mechanism for managing concurrent transactions. $\mathcal{SI}+$ provides several high-level components, that simplify database access operations. In Figure 4 the programmer executes a query that presents personal details and benefits of employees whole salary is above 50,000. The result set of a query is passed directly to one of $\mathcal{SI}+$ table models and presented in the table (table model is a Swing concept; table models control the contents of GUI tables).

## 3 Evaluation

The experimental framework has been tested on a number of small systems. Testing with larger, more realistic applications is necessary in order to make valid conclusions.

Based on the observations already made we shall now explain why we expect that use case-oriented structure facilitates the design of correct applications. The few $\mathcal{SI}+$-based applications made hitherto suggest, that the different use cases can be designed and implemented (almost) independently of each other ("low coupling"). One explanation for this fortunate property is that the different use cases

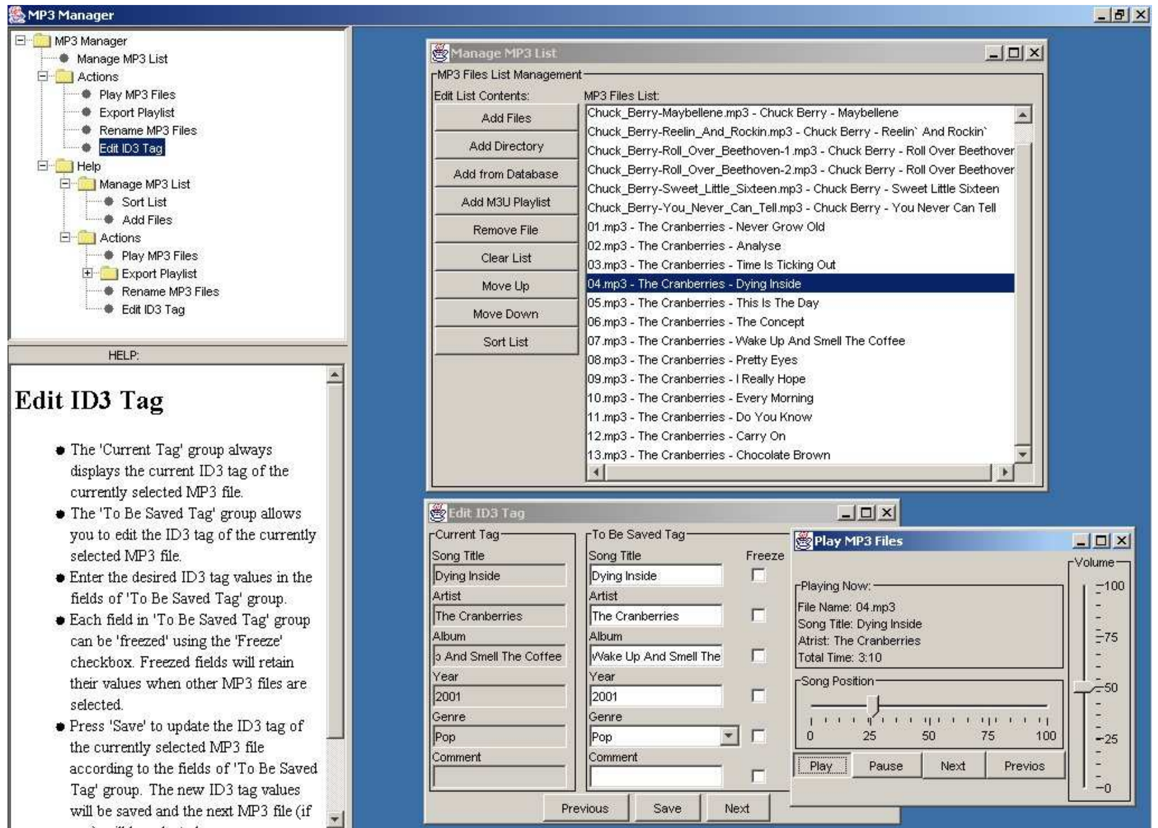



**Fig. 3.** The MP3 Manager application utilizing one of $\mathcal{SI}$+ displayers

```
SIMutableRSTableModel model = new SIMutableRSTableModel();
//...
ResultSet rs = DBProxy.getDBProxy().executeQuery("SELECT FIRSTNAME, LASTNAME, BENEFITS " +
               "FROM EMPLOYEES, EMP_STATS WHERE " +
               "EMPLOYEES.EID=EMP_STATS.EID AND EMP_STATS.SALARY > 50000");
model.setDataFromResultSet(rs);
```

**Fig. 4.** Direct presentation of a query in a table

usually do not communicate directly with each other. Typically a use case communicates only with its human user and with the database. The communication with the database is done by atomic transactions, which should be designed, in information system fashion, to retain the consistency of the database. Since the database represents the state of the application, this strategy retains the consistency of the state of the application.

The use case organization facilitates both verification by comparing the code to the natural language specifications and by black-box testing, where each use case component is tested with all its different kinds of scenarios.

Software applications tend to evolve over the years, which reflect additions and modifications to the original specifications. A discussion of ways to ensure the correctness of an application must therefore also consider the situation of changes and additions to the specifications. For an $\mathcal{SI}+$-based application the addition of new use cases may in some cases be done without touching or even knowing anything about other use cases. This is due to use case independency phenomenon discussed above, which is likely to occur when the added new use case communicates directly only with the database and the users. Using the terminology of [14] the proposed architecture seems to have an extension complexity of $O(1)$, i.e., the effort required to add a new use case is independent of the number of already existing use cases.

The schema of the database must of course be designed to represent the application domain precisely, such that it may be employed by "any" unknown future use case. Techniques for designing such database schema are well established [15].

The structuring of the software into use case components is expected to facilitate future code modifications. If we have to modify some code, we know that the resulting code must meet the specification of the use case component in which it is located.

The observations made and the above analysis suggest that the proposed use case program structures facilitate in a number of different ways the manufacturing of correct interactive information system. More experience with more complex system is needed to validate these expectations.

## References


1. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley (1992)
2. Seffa, A., Djouab, R., Antunes, H.: Comparing and Reconciling Usability-Centered and Use Case-Driven Requirements Engineering Processes. Second Australian User Interface Conference (AUIC'01) (2001)
3. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley (1999)
4. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley (1999)
5. Junko, S., Yoshiaki, F.: GUI Prototype Generation by Merging Use Cases. Proceedings of the 7th international conference on intelligent user interfaces, San Francisco (2002)
6. Elkoutbi, M., Khriss, I., Keller, R.: Generating User interface Prototypes from Scenarios. Proc. of IEEE International Symposium on Requirements Engineering (RE'99) (1999)
7. Chechik, M., Gannon, J.: Automatic Analysis of Consistency between Requirements and Designs. IEEE Transactions on Software Engineering, 27, 7 (July 2001)
8. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8, (1987) 231-174, North Holland
9. Rhapsody product of I-Logix company. `http://www.ilogix.com/products/rhapsody/index.cfm`
10. Harel, D., Marelly, R.: Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. Software and System Modeling (SoSyM), to appear (2003)
11. Kantorowitz, E., Tadmor, S.: A Specification-Oriented Framework for Information System User Interface. Workshop of Object-Oriented Information Systems 2002 (OOIS'02), Springer LNCS 2426 (2002)
12. Kantorowitz, E., Sudarsky, O.: The Adaptable User Interface. Commun. ACM, 32, 11 (Nov 1989), 1352-1352
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
14. Kantorowitz, E.: Algorithm Simplification through Object Orientation. Software Practice and Experience, 27(2) (1997) 173-183
15. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W. (contributor): Object-Oriented Modeling and Design. Prentice Hall (1990)

# Observing Component Behaviors with TLA

Nicolas Rivierre and Thierry Coupaye

France Télécom R&D (DTL/ASR),
38-40 rue du Général Leclerc, 92794 Issy Les Moulineaux Cedex 9, France
{nicolas.rivierre,thierry.coupaye}@rd.francetelecom.com

**Abstract.** Temporal logic, and especially the Temporal Logic of Actions (TLA), allows the specification of the correct behaviors of a system. This position paper outlines a practical approach to verify that components do not violate their TLA specification at runtime.

## 1 Introduction

This position paper deals with specification and verification of component-based systems. First, we describe an approach for producing observers from TLA specifications of a system [8]. Such observers are useful at runtime, for testing or monitoring purposes, to detect temporal faults by checking an actual implementation of a system against a formal and verified behavioral model. Then, we discuss why we believe that component-based systems help to enforce the correctness of software system. This discussion is illustrated with Fractal [2, 3], a workable component framework. Section 2 discusses the role of observation. Section 3 provides an overview of our design choices to implement TLA observers. Section 4 discusses the implication of observation in component frameworks. Section 5 concludes.

## 2 The role of observation

The role of observation, as discussed in this paper, should be understood as follows (cf. figure 1). The actual implementation of the system is continuously checked against a formal and verified model of some adequately selected aspects of the system behavior [5]. Ideally, observation could be avoided by proving that an actual implementation of a system is consistent with an abstract specification. However, in practice, even if this specification has been formally verified using theorem prover or model checker tools [4, 7], the designer often lacks formal support to check for the correctness of an actual implementation. Indeed, refinement analysis requires nontrivial proofs to show that properties demonstrated at an abstract level are preserved when considering actual implementation details. Also, although some specification languages, such as B [1], propose methodologies and tools that enable designers to translate formal specifications into an executable language, refinement reasoning is often available only at a specification level. This motivates the need for observation. Of course, testing the observable behaviors of an implementation is not

a panacea. It fails to provide guarantees for all possible situations (even considering continuous monitoring) and addresses only safety properties (whose violation can be illustrated on finite behaviors). But it is a common way to increase confidence that a system is correct. At least, it provides counter-examples when the design is incorrect and certainly does not exclude other complementary techniques (proof, animation, simulation...). Unit-level tests are well adapted for local debugging, tuning or performance evaluation but fail to check behavioral requirements. Executable assertions based on pre-post conditions and invariants [12], whose main reference is Eiffel, are more expressive and well understood by developers but they cannot assert temporal properties. Another approach is to check observable behaviors against a formal and verified model. This latter approach seems advisable for behavioral testing since it relies on formal specifications to decide which temporal properties to ensure. However, its practical application can be quite difficult since formal specification (temporal logic, process algebra...) are executable at model level. Formally, if the specification of a system is represented by a temporal formula $F$, the execution of $F$ consists of computing whether a (finite) model $M$ of the system satisfies $F$ ($M \models F$). For this reason, there has been some interest in deriving observers or oracles from the formal specification of the system [5, 6, 13, 15, 16].

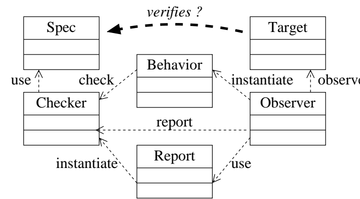



**Fig. 1.** Observation: *Spec* is a specification of the correct behaviors of some aspects of the system. *Target* is the actual component of the system that is to be checked against *Spec* (*Target* => *Spec* is not provable). *Observer* perceives the behaviors of *Target* and translates them to be understable by *Checker*. *Checker* is an oracle (a runtime checker) that raises a *Report* whenever a *Behavior* perceived by *Observer* violates *Spec*.

## 3 TLA observers

This section outlines a practical approach to implement TLA observers and illustrates the process with a simple example.

**TLA overview** The Temporal Logic of Actions (TLA) has been proposed by Lamport [8] for the specification[1] and verification of concurrent and reactive systems.

[1] The specification language is TLA$^+$. It combines the temporal logic TLA with full first-order logic and ZF set theory. We abbreviate TLA$^+$ by TLA in the sequel.

TLA allows the specification of the correct behaviors of a system, the composition of sub-system specifications and refinement reasoning about a system specified at multiple levels of abstraction. A *behavior* models an execution of the system as an infinite sequence of states. A *state* is an assignment of values to state variables. An *action* formula expresses the relation between the value of variables in two successive states, using a prime notation as in Z [14] (e.g. if $x$ represents the value of a state variable in the current state of a *step*, $x'$ represents its value in the successor state). TLA specifications are usually written in the canonical form

$$Spec \triangleq Init \wedge \Box[Next]_x \wedge L \tag{1}$$

where *Init* is a state predicate that characterizes the system's initial states, *Next* is an action formula representing the next-state relation (typically written as a disjunction of possible moves), $x$ is the tuple of state variables of interest and $L$ is a formula that describes the liveness requirements. This specification represents all behaviors satisfying formula 1. Compared to other temporal logic, TLA differs in that the notion of action allows to specify both the system and its temporal properties within the same formalism. The verification of TLA specifications has been amply studied[2]. The book [9] introduces TLA from a user perspective and serves as a reference manual for several TLA Tools. In particular, it introduces TLC, a model checker tool for debugging a large class of TLA specifications [11].

**Approach.** An approach to implement TLA observers can be sketched as follows.

*Specification.* Let *Spec* be a specification of a system (*Spec* is formula 1) and *Obs* be a specification of the expected observable behaviors of this system. *Obs* represents correct and incorrect behaviors of the system.

$$Obs \triangleq InitObs \wedge \Box[NextObs]_x \tag{2}$$

We produce an *Oracle* specification (cf. formula 3) as a simple form of composite specification of the system *Spec* and its observer *Obs*. *Oracle* is a noninterleaving specification since the conjunction of the two initial state predicates and the two next state-relations represent the simultaneous advance of any observable behavior against a correct behavior of the system. It is a shared-state specification since all parts of the state attributed to the system component can be changed by the observer component (the state is not partitioned).

$$Oracle \triangleq (InitObs \wedge Init) \wedge \Box[(NextObs \wedge Next)]_x \tag{3}$$

*Runtime.* The definition of formula *Obs* requires specific TLA operators to be satisfied by a single (observed at runtime) behavior when evaluated by a checker. These operators are evaluated as follows. Instead of computing a value from a model of the system, the checker reads a value perceived by an actual observer of

[2] References to the literature can be found at http://lamport.org/.

an implementation of the system. That way, formula *Oracle* can be used to check the behaviors of an actual implementation of a system against its correct behaviors. Any observable behavior satisfying *Obs* but not *Spec* will not satisfy *Oracle*, and be reported as a deadlock, which is the expected behavior from an oracle.

**Specifying the system.** This process is illustrated with a simple mutual exclusion protocol, due to [7] but reformulated in TLA. The first step is as usual and consists of specifying and validating whether a behavioral model of a system satisfies some expected temporal properties. Several concurrent processes share a resource. The protocol must ensure a trivial safety property: only one process is allowed to be in critical section at any time. Each process undergoes transitions in the cycle $n \rightarrow t \rightarrow c \rightarrow n \rightarrow \ldots$ (where $n$ stands for non-critical section, $t$ for trying to enter critical section and $c$ for critical section), but processes interleave with each other as illustrated in figure 2 in the presence of two processes.

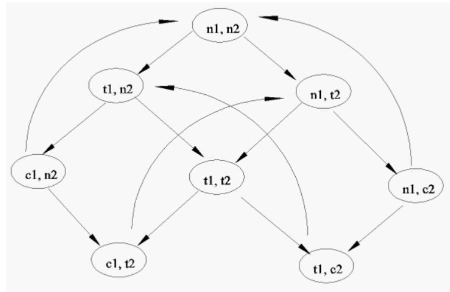



**Fig. 2.** Interleaving

The specification of this protocol appears in figure 3. It should be easy to understand, even without detailed knowledge of TLA. The constant parameter $N$ is the number of concurrent processes. The single state variable *prc* represents the concurrent processes. Formula *Spec* is similar to formula 1, ignoring liveness requirements. Its initial state predicate *Init* sets all processes in non-critical section. Its next-state relation *Next* is a disjunction of possible moves for any process $i$.

- $Try(i)$ represents process $i$ trying to enter critical section. It is enabled if process $i$ is in non-critical section.
- $Critic(i)$ represents process $i$ moving into critical section. It is enabled if process $i$ is trying to enter critical section and no other process $j$ is in critical section.
- $Free(i)$ represents process $i$ moving into non-critical section. It is enabled if process $i$ is in critical section.

To define a model, we must assign a value to the constant parameter $N$ (e.g. 1, 2, 3,...). The model checker tool TLC allows us to verify that this model satisfies the expected safety property by generating all possible behaviors (as illustrated by figure 2 in the presence of two processes).

MODULE *MutExclu*

CONSTANT $N$
VARIABLE $prc$
$Proc \triangleq 1 \,..\, N$

$Init \triangleq prc = [i \in Proc \mapsto \text{"n"}]$

$Try(i) \triangleq prc[i] = \text{"n"} \land prc' = [prc \text{ EXCEPT } ![i] = \text{"t"}]$

$Critic(i) \triangleq prc[i] = \text{"t"} \land \forall j \in Proc : prc[j] \neq \text{"c"} \land prc' = [prc \text{ EXCEPT } ![i] = \text{"c"}]$

$Free(i) \triangleq prc[i] = \text{"c"} \land prc' = [prc \text{ EXCEPT } ![i] = \text{"n"}]$

$Next \triangleq \exists i \in Proc : Try(i) \lor Critic(i) \lor Free(i)$

$Spec \triangleq Init \land \Box[Next]_{prc}$

**Fig. 3.** The mutual exclusion specification

**Observing the system.** We specify now in module *MutExcluOracle* (cf. figure 4) an oracle for the mutual exclusion protocol. *MutExcluOracle* first incorporates[3] the definitions from modules *MutExclu* (cf. figure 3) and *IO*. The TLA operator *ioStr* used in *MutExcluOracle* is defined in module *IO* as follows

$$ioStr \triangleq \text{CHOOSE } val : val \in \text{STRING} \tag{4}$$

This operator represents some arbitrary string and is evaluated by the (runtime) checker as the reading of an actual string value[4]. This value is obtained by an interaction between the observers and the checker, as depicted in figure 1.

MODULE *MutExcluOracle*

EXTENDS *MutExclu*, *IO*

$InitObs \triangleq prc = [i \in Proc \mapsto ioStr]$

$NextObs \triangleq prc' = [i \in Proc \mapsto ioStr]$

$Obs \triangleq InitObs \land \Box[NextObs]_{prc}$

$Oracle \triangleq (InitObs \land Init) \land \Box[NextObs \land Next]_{prc}$

**Fig. 4.** The observer specification

*Obs* and *Oracle* are formula 2 and 3 introduced above. According to the definition of the initial state predicate *InitObs* and next-state relation *NextObs* of formula

[3] In TLA, the EXTENDS statement leads to incorporate other TLA modules into a module. That is, to make visible their parameter and operator definitions

[4] This requirement is achieved by the model checker tool TLC that allows a TLA operator to be overridden by an executable Java method[9]. The generalization requires to define such specifics operators in module *IO* only for simple TLA values (boolean, integer. . . ) since other TLA values are construction of simple values.

*Obs*, and to the specific evaluation of the operator *ioStr* by the checker, each state of an observable behavior of the mutual exclusion protocol is an assignment of some (observed at runtime) string value to each process. This can be compared with the only correct behaviors of the protocol depicted in figure 2. The composite formula *Oracle* of the system and its observer allows to check such observable behaviors of the protocol against its correct behaviors[5]. Any behavior (perceived by an Observer of an actual implementation of the protocol) satisfying not *Spec* will not satisfy *Oracle*, and be reported as a deadlock by the (runtime) checker. The same approach holds with more sophisticated TLA specifications.

**Discussion.** This approach relies on TLA for the main part. It requires trivial TLA specifications (at least from the observer side) and an executable form of few simple TLA operators (to allow the interaction between a checker and an observer). We tried out this approach with the model checker tool TLC. In this way, TLC acts as a runtime checker. So, the same tool can be used to check behaviors from the specification to the implementation phases of a system. Among several drawbacks: this approach addresses only safety properties (whose violation can be illustrated on finite behaviors) and the provided oracle specification requires to describe the full state at each step. This latter point can be improved but at the price of more complex observer specifications.

## 4 Observing in a component framework

This section discusses some features of component-based system that make them adequate for both correctness specification and observation, as depicted in figure 1. We use the Fractal component model and framework as an illustration [2, 3].

**Observation in component models.** One of the most stringent challenge concerning a software system is probably the correctness assessment of its dynamic behaviors, i.e., the verification that its execution satisfies some formal specification. That imposes to be able i) to specify the correct behaviors of the target system as a composition of interacting sub-systems, ii) to "attach" a specification to each of these sub-systems by instrumenting the target system and iii) to actually observe the system as depicted in figure 1. We believe *structural features* of component models make them very suitable with respect to these points, much more than previous approaches such as object-orientation for instance. Interesting structural features in the Fractal component model are the following (cf. Figure 5)

[5] Note that formula *Obs* can also be used. However, this would be rather silly since this formula does not allow to check the observable behaviors against *Spec*, the correct behaviors of the mutual exclusion protocol. For example, even in presence of only one process (i.e. without considering concurrency) the behavior $n \rightarrow c \rightarrow t \rightarrow \ldots$ would be acknowledged by the checker. It should not since formula *Spec* of figure 3 specifies that each process undergoes transitions in the cycle $n \rightarrow t \rightarrow c \rightarrow n \rightarrow \ldots$.

- *Interfaces and bindings*: components can only interact with their environment through operations at identified access points called *interfaces* which can be *server (provided)* or *client (required)*. Some *bindings* must be established between components (more precisely between their interfaces) so that they can interact. Bindings are communication paths (externalized from components) that can be local/distributed, synchronous/asynchronous, secured...
- *Recursion*: the model is fully recursive. A component is composed out of two parts: *controller* and *content*. The content of a component is composed of other components, which are under the control of (the controller of) the enclosing component. The model allows components to appear at an arbitrary level. The recursion ends up with components with an empty content (these are called primitive components). Their implementation is provided in an underlying programming language, out of the scope of the component model .
- *Reflection and Control*: the model is fully reflexive. It allows a programmatic manipulation of software architectures. Fractal components are runtime entities manifest during system execution that exhibit introspection and intercession capabilities: they provide (meta)information and constructs which allow applications to dynamically access and *control* their structure and behavior.

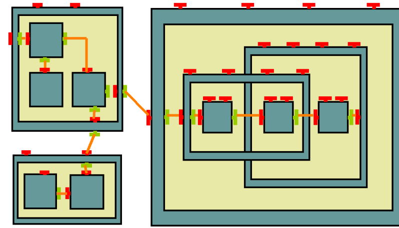


**Fig. 5.** A Fractal configuration of three components with their sub-components.

What make these features especially interesting with respect to behavioral observation is that observers can be localized anywhere, i.e., component behaviors can be observed at an arbitrary granularity by navigating in component structure.

**Instrumenting components in Fractal.** An essential concept in Fractal is that the *controller* of a component (its "membranes" in figure 5) is fully responsible of its content structure and behaviors. A *controller* is in fact an abstract entity actually implemented (in the reference implementation) by *interceptor* and *controller* objects organized in a graph of collaborating objects. The Fractal framework comes with a library of controller objects such as *binding controller*, *content controller*, *life-cycle controller*... which provide (structural) dynamic control over components. Much of the Fractal framework is devoted to controllers manipulation. Controllers are very good places where to introduce instrumentation for components behavior

observation (i.e., Observer, Checker, Behavior and Report from figure 1) in a non intrusive way . It is worth noticing that instrumentation can also be done in a more "applicative" way by building, introducing and binding explicitly an Observer component into the content of the target component. One way or the other, we believe component frameworks such as Fractal are very suitable platforms for component correctness observation as they offer infrastructures (concepts, languages, tools...) for the support of flexible and non intrusive instrumentation of target systems.

**Experiments.** We are currently conducting several experiments for validating the approach. The first experiment takes place in the implementation of the Fractal framework itself. Temporal logic is used to specify the correct behaviors associated with i) controller aspects composition and ii) shared component (as depicted in figure 5). Observation will be of great help in this context to check faulty behaviors. The second experiment takes place in the development of Jabyce [10], a software framework for compiled Java program (i.e. bytecode) transformations. Jabyce is implemented as a Fractal system (program transformations are Fractal components) and uses Design by Contract [12] to assert constraints (pre-post conditions and invariants) on valid program transformations. However, these constraints are not sufficient to ensure correctness in all cases for they cannot express constraints between several actions of a transformation. This again lets room for temporal logic specification and observation.

## 5 Conclusion

We believe component models provide structural concepts that i) help reasoning about individual components and component configurations and ii) allow instrumentation of observations to be localized almost anywhere. This in turn helps to check the adherence of components' implementations to their specifications. Of course, observation is not a panacea but we believe a benefit is to (partially) overcome the gap between formal specification and software implementation of a system, especially when considering temporal properties. To apply this idea, we have tried out an approach that enables TLC, a model checker tool for TLA, to be used as a runtime checker. So, the same tool is used to check behaviors from the specification to the implementation phases of a system. Our aim is to use this approach in a practical setting and we are currently conducting experiments in that respect with the component framework Fractal. This work is still in progress and needs further validation. Our main concern remains the *applicability* of the approach. Indeed, if we consider a complex distributed system, it is not quite reasonable to assume that every component would need a temporal logic specification (nor that programmers would enjoy this job!), nor that this would be without effect on the overall performance of the system. Therefore the goals we are pursuing with our experiments are: i) to assess the cost of the approach in terms of performance degradation, ii) to define in which situations it is worth using temporal logic and iii) to make sure that observation with temporal logic fits well with other mechanisms for correctness assessment when cohabiting in the same system.

## References

1. J. Abrial, The B-Book. Cambridge University Press, 1996.
2. E. Bruneton, T. Coupaye, J.B. Stefani, Recursive and Dynamic Software Composition with Sharing, Component-Oriented Programming Workshop at ECOOP02, 2002.
3. E. Bruneton, T. Coupaye, J.B. Stefani, Fractal Composition Framework Specification 1.0, ObjectWeb Consortium, http://www.objectweb.org/fractal, 2002.
4. E. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.
5. M. Diaz, G. Juanole, J.P. Courtiat, Observer-A Concept for Formal On-Line Validation of Distributed Systems, IEEE Trans. Software Engineering, vol 20, pp. 900-913, 1994.
6. L. Dillon, Y. Ramakrishna, Generating oracles from your favorite temporal logic specification, 4th ACM SIGSOFT Symp. Foundation of Software Engineering, 1996.
7. M. Huth, M. Ryan, Logic in Computer Science, Modelling and reasoning about systems, Cambridge University Press, 2000.
8. L. Lamport, The temporal logic of actions. ACM Trans. Programming Languages and Systems, 16(3): 872923, 1994.
9. L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison Wesley, 2002.
10. R. Lenglet, T. Coupaye, E. Bruneton, Composing Transformations of Compiled Java Programs with Jabyce, Submitted to publication, 2003.
11. P. Manolios, L. Lamport, Y. Yu, Model Checking TLA+ Specifications, Correct Hardware Design and Verification Methods, CHARME'99, 1999.
12. B. Meyer, Object-Oriented Software Construction, Prentice-Hall, second edition, 1997.
13. A. K. Mok, G. Liu, Efficient Run-Time Monitoring of Timing Constraints, Proc. 3rd Real Time Technology and Applications Symposium, Montreal, Canada, 1997.
14. B. Potter, J. Sinclair, D. Till, An Introduction to formal specification and Z, Prentice Hall Intrenational, Series in Computer science, 1991.
15. D. Richardson, S. Aha, T. Malley, Specification-based test oracles for reactive systems, 14th Inter. Conf. Software Engineering, pp. 105-118, 1992.
16. M. Rodriguez, J. C. Fabre, J. Arlat, Wrapping Real-Time Systems from Temporal Logic Specifications, Dependable Computing EDCC-4, LNCS 2485, France, 2002.

# Validation of Business Process Models

Andreas Speck[1], Elke Pulvermüller[2], and Dirk Heuzeroth[2]

[1] Intershop Research Jena
D-07740 Jena, Germany
andreas.speck@intershop.com
[2] Institut für Programmstrukturen und Datenorganisation,
Universitaet Karlsruhe,
D-76128 Karlsruhe, Germany
pulvermueller@acm.org, heuzer@ipd.info.uni-karlsruhe.de

**Abstract.** The eCommerce system development of Intershop is based on different models on various levels of abstraction. The software engineering tool ARIS represents most of these models. In this paper we focus on the validation of the business process models on an intermediate abstraction level of the ARIS model. The business processes may be derived from process patterns and have to follow specific rules (best practices). The validation of the compliance with these rules and the consistency with the original business process pattern is the focus of this paper.

## 1 Introduction

Workflows are a common technology to define the dynamic behavior of software like business systems. An almost classical application for workflow models is the application logic in eCommerce applications. These specific business-oriented workflows are business process models. The steps of the processes provided by an eCommerce application could ideally be modeled as business processes.

In order to support the workflow-driven system development specific tools have been invented. One of these tools is *ARIS* which has been developed at IDS Scheer Saarbrücken [14], [13].

*ARIS* distinguishes between four different levels of abstraction in software specification. It is used as a tool for specifying the customers‘ requirements and for modeling the business processes of the applications. However, *ARIS* doesn‘t provide means to validate the models which is the focus of our work. It is very desirable to identify errors and problems in an early state of the system modeling.

The *ARIS* models are introduced in section 2. The problems to be checked are discussed in the following subsection. In section 3 we elaborate the problem of validating business process models which is one of the issues for validation. Others like the consistency check of the static relationships of the functions in the level two is not taken into consideration in this papers since it has already been discussed in previous publications like in [16] or [12].

## 2 ARIS Models

The concept of *ARIS* is to support the different levels of abstraction. The generic *ARIS House of Business Engineering* architecture (HOBE) is the base for an eCommerce specific adaptation of *ARIS* supporting the Intershop eCommerce development – *ARIS for Enfinity* (cf. figure 1). This eCommerce version of *ARIS* applies the three upper levels of HOBE without modifications. Only the lowest and most concrete layer has been adapted to the Intershop-specific eCommerce development. The detailed workflows on the lowest level are implemented as Intershop *Pipelines* consisting of different types of connectors and *Pipelets* which provide small, reusable elements of the application business logic. These *Pipelines* are manipulated by the *Visual Pipeline Manager* (VPM) which has been integrated in *ARIS for Enfinity*.

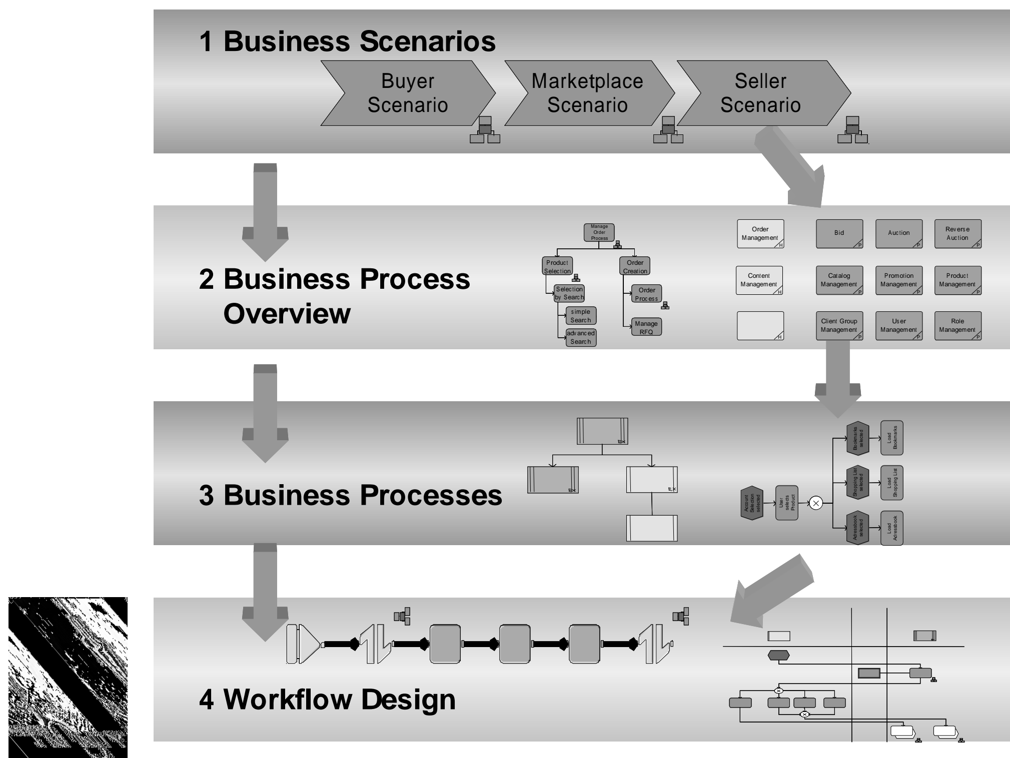



**Fig. 1.** *ARIS for Enfinity* eCommerce Model [1].

The different levels of *ARIS* (or *ARIS for Enfinity*, respectively) are:

1. *Business Scenarios*
   In the layer with the highest degree of abstraction the basic functionality of the system is described as scenarios. The set of scenarios provides an overview over the system to be developed.
2. *Business Process Overview*
   In the second layer the basic functions are divided into more detailed parts. This detailed functionality is arranged hierarchically in a tree (actually network) structure. Each of these detailed function blocks represents one ore more workflows (business processes) which are defined in the third level.
3. *Business Processes*
   The business process level presents the basic workflows derived from the function blocks and specifies their behavior. The notation used for these comparatively abstract workflows is the EPK (the abbreviation stands for the German word for *Event-driven Process Chain*, namely *Ereignisgesteuerte Prozeßkette*). This notation has been developed at the University of Saarbrücken in cooperation with SAP and IDS Scheer [8].
4. *Workflow Design*
   In this layer the business processes are more detailed and the different activities of the processes are attached to specific actors. The actors are arranged in swim lanes like the actors in the UML activity diagrams.
   Each of the functions in the EPKs are specified in detail with one or more *Pipelines*. In some cases a *Pipeline* may also cover two or more functions. *Pipelines* consist of *Pipelets* which represent small reusable code elements.

### 2.1 Issues of Validation

On each level of the *ARIS* model there are specific issues to be validated. Moreover, the overall consistency between the different layers needs to be checked. In detail this means:

– *Business Scenarios*
  Here it has to be assured that a sufficient number of reusable code elements exists for each scenario offered. Since a customer should have a real choice between different alternatives it requires more code than just to realize one scenario.
– *Business Process Overview*
  Although the functions are arranged in a hierarchical tree order they are actually a complex network with different "cross-tree" constraints. It has to be assured that all these constraints are met, that mutually required functions are part of the configuration and that mutual exclusive functions are not both in the system. The consistency of the static relations between the functions has to be assured.
– *Business Processes*
  The business processes are based on basic patterns describing the standard processes and rules (best practices) defining the knowledge of what is a correct system. Errors resulting in a wrong customization have to be addressed.
– *Workflow Design*
  Since one single *Pipeline* does not necessarily represent one function of the detailed

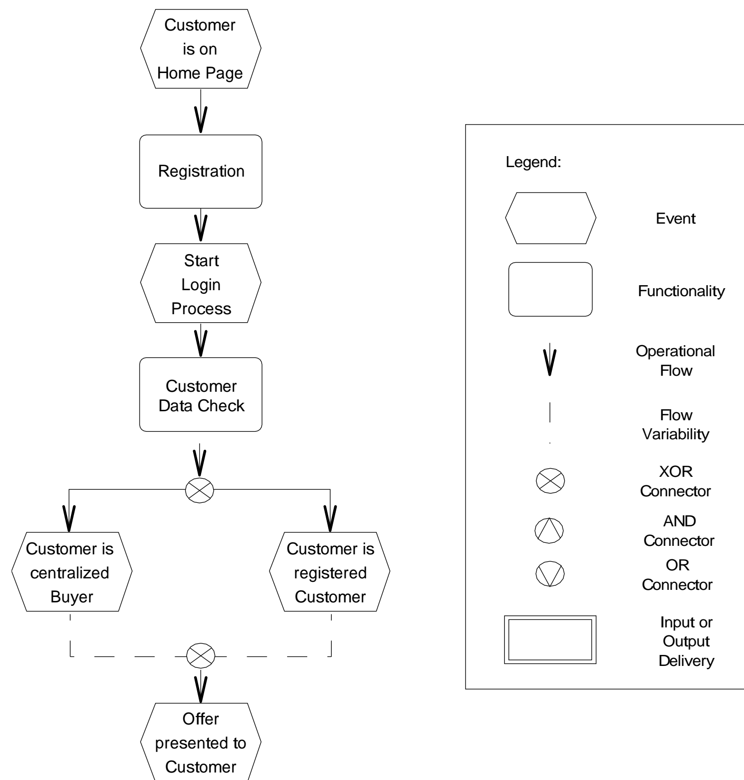



**Fig. 2.** EPK Example of an eCommerce Login Process Pattern

EPK the consistency between the workflow of the EPKs and the *Pipelines* needs to be assured.
Additionally there is no mechanism preventing the modification of the *Pipelets* code. This may result in the problem that services and data required by *Pipelets* may not be provided by the preceding *Pipelets* in the workflow.

Finally the different layers have to be kept consistent between each other which means that a pattern or structure in a more abstract level has to be found in the lower layer. Therefore explicit rules for a transition from models in one layer to the other have to be given.

## 3 Validation of Business Processes

Intershop has presented a new strategy for implementing eCommerce systems: *Unified Commerce Management* (UCM). One aspect of UCM is the development of a set of business process patterns for different typical requirements of customers. These busi-

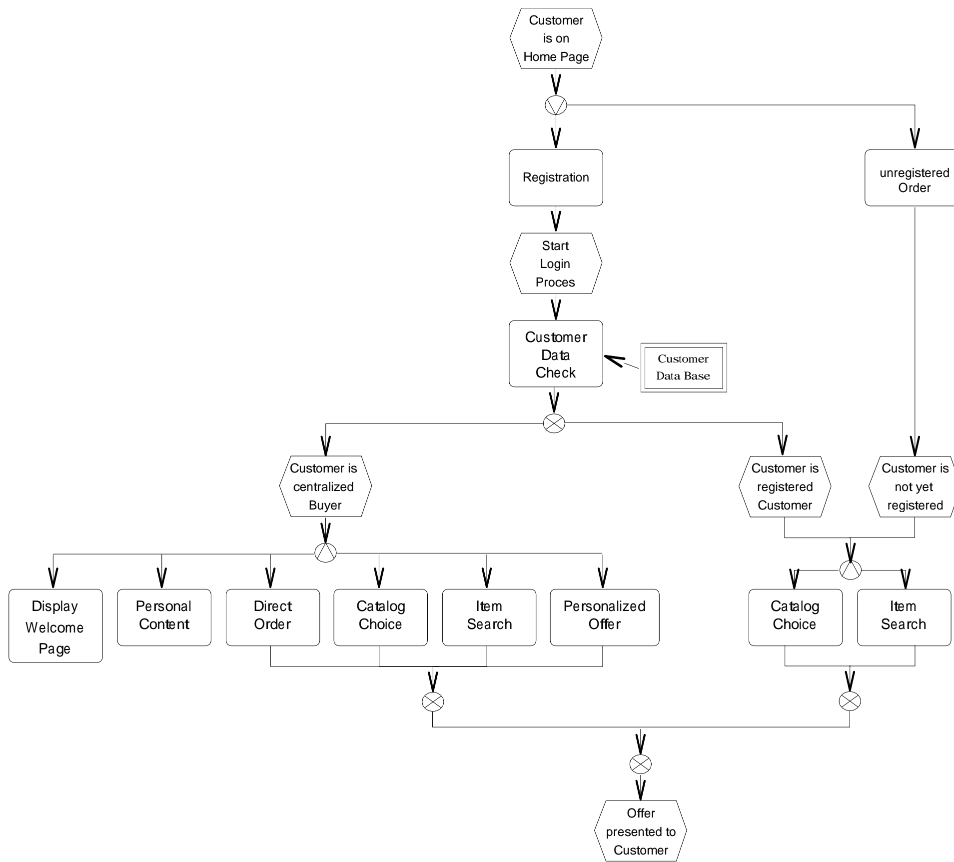



**Fig. 3.** EPK Example of an eCommerce Login Process

ness process patterns may then be customized to specific systems. They may be seen as a similar concept as architectural or analysis patterns in software engineering.

In a first step we focus on the level of business processes and the adaptation of business process patterns. A concrete application scenario may be that a consultant creates an eCommerce system. A set of business process patterns for characteristic activities and rules for specific details (in UCM terminology this is called *Best Practices*) are supporting this activity.

### 3.1 Example of Business Process Adaptation

A pattern for an eCommerce process may look like the login pattern depicted in figure 2.

The pattern starts with the event *Customer is on Home Page* followed by the function *Registration* which leads to the event *Start Login Process*. The following function named *Customer Data Check* may either lead to the event that the *Customer is cen-*

*tralized Buyer* or that the *Customer is registered Customer*. Both events will be mutual exclusive which means that a customer may belong exclusively only to one of both groups [1]. After both events (*Customer is centralized Buyer* and *Customer is registered Customer*) an undefined number of parallel functions must follow (indicated by the dashed line) which are concluded by the last event *Offer presented to Customer*.

Figure 3 presents a real business process which could be derived from the process pattern in figure 2. In this example a large number of functions have been added to the pattern, i.e. *Display Welcome Page*, *Personal Content*, *Direct Order*, *Catalog Choice*, *Item Search* and *Personalized Offer*. Moreover, the process has a new path for an *unregistered Order*. This path is for customers which are not yet registered and who, however, should also get a chance to use the web shop.

### 3.2 Issues to be Checked

The example presented in the previous section 3.1 is a comparatively small process. However, it demonstrates some of the problems that may occur when a pattern has to be customized. New functions and events have to be added at the right position. New paths have to be added without violating rules and experiences. Since the usual size of the Intershop eCommerce business processes is much larger than our simple example, the business process diagrams may become difficult to examine.

Rules to be checked may help to validate the dynamic behavior of the system. We apply model checkers such as SMV (*Symbolic Model Validation* [11]) in order to validate the business processes. These checking tools usually need a model of the issue to be validated (in our case the business processes) described as finite automata. The specification which the model is checked against is formulated in temporal logic such as CTL (*Computational Tree Logic*) or CTL* (an extension of the *Computational Tree Logic*).

Examples for concrete rules to be checked my look like:

- the customer may alway have a *Choice*:
  in CTL*: `AF Choice_F` (`_F` indicates that *Choice* is a function)
- there is at least one path to *Search*:
  in CTL*: `EF Search_F`
- a *centralized Buyer* will always get a *Personal Content* and *Personalized Offer*:
  in CTL*: `AG (centralized_Buyer_E -> AF (personal_Content_F & personalized_Offer_F))`
  `_E` represents an event and the arrow (`->`) symbolizes the logical implication operator.

The procedure of business process validation is depicted in figure 4. Here the business process is transformed into a finite automata. The specification is expressed in CTL* (extended version of the first specification example above). Both are given to the model checker (actually they are copied into one file which is then processed by SMV).

[1] The logical $AND$, $OR$ and $XOR$ operators are used for branches in the workflow (c.f. figure 2).

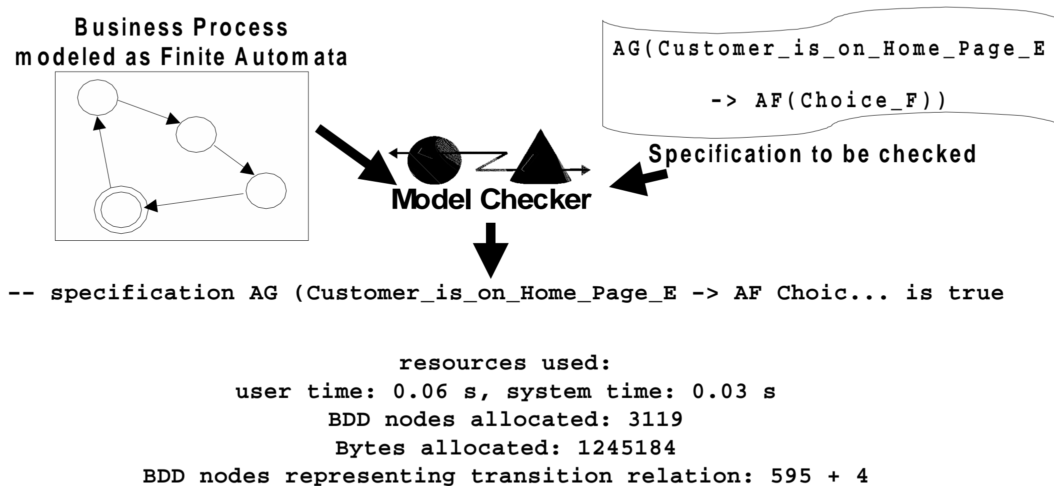



**Fig. 4.** Model Checking of Business Processes

The result produced by the model checker is then presented at the bottom of the picture: the positive validation that the customer will have a *Choice* in the given business process model.

For this problem model checking seems to be an ideal technique since we have only a very limited number of states to be considered. Therefore, the probability of running into the state explosion problem is quite low in contrast to other checking issues like code.

## 4 Related Work

The application of model checking in hardware-related domains is wide-spread and has already industrial relevance since several years [3]. The usage of this formal method in the domain of software products, however, is still in its very beginnings (the first steps may be found in [10]). This is due to the state explosion problem as well as the model construction problem. Both are even more difficult to deal with for software as compared to hardware systems [5]. In [5] first approaches to apply model checking for software systems are distinguished into monolithic approaches and translation approaches [6]. Monolithic approaches provide dedicated model checkers for a specific programming language. Translation approaches translate programs directly into a relatively expressive verifier input language like *Promela* reducing the semantic gap between software and model.

*Bandera*, an integrated collection of program analysis and transformation components [5], allows to validate Java code to a certain extend (e.g. limited lines of code and limited set of Java methods to be verified). Besides SMV, we applied *Bandera* for the validation of the Java code against the *Pipelines* and *Pipelets* on the lowest level of the

*ARIS* model. In contrast to *Bandera* we focus on the business process models in this paper. These are much more abstract than the Java source code.

The transformation from state charts to the model checking language SMV is systematically investigated in [2]. Despite the fact that state charts are usually quite close to the code, this approach deals with similar problems (building finite automata from dynamic models).

Assuming the system is developed starting with manually produced state charts (as realized in [4], for instance) it is possible to translate programs and requirements applying this technique. However, the consistency problem arises. If the model is produced independently to the implementation these different views on the same system may vary. In [4] model checking is employed to realize the composition of layers. Therefore, similar to our approach the composition of system units is supported. Each layer in [4] realizes a feature which may lead to a feature / layer explosion problem in larger systems.

The validation of the behavior of components is also related to this work since it meets the problems in the lower abstraction levels. In [17] an approach called PACC is presented. It allows component certification. The approach considers to enforce predefined and designed interaction patterns and is therefore based on comparable software analysis and documentation techniques. The focus in their work is on certification and documentation. Another approach to model and validate the dynamic activities of components may be found in [15]. In this approach model checking is explicitly applied in order to validate the behavior of the components and composites.

Some approaches deal with the validation of workflows in general by applying model checkers. [9] is a quite formal approach focusing on finite automata models. However, it does not provide a mapping between business process models such as EPKs and finite automata. Another example for a workflow checking approach is [7]. Here web service workflows are to be validated.

## 5 Conclusion

The paper proposes an approach to validate business processes against rules and patterns. This validation is done by applying model checkers like SMV. In contrast to other applications of model checking the validation of business processes doesn't bear a high risk of getting too many states.

From the viewpoint of the *ARIS* system development model the paper presents only a limited solution for the validation of business processes on level three. However, further validation techniques have to be found to compare the models and the code and to assure their consistency. Our current steps towards this issue are first investigations of the comparison of code against the most concrete models (*Pipelines* and *Pipelets*), the consistency within *Pipelines* and the development of an eCommerce-specific specification language with allows to express the standard requests in the eCommerce domain. The latter work implies that the temporal specification languages CTL, CTL* as well as LTL have to be considered and a mapping from these languages to the standard classes of problems have to be built.

## References


1. M. Breitling. Business Consulting, Service Packages & Benefits. Technical report, Intershop Customer Services, Jena, 2002.
2. E.M. Clarke and W. Heinle. Modular translation of statecharts to smv. Technical Report CMU-CS-00-XXX, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1998.
3. E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, pages 61 – 67, June 1996.
4. K. Fisler, S. Krishnamurthi, D. Batory, and J. Liu. A Model Checking Framework for Layered Command and Control Software. In *Proceedings of the Workshop on Engineering Automation for Software Intensive System Integration*, June 2001.
5. J. Hatcliff, C. Pasareanu, R. S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
6. K. Havelund and J.E. Skakkebk. Applying Model Checking in JAVA Verification. In *Proceedings of the 6th SPIN Workshop '99: Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*. Springer, September 1999.
7. C. Karamanolis, Giannakopoulou D., J. Magee, and S. Wheater. Model Checking of Workflow Schemas. In *In Proc. of the 4th International Enterprise Distributed Object Computing Conference (EDOC'00)*. IEEE Computer Society, 2000.
8. G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozeßmodellierung. Technical Report Nr. 89, Veröffentlichungen des Instituts für Wirtschaftsinformatik, Saarbrücken, 1992.
9. J. Koehler, G. Tirenni, and S. Kumaran. From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods. In *In Proc. of the Sixth International Enterprise Distributed Object Computing Conference (EDOC'02)*, pages 96 – 106. IEEE Computer Society, 2002.
10. K. Laster and O. Grumberg. Modular Model Checking of Software. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, March-April 1998.
11. K. McMillan. Symbolic Model Checking. PhD Thesis, Carnegie Mellon University, 1992.
12. E. Pulvermüller, A. Speck, and J. O. Coplien. A Version Model for Aspect Dependencies. In *Proceedings of 2nd International Symposium of Generative and Component-based Software Engineering (GCSE 2001)*, LNCS, Erfurt, Germany, September 2001. Springer.
13. A.-W. Scheer. *ARIS - Modellierungsmethoden, Metamodelle, Awendungen.* Springer, Berlin, 1998.
14. A.-W. Scheer. *ARIS - Vom Geschäftsprozeß zum Awendungssystem.* Springer, Berlin, 1998.
15. A. Speck, E. Pulvermüller, M. Jerger, and B. Franczyk. Component Composition Validation. *International Journal of Applied Mathematics and Computer Science*, 12(4):581–589, January 2003.
16. A. Speck, S. Robak, E. Pulvermüller, and M. Clauß. Version-based Approach for Modeling Software Systems. In *Proceedings of Model-based Software Reuse, ECOOP 2002 Workshop*, pages 15 – 22, Malaga, Spain, 2002.
17. J. Stafford and K. Wallnau. Predicting Assembly from Certifiable Components. In E. Pulvermüller, A. Speck, J.O. Coplien, M. D'Hondt, and W. DeMeuter, editors, *Proceedings of the Workshop on Feature Interaction in Composed Systems, ECOOP 2001, Technical Report No. 2001-14, ISSN 1432-7864*. Universitaet Karlsruhe, June/September 2001.