VRIJE UNIVERSITEIT BRUSSEL – FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
SYSTEM AND SOFTWARE ENGINEERING LAB

COMBINING ASPECT-ORIENTED AND COMPONENT-BASED SOFTWARE ENGINEERING

Wim Vanderperren

May 2004

Advisor:

Prof. Dr. Viviane Jonckers

A dissertation in partial fulfillment of the requirements of the degree of Doctor in Science

VRIJE UNIVERSITEIT BRUSSEL – FACULTEIT WETENSCHAPPEN
VAKGROEP INFORMATICA
LABORATORIUM VOOR SYSTEEM EN SOFTWARE ENGINEERING

COMBINEREN VAN ASPECT GEORIENTEERDE EN COMPONENT GEBASEERDE SOFTWARE ONTWIKKELING

Wim Vanderperren

mei 2004

Promotor:
Prof. Dr. Viviane Jonckers

Proefschrift voorgelegd tot het behalen van de wetenschappelijke graad van doctor in de wetenschappen

Abstract

The goal of this dissertation is to combine aspect-oriented and component-based software engineering and as such achieve the advantages of both aspect-oriented and component-based ideas. This research is conducted in the context of the PacoSuite component-based methodology. PacoSuite introduces explicit and reusable composition patterns that are able to describe the collaboration of a set of components in an abstract way. In addition, it is possible to automatically verify whether a composition pattern and set of components are compatible. We present the composition adapter model in order to modularize crosscutting concerns in PacoSuite. A composition adapter describes protocol transformations and can be applied onto a given composition pattern. We introduce an algorithm that allows to automatically match a composition adapter with a composition pattern. In addition, the adaptations described by a composition adapter can be automatically inserted into the composition pattern. As such, composition adapters allow to effectively modularize crosscutting concerns without lowering the abstraction level of component composition in PacoSuite.

Composition adapters are however limited to describing protocol adaptations; concerns that require invasive changes to the components cannot be captured in a regular composition adapter. This limitation disappears when a composition adapter is implemented using an aspect-oriented programming language. Therefore, we present a new aspect-oriented programming language aimed at component-based software engineering and PacoSuite in particular, named JAsCo. JAsCo allows specifying highly reusable aspect beans that are deployed onto a concrete context using connectors. In addition, connectors allow to explicitly describe precedence and combination strategies that are able to control the collaboration among several cooperating aspect beans. The JAsCo technology introduces a new component model based on built-in traps that allow aspect interference. As such, aspect beans are first-class entities at run-time and can be added and removed dynamically. Furthermore, an Adaptive Programming extension to the JAsCo language is proposed that allows an aspect bean to be instantiated as an adaptive visitor in a new kind of connector, called a traversal connector. The JAsCo contributions concerning explicit precedence and combination strategies also apply for traversal connectors.

We present the invasive composition adapter model in order to integrate JAsCo into PacoSuite. The invasive composition adapter documentation describes the protocol transformations while the aspect bean implementation in JAsCo is able to alter the internal behavior of the components. An invasive composition adapter can still be automatically applied onto a given component context. As such, an invasive composition adapter is able to realize all the benefits of a regular composition adapter with the added expressiveness of the JAsCo implementation.

In order to validate the approach presented in this dissertation, an extension of the PacoSuite environment is implemented. In addition, the necessary tools for compiling JAsCo entities as well as a JAsCo run-time infrastructure are provided. Using this tool support, an extended case study is performed. We identify several crosscutting and tangled concerns in this application and successfully modularize these concerns as (invasive) composition adapters.

Samenvatting

Het doel van deze thesis is om aspect georiënteerde en component gebaseerde paradigma's te combineren om zo de voordelen van beide te verwerven en dit in de context van de PacoSuite component gebaseerde methodologie. Eén van de belangrijkste bijdragen van PacoSuite is de introductie van expliciete en herbruikbare compositie patronen die de interactie tussen verschillende componenten op een abstracte manier beschrijven. Bovendien laat PacoSuite toe om automatisch de compatibiliteit van een compositie patroon en een verzameling componenten te verifiëren. We introduceren het compositie adapter model om "crosscutting" eigenschappen te modularizeren in de PacoSuite context. Een compositie adapter beschrijft protocol transformaties en kan op een compositie patroon toegepast worden. We ontwikkelen een algoritme om te verifiëren of een compositie adapter toepasbaar is op een gegeven compositie patroon. Bovendien is het mogelijk om de protocol transformaties beschreven door een compositie adapter automatisch in een compositie patroon te weven. Op deze manier laat een compositie adapter toe om "crosscutting" eigenschappen te vatten zonder aan het hoge abstractieniveau van de PacoSuite te raken.

Compositie adapters zijn echter beperkt in hun expressiviteit, ze kunnen enkel protocol transformaties beschrijven. Aspecten die aanpassingen aan de componenten zelf nodig hebben, kunnen niet in een compositie adapter gevat worden. Deze beperking verdwijnt wanneer we een compositie adapter van een implementatie in een aspect georiënteerde taal voorzien. Daarom ontwikkelen we een nieuwe aspect georiënteerde taal toegespitst op component gebaseerde ontwikkeling, genaamd JAsCo. JAsCo introduceert zeer herbruikbare "aspect beans" die onafhankelijk van een bepaalde context gespecificeerd worden. Expressieve connectoren passen deze aspect beans toe op concrete snijpunten en kunnen aspect composities expliciet vatten door middel van combinatie strategieën. De JAsCo technologie stelt een nieuw component model voor waar "traps" die aspect applicatie toelaten reeds ingebakken zitten. Aspecten blijven eerste klasse entiteiten tijdens de programma uitvoering met als gevolg dat aspecten dynamisch kunnen toegevoegd en verwijderd worden. Bovendien stellen we een uitbreiding van JAsCo voor die "Adaptive Programming" ondersteunt op een zodanige manier dat aspect beans zowel traditionele aspecten als "traversal" gerelateerde aspecten kunnen beschrijven, wat de bruikbaarheid van ons aspect model uiteraard ten goede komt. De bijdragen van de JAsCo taal betreffende het expliciete compositie mechanisme zijn ook voor de Adaptive Programming extensie van toepassing.

Om de JAsCo taal te integreren in PacoSuite, stellen we het invasief compositie adapter model voor. Een invasieve compositie adapter bestaat uit een aangepaste compositie adapter, die de protocol transformaties beschrijft, en een implementatie door middel van een JAsCo aspect bean, die toelaat om de componenten zelf te wijzigen. Een invasieve compositie adapter kan nog steeds automatisch op een gegeven component compositie toegepast worden. Dit nieuwe model realiseert dus alle voordelen van het gewone compositie model met de toegevoegde expressiviteit van de JAsCo aspect bean implementatie.

Om de benadering voorgesteld in deze verhandeling te valideren wordt de PacoSuite visuele component compositie omgeving uitgebreid om zowel gewone als invasieve compositie adapters te ondersteunen. Bovendien hebben we de noodzakelijke programma's ontwikkeld om de JAsCo taal te realiseren. Door middel van deze programma ondersteuning wordt een uitgebreide evaluatie van de ideeën, voorgesteld in deze verhandeling, mogelijk. In een omvangrijke gevalstudie worden verschillende "crosscutting" eigenschappen met succes gemodulariseerd door zowel gewone als invasieve compositie adapters.

Acknowledgements

The research presented in this dissertation is funded by a doctoral scholarship of the Flemish Fund for Scientific Research (FWO or in Flemish: “Fonds voor Wetenschappelijk Onderzoek”).

First of all, I would like to thank my advisor, Viviane Jonckers, for her guidance during the last four years, for providing valuable feedback about my research ideas and for proof reading my papers and thesis. Viviane has played a very important role in the achievement of this PhD. I also want to thank her for helping me finding funding for a post-doctoral position.

I owe my gratitude to my PhD committee members, for taking the time to read my dissertation in detail and for providing invaluable feedback about my dissertation. Apart from my advisor Viviane, the PhD committee members are: Theo D’Hondt, Karl J. Lieberherr, Mario Südholt and Dirk Vermeir.

I am also greatly indebted to Davy Suvee. During his Master’s Thesis, Davy developed an early prototype version of JAsCo. Afterwards, he helped shaping the ideas and concepts of this dissertation. Davy is also responsible for the JAsCo Eclipse plugin, a .NET implementation of JAsCo, most of the JAsCo benchmarks in the tool support chapter and so much more. We collaborated quite closely the last two years and wrote many papers together. Our discussions were ever enlightening and inspiring and definitely contributed to the results of this work.

Another colleague I am very grateful to is Dr. Bart Wydaeghe, who developed the PacoSuite methodology during his PhD research. Bart and I collaborated very closely in the first years of my PhD research and he contributed greatly to starting up my own research. Furthermore, I would like to thank Dr. Kurt Verschaeve, who also guided me through the first year of my PhD research. Without Bart and Kurt, I would certainly not have pursued a PhD.

I should not forget to thank all my colleagues at the System and Software Engineering Lab for the fruitful discussions and for providing a great and motivating working environment. It has been a great pleasure to work with them over the years: Maja D’Hondt, Dennis Wagelaar, Miro Casanova Paez, Bart Verheecke, Maria Agustina Cibran and Ragnhild Van Der Straeten. Special thanks to Ragnhild Van Der Straeten, who helped me with the formal parts of this dissertation, and to Dennis Wagelaar, who proofread some chapters of my thesis. I also owe special gratitude to Bart Verheecke and Maria Agustina Cibran for applying the ideas of this thesis and thereby providing me with invaluable feedback.

Furthermore, I would like to thank all my thesis students for applying and evaluating the ideas in this dissertation: Geert Lambrechts, Pieter Schollaert, Dimitri Verspecht, Niels Jonckheere, Jan Baudisch, Frederik Vannieuwenhuyse and Joris Elsocht.

Thanks to Karl Tuyls for the initial design of the cover page.

Last but not least, I would like to thank my girlfriend Inge Nakielski for her moral support during this research and especially during the writing of my thesis, when I had little time for her. I also owe Inge my gratitude for proof reading this dissertation and for drawing several figures. There’s no doubt that without Inge, my PhD would not nearly be finished. I also would like to thank our yet unborn twins for the joy that they have already brought upon us.

Table of contents

List of Figures

List of Code Fragments

List of Tables

List of Definitions

List of Algorithms

Chapter 1 Introduction

This chapter states the problem that this dissertation aims to tackle and elucidates the goal of the dissertation. The context in which this research is conducted is shortly explained, the approach is introduced and the main contributions are announced. Finally, an outline of the dissertation is presented.

1.1 Problem Statement

The so-called software crisis endures already for more than thirty years [Dij72]. It has been claimed that Object-Oriented Software Development (OOSD) offers the solution for the software crisis [CN86]. Although object-oriented software development is already an important contribution to solving this software crisis, it cannot live up to its promise. Object-oriented software development suffers from a poor separation of concerns and classes are not as reusable as originally assumed. Component-Based Software Engineering (CBSE) and Aspect-oriented Software Development (AOSD) are two novel software engineering paradigms that aim to solve some of the deficits of object oriented software development.

Component-based software engineering aims at improving the reusability of software artifacts. In objected-oriented software development, reuse is typically achieved by inheritance. Inheritance however introduces a very strong coupling between a base class and its derived classes, causing the notorious *fragile base class problem* [MS98, Szy96]. Base classes are considered fragile because a base class can be modified in a seemingly safe way, but this new behavior, when inherited by the derived classes, might cause the derived classes to malfunction. As a result, a simple change to a key base class can render an entire program inoperable [Hol03]. Evolving a base class means re-testing and possibly reviewing all derived classes. To solve this problem, component-based software engineering provides another solution for reuse that aims at minimal coupling between software artifacts and maximum cohesion of single artifacts [Mey99]. When applying component-based software engineering, a full-fledged software-system is developed by assembling a set of pre-manufactured components. Each component is a black-box entity, which can be deployed independently and is able to deliver specific services [Szy98]. Ideally, a component-based system is composed by using a graphical composition environment that allows visual plug-and-play component composition [Sho97]. The component-based paradigm drastically improves the speed of development and the quality of the produced software [HC01]. Component-based software engineering is already mature and several commercial component-based technologies are available. As a result, component-based software engineering is applied quite a lot in practice and has become one of the standard development paradigms for large-scale applications.

Aspect-oriented software development [KLM+97] is a more recent software engineering paradigm that aims at improving the separation of concerns. Separation of concerns [Par72] is a crucial property for realizing comprehensible and maintainable software. Object-oriented technologies fail to modularize certain concerns because object-orientation only allows a single dimension of decomposition, namely the class hierarchy. This problem has been identified as the tyranny of the dominant decomposition [TOHS99]. As a consequence, some concerns are spread over and repeated in several modules in the system. Due to this code duplication, it becomes very hard to add, edit or remove such a concern in the system. These concerns are called crosscutting because the concern virtually crosscuts the decomposition of the system. Typical examples of crosscutting concerns are debugging concerns such as logging [KLM+97] and contract verification [VSJ03c], security concerns [DVD02] such as confidentiality and access control and business rules [CDJ03] that describe business-specific logic. Aspect-oriented software development solves the problem of crosscutting concerns by introducing one or more additional separation dimensions that allow modularizing crosscutting concerns. As such, the concern is no longer spread over different modules and is thus no longer crosscutting. AspectJ [KHH+00], HyperJ [OT00], Adaptive Programming [LOO01] and Composition Filters [BAT01] are four aspect-oriented approaches that have been introduced in recent years. AspectJ is undoubtly one of the most well-known aspect-oriented approaches and claims to be the first production-quality aspect-oriented technology. AspectJ is an

extension of the Java programming language and introduces an extra concept, called an aspect, to capture crosscutting concerns. An aspect is able to describe the set of points in the base program where the aspect is applicable (called *pointcut*) and the concrete behavior that has to be executed at those points (called *advice*). Aspect weaving consists of merging the aspects with the base implementation of the system. Aspect-oriented software development is a rather new paradigm in comparison to component-based software engineering and is already quite well accepted in the academic world. Although some first experiments with aspect-orientation in the industry have been conducted, the aspect-oriented software development process is not yet mature enough to be fully accepted by the industry.

Current aspect-oriented research and practice is mainly focused on the object-oriented paradigm. Component-based software engineering however, also suffers from the problems that arise with the tyranny of the dominant decomposition. This is because component-based software engineering advocates very low coupling between components and high cohesion of single components to make components independently deployable [Szy98]. In fact, in CBSE, a component should never explicitly rely on concrete other components in order to increase reusability. As a consequence, component-based software engineering suffers greatly from crosscutting concerns and tangled code because a lot of functionalities are spread and repeated over different components in order to keep the coupling as low as possible [Van02a]. As such, integrating the aspect-oriented approach into the component-based world, significantly contributes to the component-based paradigm because it enables to increase the modularity of component-based applications.

Component-based ideas also contribute to the aspect-oriented field. Component-based software engineering puts a lot of stress on the reusability of components. Aspects in current aspect-oriented approaches are often tightly coupled to the base application and are thus not reusable. For example in AspectJ, the concrete context where the aspect is applicable is hard-coded in the aspect. As such, reusing an aspect requires altering the aspect. Therefore, improving the reusability of an aspect is an important contribution to aspect-oriented software development [LLM99, SVJ03]. In addition, component-based software engineering explicitly separates the composition mechanism from the components. As such, managing a component composition is quite straightforward. In current aspect-oriented technologies, the composition mechanism is often tangled with the aspect implementation. As such, managing a composition of several aspects is not straightforward at all and can lead to unpredictable errors and conflicts. This problem has been identified as the *feature interaction problem* [PSC+01]. Furthermore, in component-based software engineering, it is common practice to allow run-time component configuration management. This means that individual components as well as complete compositions can be altered or replaced at run-time. Brichau et al. [BDD00] identify that this dynamism is often required for aspects as well. As a result, introducing a similar plug-and-play concept and explicit composition mechanism in aspect-oriented software development, would make aspects reusable and their deployment easy, manageable and flexible.

1.2 Goal, Context and Approach

The goal of this dissertation is to integrate aspect-oriented and component-based software engineering and as such combine the advantages of both aspect-oriented and component-based ideas. We aim to integrate aspect-oriented ideas both at a visual component-based composition level and at the implementation level. In addition, a seamless, preferably automatic, transition from the design level to the implementation level has to be possible.

The context in which this research is conducted is the PacoSuite component-based methodology proposed in Bart Wydaeghe's PhD dissertation [Wyd01,WV01,VW01, SVSJ03]. PacoSuite is a component-based methodology that aims at lifting the abstraction level for component-based software development. PacoSuite allows easy plug-and-play component composition without requiring in-depth technical knowledge of the involved components. In order to achieve this, PacoSuite introduces explicit and reusable composition patterns that are able to describe an abstract interaction between a number of roles. A component is documented with one or more usage scenarios that specify how the component interacts with its environment. It is possible to automatically verify compatibility of a component with a role of a composition pattern using algorithms based on automata theory. In addition, glue-code that translates possible syntactical incompatibilities and constrains the external behavior of the components as specified by the composition pattern can be automatically generated.

The PacoSuite methodology does however suffer from crosscutting concerns that can not be modularized in a single entity. A typical example of a crosscutting concern is the logging concern. In order to implement logging in PacoSuite, two solutions are possible: either incorporate the logging behavior in all involved components or include the logging behavior in all involved composition patterns. Forcing every component to incorporate the logging concern causes this concern to be spread and duplicated over all these components, which is not at all a good separation of concerns. In PacoSuite, a second solution is possible to implement logging behavior, namely incorporating the logging behavior in the composition pattern. This means for example that every composition pattern has to include an additional role to which all messages between components are also sent to. As such, the component that is mapped onto this additional role can implement the actual logging behavior. This solution already modularizes the actual logging logic itself into one component. However, the composition pattern has to be altered to include a protocol fragment specific for the logging concern. The same is true for other crosscutting concerns leading to an explosion of different versions of the same composition pattern to cope with every possible combination of concerns. As such, neither modularizing the logging behavior in a component nor modularizing the logging behavior in a composition pattern, are acceptable solutions.

In this dissertation, a solution is proposed, called a *composition adapter* [Van01,Van02a,Van02b,VW02,VSJ03b], which allows cleanly modularizing crosscutting concerns in the PacoSuite methodology without producing an explosion of different versions for every possible combination of concerns. A composition adapter is able to describe transformations of a composition pattern and consists of two parts: a *context part* and an *adapter part*. The context part describes an abstract protocol fragment that has to be adapted and the adapter part specifies the actual adaptation. A composition adapter can be *applied* onto a given composition pattern. Applying a composition adapter consists of two steps: first *matching* the context part with the composition pattern and afterwards *inserting* the adapter part into the composition pattern. When the context part does not match the composition pattern at hand, the adapter part can not be inserted into the composition pattern and the application of the composition adapter is invalid. Both the matching and inserting steps are executed automatically using algorithms based on

finite automata theory. Checking compatibility of a component with a role of a composition pattern and automatic glue-code generation are still possible when a composition adapter is applied. As such, a composition adapter effectively modularizes crosscutting concerns while maintaining the high abstraction level of component composition in PacoSuite.

A limitation of the composition adapter model is that a composition adapter is only able to influence the external behavior of a component by for example re-routing or ignoring its messages. In order to realize adaptations of the internals of a component, a programming language tailored for component-based software engineering is required. In this dissertation, we motivate that current mainstream aspect-oriented approaches are not well suited for the component-based context and PacoSuite in particular. Therefore, a novel aspect-oriented programming language, called JAsCo [CDS+03,SVJ03,VVSJ03,VS04a,VS04b,VSVC04] is proposed. JAsCo leverages the contributions from component-based software engineering into an aspect-oriented programming language. JAsCo introduces two main concepts: *aspect beans* and *connectors*. An aspect bean is an extended version of the standard Java bean component that is able to contain several hooks that capture the crosscutting behavior. Aspect beans do not hard-code a concrete context and are thus easily reusable. An explicit composition mechanism is provided, called a connector, which allows connecting an aspect bean to a concrete context. Furthermore, expressive combination strategies and precedence strategies can be defined in order to manage the collaboration of several aspect beans and components. The JAsCo technology introduces a new component model with built-in traps. This allows instantiating and destroying aspects at run-time and as such the application behavior can be dynamically altered.

A JAsCo aspect bean is represented in PacoSuite by an enhanced version of a composition adapter, called an *invasive composition adapter* [VS03,VSJ03a,VSJ03c,VSWJ03,VSJ05]. The invasive composition adapter documentation describes the protocol transformations while the aspect bean implementation in JAsCo is able to alter the internal behavior of the components. An invasive composition adapter can be visually applied onto a complete component composition. It is still possible to automatically verify whether an invasive composition adapter matches with a given component composition. The protocol transformations described by an invasive composition adapter can also be automatically inserted into the composition pattern and component usage scenarios. Furthermore, a connector in the JAsCo language that connects the aspect bean to the correct component context can be automatically generated from the complete composition. Checking compatibility of a component with a role of a composition pattern and automatic glue-code generation are still possible when an invasive composition adapter is applied. As such, an invasive composition adapter is able to realize all the benefits of a regular composition adapter with the added expressiveness of the JAsCo implementation.

1.3 Contributions

The contributions of this dissertation can be divided into two major groups:

- Contributions with respect to integrating aspect-oriented and component-based ideas at the component composition level (PacoSuite):
  - We present a composition adapter to integrate aspect-oriented ideas at a component-based design level (PacoSuite). A composition adapter consists of an abstract context and adaptation specification independent of concrete component types and APIs. As such, a composition adapter is an ideal means to describe aspects that require protocol transformations. In addition, a composition adapter supports the specification of protocol-based triggering conditions for aspects.
  - We present an algorithm to automatically verify whether a composition adapter matches with a given context. Composition adapters provide a partial solution to the feature interaction problem as the abstract protocol context where the composition adapter has to be applied to is explicitly defined. As such, it is possible to verify whether a composition adapter does not destroy or alter to context of another consecutive composition adapter.
  - We present an algorithm that automatically inserts a composition adapter into a given composition pattern. As such, the abstraction level of component composition in PacoSuite is not lowered when aspects represented as composition adapters are present.
  - We present invasive composition adapters that combine the best of both worlds, namely explicit protocol transformations and internal adaptations of components through the JAsCo aspect bean implementation. The achieved contributions of regular composition adapters are also valid for invasive composition adapters.
  - We present an extension of the PacoSuite tool environment that implements the composition adapter model. Both regular and invasive composition adapters can be visually applied onto a given component composition. Glue-code that realizes the composition is generated automatically.
  - We report on a larger case study of an e-commerce application in which we succeeded to modularize several crosscutting concerns as (invasive) composition adapters.
- Contributions with respect to integrating aspect-oriented ideas and component-based ideas at the implementation level (JAsCo):
  - We introduce explicit aspect beans that capture crosscutting behavior in hooks, which are a special kind of inner classes. Aspect beans are highly reusable as they do not hard-code a concrete context, but describe an abstract pointcut. Furthermore, aspect beans are backward compatible with Java beans and can thus engage in regular component interactions. Aspect beans are also able to support a kind of aspectual polymorphism [MO03] by employing abstract methods.
  - We present explicit connectors that instantiate aspect beans onto a concrete context. In addition, connectors are able to manage the cooperation of several aspect beans by employing precedence and combination strategies. As such, a partial solution for the feature interaction problem is provided.

- We present an aspect-oriented technology that allows keeping aspect beans first-class at run-time and that allows aspect beans to be dynamically loaded and unloaded.
- We integrate ideas from adaptive programming [Lie96, LOO01] into the JAsCo language and show that the JAsCo contributions concerning reusable aspects and an explicit aspect composition mechanism are also applicable to adaptive programming.
- We present a full prototype of the JAsCo programming language in order to assess the contributions realized by JAsCo.

1.4 Outline of the Dissertation

In Chapter 2, we introduce the most important concepts and approaches of component-based software engineering and explain the original PacoSuite approach developed by Bart Wydaeghe in more detail. Furthermore, aspect-oriented software development is introduced and four representative aspect-oriented approaches are presented: AspectJ [KHH+00], Composition Filters [BAT01], Adaptive Programming [Lie96] and HyperJ [TOHS99]. Afterwards, the current state-of-the-art approaches that combine aspect-oriented and component-based software engineering are discussed. Readers familiar with these topics can safely skip the corresponding sections.

Chapter 3 motivates why a composition adapter is required in the PacoSuite approach and presents the composition adapter model. In addition, the algorithm to automatically match a composition adapter with a given composition pattern and insert the adaptations described by the composition adapter is formally defined.

In Chapter 4, we first outline the requirements for an aspect-oriented programming language targeted at component-based software engineering in general and PacoSuite in particular. Afterwards, the JAsCo language and technology are explained in full detail. Finally, an extension to the JAsCo language that integrates ideas of adaptive programming is presented.

In Chapter 5, the invasive composition adapter model is presented. In addition, an algorithm is introduced that allows applying an invasive composition adapter by automatically generating a connector in the JAsCo language. Finally, an extension of the PacoSuite concepts is proposed in order to unleash the full power of the JAsCo aspect bean implementation.

In Chapter 6, the tool support implemented for the approach elucidated in this dissertation is explained. First, the extension to the PacoSuite environment is discussed. Afterwards, the JAsCo implementation is presented. In addition, a critical performance evaluation is performed and some improvements are proposed.

Chapter 7 performs a larger case study in order to validate the approach elucidated in this dissertation. A distributed e-commerce application is created using the PacoSuite methodology and several crosscutting concerns are successfully modularized using (invasive) composition adapters.

Chapter 8 states the conclusions of this dissertation. We summarize the contributions of this dissertation and evaluate the applicability of the approach presented in this dissertation. Finally, we discuss some interesting issues that remain to be investigated.

Chapter 2 Research Context

In this chapter, the context of the dissertation is elucidated. We introduce the most important concepts and approaches of component-based software engineering and explain the original PacoSuite approach developed by Bart Wydaeghe in more detail. Furthermore, aspect-oriented software development is motivated and explained. In addition, four representative aspect-oriented approaches are presented: AspectJ, Composition Filters, Adaptive Programming and HyperJ. Afterwards, the current state-of-the-art approaches that combine aspect-oriented and component-based software engineering are discussed.

2.1 Component-Based Software Engineering

2.1.1 Definition

The term "component" is probably one of the most overloaded terms in computer science research. Even in the software engineering context, the term component is used to denote anything from Abstract Data Types (ADT) to complete frameworks. Bellinzona et al. [BFM93] for example employ objects as components while Reid et al. [RFS+00] use sets of C functions as components. Shaw and Garlan [SG96] define a component as a "*computational unit*", which implies that almost any software artifact is considered to be a software component. In [DK92], a component is defined as *"A life-cycle work product out of which other, larger life-cycle work products can be composed"*. This definition states that a component is subject to composition and that as such a larger system is established. This definition is however still too vague as a lot of software artifacts are subject to composition. Brown and Short define a component as "*an independently deliverable set of reusable services*" [BW96]. This means that a component has to offer a set of services that are reusable and that the component itself should be independent of a specific context. This definition does however not include that a component has to be subject for composition.

Partially due to this confusion, a substantial amount of software engineering researchers do not get the message component-based software engineering tries to deliver. These disbelievers claim that doing components is just reinventing the wheel, as they consider a component to be identical to for instance an Abstract Data Type. Therefore, it is crucial to have an unambiguous, consistent and exclusive definition of a component. A commonly accepted definition of a component is formulated by Szyperski [SZY98]:

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

This definition states that a component does not only define which services it is able to offer, but also defines the services it requires from its environment in order to function properly. Using this definition, it is possible to differentiate between an Abstract Data Type and a component, because a component has to be independently deployable, while an ADT like for example a complex number is not. Also objects do not automatically fall under szyperski's component definition, because an object does not explicitly define context dependencies and is generally not independently deployable. Of course, this does not mean that an object can never be a component.

The idea of component-based software engineering is to improve on current software engineering methodologies by enabling cheaper and faster development which results in higher quality software. This is achieved by plug-and-play component composition. More concretely, CBSE makes the following promises [HC01]:

- Costs will be decreased because components will be reused many times, allowing development costs to be amortized over many uses.
- Development time will be decreased because most of the development required will now involve writing only software interfaces.
- And software quality will be improved because many different projects will be using the components, and therefore defects will be caught more quickly, yielding more trustworthy software.

In order to motivate why component-based software engineering will succeed in this promise, the analogy with other engineering disciplines is often used. Many industries have matured over a long period to a point where they now consistently use components, including the manufacturing, electronics, construction and automotive industries [DW99]. The use of components is considered a big step forward in all of these industries.

2.1.2 Component Models

Current practice component-based software engineering depends heavily on the employed component model. In [HC01], a component model is defined as such:

"A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment. A component model also defines standards for an associated component model implementation, the dedicated set of executable software entities required to support the execution of components that conform to the model."

Examples of the main competing component models are OMG's CORBA Component Model (CCM), Microsoft's (D)COM/COM+ family, Microsoft's .NET, SUN Microsystems' JavaBeans and Enterprise JavaBeans and the recently introduced Web Services standard from W3C. All of these component models have their own specific characteristics which results in competitive advantages and disadvantages regarding to particular application domains. Some component models depend on one specific platform, like the COM component models. Others like for example CORBA and Web Services enable component interaction among different platforms and implementation technologies. Not all component models provide every feature listed in the definition above, but at the very least they provide a means for component composition. The Java Beans and Web Services component models for example, only offer a communication mechanism, while others, like for instance CORBA, offer tons of features, making those component models more complex.

In this dissertation, the Java Beans component model is chosen as the target model for several reasons. First of all, Java Beans is a very simplistic component model making it ideal for research experiments. Secondly, because Java Beans is widely used in the software engineering research community, it allows for easy comparison and integration with other approaches. Finally, we choose to employ Java Beans because the PacoSuite component-based methodology uses Java Beans and this dissertation builds onto the results of this research.

2.2 PacoSuite

The PacoSuite approach is used as the starting point and the component-based methodology of this dissertation. The original PacoSuite component-based methodology aims at lifting the abstraction level for component-based software engineering. The research resulted in the PhD of Bart Wydaeghe [Wyd01] in November 2001. This section summarizes the main ideas and contributions of the PacoSuite approach that are relevant for this thesis.

2.2.1 Motivation

A naive view on component-based software engineering is known as the Lego™ model [Sho97]. In this model, an application is created by simply plugging software components together in a similar manner as assembling Lego™ bricks. While this model might work in some simple examples, real-life component-based software engineering is quite different. Most of the time, components are developed independently and are thus normally not specifically designed to interoperate with a given other component. Both syntactic incompatibilities, like for example naming of the API, and semantic incompatibilities, like for example different currency types, can occur. In order to bridge these incompatibilities, glue-code has to be written. IBM [SZY97] estimates that 70% of all code written today consists of such translations between interfaces. As a consequence, even a minor automation of this process would be a huge gain. It is of course also very well possible that two components are never able to interoperate, because they both expect fundamentally different services from their environment.

Current practice component composition environments, like for example VisualAge for Java from IBM, JBuilder from Borland and BeanBuilder from Sun allow already some form of automatic glue-code generation from a given component composition. They typically allow to visually drag components onto a canvas. Communication between components is established by visually connecting an event from one component to one or more methods of another component. Optionally, the properties of the components can be changed using a wizard-based graphical user interface. Figure 1 illustrates a screenshot of the Bean Builder component composition environment. In this screenshot, two *JButton* components and a juggler component are dragged onto the composition canvas. The *JButton* is a simple button component that fires an *actionPerformed* event when it is clicked. The *Juggler* component is a dummy visual component that is able to show a juggling animation. The component composer has connected the *actionPerformed* event of one *JButton* to the start method of the *Juggler*. Likewise, the component composer is about to map the *actionPerformed* event of the other *JButton* to the stop method of the *Juggler* component. As a consequence, the *Juggler* bean can be started and stopped by clicking the appropriate buttons.

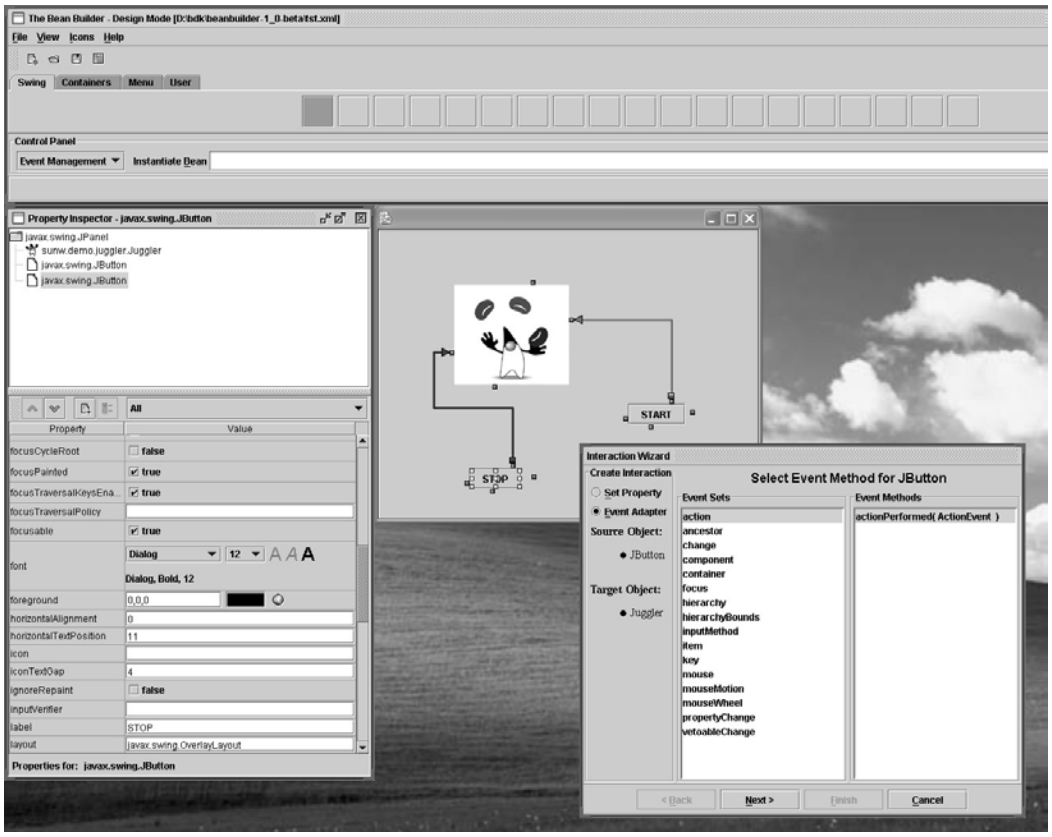

Figure 1: Visually wiring an application in the Bean Builder environment. The top window features a palette of components sorted into different categories. The center window is the visual canvas that allows to visually design a component composition. Notice that the lowest button is currently selected in the center window. The leftmost window shows the properties of the selected component in the designer window and allows changing them. The bottom-right window represents a wizard that allows connecting events from one component to methods of another component.

The problem in general with current component composition environments is that they are too low-level. For example in Figure 1, it is impossible to wire a single *JButton* that is able to toggle the *Juggler* component. This because current visual component composition environments do not allow compositions to have state. As a consequence, only simple event-method connections can be automatically generated. Collaborations that are more complex still have to be written manually. In addition, there is no support whatsoever for checking whether a set of components is even capable to cooperate. It is only after studying the API and the detailed documentation in a natural language that one can decide whether a set of components can be wired together or not. Furthermore, current component composition environments do not allow reusing the collaboration logic between the components. Even a simple replacement of a component with a newer version is impossible. As a result, the composition has to be rewired from scratch in most cases. The success of design patterns [GHJV95] however, indicates that some collaboration patterns are used over and over again.

2.2.2 PacoSuite Entities

To solve the problems described above, PacoSuite introduces explicit and reusable composition patterns that are able to describe stateful collaborations between a number of abstract roles. In addition, PacoSuite

introduces explicit component usage scenarios. Using both composition patterns and usage scenarios, it is possible to automatically check whether a component is "protocol compatible" with a given role of a composition pattern. When all the roles of a composition pattern are filled, glue-code can be automatically generated. This glue-code translates syntactical incompatibilities and limits the components' behavior as specified by the composition pattern. The next sections describe this approach in more detail. For an in-depth explanation of PacoSuite, the interested reader is referred to the PhD dissertation of Bart Wydaeghe [Wyd01].

2.2.2.1 Component Usage Scenarios

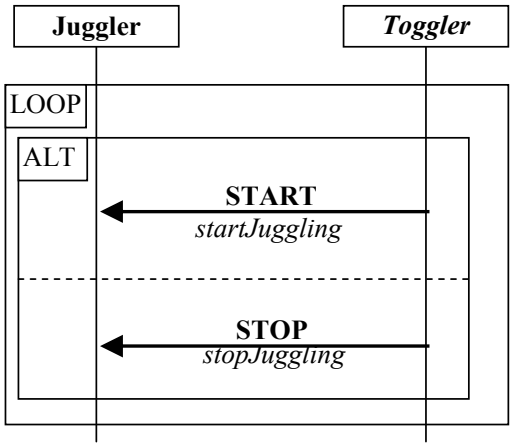

Figure 2: A usage scenario of the Juggler component. This usage scenario documents that a Juggler component is able to receive continuous START and STOP messages implemented by the startJuggling and stopJuggling methods.

In PacoSuite, a component is documented by a special kind of Message Sequence Charts (MSC) [DLVW98] that specifies how to use the component. As such, the interior logic of the component is not documented. The idea is that for black-box components only the external protocol matters, as the interior is supposed to be hidden. Figure 2 illustrates a usage scenario of the well-known *Juggler* bean. One participant of a usage scenario represents the component itself and the other participants represent the environment the component expects. In this case, only one environment participant is specified, namely the *Toggler* participant.

The messages sent between the participants are not documented using the concrete API of the component because this is rather ambiguous. Indeed, a *connect* message received by one component can mean that the component should connect with a network host while the same message sent to an agent component results in a randomized search for another agent component. In general, natural language descriptions work fairly well for interpretation by humans, but it is very hard to create automatic support and tools for component composition based on these natural language descriptions.

Therefore, PacoSuite proposes to document messages by a set of abstract semantic primitives. Of course, it is impossible to come up with such a set for all possible applications. However, case studies have shown that for specific application domains, constructing a set of semantic primitives is feasible. As a consequence, the PacoSuite approach is domain dependent and mainly applicable for creating constructing kits that allow easy and fast application development for a certain application domain. Of course, it is also possible to just skip these semantic primitives. In that case, the PacoSuite approach is still a valuable contribution. Components however, will be less reusable because they would have to match on the concrete API. Figure 3 illustrates the set of primitives that is used in this dissertation. Notice that this is a

hierarchical set. The topmost primitive is the most general one and matches any of the primitives below. An elaborate motivation for this hierarchical set of primitives is given in Bart Wydaeghe's PhD [Wyd01].

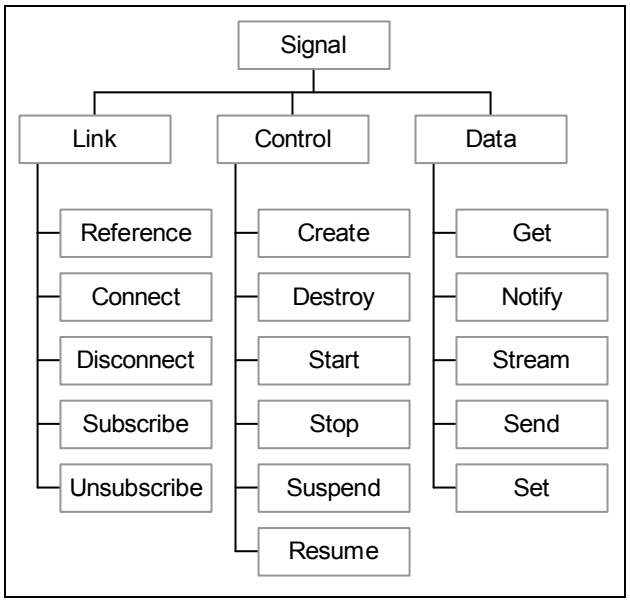

Figure 3: Example of a hierarchical set of semantic primitives.

The PacoSuite component documentation is not only abstract but also concrete because each message is documented by a concrete API that implements the message. For example, the usage scenario of the Juggler component (see Figure 2) specifies that the START primitive is implemented by the *startJuggling* method. Likewise, the STOP primitive is implemented by the *stopJuggling* method. As a result, this usage scenario documents that a Juggler component is able to receive continuous START or STOP messages implemented by the *startJuggling* and *stopJuggling* methods.

Notice that PaocSuite only documents the external protocol of a component. Invocations of a component onto itself are not documented. This is because the goal of PacoSuite consists of checking compatibility with other components. Self-invocations are irrelevant for this purpose.

2.2.2.2 Composition patterns

PacoSuite introduces explicit and reusable composition patterns that are able to describe stateful collaborations. A composition pattern is similar to an n-ary connecter found in the Architectural Description Languages (ADL) [GS93]. A composition pattern specifies an abstract interaction between a number of roles and is expressed using an adapted version of Message Sequence Charts (MSC). Every participant of the MSC represents a role in the collaboration of the composition pattern. The messages between the roles originate from the same limited set of semantic primitives as the components are documented with. This allows comparing the signals in a usage scenario of a component with these in a composition pattern.

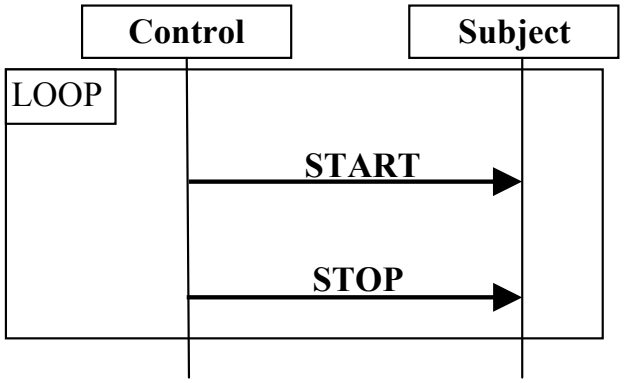

Figure 4: A toggling composition pattern. This composition pattern documents a toggling interaction, namely consecutive START and STOPS, between two collaborating roles.

Figure 4 illustrates an example of a composition pattern. This composition pattern describes the collaboration between two roles: *Control* and *Subject*. This composition pattern specifies that the *Control* participant sends consecutive STARTs and STOPs to the *Subject* participant. A possible application of this composition pattern is a simple visual interface that allows toggling the *Juggler* component from a single *JButton* component. To build this application, the *Juggler* component is mapped on the *Subject* role and the *JButton* component is mapped on the *Control* role. Notice that even this simple collaboration can not be wired by most visual composition environments because the collaboration itself requires state.

2.2.3 Short introduction to Finite Automata

In order to allow automatic compatibility checking and glue-code generation, the MSCs have to be transformed to a more operational model. Therefore, every MSC is translated to a finite state automaton representation. Finite state automata are well documented in literature [HMU01]. The following paragraphs briefly review some important concepts of finite automata theory.

Definition 1 defines the concept of a finite state machine (or also named a finite automaton) as a five-tuple consisting of a set of states, an input alphabet, a start state, a set of final states and a transition function.

Definition 1: Deterministic Finite State Machine

A deterministic finite state machine is

a five-tuple $\{Q, \Sigma, q_s, F, \delta\}$ where

- Q is a finite set of states
- Σ is a finite input alphabet of symbols
- $q_s \in Q$ and q_s is the start state
- $F \subseteq Q$ and F is the set of final states
- $\delta: Q \times \Sigma \rightarrow Q$ is the transition function

  if $\delta(q,a)$ is not defined then we notate $\delta(q,a)=\varnothing$

Definition 2 specifies the transitive transition function δ^t of an automaton M. This transitive transition function takes as input a state p and a list R containing any finite number of input symbols of the alphabet

of M. The transitive transition function returns the state that can be reached by applying the regular transition function successively for every member of the list R.

An automaton accepts a string of input symbols R when the transitive transition function applied to the start state and the string R renders a final state of the automaton as a result. The *input language* of an automaton consists of the set of all the strings of input symbols accepted by the automaton.

Definition 2: Transitive transition function δ^t

If $M=\{Q, \Sigma, q_s, F, \delta\}$ is a finite state machine then

$\delta^t: Q \times \Sigma^+ \rightarrow Q$ is the transitive transition function of M

when $\delta^t(p,(a_1,\ldots,a_n))=q$ for $n>0$

where $\exists\ q_0,\ldots,q_n \in Q \wedge q_0=p \wedge q_n=q \wedge$

$\forall 0<i\leq n: \ \delta(q_{i-1},a_i)=q_i$

Definition 3 specifies that a complete automaton is an automaton where the transition function is defined for every possible combination of states and input symbols. An incomplete automaton can be easily made complete without changing its input language by adding a so-called error state [HMU01].

Definition 3: Complete Automaton

A finite state machine $M=\{Q, \Sigma, q_s, F, \delta\}$ is complete

when $\forall q\in Q \wedge \forall a\in\Sigma: \delta(q,a)\neq\varnothing$

Definition 4 defines a path of an automaton as an ordered set of states of the automaton. In addition, for every state q in the path except the last state, there exists an input alphabet element a so that the result of applying the transition function on q and a renders the next state in the path. The length of a path is defined as the number of states in the path. A start-stop path is a path that starts at the start state of the automaton and ends at a final state of the automaton (see Definition 5).

Definition 4: Path of a DFA

$P=(q_0, \ldots, q_n)$ with $n>0$ is a path of length n+1 of

DFA $M=\{Q, \Sigma, q_s, F, \delta\}$

when $P\in Q^{n+1} \wedge \forall 0\leq i<n: \exists a\in\Sigma: \delta(q_i, a) = q_{i+1}$

Definition 5: Start-Stop Path of a DFA

$P=(q_s, q_1, \ldots, q_n)$ is a start-stop path of DFA $M=\{Q, \Sigma, q_s, F, \delta\}$

when P is a path of M and $q_n\in F$

2.2.4 Finite Automata in PacoSuite

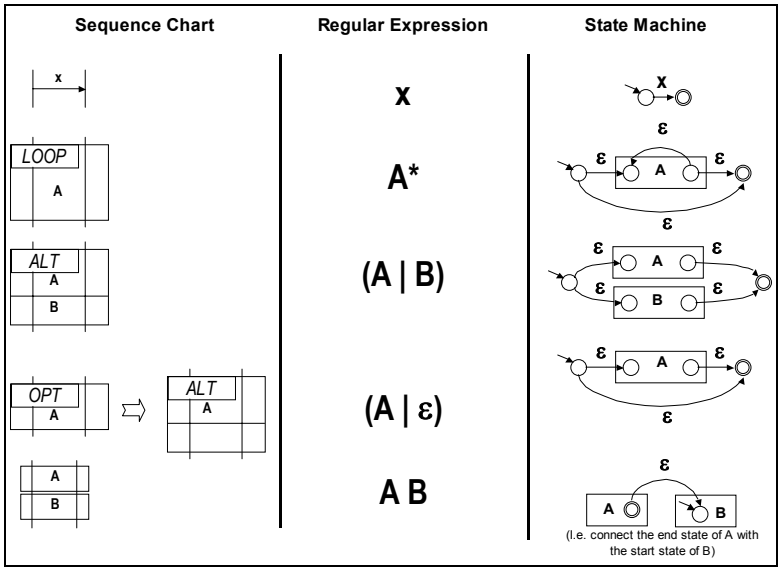

Figure 5: Converting message sequence charts to deterministic finite automata.

The translation of an MSC to an automaton representation is a well-known process and described in literature [HMU01]. Every single MSC construct can be translated to a finite automaton as illustrated by Figure 5. In order to translate a full MSC, recursively applying the translation rules and connecting the separate automata with epsilon transitions does the job. Afterwards, the obtained automaton can be made deterministic again using the standard algorithm [HMU01].

One issue that has to be coped with consists of avoiding losing the source and destination of message sends of an MSC. To this end, the transition labels specify the employed primitive of the message augmented with the source and destination roles. The following definition defines an MSC as an annotated automaton with source and destination roles. The τ function returns a mapping of roles for a given transition label.

Definition 6: MSC automaton

An MSC automaton is a three-tuple $\{M, R, \tau\}$

where $M=\{Q, \Sigma, q_s, F, \delta\}$ is a finite automaton

R is a finite set of roles

$\tau: \Sigma \rightarrow RxR$

Using the MSC automaton definition, a component usage scenario and composition pattern are formally defined (see Definition 7 and Definition 8). The finite automaton M of a component usage scenario is referred to as the component automaton. Likewise, the finite automaton M of a composition pattern is named the composition pattern automaton.

Definition 7: Component Usage Scenario

A component usage scenario C is a seven-tuple

$\{M, R, \tau, A, \alpha, P, \rho\}$

where $\{M=\{Q, \Sigma, q_s, F, \delta\}, R, \tau\}$ is an MSC automaton

A is a finite set of API terms

$\alpha: \Sigma \rightarrow A$

P is a finite set of primitives

$\rho: \Sigma \rightarrow P$

Definition 8: Composition Pattern

A composition pattern CP is a five-tuple

$\{M, R, \tau, P, \rho\}$

where $\{M=\{Q, \Sigma, q_s, F, \delta\}, R, \tau\}$ is an MSC automaton

P is a finite set of primitives

$\rho: \Sigma \rightarrow P$

2.2.5 Defining Compatibility

When a component has to be declared compatible with a role in a connecter/composition pattern is difficult define. It all depends on how reusable the connectors should be. Most approaches [AG97] define a component to be compatible with a connector if and only if the component is able to adhere to the full protocol described in the connector. In other words, the protocol of the component is a superset of the one of the connector. At first sight, this is a logical definition, because components should be generic and reusable entities. However, some approaches, including PacoSuite, claim that connectors should also be first class, generic and reusable entities. In this view, a connector that specifies several optional protocols to achieve the same goal should match with a component that only implements one of these protocols.

To define compatibility in the PacoSuite approach, the projection of a composition pattern to a role is required:

Notation: Projection of a composition pattern to a role

We denote the projection of a composition pattern CP to a role r as:

$$Proj_r(CP)$$

This projection can easily be constructed from the full composition pattern by replacing all transitions that are labeled with a message that does not interact with role r (source and destination $\neq$ r) with an epsilon transition. Calculating the epsilon closure of this automaton calculates the projection. The set of messages

of the projection is the original set without these non-interacting messages. The set of roles of the projection is the original set of roles without the roles that are not the source or the destination of any message in the new set of messages.

Using the above definition, PacoSuite defines local compatibility, meaning compatibility of one component with one role as follows:

Definition 9: Local Compatibility

Let C be a component and r be a role of the composition pattern CP. Further let L(C) be the language accepted by the component automaton and let $L(P_r(CP))$ be the language accepted by the automata defined by $proj_r(CP)$

Then C is local compatible with r $\Leftrightarrow$ $L(C) \cap L(proj_r(CP)) \neq \emptyset$

Informally, this means that a component is defined to be compatible with a role of a composition pattern if and only if the component and the composition pattern share at least one common path of protocol, which means that the intersection of the corresponding languages cannot be empty. The projection is needed in the definition to discard all interactions that have nothing to do with the role the component is mapped onto.

Defining local compatibility alone is not enough because PacoSuite allows both a component and a composition pattern to be more generic than the other. Consider for example a composition pattern that specifies two different optional protocols to reach the same goal. If two components are applied onto this composition pattern, and one component implements one alternative and the other component implements the other alternative, then both components will be locally compatible. However, the global composition is invalid and deadlocks because both components adhere to very different protocols. Therefore, a global compatibility definition is required. In order to define the global compatibility, an important helper definition is needed. The helper definition is motivated and explained in detail in Bart Wydaeghe's PhD. The following paragraphs shortly recapitulate this definition.

The definition defines the parallel composition of two automata. The parallel composition is in defined similar to the parallel composition operator found in CSP [Hoa85]. An intuitive view on the parallel composition is that it allows component or composition pattern automata to perform their own transitions as long as no synchronization is required. However, they need to perform a joint step for labels in the intersection of their alphabets. I.e. they need to agree on common steps, but can go along with their own transitions. The parallel composition is defined as follows [Wyd01]:

Definition 10: the parallel composition operator

A parallel combination of two component or composition automata $P_1=\{Q_1, \Sigma_1, q_{s1}, F_1, \delta_1\}$ and $P_2=\{Q_2, \Sigma_2, q_{s2}, F_2, \delta_2\}$ is described as:

$$P_{1\,\Sigma_1}\Big\|_{\Sigma_2} P_2$$

In this combination, P_1 can perform events only in Σ_1, P_2 can perform events only in Σ_2, and they must simultaneously engage in events in the intersection of Σ_1 and Σ_2. There are two rules that define the possible transitions of a parallel combination. One rule describes the independent execution of each of the components, and the other describes the performance of a joint step.

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_{1\,\Sigma_1}\Big\|_{\Sigma_2} P_2 \xrightarrow{\alpha} P_{1\,\Sigma_1}'\Big\|_{\Sigma_2} P_2} [\alpha \in (\Sigma_1 \setminus \Sigma_2)]$$

$$P_{2\,\Sigma_2}\Big\|_{\Sigma_1} P_1 \xrightarrow{\alpha} P_{2\,\Sigma_2}\Big\|_{\Sigma_1} P_1'$$

$$\frac{\begin{array}{c} P_1 \xrightarrow{\alpha} P_1' \\ P_2 \xrightarrow{\alpha} P_2' \end{array}}{P_{1\,\Sigma_1}\Big\|_{\Sigma_2} P_2 \xrightarrow{\alpha} P_1'\,{}_{\Sigma_1}\Big\|_{\Sigma_2} P_2'} [\alpha \in \Sigma_1 \cap \Sigma_2]$$

This corresponds with a new automaton

$\{Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, [q_{s1}, q_{s2}], F_1 \times F_2, \delta'\}$ with

δ': $(Q_1 \times Q_2) \times (\Sigma_1 \cup \Sigma_2) \rightarrow Q_1 \times Q_2$:

- $\delta'([s_1,s_2],\alpha) = [t_1,t_2] \Leftrightarrow \alpha \in \Sigma_1 \cap \Sigma_2$ and $\delta_1(s_1,\alpha) = t_1$ and $\delta_2(s_2,\alpha) = t_2$
- $\delta'([s_1,s_2],\alpha) = [t_1,s_2] \Leftrightarrow \alpha \in (\Sigma_1 \setminus \Sigma_2)$ and $\delta_1(s_1,\alpha) = t_1$
- $\delta'([s_1,s_2],\alpha) = [s_1,t_2] \Leftrightarrow \alpha \in (\Sigma_2 \setminus \Sigma_1)$ and $\delta_2(s_2,\alpha) = t_2$

Using the parallel composition operator, it is possible to define global compatibility of a composition pattern with a set of components. This is stated by the following definition [Wyd01]:

Definition 11: Global Compatibility

Let $C_1,\ldots,C_n$ be a set of components. Let $CA_1,\ldots,CA_n$ be the corresponding component automata (i.e. $\forall 0<i\leq n: CA_i$ is the component automaton of C_i)

Let CP be a composition pattern with roles $r_1,\ldots,r_n$. Let CPA be the corresponding composition automaton.

Then CP is global compatible with $C_1,\ldots,C_n$

$$\Leftrightarrow$$

$$L\left(CPA \parallel CA_1 \parallel \cdots \parallel CA_n\right) \neq \varnothing$$

In words, this definition states that a set of components and a composition pattern are compatible if the composition pattern and all possible collaborations between the set of components (as calculated by the parallel composition of all components) share at least one path. Informally, this means that there has to be at least one possible protocol where both the composition pattern and all involved components adhere to. As such, both the components and composition pattern are able to offer more than what the complete composition employs because only a single common protocol has to exist. Therefore, the definition complies with the generic reusability idea proposed by PacoSuite.

2.2.6 Checking Compatibility

Once a set of components and composition pattern are all translated to a DFA, it is in fact very easy to check both local and global compatibility because the definitions are already in terms of automata theory. For example, for checking the local compatibility between a component X and a role Y of composition pattern Z, it suffices to take the projection to role Y of composition pattern Z, transform both MSCs to DFAs and then to calculate the intersection between both DFAs. If the intersection is not empty, meaning that there is a start-stop path in the automaton, then the component X is compatible with role Y. The intersection is also a standard DFA process and widely described in literature [HMU01].

Checking global compatibility can be as easily performed. First, transform all MSCs to DFAs. Then, compute the parallel composition of all component DFAs, which in this case amounts to calculating the shuffle automaton [HMU01] because all the component languages are disjunct. Afterwards, again the intersection is computed of the whole composition pattern DFA with the shuffle automaton. If the intersection is not empty, meaning that there is a start-stop path in the resulting automaton, then the global check succeeded and the composition is valid as a whole.

Notice that especially the global checking algorithm is highly resource-intensive because it first builds up the whole parallel composition, which is a huge automaton, and afterwards shrinks it by taking the intersection. In [VW01, Wyd01], an incremental algorithm is proposed to increase the performance by incrementally verifying every component with the composition and thus avoiding the big shuffle automaton. In theory however, both algorithms are of exponential nature and when the set of components is large enough, they both perform equally.

2.2.7 Glue-code generation

Because the components possibly have syntactically incompatible APIs, glue-code is necessary to enable component interaction. The glue-code translates incoming events from the components to the correct outgoing method calls. In addition, the glue-code has to be state-aware in the sense that it has to be able to react differently on the same incoming event depending on the "state" of the whole application. In the toggle control example discussed in sections 2.2.2.1 and 2.2.2.2, the first time an *actionPerformed* event from the *JButton* is received, it should be translated into *startJuggling*. The second time however, the same *actionPerformed* event has to be translated into *stopJuggling*.

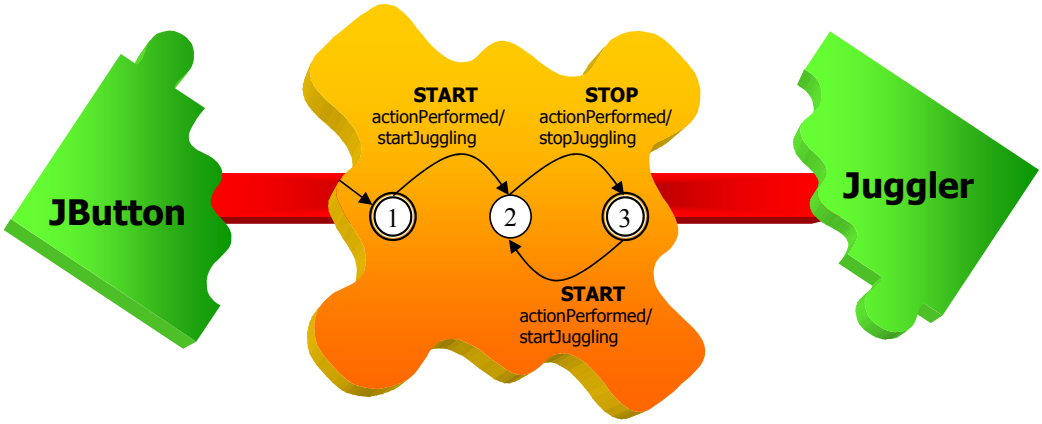

Figure 6: Illustration of the glue-code generated for the composition of a Juggler and a JButton component.

To enable state-aware glue-code, the glue-code simulates a state machine that represents the whole composition. This state machine is in fact already available as it is the result of the global checking process. As a consequence, automatic generation of the translation code is straightforward and inexpensive. PacoSuite generates highly restrictive glue-code in the sense that the glue-code acts as a kind of wrapper around the components and thus no direct component communication is possible anymore. In addition, the glue-code also ignores any events that are not expected at this point in the execution trace of the composition. As a result, the glue-code enforces the components to work as described by the composition pattern, or at least makes sure that no other collaboration protocol is allowed.

Figure 6 illustrates an example of the glue-code generated when composing the *JButton* and *Juggler* components with the *ToggleContol* composition pattern. Both the *JButton* and *Juggler* components never directly communicate with each other, instead the central glue-code is used as a mediator. The glue-code contains a state-machine that simulates the toggle control behavior. Every transition of the state machine is labeled with the name of the primitive, an incoming method and an outgoing method. For example, when the state machine is in state 1 and *actionPerformed* is received from the JButton, *startJuggling* has to be called onto the *Juggler* component

2.2.8 Tool Support

The algorithms and ideas mentioned in the previous sections have been implemented in a prototype component composition environment that improves upon current state-of-the-art. The PacoSuite tool environment consists of two main applications: PacoDoc and PacoWire. PacoDoc is a documentation tool that allows to graphically draw and edit both component usage scenarios and composition patterns. The

created MSCs can be saved using XML. Figure 7 illustrates a screenshot of the PacoDoc tool showing a simple usage scenario of a *JButton* component.

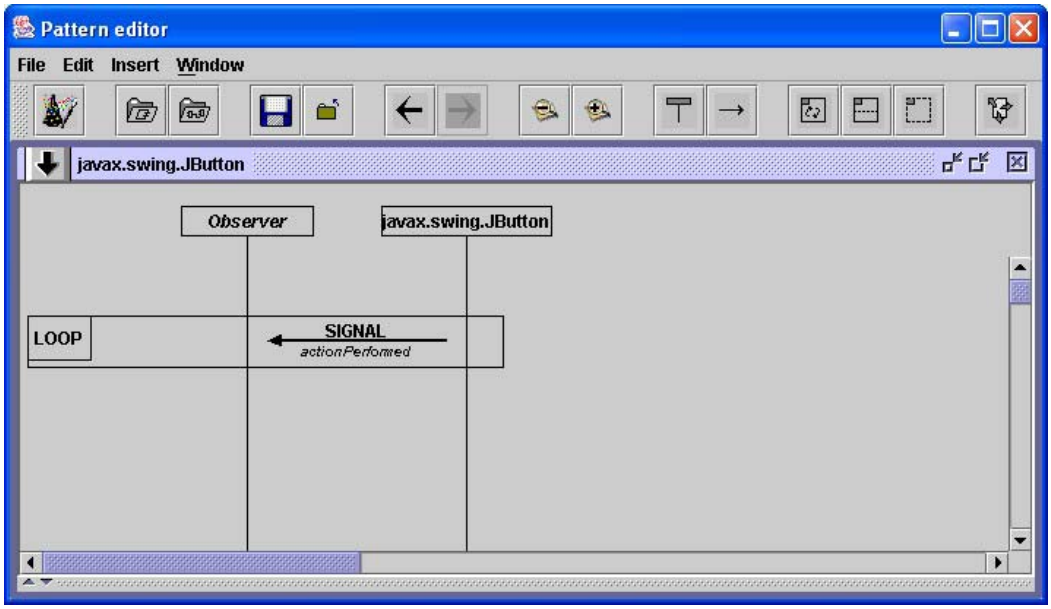

Figure 7: Simple usage scenario of a JButton component shown in the PacoDoc tool.

PacoWire is the actual component composition environment that allows to visually place components and composition patterns on a composition canvas. When a component is dragged onto a role of a composition pattern, compatibility is checked using the algorithms introduced in the previous sections. If the compatibility check fails, the drag-and-drop action is refused. As soon as all the roles of a composition pattern are filled, glue-code can be generated automatically.

Figure 8 illustrates a screenshot of the PacoWire tool. At the top, a library of components and composition patterns is available, nicely sorted into different categories. The center pane is the composition canvas where components and compositions can be dragged onto. The visual representation of a component on the canvas consists of a rectangular shape with an icon and the component's name. A composition pattern is represented by an oval connected with lines to its roles symbolized also by rectangles. In this screenshot, the component composer is about to map the *JButton* component (see Figure 7) onto the *Control* role of the *ToggleControl* composition pattern (see Figure 4). This component is clearly compatible with this role, so the drag is accepted by PacoSuite. The other role of the ToggleControl composition pattern is already filled by the *Juggler* component. When both roles are filled, glue-code generation can start.

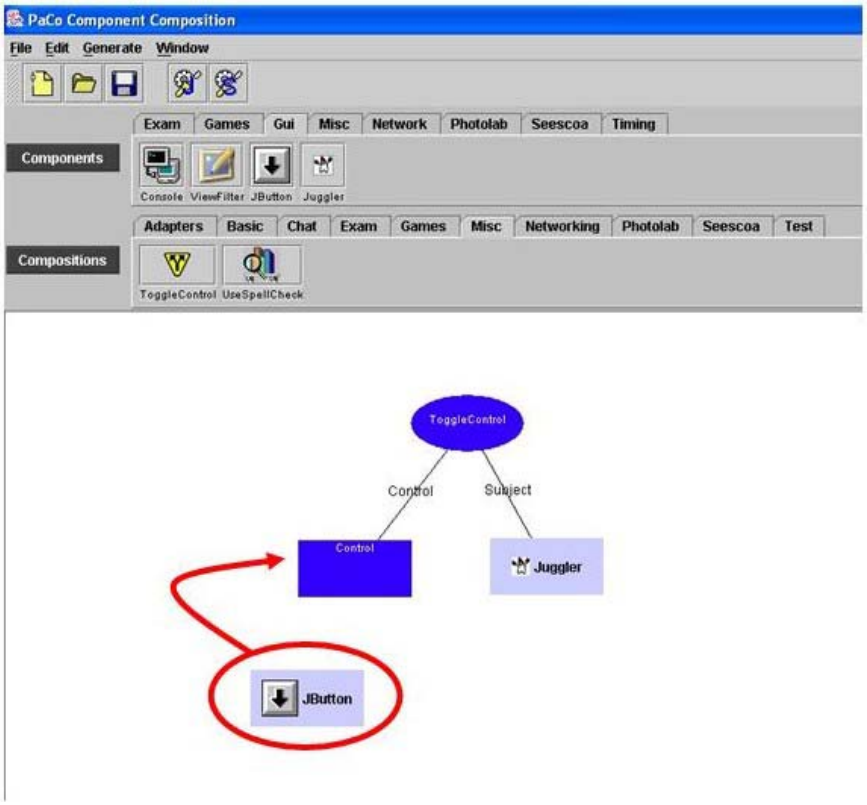

Figure 8: Screenshot of the PacoWire tool.

The tool also generates a demo application that positions every visual component in a frame. After the code generation process has finished, the demo application is launched. An example of the demo application generated from the composition in Figure 8 is depicted in Figure 9. Notice that both the *JButton* and *Juggler* components have a titled border that states the name of the role they are mapped upon. PacoSuite also supports to introspect and alter the bean properties of a Java Bean as defined by the Java Beans specification. The label of the *JButton* of Figure 9 for instance, is set to the "TOGGLE" string.

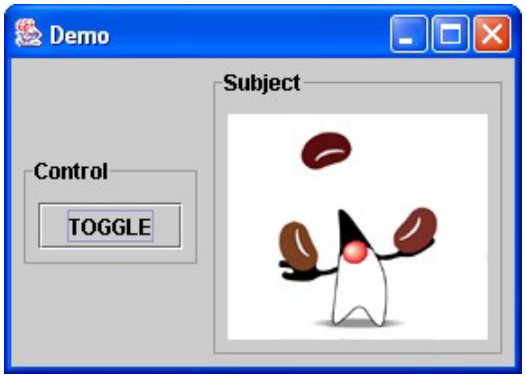

Figure 9: ToggleControl demo application that is automatically generated from the component composition of Figure 8.

2.3 Aspect-Oriented Software Development

2.3.1 Separation of concerns

Separation of concerns [Par72] is at the core of software engineering, and has been for decades. In its most general form, it refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. Many different kinds, or dimensions, of concerns may be relevant to different developers in different roles, or at different stages of the software lifecycle.

Separation of concerns involves decomposition of software according to one or more dimensions of concern. "Clean" separation of concerns has been hypothesized to [THO+00]:

- reduce software complexity and improve comprehensibility;
- promote traceability within and across artifacts and throughout the lifecycle;
- limit the impact of change, facilitating evolution and non-invasive adaptation and customization;
- facilitate reuse;
- and simplify component integration.

The ultimate goal of AOSD [KLM+97] is to achieve a better separation of concerns than possible using current software engineering paradigms. Some concerns of an application cannot be cleanly modularized as they are scattered all over the different modules of the system. Such a concern is called crosscutting because the concern virtually crosscuts the decomposition of the system. As a result, similar logic is repeated in different modules, with code duplication as a consequence. Due to this code duplication, it becomes very hard to add, edit or remove a crosscutting concern in the system.

A typical example of a crosscutting concern is logging. In a case study often cited by the advocates of aspect-orientation [KLM+97], several concerns are investigated in Tomcat, an open-source JavaServer Pages (JSP) implementation. In Figure 10, an analysis of the source-code of Tomcat is shown. Each vertical bar represents a class of the Tomcat system. The longer the bar, the more lines of code reside in that class. Some concerns, like for example URL Pattern matching are cleanly modularized using inheritance. The URL pattern matching code is situated in two dedicated classes. As a result, this concern is very well modularized and certainly not crosscutting. However, the logging concern is a lot less localized. In fact, when considering the analysis of Figure 11, one might conclude that the logging concern is not modularized at all because it is contained in almost every class. Even within one class, the logging concern is spread all over the code of that class. As a consequence, the logging concern in the Tomcat system is clearly crosscutting. The critical reader might remark that this system was not properly designed from the beginning and that using good software engineering practices, this would not occur. However, an in-depth study of the source-code revealed that the logging concern is fundamentally crosscutting and that there is no way of refactoring the application to modularize the logging concern while keeping other concerns isolated.

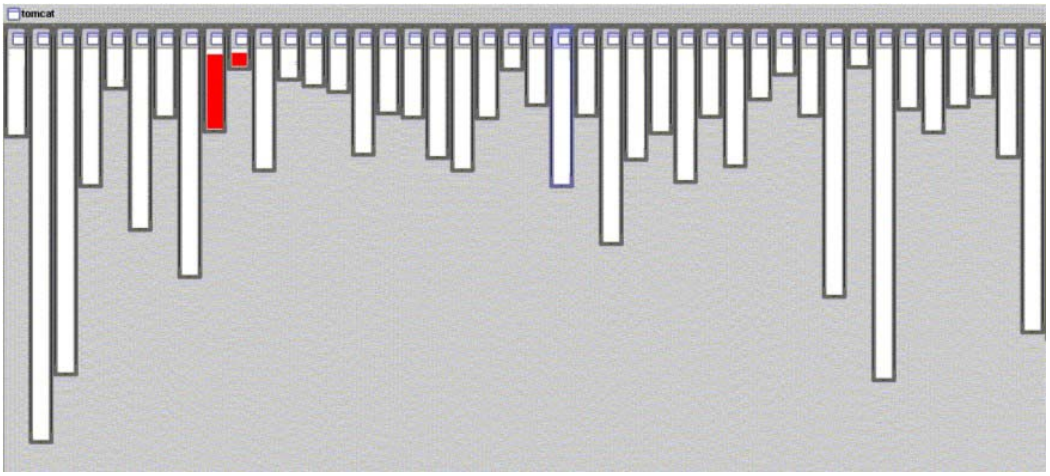

Figure 10: URL Pattern matching code only resides in two dedicated modules of the Tomcat system. Example taken from the AspectJ documentation.

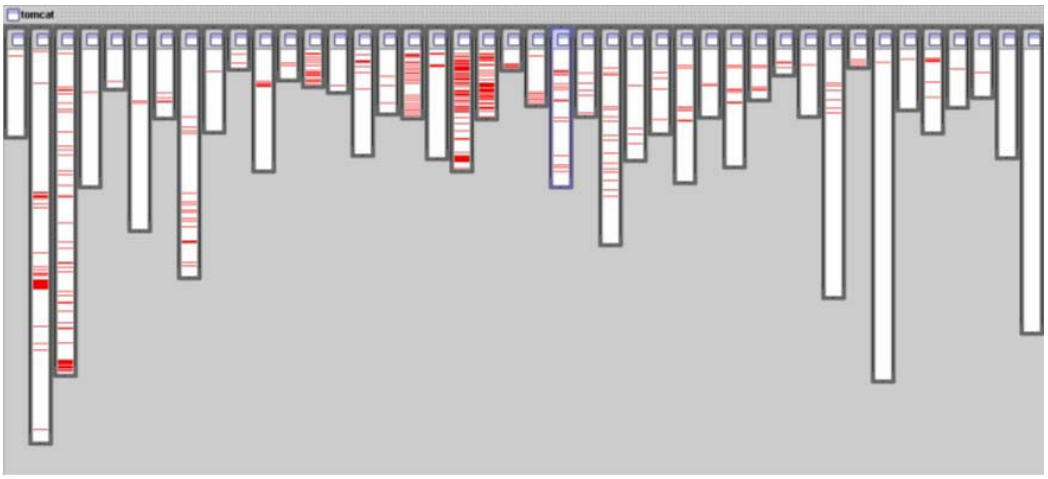

Figure 11: Logging code in the Tomcat system is spread over tons of places. Example taken from the AspectJ documentation.

Currently, aspect-orientation is mainly focused on object-oriented software development. In object-orientation, the main kind of structure to isolate a concern is the class hierarchy. This hierarchy only spans one dimension, so if we decompose our system so that one main concern or group of concerns is encapsulated nicely, some other concerns might be spread over different classes in the system. Of course, it is possible to redesign the application in order to fit the originally crosscutting concerns, but than the main concerns might very well become scattered among several classes. This problem is caused by the fact that object-orientation allows only a single dimension to decompose the features of a system. Tarr et al. [TOHS99] describe this as the "problem of the dominant decomposition". Aspect-orientation promises to solve this problem by introducing one or more additional dimensions to describe these crosscutting concerns. How this is realized in detail depends heavily on the employed aspect-oriented technology. The next section presents an overview of the four representative approaches, namely AspectJ, Composition Filters, Adaptive Programming and HyperJ. Most of the other aspect-oriented approaches are based on ideas of one or more of these approaches.

2.3.2 Overview of representative approaches

2.3.2.1 AspectJ

AspectJ [KHH+00] is the most well-known and mature aspect-oriented technology available today. AspectJ is an extension of the Java programming language and introduces an extra construct for specifying crosscutting concerns, conveniently called an aspect. AspectJ aspects are very similar to Java classes. They are able to inherit from other aspects and define methods and variables. In order to specify crosscutting features, AspectJ introduces two main methods which they call respectively static and dynamic crosscutting.

```
1  aspect StaticExample {
2
3     declare parents : ClasseA extends ClasseB;
4
5     declare parents : ClasseA implements Cloneable;
6
7     declare int ClasseA.x = 0;
8
9     declare int ClasseA.getX() { return x; }
10 }
```

Code Fragment 1: Static crosscutting in AspectJ.

Static crosscutting consists of adapting certain properties of classes like adding methods, superclasses, variables, etc… Code Fragment 1 illustrates an example aspect that introduces a method, a field, a super-interface and a superclass using the keyword *declare*. This code fragment respectively sets `ClassB` as parent of `ClassA` (line 3), adds `Clonable` to the interfaces `ClassA` implements (line 5), introduces a variable `x` in `ClassA` (line 7) and introduces a method `getX` in `ClassA` (line 9). Notice that all these additions are applied to the implementation of the target classes themselves. As a result, when a method is introduced, this method can be called like any other regular Java method.

Dynamic crosscutting at the other hand, allows changing the dynamical control-flow of the application. To this end, AspectJ introduces two additional constructs, called *pointcut* and *advice*. A pointcut defines a set of places (also called *joinpoints*) in the control-flow of a program that the aspect should influence. An advice is attached to a pointcut and implements the adaptations that should be executed to realize the desired aspect behavior. An example of dynamic crosscutting is shown in Code Fragment 2. The SimpleTracing aspect contains one pointcut *traceCall* (line 2-4) that denotes the joinpoint at the moment the *startJuggling* method is called onto the *Juggler* class. In addition, the aspects attaches a *before* advice (line 6-8) to the *traceCall* pointcut that prints a tracing string to the standard output. As a result, the *SimpleTracing* aspect specifies that some tracing string should be printed before the method *Juggler.startJuggling* is invoked.

```
1  aspect SimpleTracing {
2     pointcut traceCall() {
3        call(void Juggler.startJuggling());
4     }
5
6     before(): traceCall() {
7        System.out.println("Start Juggling");
8     }
9  }
```

Code Fragment 2: Simple tracing aspect that illustrates basic dynamic crosscutting.

In order to deploy aspects specified in the AspectJ language, a special kind of compiler, called an aspect weaver is required. An aspect weaver invasively merges the code specified in the aspects with target classes to end up with the desired application. The AspectJ weaver employs source-code weaving, which means that the aspects are inserted in the source-code of the target classes. Recently however, a first prototype of a byte-code weaver has been proposed. A byte-code weaver inserts the aspect behavior on byte-code level, eliminating the need for the source-code of target classes.

AspectJ provides more advanced features than discussed in this section. Pointcuts can be defined with in a very fine-grained and detailed way. For example, a pointcut is able to specify all executions of a certain method in the code of another method, but not in the control flow of yet another method. Also, regular expressions can be used to denote sets of methods, like for example all methods which name starts with "get". In addition, different types of advices exist, like for example, before, after, replace, after throwing, after returning and so on.

AspectJ is the first production quality general purpose AOP language. The AspectJ tools are released under a GNU LGPL license and are part of the eclipse.org project. Plug-ins are available for most common IDEs including of course Eclipse, Borland JBuilder and Sun's JavaOne Studio.

2.3.2.2 Composition Filters

The Composition Filters [BAT01] approach is quite different from AspectJ. The Composition Filters model allows expressing crosscutting concerns by attaching filters to existing classes. These filters implement modular, non-invasive and orthogonal extensions to the classes at hand. Here, modularitiy means here that the filters are completely independent of the classes on which they are applied. As a result, Composition Filters are highly reusable in comparison to AspectJ aspects. Non-invasiveness means that the filters implement extensions that do not alter the interior behavior of the target classes. Instead, only the incoming and outgoing messages are influenced. As a result, the Composition Filters model satisfies the wrapper Design Pattern [GHJV95]. Orthogonality means that Composition Filters do not refer to each other. In [BAT01], it is claimed that these three properties increase the adaptability and reusability of concerns because filters can be applied onto different programming languages and can be combined easily. The aspect-oriented expressiveness of Composition Filters is however strongly restricted if compared with for example AspectJ.

To illustrate the composition filter approach, consider the following example. In a medical system, several types of documents are used. The basic implementation of a document class is already available and needs to be extended to incorporate specific information about claims of clients. The additional information includes the motivation and size of a claim, the medical status and the approved claim amount. Of course, one could just solve this by using object-oriented inheritance, but then the specific behavior for handling claims would be captured in a subclass of the Document class. As a result, the claim logic needs to be repeated into other parts of the system when it is needed for non-document related purposes. This is an obvious example of a crosscutting concern and can be solved using the Composition Filter model. Figure 12 illustrates a schematic overview of this solution. A *ClaimDocument* class consists of attaching a *Dispatch* input filter to the *Document* class. In addition, a regular object-oriented class (implementation part in Figure 12) is used to implement the specific claim information. The dispatch input filter reroutes all messages that the implementation part is able to handle to this implementation part. Messages that are not understood by the implementation part end up at the normal *Document* class. As a consequence, this filter effectively implements specialisation of the *Document* class.

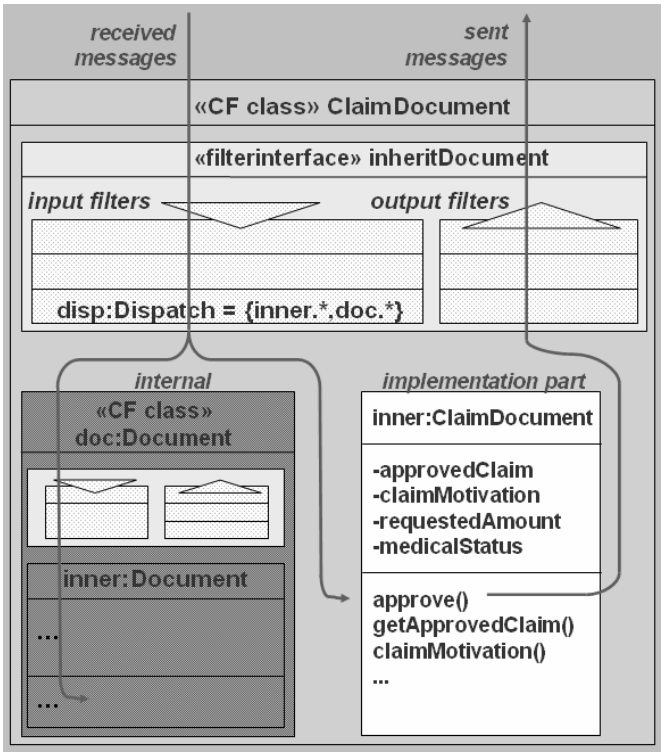

Figure 12: Composition Filter as an alternative for Object-Oriented inheritance. Picture adapted from [BA01].

In addition to input filters, the Composition Filter model allows to attach output filters to a class. Furthermore, multiple filters can be used for both input and output filters. Messages are first matched in the sequence the filters are specified. In other words, matching begins with the first filter, if this filter does not result in a method call, the next filter is matched. Classes that result from a Composition Filter application can again be used as a target for other filters. Recently, the concept of superimposition [BAT01] is introduced to the Composition Filter model to allow attachment of a filter to a range of target classes in one declaration. Before superimposition, a filter had to be applied on every target class separately, causing a lot of duplication of similar code.

In contrast to AspectJ, the Composition Filter approach is more an academic model instead of a practical language and corresponding tools. In fact, there is no stable, production quality implementation available for Composition Filters at this moment. There are however some prototype implementation languages introduced that support parts of the Composition Filter semantics, like for instance Sina for Smalltalk [Koo95] and ConcernJ for Java [Wic99, Sal01]. These languages offer a test-environment for practical experiments with the Composition Filter approach. A stable and full implementation of the Composition Filters approach targeted at the .NET platform is at this moment under development.

2.3.2.3 Adaptive Programming

The Adaptive Programming [Lie96] approach is, together with the Composition Filters approach, one of the first aspect-oriented approaches. Adaptive Programming originates from the Law of Demeter [LH89] that states in its most general form:

"Each unit should have only limited knowledge about other units: only units 'closely' related to the current unit."

In object-oriented systems this means, in short, that an object should only talk to objects it has direct references to. The Demeter rule is in fact a special case of the low-coupling principle and makes it very explicit. The Law of Demeter has been widely adopted by both academics and industry as a very important rule to follow. Adhering to the Law of Demeter is not always easy and sometimes even impossible. When an operation involves a set of cooperating classes, one can either localize the operation in one class or split the operation over the set of classes. Localizing the operation in one class causes hard-coded information about the structural relationships between these classes and as such a violation of the Law of Demeter. The other alternative, namely spreading the operation over the set of classes, conforms to the law of Demeter, but causes the logic of the desired behavior to be spread over different classes making evolution very difficult. This is in fact the relation with Aspect-Oriented Software Development: adhering to the Law of Demeter using regular object-oriented concepts might cause crosscutting concerns.

In order to solve crosscutting concerns originating from adhering to the Law of Demeter, adaptive methods are introduced. In [LOO01], an adaptive method is defined as follows: "An Adaptive method encapsulates the behavior of an operation in one place, thus avoiding the scattering problem, but also abstracts over the class structure, thus avoiding the tangling problem as well". An adaptive method consists of two main parts: a traversal strategy and an adaptive visitor. A traversal strategy is an abstract and high-level description on how the participants in the operation at hand are reached. The adaptive visitor visits the different classes required for the computation and takes care of the implementation itself. Code Fragment 3 shows an example of an adaptive method in the DJ library [OL01]. This method counts the total salary a company pays to all its employees. The string `s` declared at line 10 defines the traversal strategy to follow. The adaptive visitor is defined from line 12 till 19. It defines one before advice (line 15) that will adapt the total amount of salary paid (the `sum` variable) for every salary object that is visited. Using an adaptive method, how to reach *Salary* objects starting from a *Company* object is not hard coded. The company could have many subcompanies and divisions or could just contain a single employee. The adaptive method can be reused over all these possible company structures.

```
1  import edu.neu.ccs.demeter.dj.ClassGraph;
2  import edu.neu.ccs.demeter.dj.Visitor;
3
4  class Company {
5
6    static ClassGraph cg = new ClassGraph();
7
8    Double sumSalaries() {
9
10      String s = "from Company to Salary";
11
12      Visitor v = new Visitor() {
13        private double sum;
14        public void start() {sum=0.0}
15        public void before(Salary host) {sum+=host.getValue();}
16        public Object getReturnValue() {
17            return new Double(sum);
18        }
19       };
20
21       return (Double) cg.traverse(this,s,v);
22    }
23    // rest of company definition
24 }
```

Code Fragment 3: Example of adaptive method implemented using the DJ library and is taken from [LOO01].

Adaptive Programming is in fact complimentary to most aspect-oriented approaches because it captures a very specific kind of concern that most other AOP languages can not describe. In addition, DJ is a library that uses pure Java, so any aspect-oriented technology, or any other technology as a matter of fact that uses Java, can be combined with Adaptive Programming very easily. However, using a library often amounts to less comprehensible and longer code than using a dedicated language. Therefore, the DemeterJ language has been proposed that adds special language constructs to support adaptive programming to the Java language [LO97]. Recently, the DAJ [SL02, LL02] language has been introduced, which adds special language constructs for Adaptive Programming to AspectJ.

2.3.2.4 MDSOC and HyperJ

Multi-Dimensional Separation of Concerns (MDSOC) [TOHS99] is another very different kind of aspect-oriented approach that originated from subject-oriented programming [HO93]. The major goal of MDSOC is to allow developers to decompose their software so that it encapsulates all relevant kinds (dimensions) of concerns simultaneously, without one dominating the others [OT01].

HyperJ for Java is the current practical realization of MDSOC [OT00]. Contrary to AspectJ, which has an asymmetric model of base classes and aspects, HyperJ uses a symmetric model where no distinction is made between aspects and base implementation. Instead of separating only crosscutting concerns in an aspect, every concern (crosscutting or not) of an application is captured in a so called Hyperslice. In addition, HyperJ employs pure Java for describing Hyperslices, allowing easier integration of existing modules in comparison to dedicated languages as for example AspectJ. Different concerns described in hyperslices can be combined in a Hypermodule to reach the desired functionality. A hypermodule allows specifying a set of detailed rules on how the combination should be performed. Hypermodules are again subject to composition with other Hypermodules or Hyperslices.

Code Fragment 4 illustrates an example a Hypermodule that combines two hyperslices (line 3), named `Feature.Personnel` and `Feature.HR`. Lines 4 till 9 define a set of rules on how this combination needs to be performed. Line 5 states that entities from both Hyperslices that share the same name should be merged. Lines 6 till 7 state that the class `RegularMgr` from the first hyperslice has to correspond to `Manager` from the second hyperslice although their name is different. Finally, line 8 till 9 states that the name entities should not be merged, instead the name operation of the second hyperslice has to be used.

```
1  hypermodule PersonnelPlusHR
2    hyperslices:
3       Feature.Personnel, Feature.HR;
4    relationships:
5       mergeByName;
6       equate class Feature.Personnel.RegularMgr,
7          Feature.HR.Manager;
8       override operation Feature.HR.name
9       with Feature.Personnel.name;
10 end hypermodule
```

Code Fragment 4: Example of a hypermodule in the HyperJ language taken from [OT01].

HyperJ can automatically generate a composed class hierarchy from a declared Hypermodule. This class hierarchy integrates all concerns specified in the various hyperslices. Technically, HyperJ merges the hyperslices, which are essentially Java classes, using byte-code transformations. As a result, HyperJ is not dependent on the availability of source-code.

One of the interesting features of HyperJ is that it also allows to go the other way around, namely decomposing a class hierarchy into different hyperslices. This allows integrating already built systems into

the MDSOC approach very easily. Of course, in order to achieve this in practice, some user input is required. Code Fragment 5 shows a couple of declarations needed to decompose an existing class hierarchy into two separate and reusable class hierarchies. The package `Personnel` is assigned to the hyperslice `Feature.Personnel` except for the operations `position` and `pay`. These are assigned to the hyperslice `Feature.Payroll`. Notice that HyperJ automatically generates full and valid Java classes from a decomposition description, even if only one operation is extracted into another hyperslice.

```
1  package Personnel: Feature.Personnel
2  operation position: Feature.Payroll
3  operation pay: Feature.Payroll
```

Code Fragment 5: Decomposing a class hierarchy into two separate and reusable hyperslices. This example is taken from [OT01].

2.4 Combining AOSD and CBSE

Currently, aspect-oriented practice and research is mainly focused on the object-oriented paradigm. Aspect-oriented ideas are however also applicable in many other paradigms, including Component-Based Software Engineering (CBSE). As argued in section 1.1, component-based software development suffers greatly from crosscutting concerns as a lot of functionalities are spread and repeated into the different components in order to keep the coupling as low as possible. Typical examples of crosscutting concerns encountered in component-based systems are transaction management and security. Component-based ideas concerning highly reusable components and an explicit composition mechanism do however also contribute to the aspect-oriented field. Integrating a component-based plug-and-play concept and explicit composition mechanism in aspect-oriented software development, would make aspects reusable and their deployment dynamic, manageable and flexible. As such, combining ideas of component-based and aspect-oriented software engineering contributes to both paradigms. This section discusses the current state-of-the-art approaches that aim at achieving this combination. There are in fact two groups of approaches that combine component-based and aspect-oriented ideas. One group focuses on integrating aspect-orientation into specific component technologies or component models, while another group is more fundamental research that aims at combining aspect-orientation and modular systems.

2.4.1 Approaches aimed at component-based technologies

2.4.1.1 JAC

JAC is one of the first aspect-oriented approaches that targets component-based systems. JAC is an aspect-oriented framework that is introduced as an alternative of a J2EE application server [PDFS01, PSDF01]. J2EE application servers typically already provide a set of predefined non-functional services, such as transaction management and security, which are in fact crosscutting concerns. It is however not possible to add new services or alter the behavior of the existing ones. Therefore, JAC introduces extendable *wrappers* that are able to implement these non-functional services and can be attached to existing components. JAC wrappers are implemented using plain Java. Each wrapper extends the Wrapper class and implements some methods that wrap certain behavior. Code Fragment 6 illustrates a logging wrapper implemented in JAC. The `LoggingWrapper` class implements one method, named `logMethod`, that implements logging before and after a method is executed. JAC wrappers have to be employed in *aspect components* in order to apply them onto a given context. Code Fragment 7 illustrates the `MyAspect` aspect component that applies the `loggingWrapper` to all method executions in the system by invoking the `pointcut` method. By employing aspect components, JAC wrappers are independent of a concrete context and thus easily reusable. Another nice feature of the JAC approach is the meta-data that aspect components are able to attach to certain classes. Wrappers can exploit this meta-data when implementing the wrapping logic.

```
1   public class LoggingWrapper extends Wrapper  {
2
3      public LoggingWrapper(AspectComponent ac) {
4         super(ac);
5      }
6
7      public Object logMethod(Interaction interaction) {
8         Object ret=null;
9         System.out.println("Before "+interaction.method);
10        ret=proceed(interaction);
11        System.out.println("After "+interaction.method);
12        return ret;
13     }
14  }
```

Code Fragment 6: Logging aspect implemented as a wrapper in JAC.

```
1   public class MyAspect extends AspectComponent {
2
3       public MyAspect() {
4           pointcut("ALL","ALL","ALL", LoggingWrapper.class,
5                "logMethod",true);
6       }
7   }
```

Code Fragment 7: Connecting the logging aspect to a certain pointcut.

The JAC approach is easy to use from a Java point of view as no new language is introduced. The drawback is of course that JAC code is less readable and maintainable than aspects written in a specialized aspect-oriented programming language. Another limitation of this model is the poor support for controlling wrapper instantiation as this is handled by the system. Either a new wrapper instance is attached to every joinpoint or a single wrapper instance is attached to all joinpoints. It is also difficult to pass data to the wrappers from the system. The wrappers are first-class entities, but are shielded by the JAC system and thus not accessible. However, the most important drawback of JAC from a component-based point of view is that JAC is not compatible with J2EE. The authors state that it is meant as an alternative to J2EE and JAC is indeed an excellent alternative when it comes to implementing custom-made non-functional container services. However, the standard services of J2EE cannot be employed together with JAC and thus have to be re-implemented. JAC does already include a set of pre-defined aspects that are meant as a replacement of these container services, but they are quite simple in comparison to the feature-rich container services offered by most application servers. In addition, J2EE offers a lot more features than these non-functional container services that thus cannot be employed when using JAC.

2.4.1.2 JBoss/AOP

JBOSS/AOP [BCF+03] is an extension of the JBoss J2EE application server that overcomes the main shortcoming of the previous approach, namely incompatibility with J2EE. JBoss is an open-source J2EE application and claims to be the most used application server worldwide [Fle03, FR03]. The JBoss AOP extension allows implementing aspects as *interceptors* that are able to intercept all invocations to a certain range of components. Interceptors are able to intercept both method invocations and field accesses. An interceptor defines one standard method *invoke*, that receives all the invocations (including field accesses) of a certain component. Code Fragment 8 illustrates a logging interceptor that prints a log message before every constructor or method execution and before every field access. The log message consists of the type of invocation intercepted (method, constructor, or field) and the name of the intercepted invocation.

```
1   public class LoggingInterceptor implements Interceptor {
2
3     public String getName() {
4        return "MetaDataInterceptor";
5     }
6
7     public InvocationResponse invoke(Invocation invoc)
8             throws Throwable {
9        String lMessage = null;
10       if (invoc.getType() == InvocationType.METHOD) {
11          MethodInvocation mi = (MethodInvocation) invoc;
12          lMessage = "method: " + mi.method.getName();
13       }
14       else if (invoc.getType() == InvocationType.CONSTRUCTOR) {
15          ConstructorInvocation ci=(ConstructorInvocation) invoc;
16          lMessage = "constructor: " + ci.constructor.getName();
17       }
18       else {
19          FieldInvocation ci = (FieldInvocation) pInvocation;
20          lMessage = "field: " + ci.field.getName();
21       }
22
23       System.out.println(lMessage);
24
25       InvocationResponse rsp = pInvocation.invokeNext();
26       return rsp;
27    }
28  }
```

Code Fragment 8: LoggingInterceptor implemented in JBoss/AOP.

Interceptors have to be attached to a given set of components in an XML configuration file. Code Fragment 9 illustrates an example configuration file that attaches the `LoggingInterceptor` to all classes of which the name starts with *POJO*. It is also possible to specify more fine-grained pointcuts like only those methods of which the name starts with "set". JBoss/AOP allows to pass configuration data to the interceptor as illustrated by the logging-level XML tag at line 5. It is however the responsibility of the interceptor to interpret the associated XML tree. As such, no static type checking is possible and errors in the XML file might give raise to cryptic run-time error reports. The JBoss/AOP extension allows explicit control over interceptor instantiation by defining factory classes for one or more interceptors. Another contribution of the JBoss/AOP approach is that an explicit sequence of interceptors can be defined in the XML configuration file. JBoss/AOP also supports static crosscutting by introducing new methods and fields in selected target components. Likewise to JAC, meta- data can be attached to any given method, field or class and can be accessed in the interceptors.

```
1   <aop>
2      <interceptor-pointcut class="POJO.*">
3        <interceptors>
4          <interceptor class="LoggingInterceptor">
5              <logging-level>verbose<logging-level>
6        </interceptors>
7      </interceptor-pointcut>
8   </aop>
```

Code Fragment 9: LoggingInterceptor implemented in JBoss/AOP.

JBoss/AOP claims to be a dynamic AOP approach, but currently the dynamism is quite limited. JBoss/AOP only allows attaching or removing interceptors at run-time from components that already are subject to interceptor application. Other components cannot be affected. Furthermore, at run-time, the

XML configuration file cannot be altered; the only way to add or remove interceptors is by invoking the appropriate methods on the target components themselves. As such, attaching an interceptor to a set of components requires iterating over all component instances of these components and invoking the `appendInterceptor` method on every instance. Another drawback is that the XML configuration files do not have the expressiveness of a programming language as for example employed by JAC aspect components. The XML configuration files are also not so flexible because they have to have a fixed name ("META-INF/jboss-aop.xml"). As a result, it is not possible to structure and group different logically related interceptor specifications into a single entity, which causes a configuration file to become hard to read and difficult to maintain when a lot of interceptors are employed.

Although JBoss/AOP is meant as an extension of J2EE, it is also possible to employ the AOP approach separately without the JBoss application server. JBoss/AOP is currently one of the most used AOP approaches because of its low learning curve and good maturity. JBoss has also inspired other application server vendors to offer similar functionality; BEA has for example recently announced an AOP implementation for their WebLogic application servers.

2.4.1.3 DAOP

The Dynamic Aspect-Oriented Platform (DAOP) [PFFT03, PFT03] is a distributed platform where components and aspects are first-class entities that are dynamically composed at runtime using the composition information stored in a middleware layer. DAOP is proposed as an alternative for J2EE application servers and introduces a new component model. In this new component model, components never communicate directly, instead they communicate through a central middleware layer. Components send messages to other components by specifying a destination role name instead of a concrete type. As such, components are very loosely coupled and a component can easily be replaced by another component. This idea is in fact somewhat similar to the PacoSuite approach as there is a single orchestrating layer (the middleware layer in DAOP) that makes sure that incoming calls are sent to the correct destinations. Because there is a single middleware layer that receives all messages, aspects can be easily integrated in the approach. The middleware layer re-routes all messages where one or more aspects are interested in, to the corresponding aspects.

```
1   ...
2   <aspectEvaluationRules>
3      <aspectRule>
4        <compRole>whiteboard</compRole>
5          <inputAspects>
6              <messages>joins</messages>
7              <aspectList>accessControl</aspectList>
8              <aspectList>unmarshall</aspectList>
9          </inputAspects>
10         <outputAspects>
11             <messages>draws</messages>
12             <aspectList>marshall</aspectList>
13         </outputAspects>
14     </aspectRule>
15  </aspectEvaluationRules>
16  ...
```

Code Fragment 10: Aspect application specification in DAOP. Example adapted from [MT00].

DAOP employs a custom architectural description language (ADL) [MT00] in order to describe a complete component composition including aspects. Code Fragment 10 illustrates a fragment of an aspect composition in the DAOP-ADL language [PFT03]. In this fragment, the `accessControl` and `unmarshall` aspects are applied to the `joins` message received by the `whiteboard` component role.

In addition, the `marshall` aspect is applied to the `draws` message that is sent by the same `whiteboard` component role. Notice that DAOP-ADL allows specifying an explicit sequence of aspects for a certain message (line 6-8). Unfortunately, this sequence has to be re-specified for every message in the system. JBoss/AOP for example, allows reusing the same sequence of aspects over various pointcuts. The composition language of DAOP-ADL is also quite simple in comparison to those of JAC and JBoss/AOP. An aspect can only be attached to a certain message; more complicated pointcut expressions are not supported.

DAOP allows dynamic addition, replacement and removal of both aspects and components. In addition, aspects remain first-class entities at run-time. The main advantage of the approach is the very loose coupling among components and aspects. The main drawback of DAOP is however that DAOP is not backward compatible with any existing system. This is because a custom communication model is employed for inter-component communication. Instead of regular method invocations, both components and aspects have to explicitly communicate with the central middleware layer. In addition, both components and aspects have to inherit from DAOP specific classes. As such, existing systems have to be manually refactored in order to comply with the DAOP framework.

2.4.1.4 Dynamic Injectors

Filman et al. focus on separating crosscutting concerns in large-scale distributed component-based systems. They claim that most code is not devoted to implementing the desired input-output behavior but to providing system-wide properties like reliability, availability, responsiveness, performance, security, and manageability [FBLL02]. They call these kinds of qualities, *ilities*. They also observe that although current component-based technologies typically support these ilities, these technologies typically provide (1) only a finite number of choices for the application architect, and (2) require a good understanding and diligent application of the mechanism by the application programmer [Fil00]. In order to be able to separate these ilities in reusable modules, Filman et al. propose the concept of *dynamic injectors*. A dynamic injector is a kind of proxy component that can be placed on a communication channel between components and is as such able to inject the aspect's logic into a given component composition. Dynamic injectors are first class objects and can be added or adapted at run-time. It is possible to specify an explicit sequence of injectors in such a way that each component instance and even each method of a specific component instance can have a distinct sequence of injectors. Another important contribution of dynamic injectors is the support for communication between injectors by employing *annotations*. Annotations contain additional meta-information that injectors are able to pass along with requests. Injectors are capable of reading and modifying the annotations of requests and as such are able to communicate about requests. Examples of annotations are the users credentials associated with the request, priority of the requests, etc… The main advantage of the annotation system is that it allows communication between related injectors in a flexible and light-weight fashion. Without such an annotation system, the aspects have to communicate directly in order to pass aspect related information. This requires an extensive run-time architecture in order to fetch all injectors that act on this communication channel. The aspects are also distributed, which makes this direct communication even more complex. The drawback of the annotation system is of course that all injectors interested in a certain annotation have to agree on the name and semantics of that annotation. In addition, no type checking is possible and as such, possible mismatches cannot be detected and might cause cryptic run-time error reports.

Dynamic injectors are incorporated into OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. In CORBA, each component has one or more associated *stubs* for outgoing communication and *skeletons* for incoming communication. OIF generates

enhanced stubs and skeletons that are able to incorporate one or more injectors. Figure 13 illustrates four different injectors that are attached at both the client and server side of a communication channel. The injectors can be added, altered and removed at run-time. Because the injectors act as a kind of proxy outside of the components, they are only able to alter the exterior behavior of a component by changing request arguments or filtering requests. Injectors in OIF are also not able to alter the destination of a given request; they merely act as a sequence of filters.

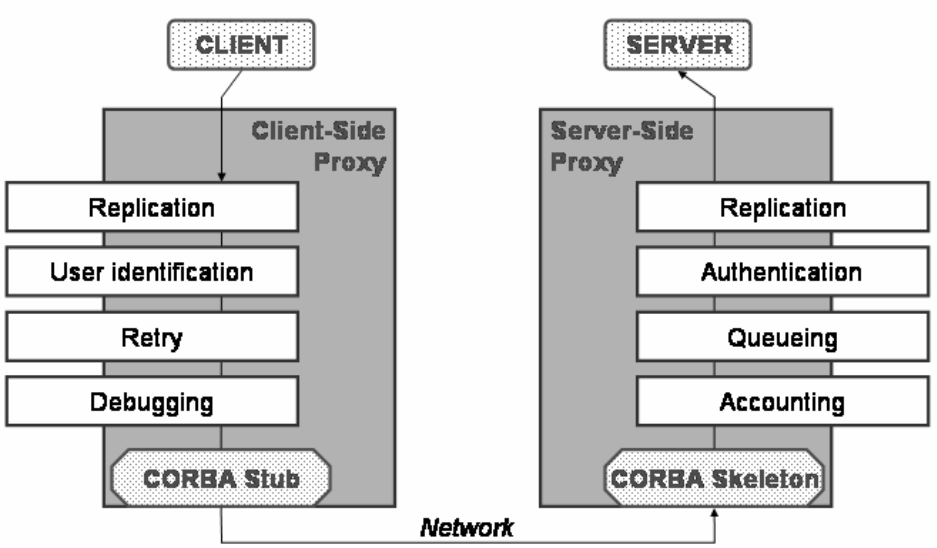

Figure 13: Dynamic injectors attached at both the client and server side of a communication channel. Picture adapted from [FBLL02].

2.4.1.5 CCM/AOP

Duclos et al. propose a different approach to modularize crosscutting concerns in a CORBA context [DEM02]. Their approach is an extension of the Corba Component Model (CCM). A component in the Corba Component Model consists of one *abstract definition* and one or more *implementation descriptions* that each target different platforms or offer different quality of service. CCM/AOP extends this component model by introducing the container concept known from enterprise Java Beans. Component instances live in one or more containers that are able to offer several non-functional concerns, as for example security or transaction management. For specifying these containers, CCM/AOP introduces two languages: an *aspect definition language* and an *aspect user language*. The aspect definition language is able to describe a container in an abstract way. The aspect user language is able to concretize such a container by connecting it to concrete components. From a given aspect definition and aspect usage specification, the CCM/AOP technology generates individually tailored CCM containers. A container includes the aspect's logic as specified by the aspect definition. The CCM/AOP technology is in fact a kind of wrapping approach because aspects are only able to influence the exterior behavior of a component. The containers cannot be dynamically generated, so run-time aspect attachment and removal is not possible. In [DEM02], Duclos et al. claim that their approach is limited to aspects that describe non-functional concerns, although there is no apparent reason that other aspects could not be described.

2.4.2 Aspect-oriented research aimed at generic component-based systems

2.4.2.1 Aspectual Components

Aspectual components is one of the first approaches that recognizes the benefits of combining aspect-oriented and component-based ideas [LLM99]. The Aspectual component approach identifies the following two main problems with current aspect-oriented approaches:

- Current aspect-oriented technologies like for example AspectJ do not allow describing reusable aspects. The context where the aspect is applied, is hard-coded in the aspect.
- There is no support for explicit composition, which makes it very difficult to manage the collaboration of several aspects. Another major problem that is caused by the lack of an explicit composition mechanism is the ability to understand programs in a modular manner [LLO03]. If each aspect can possibly be attached to every joinpoint in the system, one has to discover and understand all aspects globally in order to be able to understand a single module.

In order to solve these deficits, the aspectual component approach proposes the notion of explicit connectors that are able to connect aspects to a concrete context. The connector concept allows aspects to be unaware of their concrete deployment context, increasing their reusability. In addition, a connector is able to describe an explicit composition. The aspectual component model itself is based upon the Adaptive Plug&Play component model [ML98]. An aspectual component is specified in terms of its own formal class graph, called participant graph. This reduces the knowledge that is required about the structure and functionality of the application as well as other aspects with which it is going to be combined. Each participant in the participant graph consists of a set of expected features, a set of reimplementations (aspectual behavior) and a set of regular object-oriented operations. In addition, an aspectual component is also able to contain local classes, data and operations.

```
1   component EncryptionComponent {
2     participant DataToAccess
3        expect String getData();
4        replace String getData() {
5           String res =  expected();
6           return encrypt(res);
7        }
8        private String encrypt(String s) {
9           //encryption implementation
10       }
11    }
12  }
```

Code Fragment 11: Simple encryption aspectual component. Example adapted from [LLM99].

Code Fragment 11 illustrates a simple encryption aspectual component containing one participant, named `DataToAccess`. The `DataToAccess` participant expects a single method `getData` and replaces it so that the `getData` method returns an encrypted result instead. Notice that this aspectual component is completely independent from a concrete deployment context. A connector is employed to connect the aspectual component to a concrete context. Code Fragment 12 illustrates a connector that connects the getData expected method of the DataToAccess participant of the `EncryptionComponent` aspectual component to all methods which name starts with `get` of the `Network` component.

```
1   connector EncryptionConnector {
2      Network is EncryptionComponent.DataToAccess with {
3           getData=get*;
4      }
5   }
```

Code Fragment 12: EncryptionConnector that connects the EncryptionComponent of Code Fragment 11 to a Network component. Example adapted from [LLM99].

Connectors also allow to explicitly compose several aspectual components. For example, suppose the encryption concern and a logging concern both have to be applied at the same network component. The logging has to happen at all methods of the network component, while the encryption concern is still only applied at all the getter methods. However, when both concerns are both applicable, the logging concern has to be executed first. Aspectual component connecters are able to specify such behavior as illustrated by the connector of Code Fragment 13.

```
1   connector EncryptionConnector {
2      Network is LoggingComponent.DataToLog with {
3           toLog=*
4      } is EncryptionComponent.DataToAccess with {
5           getData=get*;
6      }
7   }
```

Code Fragment 13: Connector that composes a logging and encryption concern to the same Network component. Example adapted from [LLM99].

The aspectual component model is implemented by a prototype byte-code weaver [LLM99]. This weaver takes as input a set of aspectual components and one or more connectors. For each declared method that is expected by some aspectual component, an observer system is inserted. When the method is executed, all interested observers are notified. The weaver makes sure that the defined aspectual methods are attached to the correct expected methods using this observer system. Herrmann et al [HM01] introduce an alternative implementation of the aspectual components model using Lua [IFC96].

The main contribution of the aspectual component model is the novel connector notion allowing highly reusable aspects. Several of the recent aspect-oriented technologies, including production software like for example JBoss/AOP [BCF+03] and AspectWerkz [BV03], are inspired by this idea and also introduce a connector concept.

A limitation of the aspectual component model is that the aspect-oriented expressiveness is quite limited in comparison with AspectJ. For example, AspectJ allows attaching aspects on dynamic control-flow conditions, which aspectual components cannot specify explicitly. Another limitation is that the composition mechanism cannot describe more involved combinations of aspectual behavior, like for example that when a certain aspect is applicable, another cannot be. In [Ovl04], it is also claimed that the semantics of the aspectual components model is not always clear. In some cases, attachments based on wildcards cannot be unambiguously interpreted.

2.4.2.2 Aspectual Collaborations

Aspectual Collaborations [LLO03, Ovl04] builds upon the results of aspectual components and aims at combining both the benefits of module systems and aspect-oriented software development. The main goal of aspectual collaborations is to enable modular reasoning in the presence of AOP. Therefore, a better aspectual modularity than possible using current aspect-oriented approaches has to be achieved and type-safe composition, even for aspectual behavior, is necessary.

Likewise to an aspectual component, an aspectual collaboration consists of a set of participants that describe the behavior of the collaboration. Aspectual Collaborations offer strong encapsulation as both the expected and provided functionality is explicitly described in each participant. Code Fragment 14 illustrates a simple aspectual collaboration that implements a caching aspect. The collaboration consists of a single participant, named `Cached`. The caching aspect expects a method `getName`, which is used for logging purposes. This method is provided with an implementation when the caching aspect is composed.

```
1   collaboration caching;
2
3   participant Cached {
4
5         ChdRetVal cachedValue;
6         expected String getName();
7
8         public void clearCache() {{
9             System.out.println(getName()+"clearing cache");
10            cachedValue = null;
11        }}
12
13        aspectual ChdRetVal cache(ChdMth e) {{
14            if (cachedValue==null)
15                cachedValue = e.invoke();
16            return cachedValue;
17        }}
18  }
```

Code Fragment 14: Aspectual Collaboration that implements caching behavior. Example adapted from [LLO03].

Aspectual Collaborations offer aspect-oriented support by aspectual methods, which are able to intercept the execution of other methods. In the `caching` aspectual collaboration of Code Fragment 14, the actual caching behavior is implemented by an aspectual method. This aspectual method caches the return type of the intercepted method. One of the major contributions of the aspectual collaboration model is the introduction of reification types. Return types and arguments of aspectual methods are not real types, but automatically generated types, existentially quantified over the context of the collaboration. This allows an aspectual method to be completely oblivious of the concrete method signature of the intercepted method. For example, the `ChdRetVal` and `ChdMth` types are reification types that are automatically generated. The `invoke` method (line 15) is supported per default by reification types and causes the original method invocation to be executed. By invoking other methods or accessing members of such a reification type, the context on which the aspectual method can be applied, is however limited to types that offer the required members or methods. This system does allow type safe composition, as the reification types can be matched with concrete types at composition time. A lot of other aspect-oriented approaches (most notably AspectJ) do not have this property. Some typing errors cannot be detected statically in these approaches, which causes cryptic run-time error reports.

In order to compose aspectual collaborations, the concept of external assembly is borrowed from aspectual components, but however differently realized. Instead of an explicit connector construct, an aspectual collaboration itself is able to compose several child aspectual collaborations. This mechanism removes the need for an explicit connector concept, as composite aspectual collaborations function as a kind of connector, while keeping all the advantages of a first-class composition mechanism. The main advantage of this hierarchical composition concept is of course that a composite aspectual collaboration can again easily be subject of composition. In addition, the aspectual behavior is composed with other aspectual collaborations using the same composition mechanism as regular behavior is composed. As such, the

application design is greatly simplified; the only main modularization construct available is an aspectual collaboration.

```
1  collaboration containerSizeCaching;
2  extends basicContainer;
3  attach caching {
4    Container+=Cached {
5       provide getName with toString;
6       around getSize do cache;
7    }
8  }
```

Code Fragment 15: Aspectual Collaboration that composes the caching collaboration with a basic container implementation. Example adapted from [LLO03].

Code Fragment 15 shows a composite aspectual collaboration that composes the `caching` collaboration with a `basicContainer` collaboration. The `caching` collaboration is employed to cache the result of the `getSize` method, which is implemented naively by iterating over all elements in the `basicContainer` collaboration. The collaboration also illustrates that a composite aspectual collaboration is able to compose both regular behavior, namely the `getName` method, and aspectual behavior, namely the `caching` behavior. Notice that in order to implement an effective caching system for computing the size of a container, the cache has to be invalidated when elements are added or removed. This is omitted in the example for the sake of simplicity.

The aspectual collaborations model offers a lot more features that are not illustrated here. Composite aspectual collaborations are also able to explicitly export functionality of the child collaborations as part of the public behavior of the composite collaboration. Furthermore, composite collaborations are able to contain participants like atomic collaborations. Another important contribution of the aspectual collaboration approach is that is based upon a formal model, which has been proven to be sound.

The main drawback of the aspectual collaborations model is that although it introduces only one main modularization construct, it is quite a complex model and not as intuitive as for example AspectJ. The Aspectual Collaborations model requires a different way of structuring, composing and implementing an application and is as such not so straightforward to learn and comprehend as more conservative extensions to traditional development paradigms. The aspectual power is also somewhat weaker in comparison to AspectJ. In order to realize type safety, wildcards cannot supported in attach clauses. Most of the other typical aspect-oriented features could probably be integrated in aspectual collaborations though. Another limitation of aspectual collaborations is that it is a static approach. Aspects cannot be added or removed dynamically.

2.4.2.3 Caesar

Caesar is also based upon the aspectual component model, but separates the interface declaration from the actual implementation. Caesar is proposed to improve the modularization of aspect code in comparison with the joinpoint interception mechanism employed in AspectJ [MO02]. In [MO03], the following deficits are identified in the joinpoint interception approach employed by AspectJ:

- Lacking support for multi-abstraction aspects
- Lacking support for a sophisticated mapping
- Lacking support for reusable aspect bindings
- Lacking support for aspectual polymorphism

```
1   interface ObserverProtocol {
2       interface Subject {
3           provided void addObserver(Observer o);
4           provided void removeObserver(Observer o);
5           provided void changed();
6           expected String getState();
7       }
8       interface Observer {
9           expected void notify(Subject s);
10      }
11  }
```

Code Fragment 16: ACI for an observer protocol. Code fragment adapted from [MO03].

In order to overcome these problems, the Caesar model is introduced. The core feature of Caesar is the notion of an *Aspect Collaboration Interface* (ACI). An ACI is an abstract interface definition with mutually recursive nested types. As such, multiple abstraction interfaces are specified. Each nested interface describes its expected and provided methods. The same idea, namely an expected and provided interface, is also employed in many component description languages like for example CoCoNut [Reu03]. Code Fragment 16 illustrates the ACI for an observer protocol. The `ObserverProtocol` ACI contains two nested interfaces: `Subject` and `Observer`. The `Subject` interface provides the `addObserver`, `removeObserver` and `changed` methods and expects a `getState` method. The `Observer` interface only expects the `notify` method.

An ACI's provided and expected methods are implemented in different modules, called *aspect implementations* and *aspect bindings* respectively. An aspect implementation implements all provided methods in the ACI's interfaces while an aspect binding is responsible for binding the expected methods to concrete methods in the core application. Code Fragment 17 illustrates an example aspect implementation for the ACI of Code Fragment 16. Notice that the `Observer` interface is not implemented because it contains only expected methods.

```
1   class ObserverProtocolImpl implements ObserverProtocol {
2      class Subject {
3         List observers = new LinkedList();
4         void addObserver(Observer o) {
5             observers.add(o);
6         }
7         void removeObserver(Observer o) {
8             observers.remove(o);
9         }
10        void changed() {
11            Iterator it = observers.iterator();
12            while ( iter.hasNext() )
13               ((Observer)iter.next()).notify(this);
14        }
15     }
16  }
```

Code Fragment 17: Simple implementation of the ACI of Code Fragment 16. Code fragment adapted from [MO03].

Code Fragment 18 illustrates a possible aspect binding for the ACI of Code Fragment 16. This aspect binding implements a color observer that prints a message each time the color of a drawing object is changed. For the `Subject` interface, two bindings are available, one for a `Line` class and one for a `Point` class. The `Observer` interface is only bound to a single class, namely `Screen`. Bindings are also able to implement a combined pointcut and advice that are able to activate the provided interface of

the ACI. In this example, the changed method of the `PointSubject` or `LineSubject` is invoked when a `setColor` method is invoked on respectively a `Point` or `Line` object.

An aspect binding and aspect implementation of an ACI have to be composed in a so-called *weavelet composition*. Code Fragment 19 illustrates a weavelet composition for the observer ACI of Code Fragment 16 that composed the `ColorObserver` aspect binding with the `ObserverProtocolImpl` aspect implementation. The weavelet composition also employs the `deploy` keyword, which means that the composition has to be statically weaved. It is however also possible to dynamically deploy a composition using *deploy blocks*.

```
1   class ColorObserver binds ObserverProtocol {
2      class PointSubject binds Subject wraps Point {
3         String getState() {
4            return "Point colored "+wrappee.getColor();
5         }
6      }
7      class LineSubject binds Subject wraps Line {
8         String getState() {
9            return "Line colored "+wrappee.getColor();
10        }
11     }
12     class ScreenObserver binds Observer wraps Screen {
13        void notify(Subject s) {
14           wrappee.display("Color changed: "+s.getState());
15        }
16     }
17     after(Point p): (call(void p.setColor(Color)))
18         { PointSubject(p).changed(); }
19     after(Line l): (call(void l.setColor(Color)))
20         { LineSubject(l).changed(); }
21  }
```

Code Fragment 18: A possible aspect binding for the ACI of of Code Fragment 16. Code fragment adapted from [MO03].

```
1   deploy class CO extends
2         ObserverProtocol<ColorObserver,ObserverProtocolImpl>{};
```

Code Fragment 19: Weavelet composition for the ACI of Code Fragment 16. Code fragment adapted from [MO03].

One of the major contributions of the Caesar approach is the support for aspectual polymorphism. Aspect bindings are able to implement a binding for different types and the concrete binding is resolved dynamically using the type of the object at hand. As such, aspects are polymorphic with respect to the type which they are applied to. Furthermore, both aspect implementations and bindings can be reused. Of course, the drawback of this sophisticated model is that it is a lot more complex than for example the AspectJ aspects. Another drawback of the approach is that aspect bindings and implementations can only be combined and reused when they both implement the same ACI. It is for example perfectly possible to come up with a different observer ACI. Aspect bindings and implementations for this second ACI cannot be combined with those for the first ACI.

2.4.2.4 Event based AOP

Event based AOP is based on a formal model for representing crosscutting interactions [DFS02]. One of the main advantages of this formal model is that it allows to statically analyze possible conflicts between aspects, which is a major contribution to solving the so-called feature interaction problem. Event based AOP introduces the idea of execution monitors as a framework for AOP [DMS01]. An execution monitor

has a global view of the complete program execution. The monitor observers the execution of the base program and intercepts *events* emitted during this execution. An event is a point in the execution in the base program and is as such similar to a joinpoint in AspectJ. An event contains information about the nature of this point (e.g. method call or exit) and the dynamic context (e.g. arguments or control flow). An *aspect* is a pure Java program which is able to take events as input. Aspects in event-based AOP can be seen as event transformers [DS02]. An event is processed sequentially by every aspect in the system and aspects are able to modify the event before it is passed to the next aspect. The modifications to the event are also reflected in the base program. When an aspect changes the arguments of a method call event, these altered arguments are passed to the actual method call instead of the original arguments. As such, EAOP allows specifying a more explicit transformation than typical AOP approaches as for example AspectJ.

Another important contribution of EAOP is the support for explicit aspect composition. Aspects can be composed using binary composition operators. The following composition operators are currently defined [DS02]:

- `Seq:` propagates the current events (coming from the parent) to its first operand and then to its second operand.
- `Any:` propagates the event to its two operands in an arbitrary order.
- `Fst:` propagates the event to its first operand, and then, if and only if the first operand did not detect a crosscut, the event is forwarded to its second operand.
- `Cond:` propagates the event to its first operand, and then, if and only if the first operand did detect a crosscut, the event is forwarded to its second operand.

It is of also possible to extend this basic set of composition operators by custom composition operators.

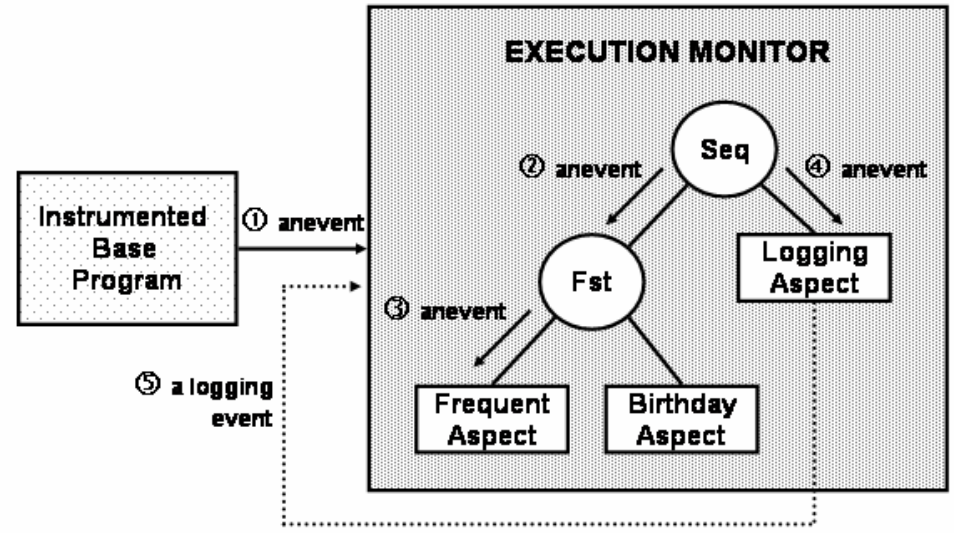

Figure 14: Aspect composition in EAOP. Figure adapted from [DS02].

Figure 14 illustrates an example aspect composition in EAOP. The example application consists of an e-commerce shop. The left-hand side of Figure 14 represents the example application, which is instrumented to submit events to the execution monitor. The execution monitor contains three aspects: a frequent customer discount aspect, a birthday discount aspect and a logging aspect. The frequent customer discount aspect gives a discount for frequent customers while the birthday discount aspect gives a discount when

the customer has its birthday. The aspect composition specifies that the birthday discount cannot be executed when the frequent customer discount is applicable by employing the `Fst` composition operator. After either one or none of the discount aspects have executed, a logging aspect is executed. This is specified by employing the `Seq` operator. EAOP also allows defining aspects of aspects. In Figure 14 for example, the logging aspect also submits an event to the execution monitor, which might be processed by other aspects.

A prototype implementation of the EAOP model is currently available [DS02]. Technically, this tool instruments the base program to emit events to the execution monitor by source-code transformations. The following events are currently supported: method call and exit, and constructor call and exit. The tool also allows dynamically adding and removing aspects.

One of the limitations of the EAOP model and implementation is that the model is sequential. As such, concurrent applications cannot be supported because all aspects are always executed sequentially, which causes the concurrent threads to synchronize at all event points. In addition, a central execution monitor might be a bottleneck in distributed environments. Achieving a decentralized set of execution monitors should however not be a fundamental problem. The current EAOP implementation depends on the source-code of the base program. This might be a problem, especially in a component-based context where source-code is rarely available.

In the PhD thesis of Andrés Farías [Far03], an extension of the formal model behind EAOP is presented for specifying protocol-based aspects. Both the pointcut and action language are extended to cope with protocol-based issues. The main contribution of this formal model, namely static analysis of possible conflicts, is preserved. Currently, the approach is mainly theoretical; there is no practical realization of the extended model.

2.4.2.5 Invasive software composition

Invasive software composition [Aßm03] is a component-based approach that unifies several software engineering techniques, such as generic programming [CE00], architecture systems [MT00] and aspect-oriented programming. Invasive Software Composition aims at improving the reusability of software components. To improve reuse, this method regards software components as greybox components and integrates them during composition.

The invasive composition model introduces a *box* as a generic and programming language independent component. A box contains *hooks* that are the variation points of a component. Two different types of hooks exist: implicit or declared hooks. An implicit hook is a hook offered by the programming language, like for example a method declaration. Declared hooks are hooks explicitly declared by the box implementer, like for example a specific method invocation inside a certain method. Hooks are composed in so-called *composer* programs that are responsible for composing the hooks of different boxes. Composer programs are written in the host programming language and are thus very expressive. In order to realize the composition described by a composer, invasive software composition proposes to adapt the boxes themselves at the implicit and declared hooks. This in contrast to typical component composition approaches (including PacoSuite) that generate glue-code for enabling cooperation between components. Invasive software composition alters the components themselves in order to enable collaboration. Because invasive software composition supports a wide range of possible hooks, typical aspect-oriented compositions, like for example logging at every method entry, can also be described. In a sense, invasive software composition generalizes both component-based and aspect-oriented composition.

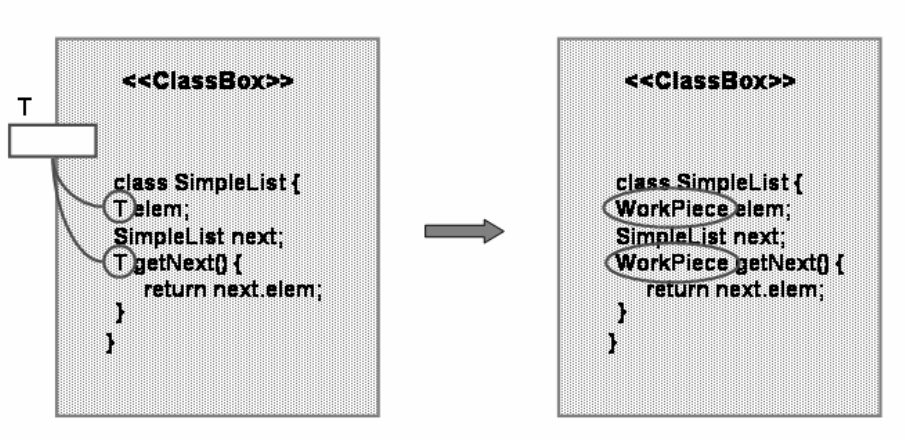

Figure 15: Template composition in invasive software composition. Example adapted from [Aßm03].

Figure 15 illustrates a classbox that specifies one declared hook to be a template type parameter. The declared hook conveys the knowledge about where to parameterize a generic class with an actual type parameter. By employing a composer program, the declared hook can be bound to a concrete type. In Figure 15 for example, the `T` hook is bound to the `WorkPiece` type. The result is that the classbox is invasively altered to include the `WorkPiece` type instead of type `T`.

Figure 16 illustrates an example of an aspect-oriented composition. The box contains two implicit hooks, method entry and method exit. A composer program defines that method exit has to be bound to the `info` method invocation on the `Debug` box. The result of this composition is that the invocation is inserted at all joinpoints defined by the method exit hook.

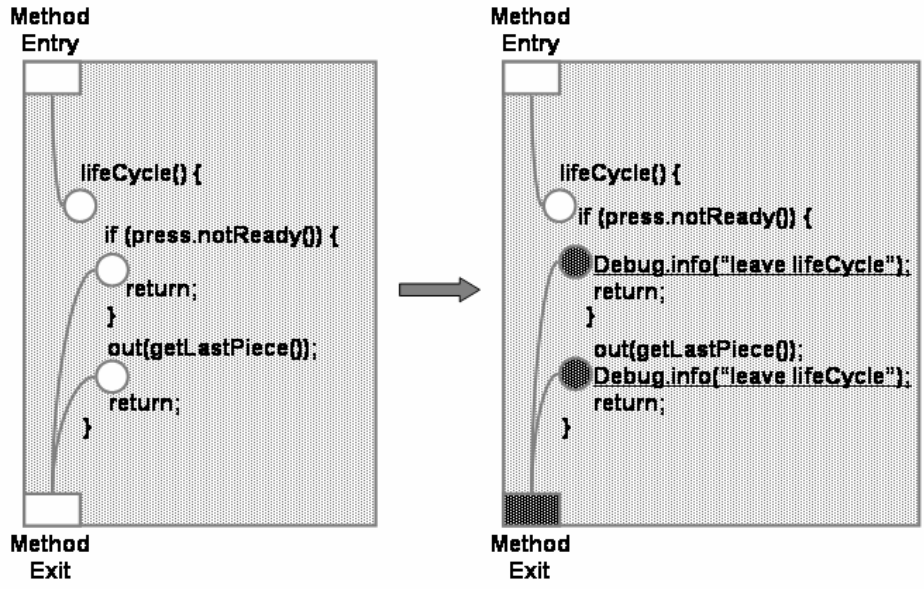

Figure 16: A method with two implicit hooks. After composition, the method exit implicit hook is bound to a logging invocation. Example adapted from [Aßm03].

One might wonder whether adapting components to make them work together is safe in the sense that the components still perform "valid" behavior. In order to validate this, invasive software composition introduces the concept of sound extension and sound composition. Sound extension means that the

semantics of the component is not changed, only extended. Sound composition means that the component is still able to fulfill a set of predefined contracts. Invasive software composition proposes an approach that is able to validate whether these properties hold for a given composer program.

The current technical realization of invasive software composition is the COMPOST library [LH00], which employs source code transformations in order to implement a composer program. COMPOST merges the components represented as boxes to one unified application. Undoubtedly, merging components renders a very efficient result. The drawback is that components loose their identity at run-time. Dynamic composition is also not possible with this approach. Another drawback of the approach is that it relies on the availability of source-code, which is seldomly available in component-based software engineering.

Chapter 3 Composition Adapters

In this chapter, the composition adapter model is proposed as a means to modularize tangled and crosscutting concerns in the PacoSuite methodology. A composition adapter allows describing transformations of a composition pattern. The transformation can be automatically applied using algorithms based on finite automata theory. The first section motivates the need for a composition adapter in PacoSuite. The second section explains the composition adapter model in detail. Section 3.3 formally defines an algorithm that allows to automatically apply the transformations described by a composition adapter onto a given composition pattern. Finally, section 3.4 introduces an algorithm to automatically resolve mappings of composition adapter roles on composition pattern roles.

3.1 Motivation

The component-based composition methodology we use as a starting point in this dissertation is the PacoSuite methodology discussed in section 2.2. In PacoSuite, several concerns exist that can not be nicely confined into a single and reusable component. Take for instance the *ToggleControl* composition elaborated in section 2.2.2. If we want to introduce debugging behavior that logs every message sent, two different approaches are possible using the normal PacoSuite entities: adapting the components or adapting the composition pattern.

One possible solution consists of adapting both the *Juggler* and *JButton* components to incorporate the extra logging behavior. Of course, this is quite a drastic approach and in most cases even impossible because components are black-box and generally no source-code is available. In addition, incorporating the logging behavior in every component that requires logging causes the logging concern to become duplicated and spread over all components. In the end, components become very heavy-weight entities, because every possible foreseeable extra functionality that is basically not the core functionality of the component has to be included. First of all, it is impossible to foresee all possible extra functionalities that are going to be used when evolving the components. Secondly, these concerns are effectively tangled with the components. An explicit test has to be included that decides whether a concern should be executed or not, at all points where the concern is applicable.

Another possible solution in PacoSuite consists of adapting the ToggleControl composition pattern to include the logging functionality. Figure 17 illustrates an example of how this could be achieved. An extra role, named *Logger*, is added and every message sent between the other roles causes a notification being sent to the *Logger* role. This solution requires adding an extra construct to the PacoSuite composition pattern syntax to denote the fact that one single event results in two messages sent. This is depicted with the bold line between START - NOTIFY; and STOP - NOTIFY. While this new construct is feasible to realize, it does not solve the problem that every composition pattern has to be manually adapted to incorporate the extra logging behavior. When the application goes into production, this logging behavior is not required any longer, and has to be manually removed again. As such, two different versions of each composition pattern are required for the logging concern alone. When confronted with other crosscutting concerns, a wealth of composition patterns is required in order to implements all those concerns. This is because all combinations of concerns are possible. As such, an exponential amount of alternatives are required for every basic composition pattern. Managing all these different versions of the same composition pattern is of course unfeasible. For example, if the basic logic of the composition pattern has to be altered, all these versions have to be altered too.

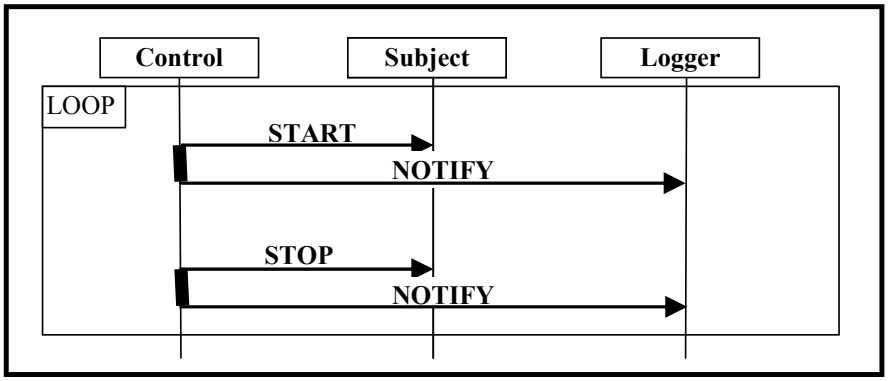

Figure 17: Adding an extra role to the composition pattern of Figure 4 (page 44) to enable logging behavior.

None of the two solutions presented above provide a satisfying solution for modularizing the logging concern and other similar concerns. As a result, component composition in PacoSuite is not so easy and flexible and requires in-depth knowledge of the components at hand when confronted with such crosscutting concerns.

Aspect-oriented ideas are thus very welcome in the component-based world and PacoSuite in particular, because AOSD allows specifying functionalities in a modular way without increasing the coupling between the modules. Current mainstream aspect-oriented approaches can however not be integrated straightforwardly in PacoSuite without lowering the abstraction level of component composition in PacoSuite. Indeed, PacoSuite allows to automatically verify compatibility of a component with a given composition pattern role. In addition, automatic glue-code generation is possible. This means that the following requirements have to be fulfilled for integrating aspect-oriented support in PacoSuite:

- It has to be possible to visually represent the aspects in PacoSuite and compose them graphically with a given component composition. A component composition consists of one or more composition patterns with filled-in roles.
- The tool has to be able to automatically verify whether the aspect is valid for the given component composition.
- If the aspect is valid for the given composition, the adaptations described by the aspects have to be automatically "weaved" into the target component composition.
- Checking compatibility of a component with a composition pattern role and automatic glue-code generation still have to be possible when aspects are present.

When the above requirements are fulfilled, the abstraction level of component composition augmented with aspect-oriented ideas is not lowered in comparison to the original PacoSuite and the PacoSuite development process is maintained as far as possible. Hence, component and aspect composition is possible without requiring in-depth knowledge of the involved components and aspects.

Current aspect-oriented research and practice are however mainly focused at providing new aspect-oriented programming languages and technologies. Integrating aspect-oriented ideas at a component-based composition level similar to PacoSuite is up till now unexplored. Therefore, a new kind of construct, called a composition adapter, is introduced in PacoSuite. A composition adapter is able to represent crosscutting concerns while satisfying the above requirements. The next sections present this model in more detail.

3.2 Composition Adapter Model

To solve the problems described in the previous section, we propose the concept of a composition adapter. The following section introduces the composition adapter model using an example. The subsequent sections present advanced features of the model.

3.2.1 A simple example

The running example used in this dissertation consists of dynamically validating timing contracts. This means that the application validates predefined timing contracts at run-time. An example of a timing contract is the following: *"the time between event A and event B has to be smaller than 100 ms"*. In order to achieve this, timestamps have to be taken of application events A and B. Similar to the logging concern, two different solutions are possible, namely either adapting all involved components or all involved composition patterns. Suppose the timestamping concern is implemented by adapting all involved composition patterns. This means that an extra role is introduced that is notified of the required events in the application. The component mapped onto this role is then able to take timestamps of these events. The timestamping logic is clearly crosscutting as it possibly affects all composition patterns in the system. As such, all composition patterns have to incorporate the timestamping logic in order to be able to validate timing contracts. Therefore, editing or removing the timestamping concern becomes a cumbersome and error-prone task as it is spread and duplicated over all composition patterns in the system.

In order to modularize crosscutting concerns in PacoSuite, the composition adapter model is proposed. A composition adapter is able to specify transformations of a composition pattern in a modular and reusable way. The changes described by a composition adapter are independent of a specific API, because a composition adapter also employs the PacoSuite semantic primitives. A composition adapter consists of two parts, a context part and an adapter part. A composition adapter specifies that every occurrence of the context part in the target composition pattern has to be replaced by the adapter part. In other words, the context part describes what has to be altered and the adapter part describes the transformations themselves.

Figure 18 illustrates a composition adapter for dynamically checking timing contracts. This composition adapter specifies in its context part that it is applicable to every signal sent between certain *Source* and *Dest* roles. The adapter part specifies that this signal should be re-routed through a *Timer* role and that a *ConstraintChecker* role should be notified. The *Timer* role is responsible for taking a timestamp and notifying the *ConstraintChecker* role. The *ConstraintChecker* role is responsible to verify whether every signal it is notified of, does not violate a timing contract. The component that is mapped on the *ConstraintChecker* role could do the verification process offline and/or run on a different CPU to minimize the disruption of the system.

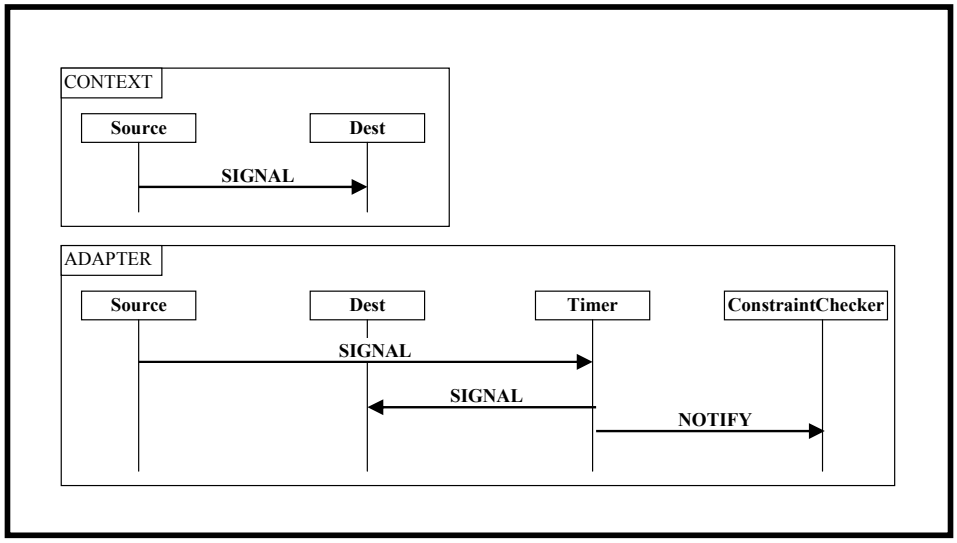

Figure 18: Composition adapter for dynamically verifying timing contracts.

When applying a composition adapter onto a composition pattern, roles in a composition adapter context part have to be mapped onto the roles of the composition pattern in order to "pattern match" the context part. Roles occurring only in the adapter part operate as newly introduced roles for the target composition pattern. For example, in order to apply the composition adapter depicted in Figure 18 onto the composition pattern of Figure 4 (page 44), the *Source* role has to be mapped onto the *Control* role and the *Dest* role onto the *Subject* role. The result of applying this composition adapter is depicted in Figure 19. Remember that the PacoSuite primitives are organized into a primitive hierarchy as explained in section 2.2. The SIGNAL primitive is the top-most primitive in the hierarchy and matches with all the lower primitives. As a result, the SIGNAL primitive matches with both the START and STOP primitives. Therefore, the composition adapter is applicable at both these protocol fragments and these are transformed as specified by the adapter part. As such, the START and STOP primitives are not sent directly to the *Subject/Dest* role but are rerouted through the *Timer* role. The component mapped onto the Timer role is able to take a timestamp of the corresponding event and notifies the *ConstraintChecker* role.

As such, the timestamping concern can be inserted in a composition pattern while it is effectively modularized by a composition adapter. Removing the timestamping concern from the composition pattern is as simple as deleting the composition adapter. The composition adapter of Figure 18 is also reusable as it specifies an abstract protocol in both its context and adapter parts. As such, the composition adapter is applicable on all composition patterns that contain a protocol fragment that matches the context part of the composition adapter.

Notice that the context roles of the composition adapter are actually merged with the roles of the target composition pattern. It is still possible to apply the JButton and Juggler components respectively onto the merged *Control* and merged *Subject* roles as is possible with the original ToggleControl composition pattern (see section 2.2.2) because the composition adapter does not alter the external protocol of the *Control* and *Subject* roles. This is however not always the case as a composition adapter might adapt the protocol instead of merely re-routing the messages.

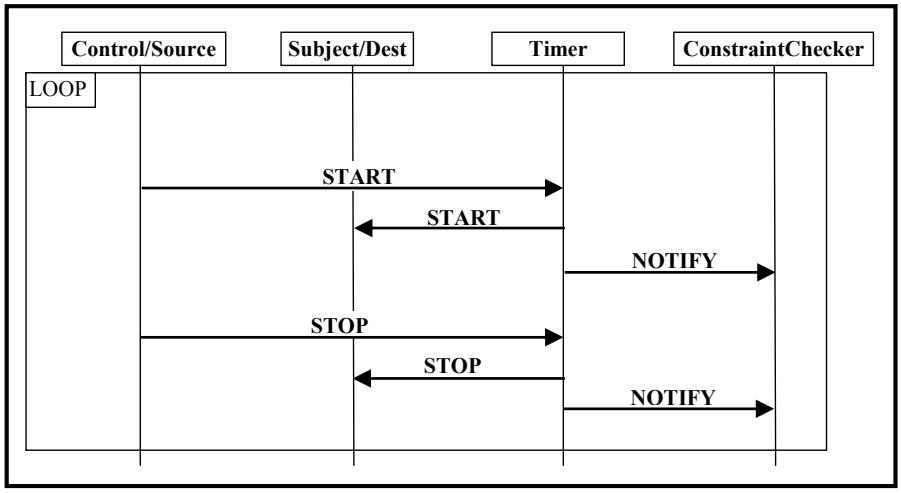

Figure 19: Result of the application of the composition adapter of Figure 18 onto the composition pattern of Figure 4 (page 44) depicted in a MSC.

3.2.2 Downcasting adapter part primitives

The transformations specified by the adapter part of a composition adapter are applied to all protocol fragments that match the context part. The primitives of the inserted adapter parts are however not necessarily the same as those specified in the adapter part of the composition adapter, instead they have to be downcasted to the primitive that matches the corresponding message in the context part. This is because otherwise the resulting MSC would be too generic and in fact the original logic of the composition pattern is changed. In the first part of the MSC of Figure 19 for instance, the two SIGNAL primitives contained in the adapter part are downcasted to START primitives because the SIGNAL primitive of the context part matches with a START primitive. Likewise, the SIGNAL primitives inserted because the context part matches with the STOP primitive, are downcasted to STOP primitives. Without downcasts, the MSC of Figure 19 would specify SIGNAL primitives instead of STARTs and STOPs. As a consequence, the behavior is changed quite a lot and does not make much sense anymore because the MSC specifies two identical parts. This is of course not the desired result in this case as the composition adapter aims at merely re-routing the original messages through a timer role in order to take timestamps.

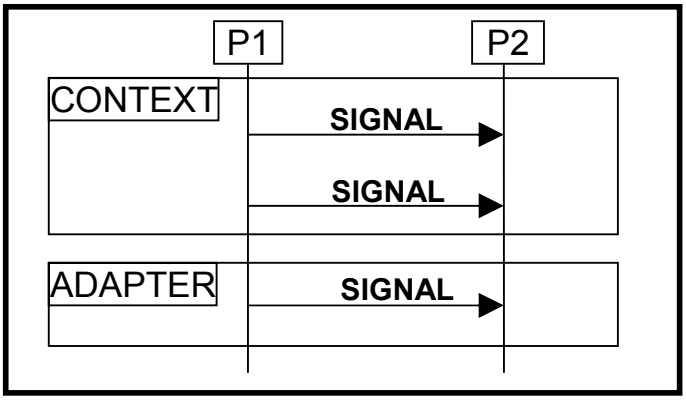

Figure 20: Composition adapter where downcast resolving is not unambiguous.

In this simple example, where the context part only specifies a single primitive, it is very simple to resolve to which primitive the adapter part signals have to be downcasted. This is however not always the case.

The composition adapter of Figure 20 for example, specifies a sequence of two SIGNAL primitives in its context part that has to be replaced by a single SIGNAL. When both the SIGNAL primitives of the context part are matched with different primitives in the composition pattern, there is no way to know to which concrete primitive the adapter part SIGNAL has to be downcasted. Indeed, the adapter part might delete the first SIGNAL and keep the second SIGNAL or vice versa. To address this issue, *primitive names* for messages sent in a composition adapter are introduced. Primitive names are specified between parentheses after the primitive type. As such, the messages contained in the adapter part can be unambiguously related to messages in the context part. Therefore, it is possible to resolve to which concrete primitive an adapter part primitive has to be downcasted. The primitive names are ignored after inserting the composition adapter into the composition pattern. As such, the glue-code generation and compatibility checking algorithms of PacoSuite are not affected by this extra information.

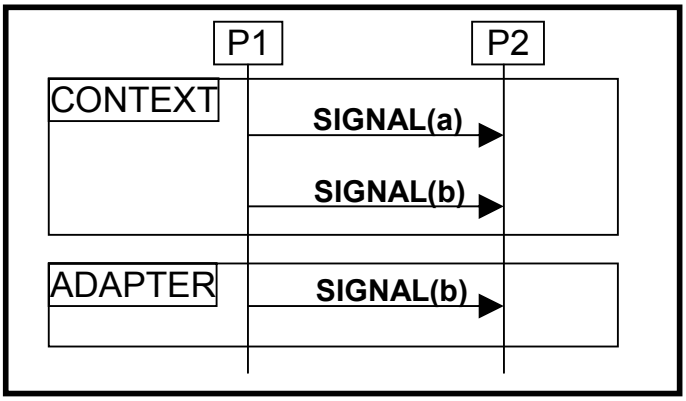

Figure 21: Composition adapter where downcast resolving is solved using explicit primitive names.

Figure 21 illustrates an example composition adapter with messages containing primitive names. The composition adapter describes the same transformation as the composition adapter of Figure 20. However, it is unambiguously defined that the first SIGNAL is removed by the adapter part because the second signal with primitive name b remains in the adapter part. As such, the SIGNAL in the adapter part is downcasted to the primitive matching the second SIGNAL of the context part.

Of course, this downcasting of adapter part primitives is not always the desired behavior. Therefore, it is also possible to specify that a certain adapter part primitive can never be downcasted. This can be done by defining a minus sign as the name of the primitive. Figure 22 illustrates an example adapter part of where the SIGNAL primitive is declared to remain as such and is never downcasted.

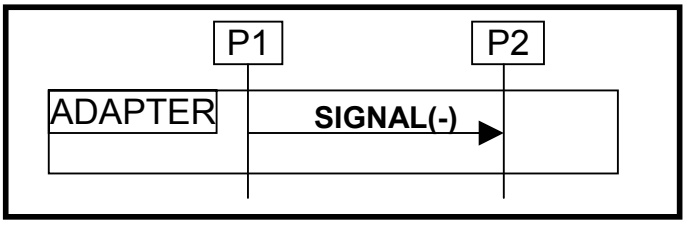

Figure 22: Composition adapter where downcasting of the SIGNAL is not allowed.

3.2.3 Mapping composition adapter roles

It is important that composition adapter context part roles are mapped on different composition pattern roles. In other words, no composition pattern role can be mapped upon more than one context part role. As a result, one can only apply a composition adapter onto a composition pattern when there are more

composition pattern roles than context part roles. The reason is that a composition pattern only documents a protocol between the roles. Self-invocations are not documented[1]. When one maps several composition adapter context part roles onto the same composition pattern role, the application of the composition adapter can never be valid. This is because the context part protocol is never able to match the protocol in the composition pattern as composition patterns do not document self-invocations. Another solution consists of merging the context part roles and as such ignoring all interactions between these roles. All possible interactions between these context part roles are then neglected with respect to checking whether the composition adapter is applicable and with respect to applying the transformations. Because the composition adapter explicitly specifies two separate roles, this might cause unwanted side effects. Therefore, mapping several composition adapter context part roles onto a single composition pattern role is not allowed.

3.2.4 Concluding remarks

The composition adapter illustrated in Figure 18 is of course a very simple example because the context part only contains a single message. Both the context and adapter parts are allowed to contain a full protocol using the ALT, OPT and LOOP constructs of the MSC syntax. When a composition adapter context part consists of multiple alternatives, it suffices for one alternative to match with the composition pattern in order to apply the transformation described by the adapter part. For example, if a composition adapter contains an alternative START and STOP in its context part and is applied to the *ToggleControl* composition pattern, the adapter part is inserted once for the START and once for the STOP message in the composition pattern of Figure 4 (see page 44). The adapter part is always completely inserted, so all alternatives specified by the adapter part are inserted at the protocol fragments that match the context part. It is even possible to specify an empty adapter part. As a result, all protocol fragments that match the context part are deleted.

The advent of the composition adapter concept enhances the reusability of the composition patterns as they focus on describing only their main behavior. As such, the composition pattern can describe the desired protocol itself without tangling other concerns. In addition, the protocol transformations described by a composition adapter are independent of concrete component types and APIs. As such, they are truly reusable over all components and composition patterns that contain a protocol fragment that matches the context part.

Composition adapters can also be stacked upon the same composition pattern to achieve a chained transformation. When stacking composition adapters onto the same composition pattern, the component composer explicitly specifies a sequence of the involved composition adapters. Hence, the composition adapters are applied in the specified sequence. This allows to effectively realize precedence strategies of the involved aspects represented as composition adapters. In addition, the precedence is able to vary over different applications of the same aspects. In AspectJ for example, this is not at all possible as precedence has to be hard coded in the aspects themselves. As such, aspects have to be aware of all possible other aspects with which they possibly interact. Obviously, this is impossible in general and the aspects themselves have to be adapted when confronted with previously unknown aspects. In addition, by hard-coding precedence in the aspect, the precedence is never able to vary over different instantiations of the aspect. For example, in some cases, the logging aspect has to be executing before the synchronization

[1] Section 5.4 presents an extension of the PacoSuite model that supports self-invocations.

aspect while in other cases, it is the other way around. This is impossible to achieve in AspectJ without duplicating the aspect behavior in several aspects.

When comparing the composition adapter model to the four mainstream aspect-oriented approaches, a composition adapter seems very similar to a composition filter. Composition filters also allow specifying protocol transformations. In addition, composition filters can also be easily stacked onto the same context in order to achieve a combination of all the composition filter specifications. The composition filters themselves are however implemented in a normal programming language like for example Java or Smalltalk. This gives the composition filter model more expressive power in comparison to the MSC based protocol transformations specified in a composition adapter. The MSC based protocol transformations used in composition adapters are however an ideal means to clearly describe protocol transformations. Using the graphical documentation, the described transformations are easily understood, even by business-oriented people without a computer science background. In addition, by making the transformations themselves more explicit in comparison to a programmed transformation, it is possible to verify whether the transformations applied to the given context make sense. The composition adapter model also employs the set of semantic primitives to describe the context where the transformations have to be applied. This allows applying a composition adapter onto any given context that is also documented with the same set of semantic primitives. In the composition filter model, the "context part" needs to be specialized for each application of a composition filter in order to apply the composition filter to the concrete methods and events of the context at hand.

Another contribution of the composition adapter model consists of the protocol history condition that can be defined. Indeed, composition adapters are able to specify a protocol history that has to have occurred before the transformations specified by the composition adapter are applicable. Other AOSD approaches like for example AspectJ only offer control flow conditions, meaning that an aspect can only be triggered in the control flow of certain methods. The protocol documented by a composition adapter allows specifying a full history of messages that have to be occurred before the transformations of the composition adapter are applicable. Whether or not these previous methods or events still have an activation record on the stack does not matter. As such, the composition adapter model realizes a more generic history condition than in control-flow based approaches like AspectJ.

When reviewing the requirements for integrating aspects in PacoSuite outlined in the previous section, the composition adapter model is able to fulfill all four requirements. It is evident that a composition adapter can be easily composed onto a given composition pattern. In addition, the resulting composition pattern can be used to check compatibility with mapped components and to automatically generate glue-code. Automatically checking whether applying a composition adapter onto a composition pattern is valid and automatically inserting the adapter part are two requirements that are not straightforwardly fulfilled. The following section presents an algorithm that fulfills both these requirements.

Composition adapter roles have to be manually mapped onto roles of the context part to "pattern match" this context part. This requires knowledge about the concrete semantics of these roles. In section 3.4, we propose an algorithm to automatically map these roles or propose all possible role mappings if there are multiple solutions.

3.3 Automatically Applying a Composition Adapter

3.3.1 Introduction

Inserting a composition adapter seems obvious from the example of Figure 18 explained above. In this example, merely syntactically scanning the affected composition pattern would do the job. In case the context part specifies a full protocol however, a more involved algorithm is needed. This section presents an algorithm that achieves this.

In order to insert a composition adapter into a composition pattern, the composition adapter and composition pattern have to be transformed to a more operational model. Similar to the original PacoSuite approach, Finite State Automata are used. The composition adapter and composition pattern have to be translated into the automaton model in order to compute the transformations specified by the composition adapter. Section 2.2.6 discusses how this translation is done for composition patterns. One issue that has to be coped with consists of avoiding to lose the source and destination of message sends of an MSC. To this end, the transition labels specify the employed primitive of the message augmented with the source and destination roles. Figure 23A illustrates the result of translating a composition pattern to a DFA.

Definition 12: Composition Adapter

A composition adapter automaton CA is a tuple {CAC, CAA}

where CAC={M_C,R_C,τ_C,P,ρ} and CAA={M_A,R_A,τ_A,P,ρ}

{M_C={Q_C, Σ, q_{sC}, F_C, δ_C}, R_C, τ_C, P, ρ} is a composition pattern

{M_A={Q_A, Σ, q_{sA}, F_A, δ_A}, R_A, τ_A, P, ρ} is a composition pattern

A composition adapter is considered as two separate MSCs and as such translated into two automata: a context automaton and an adapter automaton. Definition 12 formally defines a composition adapter in terms of automata theory. The automata M_C and M_A are referred to as respectively the context part automaton and the adapter part automaton of the composition adapter.

The labels of the transitions of the context part automaton have to be specified in terms of composition pattern roles. Otherwise, the input alphabet of both the composition pattern DFA and composition adapter context DFA would be disjunct and no paths of the composition pattern DFA would match the context part DFA. Therefore, the component composer has to manually map the composition adapter context roles to composition pattern roles. We have developed a tool to support this, see section 6.2 for detailed information about this tool. Using the specified mapping, the composition adapter is translated into two DFAs with transitions in terms of the composition pattern roles instead of composition adapter context roles. Roles occurring only in the adapter part of a composition adapter are additional roles and do not need to be matched onto roles of the composition pattern. As such, these role names can remain in the adapter part DFA. Figure 23B shows an example of the DFA representation of a composition adapter. In this example, the composition adapter role P1 is mapped upon composition pattern role R1. Likewise, the

roles P2 and R2 are manually mapped. The transitions of the composition adapter automata are qualified using this mapping. As such, role P1 is translated to R1 and P2 is translated to R2.

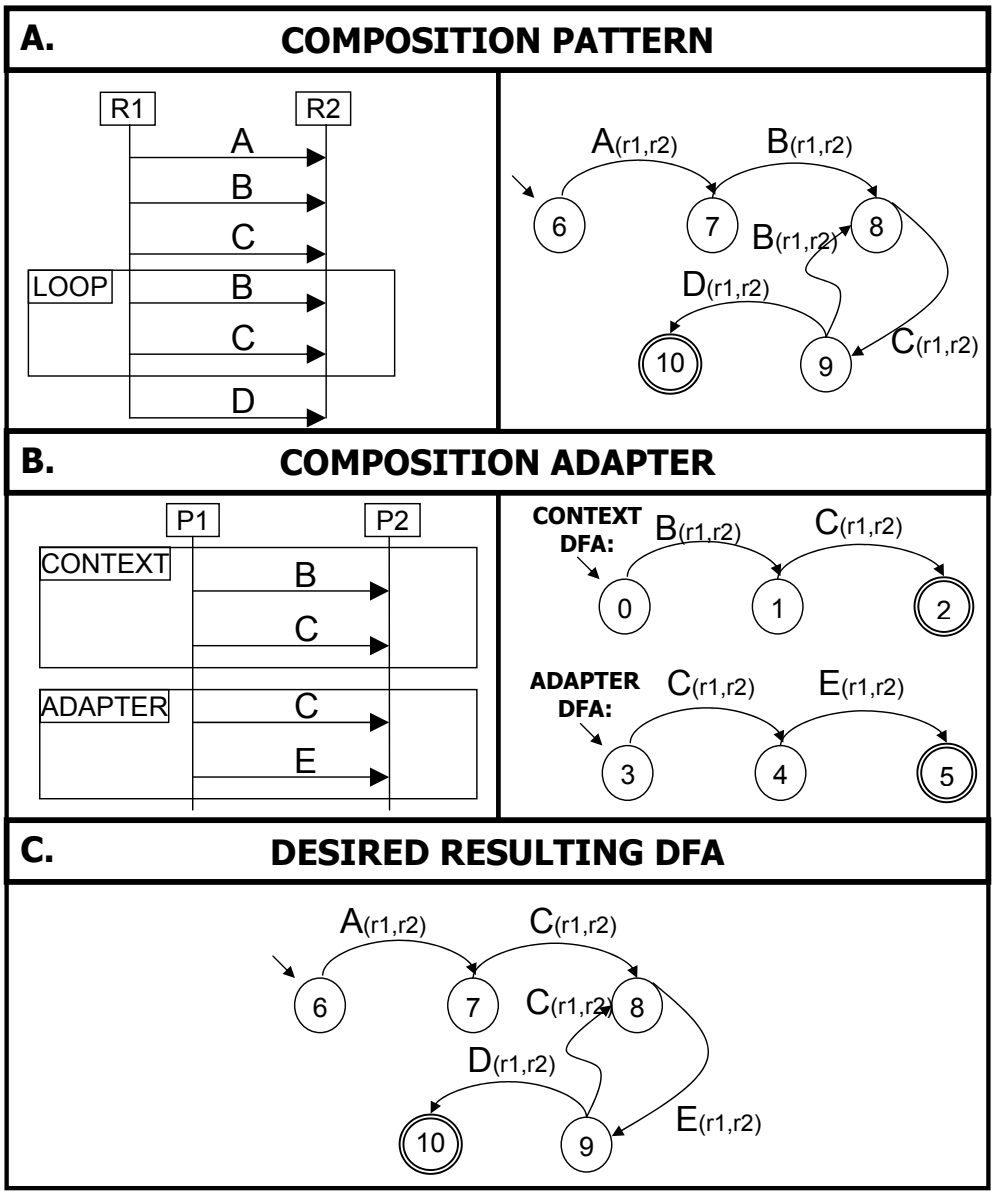

Figure 23: Desired resulting automaton when matching composition adapter roles P1 and P2 respectively on composition pattern roles R1 and R2. Notice that the label of a state has no real meaning, the label is merely used to be able to refer to the state at hand.

To illustrate the desired result of applying a composition adapter onto a composition pattern, consider again the example of Figure 23. When applying composition adapter roles P1 and P2 on respectively composition pattern roles R1 and R2, we expect the B-C protocol in the composition pattern DFA to be replaced by C-E. This is illustrated by Figure 23C. The resulting DFA can then be employed for checking compatibility with the components that fill the roles using the algorithms proposed in [Wyd01]. In addition, the DFA can also be exploited for implementing the glue-code that eventually connects the components. However, when the composition adapter context part roles are mapped in such a way that the context part does not occur in the composition pattern, the application of this composition adapter is not

possible. For example, when applying composition adapter roles P2 and P1 respectively on composition pattern roles R1 and R2, the labels of the context part DFA would be B(r2,r1) and C(r2,r1). As a result, the context part of the composition adapter does not occur in the composition pattern. Consequently, the composition adapter is not able to realize its transformation and the component composer needs to be notified.

In order to automatically apply a composition adapter onto a composition pattern, we propose a three-step algorithm based on automaton theory. The first step consists of finding all paths in the composition pattern DFA that match with the context part DFA. If no such paths are found, the application of this composition adapter is not possible. In the second step, the adapter part DFA is inserted at all the paths resulting from the previous step. In other words, the adapter automaton is inserted at all paths of the composition pattern automaton where the context part matches with. The final step consists of removing all the paths from the composition pattern DFA that match with the context part in such a way that the newly inserted paths remain. The next three sections explain these steps in more detail.

Notice that the algorithm defined in the next sections assumes that no non-determinism is encountered both in the starting automata as in the resulting automaton. How to deal with non-determinism is explained in section 3.3.5. In addition, the following sections assume that the roles of a composition adapter are already mapped onto the roles of a composition pattern and this in a valid way. This means that no two composition adapter context part roles are mapped onto the same composition pattern role.

3.3.2 Step 1: Searching paths that match with the context part

3.3.2.1 The approach

A straightforward manner to search for all paths in the composition pattern automaton that match with the context part automaton, consists of iterating over all states of the composition pattern automaton and checking whether the automaton with the state at hand as starting state is able to accept at least one input sequence that the context part automaton also accepts. This means that for every state in the composition pattern automaton, the intersection with the context part automaton has to be calculated. The intersection of automata is a standard operation and is described in literature [HMU01]. When the intersection is not empty, each start-stop path of the intersection automaton corresponds to a path in the composition pattern automaton that matches the context part. The result of this phase consists of the set of all the pairs of states, denoting matching paths, accumulated by performing the intersection for every state of the composition pattern automaton.

Definition 13: State-pair intersection $A \cap_S B$

Let $A=\{Q_A, \Sigma, q_{sA}, F_A, \delta_A\}$ and $B=\{Q_B, \Sigma, q_{sB}, F_B, \delta_B\}$

and $A \cap B = \{Q_A \times Q_B, \Sigma, (q_{sA}, q_{sB}), F_A \times F_B, \delta_{A\cap B}\}$

then $A \cap_S B = \{(q_{sA}, f) \mid f \in F_A \wedge \exists n>0 \wedge \exists R \in \Sigma^n \wedge$

$\exists k \in F_B: \delta^t_{A\cap B}((q_{sA}, q_{sB}), R) = (f, k)\}$

One helper definition is required in order to clearly specify the algorithm to find all paths matching the context part in the composition pattern automaton. Definition 13 defines a special kind of intersection between automata A and B where the result is not a new automaton, but a set of state pairs of automaton A. Every pair contains the start state of automaton A and a final state of automaton A. A pair of the start state

q_{sA} and a final state f is only contained in the resulting set when a path exists in the regular intersection of A and B starting from a combined start state (q_{sA}, q_{sB}) and ending at a combined final state (f,k) where k is a final state of automaton B. This is specified by employing the transitive transition function defined in Definition 2 as follows: if there exist a list of input symbols and a final state k of automaton B in such a way that the transitive transition function of the normal intersection automaton A∩B is able to reach a combined state containing f and k starting from the combined start states, then the pair (q_{sA}, f) is contained in the result. In other words, every pair denotes one or more start-stop paths of the intersection automaton from the viewpoint of the automaton A. Although the number of start-stop paths in the intersection automaton is possibly infinite, the number of start-stop pairs is not because the number of states is finite. Notice that only paths with a length longer than one are considered by restricting the list of input symbols of the transitive function to a list of at least one element. In other words, when the start state s of A is also a final state, it is only added as a pair (s,s) if there exists at least one transition from s to s.

Algorithm 1 specifies the algorithm to find all paths in the composition pattern automaton that match with the context part of a composition adapter. The algorithm starts from a composition pattern automaton and a composition adapter context part automaton. The result is accumulated in the set R. For each state q of the composition pattern automaton, a new transformed version of the composition pattern automaton is created where q is the start state and all states are final states (line 5). Afterwards, the state-pair intersection between the transformed composition pattern automaton and the context part automaton is calculated and united with the accumulated result from the previous steps contained in the set R. The result of the state-pair intersection contains all pairs of states of the composition pattern DFA that denote paths starting from state q that match the context part DFA. This is because the regular intersection contains a start-stop path if a final state can be reached in both automata. Since all states of the modified composition pattern automaton P′ are final states, it suffices to reach a final state in the context part automaton to find a matching path. Thus, the intersection contains all paths of the composition pattern automaton that match with the context part automaton. Finally, when the result set R is empty after iterating over all states of the adapted composition pattern DFA, the application of this composition adapter is considered to be invalid. If not, the result set R contains all pairs of states of the composition pattern automaton where the context part matches. Consequently, the adapter part of the composition adapter has to be inserted at all these pairs of states.

Algorithm 1: Finding all paths that match with the context part DFA.

1. $P=\{Q_P, \Sigma, q_{sP}, F_P, \delta_P\}$ is the composition pattern DFA
2. $C=\{Q_C, \Sigma, q_{sC}, F_C, \delta_C\}$ is the context part DFA
3. $R := \emptyset$
4. foreach $q \in Q_P$
5. $\quad P' = \{Q_P, \Sigma, q, Q_P, \delta_P\}$
6. $\quad R := R \cup (P' \cap_S C)$
7. end foreach

The complexity of Algorithm 1 is $O(m.n^2)$ where n is the number of states of the composition pattern automaton and m is the number of states of the context part automaton. Indeed, one has to iterate over all states in the composition pattern automaton and then calculate the intersection, which can be computed in O(m.n) time [HMU01].

3.3.2.2 An example

For example, let's apply Algorithm 1 to the example of Figure 23. This is illustrated in Figure 24. The algorithm starts by transforming the composition pattern DFA with state 6 as start state and all states as final states. When calculating the intersection with the context part DFA, an automaton is returned which contains no start-stop path with a length longer than 1. In other words, an empty automaton is returned and as such, the context part can not be matched here. However, the second iteration reveals a result. The automaton with state 7 as start state returns a result after the intersection with the context part DFA, namely a single path starting from state 7 and ending at state 9. Consequently, state pair (7,9) is added to the resulting set of matching state pairs. The next iteration does not return any results. Starting from state 9 however, a matching path can be found with state 9 both as start state and final state. The result of applying Algorithm 1 is then a set containing two pairs, namely (7,9) and (9,9).

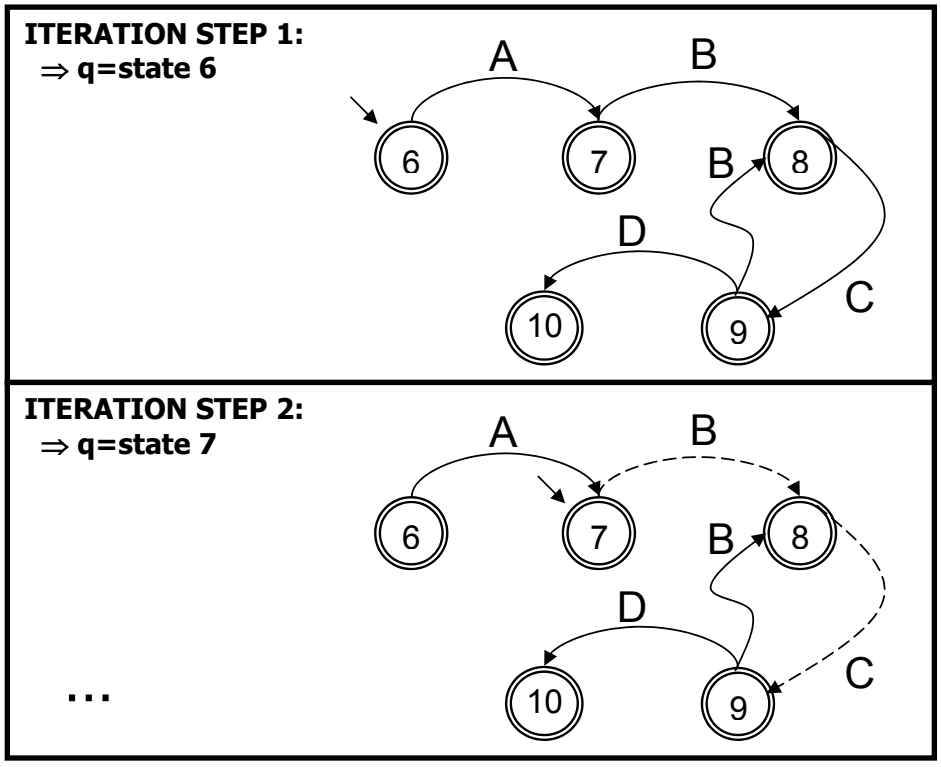

Figure 24: Transforming the composition pattern DFA for Algorithm 1. Dashed transitions are contained in the intersection with the context part automaton.

Notice that the source and destination of the primitives is omitted in Figure 24 in order to avoid cluttering the figure with long transition labels.

3.3.3 Step 2: Inserting the adapter part

3.3.3.1 The approach

In this step, the adapter automaton has to be inserted at all places in the composition pattern where the context part DFA matches. These places are calculated in the previous step, so this step comes down to copying the adapter part DFA for each pair of start-stop states and connecting it to the composition pattern DFA.

Two helper definitions are required in order to clearly specify the algorithm of this step. The insertion of an automaton A into another automaton P at states s and f is defined as the insertion automaton $A\downarrow_{(s,f)}P$ (see Definition 14). The resulting automaton is the combination of both automata P and A in such a way that state s connects to the start state of A using an epsilon transition and all the final states of A are connected to state f using an epsilon transition. All states of the previous automaton A are non-final in the insertion automaton. In addition, the start state of the insertion automaton equals the start state of automaton P and the set of final states of the insertion automaton is the set of final states of automaton P.

Definition 14: Insertion automaton $A\downarrow_{(s,f)}P$

Let $P=\{Q_P, \Sigma_P, q_{sP}, F_P, \delta_P\}$ and $A=\{Q_A, \Sigma_A, q_{sA}, F_A, \delta_A\}$ and $s,f \in Q_P$

$A\downarrow_{(s,f)}P = \{Q_P \cup Q_A, \Sigma_P \cup \Sigma_A, q_{sP}, F_P, \delta'\}$

where $\delta'(s,\varepsilon)=q_{sA}$

$\delta'(q,\varepsilon)=f$ if $q\in F_A$

$\delta'(q,a)=\delta_P(q,a)$ if $q\in Q_P \wedge a\in\Sigma_P$

$\delta'(q,a)=\delta_A(q,a)$ if $q\in Q_A \wedge a\in\Sigma_A$

The other helper definition that is required is specified by Definition 15. A marked automaton is an automaton where the input alphabet is changed by applying a marking function to every symbol of the alphabet. The transition function is changed accordingly in order to accept the transformed input alphabet. From the point of view of the state chart of the marked automaton, the labels of every transition are changed through the marking function. It is evident that when the marking function is a bijective function, the marked automaton can always be "unmarked" in such a way that the resulting automaton equals the original one. This can be achieved by applying the inverse marking function to the marked automaton.

Definition 15: Marked automaton f(M)

Let $M=\{Q, \Sigma, q_s, F, \delta\}$ a finite state machine and $f:\Sigma\rightarrow\Sigma'$ a marking function. The marked automaton is defined as

$f(M)=\{Q, \Sigma', q_s, F, \delta'\}$

where $\Sigma' = \{f(a) \mid a \in \Sigma\}$

and $\delta'(q,f(a))= \delta(q,a)$

Algorithm 2 specifies the algorithm to insert the adapter part into a composition pattern using Definition 14. The algorithm starts with the composition pattern automaton and the adapter part automaton and a bijective marking function g. Notice that both the input automata have the same input alphabet Σ which corresponds to the set of semantic primitives augmented with direction information. The bijective marking function g translates every input symbol of the alphabet Σ of the automata at hand to a new input symbol in such a way that the new alphabet Σ' has no symbols in common with the original alphabet Σ. It is easy to proof that this is always possible because the input alphabet is finite. The bijective property is required to be able to translate the marked input alphabet back to the original one by applying the inverse function.

The set R is the result from the previous phase, meaning a set of pairs denoting the start and stop states where the context part matches. For each of these pairs, a marked copy of the adapter part automaton is computed using the marking function g. The insertion automaton is calculated for inserting the marked

copy of the adapter part into the previous insertion automaton (or the composition pattern automaton at iteration step one) at the states denoted by the current pair Y_i in the set R. The result is then the last insertion automaton calculated.

Notice that the resulting insertion automaton contains epsilon transitions, making this automaton difficult to work with. Consequently, the last step of this phase consists of transforming the last insertion automaton in such a way that the epsilon transitions are removed. In literature, it is proven that it is always possible to remove epsilon transitions while keeping the accepted language unchanged [HMU01]. The idea of the epsilon removal algorithm consists of bridging the epsilon transitions for every state q, by adding to q all transitions starting from states contained in the epsilon closure of q. The epsilon closure of a state q is defined as the set of all states that are reachable from state q using only epsilon transitions. The result of Algorithm 2 is thus an insertion automaton that does not contain any epsilon transition.

Removing epsilon transitions causes the structure of the automaton to change. This is no problem because the accepted input language of the automaton remains the same. The structure of the automaton itself is of no importance as long as the input language remains the same because the input language represents the protocol. As such, removing epsilon transitions is no problem.

Algorithm 2: Inserting the adapter part

1. $P=\{Q_P, \Sigma, q_{sP}, F_P, \delta_P\}$ is the composition pattern DFA
2. $A=\{Q_A, \Sigma, q_{sA}, F_A, \delta_A\}$ is the adapter part DFA
3. $g:\Sigma\rightarrow\Sigma'$ is a bijective function and $\Sigma\cap\Sigma'=\varnothing$
4. R is the result of the previous step (Algorithm 1)
5. P_0 := P
6. foreach $Y_i \in R$ where $Y_i = (s_i, f_i) \wedge i=1...n=\#R$
7. $\quad P_i := g(A)\downarrow_{(si,fi)} P_{i-1}$
8. end foreach
9. result=removeEpsilons(P_n)

The reason for applying the marking function is that an adapter part possibly contains a path that matches the context part. When in the next step, all paths that match the context part are deleted, these paths in the inserted adapter part have to remain untouched. By renaming the alphabet of the adapter part in such a way that the new alphabet is disjunct with the original one, the marked adapter part never matches the context part and is thus not affected by the deletion step. Section 3.3.4 explains this in more detail.

The complexity of Algorithm 2 is mainly influenced by the number of pairs of states returned by Algorithm 1. In a worst case scenario, this is quadratic to the number of states of the composition pattern automaton. Taking a marked copy of an automaton can be calculated in a time linear to the number of states of the adapter part automaton. As a result, computing the resulting insertion automaton has a complexity of $O(m.n^2)$ with n the number of states of the composition pattern automaton and m the number of states of the adapter part automaton. In addition, removing epsilon transitions from the insertion automaton can be computed in a time of $O(n^3)$ where n is the number of states of the composition pattern automaton. This because the number of additional epsilon transitions is limited to the number of adapter parts inserted, which is maximum quadratic to the number of states of the composition pattern automaton. Furthermore, calculating the epsilon closure of a certain state is possible in a time linear to the number of

states. As a result, the complexity of the overall algorithm consists of $O(m.n^2 + n^3)$ with n the number of states of the composition pattern automaton and m the number of states of the adapter part automaton.

3.3.3.2 An example

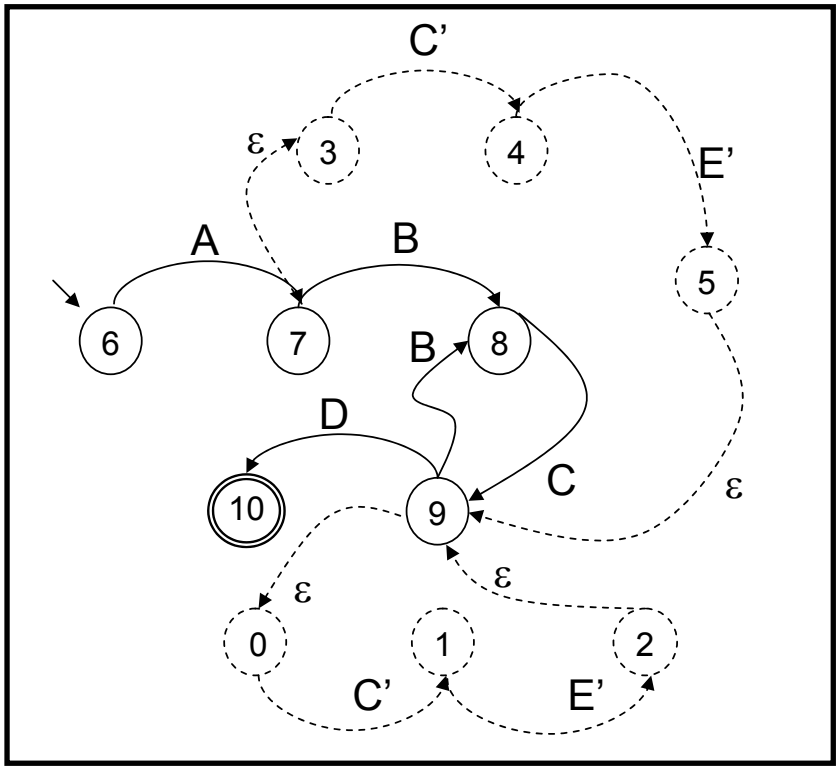

Figure 25: Result after the second phase of the composition adapter application algorithm applied to the example of Figure 23.

The final insertion automaton resulting from applying Algorithm 2 to the example of Figure 23 yields the result as illustrated in Figure 25. The states and transitions added to the composition pattern automaton are dashed. The result of the previous step, namely the places where the composition adapter context part matches, are state pairs (7,9) and (9,9). Consequently, for each of these pairs a marked copy of the adapter part automaton is added using epsilon transitions. At state 7 for instance an epsilon transition connects to the first state of the adapter part automaton. The last state (state 5) of this adapter part automaton is connected to state 9. Likewise, the pair (9,9) is connected to another copy of the adapter part automaton.

In the last step of Algorithm 2, the epsilon transitions are removed using the standard algorithm. The resulting automaton is illustrated by Figure 26. For example at state 7, the epsilon transition to the former state 3 is bridged by a C' transition to state 4. Likewise, at state 5 the epsilon transition to state 9 is bridged by three transitions: one transition connecting to state 10 using input symbol D, one transition connecting to state 8 using input symbol B and one transition connecting to state 1 using input symbol C'. This last transition is added because the epsilon closure of state 5 contains all states reachable from state 5 using epsilon transitions. State 0 can be reached by following two epsilon transitions. As a result, transition C', which is the only transition starting at state 0, is also connected from state 5. Likewise, for states 9 and 2, transitions are added that bridge the epsilon transitions. Notice that the input language of the automaton remains identical to the input language of the original automaton which contained epsilon transitions.

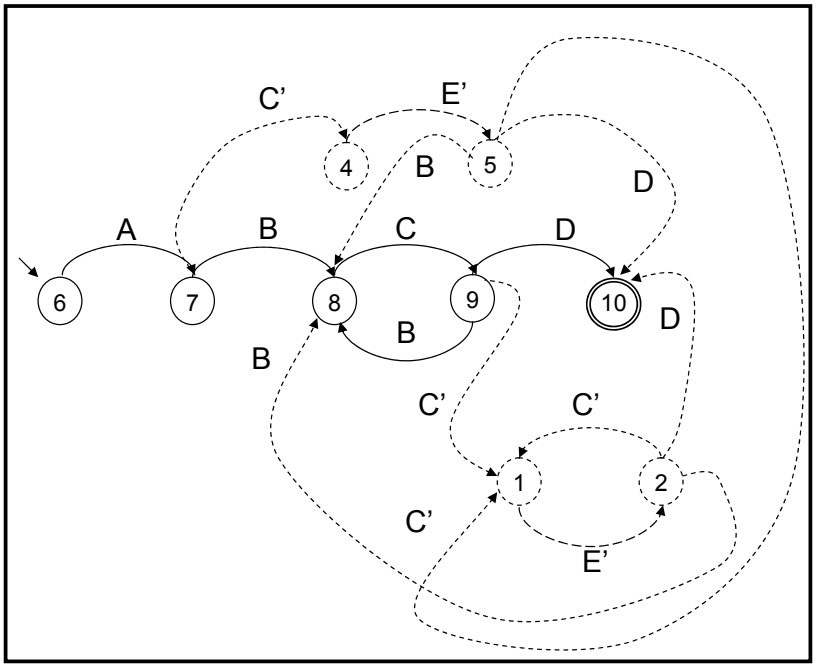

Figure 26: Resulting insertion automaton after removing epsilon transitions.

3.3.4 Step 3: Removing paths that match with the context part

3.3.4.1 The approach

In this step, all paths that match with the context part have to be removed from the automaton resulting from the previous step. Remember that all paths that match with the context part are computed in step one. A naïve and wrong approach to accomplish this removal step consists of just deleting all transitions contained in paths that match the context part. Merely deleting all those transitions is however not a possible solution because we might delete too much. In the automaton of Figure 27 for instance, a single path matches with the context part of Figure 23, namely the path starting at state 1 and ending at state 3. Neither the B and C transitions of that path can however be simply deleted. Deleting the B transition causes the path from state 0 to state 6 that accepts input string ABF to be removed as well, which is not the intention. Likewise, deleting the C transition causes the input string AAECD to be no longer accepted, which is also not the intention. Therefore, a more intelligent approach is required that makes sure that input strings are no longer accepted by the automaton resulting from the previous step if and only if they contain an input string accepted by the context part. This is achieved by constructing a special version of the context part automaton that accepts all possible input strings of the composition pattern automaton that contain an input string accepted by the context part automaton. Afterwards, the difference between the automaton resulting from the previous step and this new context part automaton is computed. As a result, all input strings that contain an input string accepted by the context part are no longer accepted.

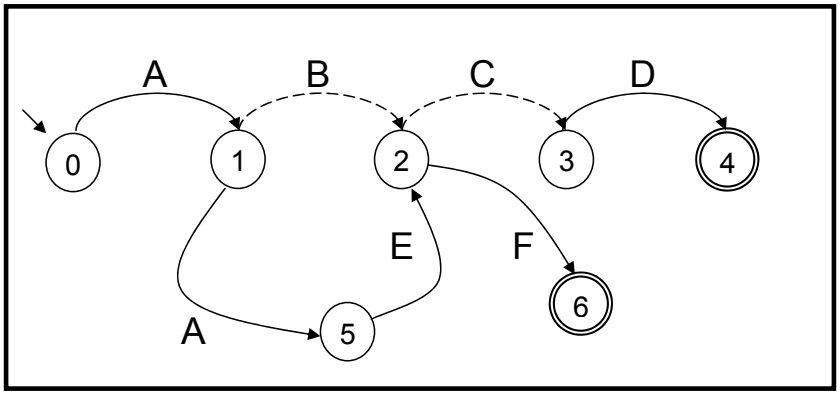

Figure 27: Automaton where no transition may simply be deleted for removing the B-C protocol that matches the context part.

In order to clearly define the algorithm of this step, two helper definitions are useful. The first helper definition defines the *path matching automaton* of an automaton C for a given automaton P (see Definition 16). This automaton is a completed version of the original automaton C, but the completion strategy renders an automaton which accepts another language. More specifically, the path matching automaton of an automaton C for a given automaton P accepts all combinations of input symbols of the alphabet of automaton P which contain an input string accepted by C. As such, the input alphabet of the *path matching automaton* accepts the union of the input alphabets of both automata. In contrary to the regular completion algorithm defined in literature, the transition function does not define transitions to an error state for undefined input symbols at a certain state. Instead, the following transitions are added to the automaton C:

(1) For every non-final state, a transition is added towards the start state for input symbols where the transition function is undefined, if the transition function is undefined from the start state with the input symbol at hand.

(2) For a non-final state `q` and an input symbol `a` for which the transition function is undefined for this state and where the transition function is defined starting from the start state for the input symbol `a`, an additional transition is made connecting state `q` to the result of applying the transition function on the start state and the input symbol `a`.

(3) For a final state, a transition to the state itself is added for every input symbol for which the transition function is undefined for the state at hand.

By construction this automaton is complete because the transition function is defined for every input symbol and state. In addition, this automaton accepts any string of symbols of the united input alphabet that contain an input string accepted by the original automaton C. In other words, the automaton matches with all start-stop paths that contain paths of the original automaton C.

Definition 16: Path matching automaton C→P

Let $P=\{Q_P, \Sigma_P, q_{sP}, F_P, \delta_P\}$ and $C=\{Q_C, \Sigma_C, q_{sC}, F_C, \delta_C\}$

the $C \rightarrow P=\{Q_C, \Sigma_P \cup \Sigma_C, q_{sC}, F_C, \delta'\}$

where (1) $\delta'(q,a) = q_{sC}$ if $\delta_C(q,a)=\emptyset \wedge \delta_C(q_{sC},a)=\emptyset \wedge q \notin F_C$

(2) $\delta'(q,a) = \delta_C(q_{sC},a)$ if $\delta_C(q,a)=\emptyset \wedge \delta_C(q_{sC},a)\neq\emptyset \wedge q \notin F_C$

(3) $\delta'(q,a) = q$ if $\delta_C(q,a)=\emptyset \wedge q \in F_C$

(4) $\delta'(q,a) = \delta_C(q,a)$ otherwise

Definition 17 defines the other helper definition needed for specifying the deletion algorithm. The union of two functions where the domain of the functions is disjunct is defined as a function starting from the union of both domains and ending at the union of both ranges. The result of applying the united function equals either the result of the first function when the argument is an element of the first domain. If the argument is an element of the second domain, the second function is used. Because both domains are disjunct, it is unambiguous to which of the functions the union function equals for a given argument. If both functions are total, meaning that every element of the domain is in relation with an element of the range, then the resulting union function is also total.

Definition 17: Union of functions f ∪ g

Let $f:S \rightarrow D$ and $g:T \rightarrow E$ where $S \cap T = \emptyset$

then $f \cup g: S \cup T \rightarrow D \cup E$

where $(f \cup g)(x) = f(x)$ if $x \in S$

$(f \cup g)(x) = g(x)$ if $x \in T$

Algorithm 3 specifies the algorithm for deleting all paths that match with the context part. The algorithm starts with the completed automaton resulting from the previous step (line 1) and the context part DFA (line 2). In addition, the bijective marking function g used in the previous step is needed. Subsequently, the identity function i for the original input alphabet Σ of the automata is defined. Afterwards, the path matching automaton of the context part automaton given the composition pattern automaton is constructed (line 5). The idea is that the intersection of this path matching automaton with the composition pattern automaton contains all start-stop paths of the latter automaton that contain a path of the context part automaton. As a result, when calculating the difference, all start-stop paths containing context part paths are removed. The next step of the algorithm (line 5) thus computes the difference between the completed result of the previous phase and the path matching automaton. The result is an automaton where start-stop paths matching the context part are removed. The last step of this algorithm translates the input symbols of the inserted adapter parts back to the original input symbols, rendering an automaton which is again in terms of the semantic primitives of the PacoSuite approach augmented with direction information. This is done by calculating the union function of the inverse marking function used in the previous step and the identity function for the input alphabet Σ containing the semantic primitives augmented with direction information. Because the domains of both the identity and inverse marking functions are disjunct and both

functions are total, the resulting union is a total function. As such, all marked input alphabet symbols are "unmarked" and other input symbols are untouched.

The final result of the three-step algorithm is thus an automaton where the adapter part is inserted at all places that match the context part. In addition, the paths matching with the context part are deleted. In other words, this algorithm replaces all paths that match the context part with adapter part paths, which is the goal that needs to be achieved.

Algorithm 3: Deleting paths matching the context part

1. $D=\{Q_D, \Sigma \cup \Sigma', q_{sD}, F_D, \delta_D\}$ is the completed DFA resulting from Algorithm 2
2. $C=\{Q_C, \Sigma, q_{sC}, F_C, \delta_C\}$ is the context part DFA
3. $g:\Sigma \rightarrow \Sigma'$ is the bijective function used in the previous step
4. $i:\Sigma \rightarrow \Sigma$ is the identity function for Σ
5. $D' = D \setminus (C \rightarrow D)$
6. result = $(g^{-1} \cup i)\ (D')$

Notice that when an adapter part automaton contains a path that matches the context part, this path is not removed by calculating the difference. Remember that in step two, the input symbols of the adapter part are translated to a separate alphabet that is disjunct to the original alphabet. As a result, the marked adapter part never matches the context part and no adapter paths are removed.

The complexity of Algorithm 3 corresponds to O(n.m) with n the number of states of the composition pattern automaton and m the number of states of the context part automaton. This is because computing the difference automaton of two automata with respectively n and m states, has a complexity of O(n.m). All the other operations, like removing the marks of the input symbols and computing the special completed version of the context part automaton, can be executed in a time linear to the number of states of the automata.

3.3.4.2 An example

Figure 28 illustrates the path matching automaton of the context part DFA of Figure 23B given the composition pattern DFA of Figure 23A that is constructed by Algorithm 3. Dashed transitions are newly introduced transitions in comparison to the automaton of Figure 23. For every non-final state and for every input alphabet symbol where the transition function is not defined, a transition is added towards the start state if the transition function for the input symbol at hand starting from the start state is not defined in the automaton of Figure 23. For example, at state 1, transitions are added for the following symbols A,C,C',D,E', because state 1 does not have a transition for these input symbols and no transition is defined from the start state 0 with either one of these input symbols. For input symbol B however, a transition function is defined starting from the start state. As a result, the transition with input symbol B is connected to the state that is the result of applying B to the start state 0, which is state 1. Likewise, from state 0, some transitions are added that connect to the start state 0. For final states, input alphabet symbols where the transition function is not defined are connected to the state itself. In this case, there is only one final state, namely state 2. Notice that the alphabet is the union of the alphabet of the automaton of Figure 25 and the context part DFA. Hence, marked symbols of the adapter parts in the insertion automaton are also handled.

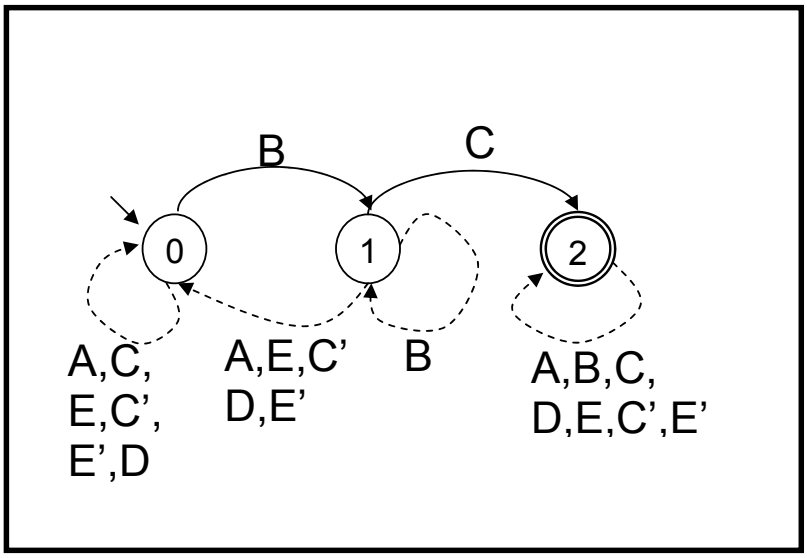

Figure 28: The path matching automaton of the context part DFA of Figure 23B given the composition pattern DFA of Figure 23A.

The automaton of Figure 28 can now be used to compute the difference automaton with the automaton of Figure 26. The minimized result is illustrated by Figure 29. After translating the marked transitions of the automaton of Figure 29, the automaton accepts the same input language as the desired result automaton of Figure 23. As a result, the replacement of the B-C protocol with the C-E protocol has succeeded. Consequently, this automaton can be used to verify compatibility with the mapped components and to generate glue-code that realizes the component-based application.

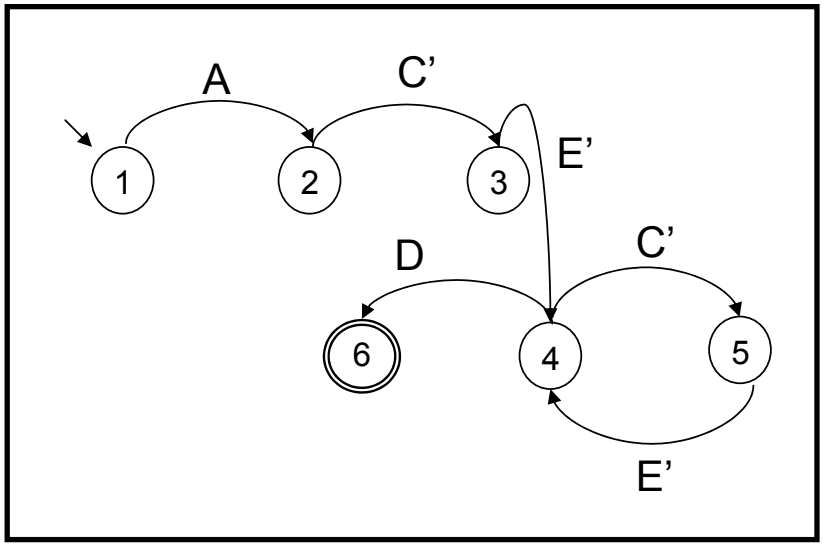

Figure 29: Result after computing the difference.

3.3.4.3 Why not use the path matching automaton to improve Algorithm 1?

One might wonder why the path matching automaton of the context part given the composition pattern is not employed to improve Algorithm 1. The goal of Algorithm 1 consists of finding all pairs of states of the composition pattern automaton denoting paths matching the context part. Conveniently, the path matching automaton matches with all start-stop paths of the composition pattern automaton containing context part

paths. As such, when computing the intersection with the composition pattern, all start-stop paths containing context part paths remain. Computing the intersection is only of quadratic nature, while Algorithm 1 has a complexity of $O(n^3)$. The problem is however that the path matching automaton is designed to match global start-stop paths that contain context part paths. As such, the path matching automaton matches with any path that contains one or more context part paths. In contrary, the goal of the first step consists of finding the concrete paths of the composition pattern that match with the context path. The result of computing the intersection with the path matching automaton consists of all start-stop paths that contain paths matching the context part. However, the concrete paths matching the context part contained in the start-stop paths still have to be located. Therefore, the resulting state machine has to be traversed to find the concrete paths that match with context part paths. As such, the complexity of employing the path matching automaton also equals $O(n^3)$, which does not improve on Algorithm 1.

3.3.5 Dealing with non-determinism

As explained before, the algorithm does not cope with non-deterministic automata. In addition, a non-deterministic result is not an option because the automaton has to be used for checking compatibility with the components that fill the roles and to generate glue-code. Non-deterministic glue-code, and as a result non-deterministic application behavior, is of course not desirable. Consequently, when non-determinism is encountered, this has to be resolved by converting the non-deterministic automaton to a deterministic automaton. The conversion process is however of exponential nature causing a serious performance overhead for the overall algorithm. Non-determinism can in fact only occur in a limited number of cases, the following paragraphs outline these cases and propose a solution if possible.

3.3.5.1 Starting automata are not deterministic

It is very well possible that the starting MSCs describe non-deterministic behavior and thus results in non-deterministic automata. As such, the corresponding NFAs have to be transformed to deterministic automata before the algorithm is able to start. This problem can never be avoided, as the component composer is free to draw any MSC. However, when the component composer attempts to save a non-deterministic scenario, the tool could check for non-determinism and issue a warning. Another complementary option consists of storing the deterministic version of the automaton corresponding to the MSC at the moment the scenario is created or when it is loaded by the composer tool for the first time. This caching approach avoids computing the NFA-DFA conversion each time a composition adapter is applied, improving the performance of composition adapter application algorithm greatly.

3.3.5.2 Insertion automaton resulting from Algorithm 2 is not deterministic

The automaton resulting from Algorithm 2 could be a non-deterministic automaton in one specific case, namely when the sequential application of the adapter part automaton is non-deterministic. Sequential application means that two copies of the adapter part MSC are placed after each other. In other words, the adapter part protocol is followed twice. For example, suppose the insertion state where the adapter part automaton starts from and ends to coincides for a couple of state pairs. This means that the adapter part automaton is inserted at state pair (x,y) and at state pair (y,z). As such, the adapter part automaton is connected with epsilon transitions to another copy of itself. The resulting automaton is non-deterministic,

This is the only case of non-determinism possible in the automaton resulting from Algorithm 2, because it is the only place in the automaton where two previously separate automata that have a non-disjunct input alphabet are connected. Remember that the adapter part automaton is marked and as such has an input alphabet disjunct with the input alphabet of the composition pattern automaton. Under the assumption that

both the adapter part automaton and composition pattern automaton are already deterministic, no non-determinism can follow from connecting the adapter part automaton to the composition pattern automaton. Only connecting adapter part automata are able to cause non-determinism.

At first sight, this problem could be solved by changing Algorithm 2 to use a different marking function for each adapter part that has to be inserted. Every marking function results in another output alphabet which is disjunct with respect to all output alphabets of other marking functions. This approach could solve some cases of non-determinism. However, it is also possible to insert the adapter part at a state pair containing twice the same state. For example, in section 3.3.2.2, state pair (9,9) is a matching state pair. Consequently, even when using a different marking function for each inserted adapter part, the result might still be non-deterministic.

A possible partial solution to this problem consists of checking whether a composition adapter might cause non-determinism by sequential insertion. This can be achieved by creating a new MSC where the adapter part is copied twice. If this new MSC results into a deterministic automaton, then sequential insertion is not a problem. If not, a warning could be issued when such an MSC is created. If possible, the component composer might draw the MSC in another way to avoid the non-determinism. In the general case however, the NFA-DFA transformation can not be avoided. Nevertheless, it is possible to warn the component composer beforehand that when applying such a composition adapter, the composition adapter application algorithm is very likely to be of exponential nature and thus might take a long time.

3.3.5.3 Final resulting automaton is not deterministic

Even when the starting automata are deterministic and the composition adapter does not cause non-determinism when sequentially inserted, non-determinism is still possible. After the final result automaton is unmarked using the inverse marking function, the alphabets of the inserted adapter parts and composition pattern are no longer disjunct, as they both correspond to the limited set of semantic primitives. Consequently, when the adapter part is inserted at a state where a transition leaves with the same input symbol as a transition connecting the first state of the adapter part, the resulting automaton becomes non-deterministic. Likewise, non-determinism can also occur at the other connection point, namely the previously final states of the adapter parts. This kind of non-determinism is a problem as it can neither be avoided, nor detected before the start of the algorithm. As a result, when the final automaton is non-deterministic, the NFA-DFA conversion still has to be computed after all.

3.3.6 Stacking several composition adapters

As explained before, it is also possible to stack several composition adapters onto the same composition pattern. The transformations described by the composition adapters are applied sequentially in the sequence defined by the component composer. To automatically apply several composition adapters, the algorithm elucidated in the previous sections can still be used. The result of applying a composition adapter can be perceived as a new composition pattern that then serves as the input composition pattern for the following composition adapter.

Stacking several composition adapters onto the same composition pattern might however cause conflicts. This is because a composition adapter that is applied latest might destroy the effect of a former composition adapter. Furthermore, a composition adapter applied first might destroy the applicability of a following composition adapter. This problem is related to the more general “feature interaction” problem encountered in most aspect-oriented approaches. Several workshops at ECOOP [PSC+01] and other conferences have addressed this issue and some approaches presenting a solution to this problem are

already emerging. These approaches can be categorized in three groups: *manual conflict prevention, semi-automatic conflict detection* and *automatic conflict detection*.

Manual conflict prevention consists of describing how a set of aspects have to cooperate. For example, Brichau et al. [BMD02] modularize aspects as logic metaprograms. For combining aspects, an aspect combination module is used. This logic module is parameterized with one or more aspects and contains rules that describe how the functionality of these aspects should be combined. As such, conflicts can be manually prevented by defining one or more aspect combination modules.

Semi-automatic conflict detection consists of manually adding a description concerning the different manners several aspects are able to cooperate. Using this description, the conflicts can be automatically detected. An example of a semi-automatic conflict detection approach is described by Klaeren et al. [KPRS00]. Here, the interaction between aspects is described as a set of invariants and post conditions, which can be checked both at compile-time and run-time. Although this system promotes automatic conflict detection, it is still the responsibility of the software developer to describe which aspects are able to cooperate and which aspects not.

A last approach consists of automatically checking whether the deployment of a set of aspects is a valid combination or not. Typically, the aspects are documented in some formalism in order to allow automatic conflict detection. It is however not required to describe all possible aspect combinations beforehand as is needed when using semi-automatic conflict detection approaches. Instead, a documentation of single aspects suffices for automatically checking conflicts. In [DFS02], a formal model is presented which is used to describe the join points and the behavior of aspects. As aspects are described formally, it is possible to detect if a combination of aspects induces conflicting behavior or not.

Automatic conflict detection is also possible with composition adapters because composition adapters contain a semi-formal documentation of the context where they apply to, namely their context part. Using step 1 of the composition adapter application algorithm, it is possible to automatically verify whether the application of a composition adapter is valid. Because the composition adapter documentation is already available, no additional specification or model has to be created in order to be able to realize automatic conflict detection. In addition, in some cases, it is possible to resolve conflicts automatically by, for example, re-arranging the sequence in which composition adapters are inserted. The expressiveness of the base language is however limited to protocols, while in the approach of Douence et al [DFS02], a Turing complete language is supported. The PacoSuite tool includes a preliminary implementation of such an automatic conflict resolution approach which is proposed in [VSJ03b]. A complete and sound approach is however subject of further research.

3.3.7 Concluding remarks

Notice that in the composition adapter application algorithm, two transitions are defined to match if their labels are equal. As explained in section 2.2.2, the semantic primitives are typically a hierarchy instead of a flat set. For example, the top-most primitive of the hierarchy should match all the other primitives. As such, the algorithms of the previous section have to be adapted to cope with this. This is however not a big issue, because it merely means that when implementing the product automaton and other DFA algorithms, a more sophisticated transition equality check has to be used. The algorithm also does not cope with downcasting the adapter part primitives to the concrete context. This can be easily achieved by remembering the mappings of primitives of the context part onto primitives of the composition pattern when calculating the paths matching the context part in step one. In step two, the inserted adapter parts are translated using the computed primitive mappings.

In case no non-determinism occurs, the complexity of the overall algorithm by combining the three steps, corresponds to $O(n^3+n^3+n^2)$ with n the maximum number of states of the composition pattern, the context part and the adapter part automaton. As a result, the overall algorithm executes in a time polynomial to the maximum number of states of the involved automata. However, when the resulting automaton is non-deterministic, the NFA-DFA conversion has to be calculated which seriously constrains the performance of the overall algorithm. Currently, there is no adequate solution to this problem. As such, the algorithm is not well suited for run-time reconfiguration of component-based applications, because in that context, execution time is an important factor. At component composition time however, there are no hard deadlines, but still, the computation has to be finished in a reasonable time. Because a composition pattern and a composition adapter describe a reusable and abstract protocol, the resulting automaton is quite limited in its size. As such, the performance of the composition adapter application algorithm is not a big issue at component composition time. Also notice that the algorithm presented here is a first version of the composition adapter application algorithm. It is very well possible to improve on the performance of this algorithm. Chapter 6 describes the implementation of the algorithm discussed here in the PacoSuite tool environment and Chapter 7 introduces a larger case study as a validation of the approach.

3.4 Automatic role mappings

3.4.1 First approach

In the previous section, an algorithm is introduced for applying a composition adapter onto a given composition pattern. A limitation of this algorithm is that the algorithm assumes a mapping is known from the roles of the context part of the composition adapter to composition pattern roles. This forces the component composer to manually map the roles. Consequently, one has to be familiar with not only the semantics of the involved composition adapters and patterns, but also thoroughly understand the meaning of the roles. As such, the approach requires more in-depth knowledge of the involved artifacts from the component composer. In a lot of cases however, only one possible mapping is available. It would be a great help for the component composer if the composition tool could map the roles automatically when only one solution is possible or could present a set of possible role mappings when multiple role mappings are possible. A straightforward algorithm to achieve this consists of iterating over all possible role mappings of composition adapter context roles and composition pattern roles and checking for each role mapping whether the context part matches with the composition pattern. If so, this role mapping is valid and can be added to the set of valid role mappings.

In order to clearly specify the algorithm, two auxiliary definitions are needed. Definition 18 defines a role mapping as a set of pairs of composition adapter context roles and composition pattern roles where no composition adapter context role is mapped upon more than one composition pattern role and no composition pattern role is mapped upon more than one composition adapter context role. A *complete* role mapping is defined as a role mapping that contains all context part roles. A *partial* role mapping does not contain all context part roles.

Definition 18: Role Mapping

S is a role mapping of a composition adapter A and a composition pattern P

$\Leftrightarrow$

$S \subset roles(A_C) \times roles(P) \wedge \forall(c_1,p_1), (c_2,p_2) \in S: c_1=c_2 \Leftrightarrow p_1= p_2$

where A_C is the context part of the composition adapter

The other helper definition (Definition 19) specifies that two role mappings are compatible if both role mappings do not map the same composition adapter context role onto another composition pattern role. Likewise, a composition pattern role contained in one role mapping may not be combined with a different composition adapter context role in the other role mapping. In other words, the union of both role mappings has to be a role mapping.

Definition 19: Compatible Role Mappings

Role mappings R and S of a composition adapter A and composition pattern P are compatible

⇔

R ∪ S is a role mapping of composition adapter A and composition pattern P

Algorithm 4 is a first attempt for finding all possible role combinations. This algorithm starts with a composition adapter and composition pattern. Then, the algorithm iterates over all possible role mappings of the composition adapter and the composition pattern. For every role mapping, Algorithm 1 is applied. When Algorithm 1 does not return any matching pairs, the role mapping at hand does not allow a valid application of the composition adapter. Otherwise, the role mapping is added to the set of valid role mappings `resultset`. When the `resultset` is still empty after iterating over all possible role mappings, the application of this composition adapter on the composition pattern at hand is invalid. If only one result is returned, steps two and three of the composition adapter application algorithm can be taken, in order to automatically insert the adapter part. If more than one result is returned, either an automatic choice has to be made on the basis of some heuristics or the component composer is queried to make the decision.

Algorithm 4: Finding all valid role mappings

```
1. P is a composition pattern
2. A is a composition adapter
3. resultset := ∅
4. foreach R is a complete role mapping of A and P
5.    result = compute matching paths using Algorithm 1
6.    if result ≠ ∅
7.          resultset := resultset ∪ R
8.    end if
9. end foreach
```

Notice that the algorithm as specified above does not remember the concrete composition pattern paths where the context part matches with. As such, when accepting a certain role mapping, these paths have to be recomputed. A more efficient version of this algorithm remembers these paths in order to avoid this unnecessary overhead. This is not specified in Algorithm 4 in order to prevent complicating the algorithm with unessential parts.

Algorithm 4 can be computed in a time $O(m^2n^3)$ with m the maximum number of roles in the composition pattern and composition adapter context part and n the maximum number of states contained in the corresponding automata. While this might seem not too bad at first sight, a lot of unnecessary work is performed. Remember that Algorithm 1 computes the intersection of a transformed composition pattern DFA with the context part DFA for every state of the composition pattern DFA. First of all, the transformation of the composition pattern DFA stays the same regardless of the role mapping at hand. In addition, computing the intersection for every state of the composition pattern DFA is structurally very similar for every possible role mapping.

3.4.2 Optimizing the first approach

The idea of the optimized algorithm consists of computing an intersection automaton between the composition pattern and the context part of a composition adapter in such a way that the result contains all paths that match the context part disregarding any role information. In addition, the direction information of both composition adapter and composition pattern roles is stored on every transition label of the resulting intersection. As such, the resulting intersection automaton contains for every transition a partial role mapping. Computing a complete role mapping for a given start-stop path in the automaton is then possible by accumulating the union of all partial role mappings. If the partial role mappings for a certain start-stop path are not compatible, the united role mapping is not a possible solution. All possible role mappings can be accumulated by iterating over all start-stop paths. Of course, the number of start-stop paths in an automaton is possibly infinite, so iterating over all paths is not possible. However, the number of transitions is finite and as such the number of different partial role mappings is finite. A path that repeats a transition for a number of times does not result in a different solution as the same path that only visits the transition once. As a result, the set of all valid role mappings can be computed.

3.4.2.1 The global algorithm

Algorithm 5: Optimized finding of all valid role mappings

1. $P=\{Q_P, \Sigma, q_{sP}, F_P, \delta_P\}$ is the composition pattern DFA
2. $C=\{Q_C, \Sigma, q_{sC}, F_C, \delta_C\}$ is the context part DFA
3. resultset := $\emptyset$
4. foreach $q \in Q_P$
5. $P' = \{Q_P, \Sigma, q, Q_P, \delta_P\}$
6. $R = P' \cap' C$
7. result = compute role mappings of R (Algorithm 6)
8. resultset := resultset $\cup$ result
9. end foreach

The optimized algorithm to find all valid role mappings is specified in Algorithm 5. The algorithm starts with the composition pattern and context part represented as a DFA. Afterwards, the strategy of the first step of the composition adapter application algorithm elucidated in Algorithm 1 is employed. This means that a transformed version of the composition pattern automaton is constructed for every state q of the composition pattern automaton. The state q is the start state and all other states are final states in this transformed version of the composition pattern automaton. Afterwards, the intersection with the context part is computed. The intersection is however slightly changed in order to ignore role direction information. In addition, a resulting transition label of the intersection contains a role mapping computed from the roles defined on the corresponding transitions in the context part automaton and composition pattern automaton. This is illustrated by Figure 30. At the left hand side a composition adapter context part DFA is shown with a single transition. The transition label specifies a B message from role *r1* to role *r2*. The middle DFA represents the composition pattern DFA containing a transition label that specifies a B message from composition pattern role *p1* to role *p2*. The intersection is taken disregarding any role information, so the two transitions match and are combined in the intersection. The resulting role mapping stored on the transition label is computed from the source and destination roles of the corresponding

transitions in the composition adapter context part DFA and composition pattern DFA. As such, the source roles *r1* and *p1* are mapped and the destination roles *r2* and *p2* are mapped.

The intersection is then traversed to find all valid role mappings. The next section explains how this is done. The resulting role mappings are then united with the resultset that has been accumulated from the previous iteration steps. The final `resultset` is then the result of the algorithm, namely all possible role mappings of composition adapter context part roles onto composition pattern roles.

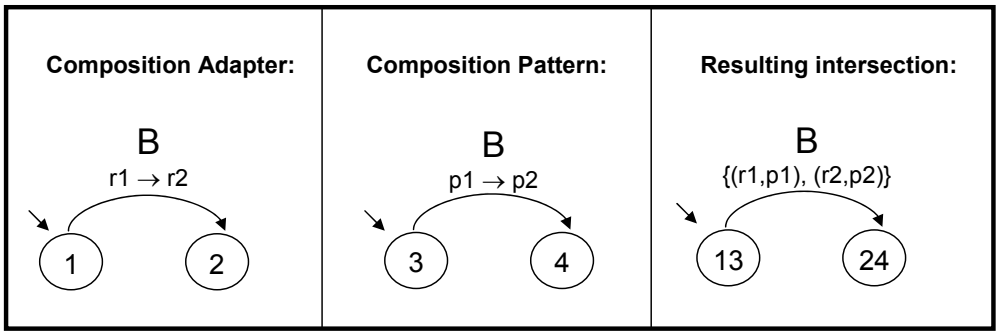

Figure 30: Combining composition adapter and composition pattern transition labels to form a partial role mapping.

Notice that an additional post-processing step is required to filter all valid role mappings from the result of Algorithm 6. The resulting set might contain partial role mappings, meaning that not all roles of the context part automaton are mapped. As a partial role mapping is not a valid result for applying the composition adapter application algorithm, they have to be filtered out. If the resulting set is empty after filtering out incomplete role mappings, the application of the composition adapter is not possible.

The complexity of Algorithm 5 corresponds to O(p(pc + z)) where p is the number of states of the composition pattern automaton, c is the number of states of the context part automaton and z is the complexity of Algorithm 6. Indeed, the algorithm iterates over all states in the composition pattern automaton. For each of these states, the intersection with the context part automaton is computed and Algorithm 6 is applied to traverse this intersection.

3.4.2.2 Traversing the intersection automaton

In order to compute the role mappings contained in an intersection automaton, the automaton resulting from the intersection is traversed to find all valid role mappings using ideas of dynamic programming [BD62]. Dynamic programming is a mathematical technique used as an approach to improve the computational efficiency of optimization problems via sequential decision making [Law96]. Dynamic programming ideas are employed in many application domains, like for example control theory, scheduling algorithms and shortest path algorithms. It is however the application of dynamic programming ideas to the world of network routing that can be recuperated to find all valid role mappings of the composition adapter and composition pattern at hand. In the standard algorithm, the network is represented as a graph and optimal paths between the nodes in the graph have to be calculated. Instead of using a global approach, the algorithm divides computing optimal paths over every vertex and constructs the global solution bottom-up from the partial solutions computed for every vertex. As such, the optimal path algorithm is reduced to a less complex problem and can be easily parallelized. The DFAs with a partial role mapping on every transition label computed by Algorithm 5 can in fact be considered as a graph where for each final state, the start state has to be reached. Instead of finding optimal paths cost-wise, our algorithm searches for paths containing a valid role mapping.

The algorithm is specified in detail in Algorithm 6. The algorithm starts by defining a mapping set A that maps each state to an empty set of role mappings (line 2). In addition, all final states of the intersection are added to a queue S (line 4). Then, the algorithm continuously pops states from the queue S (line 7). For each state s popped from the queue, all transitions resulting in s are enumerated (line 11). For each of these transitions, the role mapping N is fetched of this transition (line 12). In addition, the set of role mappings U stored for the state s is obtained using the map A (line 13). Likewise, the set of role mappings W stored for the origin state q of the transition is acquired (line 14). Furthermore, a new set of role mappings U′ is initialized to the empty set if U does not equal the empty set (line 16). Otherwise, the new set of role mappings U′ is initialized to a set with a single element N (line 18). The set U′ is used to accumulate the role mappings contained in U which are compatible with the role mapping N. Therefore, the role mapping N corresponding to the transition at hand is compared to each role mapping M contained in U (line 21). If the role mapping N is compatible with M, a new role mapping M′ is constructed from uniting M and N (line 22). In addition, if M′ is not already contained in W, then M′ is added to the new set of role mappings U′ (line 23-24). After iterating over each role mapping M contained in U, the new set of role mappings U′ is united with the original set of role mappings W and serves as the new set of role mappings for the state q (line 29). If for a single transition the set of role mappings of state q is updated and the queue does not already contain the state q, the state q is added to the queue for further processing (line 31-32). The result of this algorithm is the set of role mappings stored for the start state.

From the specification of Algorithm 6, it might not be evident to see that the algorithm always finishes. However, Algorithm 6 does always finish because of the test at line 23, where a new role mapping is not included in the result of a certain state, if the state at hand already has that role mapping in its result. After processing all transitions, no new role mappings can come up and as such, no states are updated any longer. As a result, no states are added for further processing and the algorithm finishes.

The best case scenario for Algorithm 6 consists of a state machine containing only a single start-stop path. This means that each state is connected with exactly one other state and a state is never connected with itself. As such, the algorithm just has to iterate over all states once in order to find the single role mapping. As such, the best case complexity of Algorithm 6 corresponds to O(m) where m is the number of states in the automaton. In worst case however, the automaton contains only a single state and all transitions start from and end at this state. In this case, the algorithm computes all possible combinations of transitions, which is of exponential nature. As such, the worst case complexity of Algorithm 6 consists of $O(2^t)$ where t is the number of transitions in the automaton. In the average case, the algorithm iterates over all states contained in start-stop paths that do not visit the same transition more than once. As such, the average complexity of Algorithm 6 corresponds to O(mx) where m is the number of states in the automaton and x is the number of start-stop paths that do not visit the same transition more that once. This is however an upper bound for the average-case complexity. It is possible that not all start-stop paths have to be traversed if they result in the same role mapping.

Algorithm 6: Optimized calculation of all valid role mappings (STEP 2)

```
1. {Q_R, Σ_R, q_sR, F_R, δ_R} = result of step 1 (Algorithm 5)
2. map A = {(q, ∅)|q ∈ Q_R}
3. queue S = ∅
4. foreach f ∈ F_R
5.     push(S,f)
6. end foreach
7. while #S ≠ ∅
8.    s = pop(S)
9.    foreach q ∈ Q_R
10.      isUpdated=false
11.      foreach a ∈ Σ_R where δ_R(q,a)=s
12.         N = getRoleMappingForTransition(q,a)
13.         U = getSetOfRoleMappingsForState(A,s)
14.         W = getSetOfRoleMappingsForState(A,q)
15.         if U = ∅
16.              U' := {N}
17.              isUpdated:=true
18.         else U' := ∅
19.         end if
20.         foreach M ∈ U
21.            if isCompatible(N, M)
22.               M' = M ∪ N
23.               if M' ⊄ W
24.                    U':= U' ∪ {M'}
25.                    isUpdated:=true
26.               end if
27.            end if
28.          end foreach
29.          pushSetOfRoleMappingsForState(A,q,U' ∪ W)
30.       end foreach
31.       if isUpdated ∧ ¬contains(S,q)
32.          push(S,q)
33.     end foreach
34. end while
35. result = getSetOfRoleMappingsForState (q_sR)
```

It is possible to describe this result in terms of the input automata of Algorithm 5, which are the composition pattern and context part automata, because Algorithm 6 traverses the intersection of the input automata. The number of states in the intersection automaton is quadratic to the number of states of the input automata. The number of start-stop paths that visit each transition maximum once is in fact at most the number of those start-stop paths contained in the input automata. This is because the intersection is calculated and as such only those start-stop paths remain that are contained by both automata. In other words, only those input strings remain that are accepted by both automata. As such, the average complexity of Algorithm 6 equals $O(n^2.y)$ where n is the maximum number of states contained in the composition pattern or context part automaton and y is the maximum number of start-stop paths that visit each transition only once contained in the composition pattern or context part automata. Combined with the overall algorithm step, which is specified in Algorithm 5, the overall complexity equals $O(n^3+ n^3y)$ where n is the maximum number of states contained in the composition pattern or context part automaton and y is the maximum number of start-stop paths that visit each transition only once contained in the composition pattern or context part automata. The number of start-stop paths that visit each transition only once are in the best and average cases quite limited and as such the optimized algorithm improves on Algorithm 4 elucidated in section 3.4.1. In worst case however, this algorithm is of exponential nature with respect to the alphabet of the automata. This is because the number of start-stop paths visiting each transition only once equals in worst case the number of all possible subsets of the alphabet. In that case, Algorithm 4 is the better choice.

3.4.3 An example

3.4.3.1 A simple example

In order to illustrate the algorithm to automatically find all valid role mappings, consider again the example of Figure 23. There is only a single role mapping possible, namely {(p1,r1),(p2,r2)}, so this should be the outcome of the algorithm. The algorithm starts by transforming the composition pattern DFA with state 6 as start state and all states as final states. When calculating the intersection with the context part DFA, an automaton is returned which contains no start-stop path with a length longer than 1. In other words, an empty automaton is returned and as such, the context part cannot be matched here. The automaton with state 7 as start state returns a result after the intersection with the context part DFA. This automaton is illustrated by Figure 31. Notice that the transition labels contain a role mapping of composition adapter context part roles onto composition pattern roles. This role mapping is computed from the role information specified onto the corresponding transition labels of the context part and composition pattern automata.

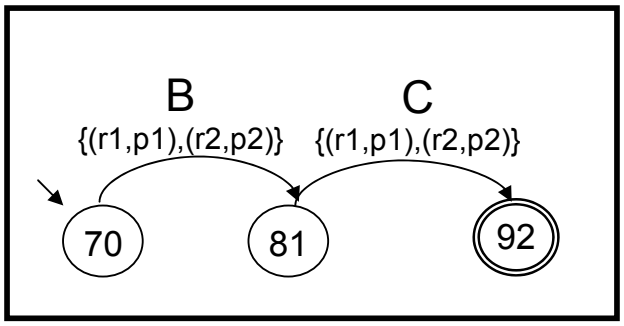

Figure 31: Resulting automaton after computing the intersection between the transformed composition pattern automaton for state 7 and the context part automaton.

The algorithm to traverse the automaton of Figure 31 starts by adding all final states to the queue. In this case, there is only one final state, namely state 92. In the first iteration step, state 92 is popped from the queue and processed. This means iterating over all transitions connecting to state 92. In this case, there is only one transition, namely the transition connecting to state 82. The set of role mappings of states 92 and 81 are empty, so the set of role mappings of state 81 is changed to the set {{(r1,p1),(r2,p2)}}. State 81 is updated and as such added to the queue. In iteration step two, state 81 is popped from the queue. State 81 also has only one incoming transition starting from state 70. The role mapping defined on this transition does not conflict with the single role mapping contained in the set of role mappings of state 81. As such, the set of role mappings of state 70 is updated to {{(r1,p1),(r2,p2)}} and this state is added to the queue. In iteration step three, state 70 is processed. State 70 does not have any incoming transitions, so the algorithm ends. Notice that this automaton corresponds to the best case scenario for Algorithm 6 as the algorithm finishes in three steps.

Subsequently, the resultset is updated to the result of traversing this intersection automaton, namely {{(r1,p1),(r2,p2)}}. Next, the algorithm continues iterating over all states of the composition pattern automaton and computes the intersection between the context part and transformed composition pattern automaton. At state 9, another non-empty intersection automaton is encountered. This automaton is in fact very similar to the automaton of Figure 31 and thus also results in the single role mapping {(r1,p1),(r2,p2)}. After iterating over all composition pattern roles, the result thus equals the single role mapping {(r1,p1),(r2,p2)}, which is the expected result. As such, the composition adapter can be directly inserted into the composition pattern using this role mapping.

3.4.3.2 A larger example

The automaton illustrated in Figure 31 is a very simple automaton. As such, traversing this automaton to find all valid role mappings is quite straightforward. In order to illustrate Algorithm 6 on a more involved intersection automaton, consider the example of Figure 32. This automaton corresponds to the intersection between a transformed composition pattern automaton and context part automaton as computed by Algorithm 5. The traversal of Algorithm 6 starts by pushing all final states on the queue, which is only state 5 in this case. In the first iteration step, all transitions connecting to the final state 5 are processed. State 5 has only one incoming transition B which has a role mapping {(r1,p1),(r2,p2)}}. State 4 and 5 have an empty set of role mappings, so the set of role mappings of state 4 is updated to the set {{(r1,p1),(r2,p2)}} after iteration one. State 4 is thus updated and added to the to-do queue. Iteration step two then processes state 4 and updates the set of role mappings of state 5 to {{(r1,p1),(r2,p2)}} and state 3 to {{(r1,p1),(r2,p2),(r3,p3)}}. As such, states 3 and 5 are added to the to-do queue. In iteration step three, the set of role mappings of state 2 is updated to the set {{(r1,p1),(r2,p2),(r3,p3)}, {(r1,p1),(r2,p2),(r3,p3),(r4,p4)}}. This is because state 3 connects with two transitions to state 2, each containing different role mappings. The other state on the queue, state 5, is now processed. State 5 has a single incoming transition leaving from state 4. The role mapping of this transition is united with the single role mapping of state 5, which renders the role mapping {(r1,p1),(r2,p2)}. This role mapping is already contained in the set of role mappings of state 4. As a result, sate 4 is not updated and not added to the to-do queue. In iteration step four, there is only a single state left on the queue, namely state 2. The set of role mappings of state 1 is updated to {{(r1,p1),(r2,p2),(r3,p3)}, {(r1,p1),(r2,p2), (r3,p3),(r4,p4)}}. The role mapping defined on the transition B from state 1 to state 2 is not added because it conflicts with both the role mappings contained in state 2. This is because the role r1 is mapped upon role p3 whereas the role mappings of state 2 each map r1 on p1. In the final iteration step, only state 1 is left on the queue. For state 1, there are no incoming transitions, so nothing has to be done for this state. No states are left on the queue and consequently the algorithm ends. The result of this algorithm is obtained from the set of role mappings

of state 1, namely {{(r1,p1),(r2,p2),(r3,p3)}, {(r1,p1),(r2,p2),(r3,p3),(r4,r4)}}. Because the first role mapping is a subset of the second, they can be merged together into one role mapping {(r1,p1),(r2,p2),(r3,p3),(r4,p4)}. When this is the only role mapping possible after computing all intersections, this mapping could be employed directly.

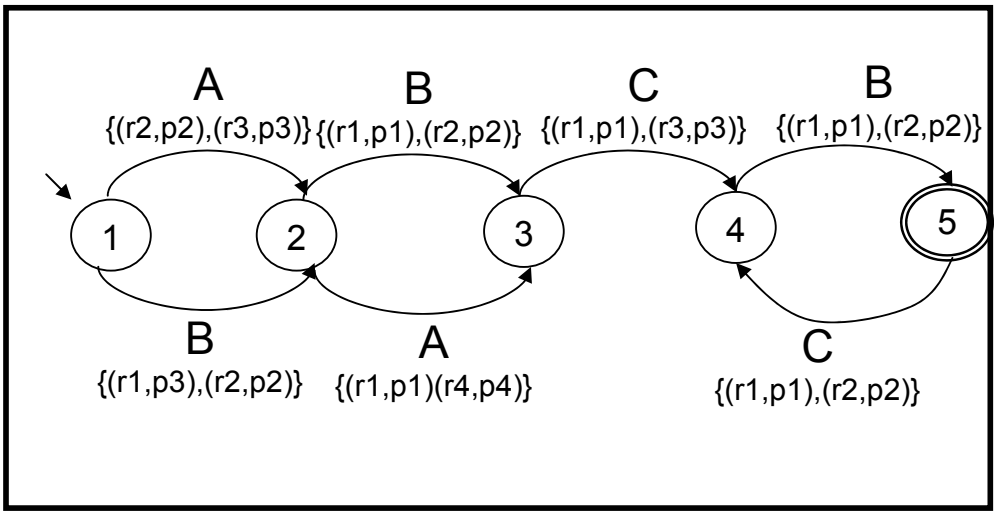

Figure 32: An automaton resulting from computing the intersection between a transformed composition pattern automaton and context part automaton.

Chapter 4 JAsCo

In this chapter, the JAsCo aspect-oriented programming language tailored for component-based software engineering is presented. The first section motivates why a new programming language is required. The second section explains the JAsCo language in full detail. Afterwards, different alternatives for implementing JAsCo are discussed and the alternative selected as technology for JAsCo is presented in detail. Finally, an extension of the JAsCo language that recuperates ideas of Adaptive Programming using traversal connectors is presented.

4.1 Introduction

4.1.1 Motivation

Employing the composition adapter introduced in the previous chapter (see Figure 18) to implement run-time checking of timing constraints is in fact not a very accurate approach. Remember that this composition adapter re-routes all signals through a timer role that is responsible for taking timestamps. Therefore, the timestamp of a sent message is only computed when the message arrives at the component mapped on the Timer role. As a result, there is at least an inaccuracy because of the delay of this additional re-routed message sent. Because the PacoSuite approach generates glue-code represented by a state-machine in between the components, the sent message is first processed through this state-machine in order to decide which component(s) to invoke. Therefore, the delay of the message sent is not a constant factor and can thus not be neglected. In addition, if the application works distributed, the network comes into play. The time needed for a message sent to and from the glue-code over the network is normally not predictable. This is especially true for internet applications. Furthermore, certain sophisticated component systems use a scheduler to pass messages to components. The scheduler might decide to give priority to another message because the deadline for the task related to that message is sooner. This scheduling process imposes yet another delay, making the timestamp even less accurate. As a result, our composition adapter approach to check timing constraints at run-time is not very well suited if a high precision is desired. The only way to achieve a correct timestamp consists of altering the components mapped on the composition adapter in such a way that the timestamp is computed in place at the exact moment a message is sent or received. A composition adapter however, is only able to alter the exterior behavior of a component by for example ignoring or rerouting messages. Aspects that require other adaptations can not be described using composition adapters, which is a major limitation. As such, an entire range of aspects cannot be represented by a composition adapter. Security aspects like confidentiality and access control [DVD02] for instance have to be inserted into the components in order to ensure non- bypassability.

In order to enhance the composition adapter model, we propose to implement a composition adapter in an aspect-oriented programming language. The idea is that the implementation in the aspect-oriented programming language is able to describe adaptations to the components themselves, while the composition adapter is used to describe the result of these adaptations to the external protocol of the components. However, the black-box idea is crucial in component-based software engineering. Components never depend on the internals of other components in order to achieve a loosely coupled system. Therefore, a middle ground has to be found between allowing adaptations to the internals of a component and keeping components as loosely coupled as possible. In fact, this discussion is currently also a hot topic in the general (not specifically targeted at components) aspect-oriented community[2]. Two main viewpoints exist regarding the expressiveness of the adaptations that aspects are allowed to describe and realise. One viewpoint, mainly advocated by the AspectJ supporters, claims that in general there is no reason to limit the expressiveness of the adaptations. As such, an aspect is allowed to adapt virtually everything of the target program. The drawback of this approach is that such expressive aspects are able to completely break the encapsulation of the target program. In addition, aspect compositions are very difficult to describe and control as aspects easily conflict due to the expressiveness of their adaptations. The other viewpoint, known as the restrictive viewpoint, argues that in order to keep the resulting

[2] See for example the keynote speech of Gregor Kiczales, “The Fun has Just Begun”, at AOSD 2003.

application manageable and traceable, aspects should be restricted in the adaptations they are allowed to describe. While sacrificing expressiveness of the aspect adaptations, compositions of restricted aspects are easier to describe and control. A common restriction on the adaptations that an aspect is able to describe is that the interface of the adapted component has to remain unchanged. Otherwise, the other components have to be aware of the inserted aspects in order to use the additional services offered by the adapted component [HC02].

In this dissertation, the second viewpoint is followed because it is especially true in the component-based context where loosely coupled components are considered crucial. Therefore, the aspect-oriented language that is used as an implementation of a composition adapter is limited to describe adaptations to methods and events contained in the public interface of the component. The public interface of a Java Bean consists of all the methods that can be invoked onto the component and all the events that are possibly fired by the component. As such, the AOP language is limited to describe adaptations of the public methods and fired events. No additional methods can be introduced, nor can internal fields or internal methods be altered. Of course, this restriction limits the expressiveness of the aspects. The black-box principle of the components is not seriously violated as aspects do not depend on the internals of the components, they only depend on the public interface.

Notice that in spite of the restricted set of possible joinpoints, the run-time checking of timing contracts concern can be implemented. Indeed, the proposed aspect language is able to alter the methods and events contained in the public interface of the component. As such, an aspect can be inserted that takes a timestamp inside public methods that have to be timed. It is however not possible to time internal methods of a component. This is not an issue because in component-based software engineering, one should not depend on the internals of a component. Therefore, it is logical that timing contracts are only specified on the public protocol of a component. Of course, the developer of a component has to be able to time the internals of its component. If this component is again built out of reusable black-box components, the proposed AOP approach is still applicable. Otherwise, if the component is implemented using another paradigm like for example Object Oriented Software Engineering (OOSE), another AOP approach that is suitable for OOSE has to be used.

4.1.2 Requirements of the language

Due to the specifics of component-based software development, the aspect-oriented language has to satisfy the following requirements:

- **General Purpose:** Component-based software engineering technologies are typically not limited to certain application domains. Therefore, the aspect language has to be able to specify aspects for a broad range of application domains.
- **Interaction:** The interaction between the various components from which an application is composed, is in most cases specific to the employed component model. For example, Java Beans communicate using the event model (i.e. firing events to interested listeners). The aspect language has to be able to declare aspects on typical kinds of interaction in component models.
- **Independence:** As specified by Szyperski's definition of a component (see section 2.1.1), the independence idea is crucial to component-based software engineering. Components never hard code a reference to a specific other component in order to increase reusability. Instead, they interact in an abstract way by for example firing events or posting messages to ports. Therefore,

aspects in this context should also be independently specified and it should be possible to reuse aspects over several compatible components.

- **Aspect Composition:** A consequence of the independence idea is that aspects do not directly refer to other aspects. For this reason, a strong mechanism is needed to specify combinations of independent aspects and components. For example, it should be possible to explicitly specify precedence relationships between aspects.

In addition, the technology that realizes the language has to comply with the subsequent requirements:

- **Independence from source-code:** Source-code of third party components is almost never accessible. Therefore, the aspect-oriented technology can not rely on the availability of source-code nor for analyzing the components or applying the aspects.
- **No invasive changes:** Invasively adapting third party components in order to weave aspects becomes unfeasible when considering quality of service (QOS) guarantees and digitally signed components. Third party components often ship with several quality of service (QOS) guarantees, like for example memory usage under certain conditions. If one weaves aspects into a component, obviously all the guarantees become void. In addition, it is possible that a component is encrypted or digitally signed so that it becomes impossible to modify the component at all. Therefore, the aspect-oriented technology should avoid invasive adaptations to the components at aspect weaving time.
- **First class at run-time:** Components are independent entities and remain often independent during execution. This allows easier configuration, maintenance and debugging of the system, as there is a one-to-one mapping of the components at design-time, compile-time and run-time. Aspects in this context also need to be first-class, even at run-time, in order to benefit from those advantages.
- **Dynamic:** For some application domains, like for example Web Services, it is vital that the system remains online as long as possible. Because aspects often represent non-functional concerns, like for instance logging or business specific properties [CDS+03, CDJ03], they need to be enabled and disabled quite frequently. Therefore, it has to be possible to add and remove aspects at run-time with minimal or no disruption of the system to which they are applied.

Table 1 illustrates a comparison of the mainstream aspect-oriented approaches discussed in section 2.3.2 with the requirements outlined above. AspectJ fulfills the first requirement, as it is a general purpose aspect language. AspectJ does not support applying aspect on typical component interaction schemes, independent aspects, and explicit aspect combinations. In addition, AspectJ employs source-code weaving in such a way that the advice of the aspects is completely inserted into the target components. As such, the aspects do not remain independent entities at run-time and dynamic weaving and unweaving of aspects is impossible. Notice that at this moment, a prototype byte-code weaver is available for AspectJ, but because it does not yet offer the same functionality as the traditional source-code weaver, it is not considered here.

Likewise to AspectJ, the composition filter approach is a general purpose language and does not allow applying aspects on typical component interaction schemes. Composition filters are however independent entities as there is a clean separation between the filter specification and implementation. Furthermore, filter declarations allow declaring explicit combinations of filters. As there is currently no stable implementation for the composition filter model, assessing the technology is rather difficult. The concernJ

[Wic99, Sal01] implementation for Java for example, employs source-code weaving in a way similar to AspectJ. Therefore, the concernJ implementation of composition filters suffers from the same limitations as the AspectJ approach.

Adaptive Programming aims at isolating a very different concern than the other three approaches, namely general-purpose traversal related concerns. Adaptive Programming does not allow applying aspects on specific component interactions. There is only limited support for combinations of non-orthogonal adaptive visitors. Adaptive visitors are however independently specified from a concrete context. The technology behind adaptive visitors is based on reflection and as such does not need invasive changes to the target components at all. Adaptive visitors remain first-class entities at run-time and it is also possible to easily add and remove adaptive visitors at run-time.

Table 1: Evaluation of current aspect-oriented technologies with respect to the requirements for intergration into PacoSuite.

Require-ments / Approach	General Purpose	Interaction	Independence	Aspect Composition	Independence from source-code	No invasive changes	First class at run-time	Dynamic
AspectJ	X							
Composition Filters	X		X	X				
Adaptive Programming	X		X		X	X	X	X
HyperJ	X		X	X	X			

The last approach, HyperJ, is a general purpose aspect language and allows to independently specify aspects as hyperslices and to explicitly combine them in a hypermodule. HyperJ does however not allow attaching aspects to typical kinds of component interactions. HyperJ is not dependent on source-code and employs a byte-code weaving approach to invasively merge the hyperslices as specified by a hypermodule. As such, the hyperslices are merged together and at run-time, the hyperslice notion is lost. Adding and removing hyperslices at run-time is also not possible.

Notice that Table 1 is by no means meant to be a complete and comprehensive evaluation of the aspect-oriented approaches at hand. Instead, Table 1 aims at providing the intuition that current mainstream aspect-oriented approaches do not completely fulfill the requirements outlined in the previous paragraphs. Although some of the approaches score significantly better than others, none of these approaches are entirely suitable to be used as an implementation of a composition adapter.

4.1.3 JAsCo overview

Because no suitable approach currently exists, we propose a new aspect-oriented approach that satisfies all these requirements, named JAsCo (**J**ava **As**pect **Co**mponents). JAsCo is an aspect-oriented extension to the Java language. The JAsCo approach is primarily based upon two existing AOSD approaches: AspectJ and Aspectual Components [LLM99]. AspectJ's main advantage is the expressiveness of its "pointcut"-

language. The pointcut language describes the condition on which the corresponding aspect advice is executed. These conditions are able to contain a wealth of constructs ranging from lexical properties of the program to dynamic information as activation records on the stack. However, aspects are not reusable, since the context on which an aspect needs to be deployed is specified directly in the aspect-definition. To overcome this problem, Karl Lieberherr et al. introduce the concept of Aspectual Components. In Aspectual Components, aspects are specified independently as a set of abstract join points. Explicit connectors are then used to bind these abstract joinpoints with concrete joinpoints in the target application. This way, the aspect-behavior is kept separate from the base components, even at run-time. JAsCo combines the expressive pointcut declarations of AspectJ with the aspect independency idea of Aspectual Components. JAsCo does however restrict the joinpoints that are possible to the public interface of the components, meaning public methods and fired events.

The JAsCo language is kept as close as possible to the regular Java syntax and concepts. Only a minimal number of new keywords and constructs are introduced. The JAsCo language introduces two new concepts: *aspect beans* and *connectors*. An aspect bean is a regular Java bean that is able to declare one or more logically related hooks as a special kind of inner classes. Hooks are generic and reusable entities and can be considered as a combination of an abstract pointcut and advice. Because aspect beans are described independently from a specific context, they can be reused and applied upon a variety of components. Connectors have two main purposes: instantiating the abstract aspect beans onto a concrete context and thereby binding the abstract pointcuts with concrete pointcuts. In addition, a connector allows specifying precedence and combination strategies between the aspects and components.

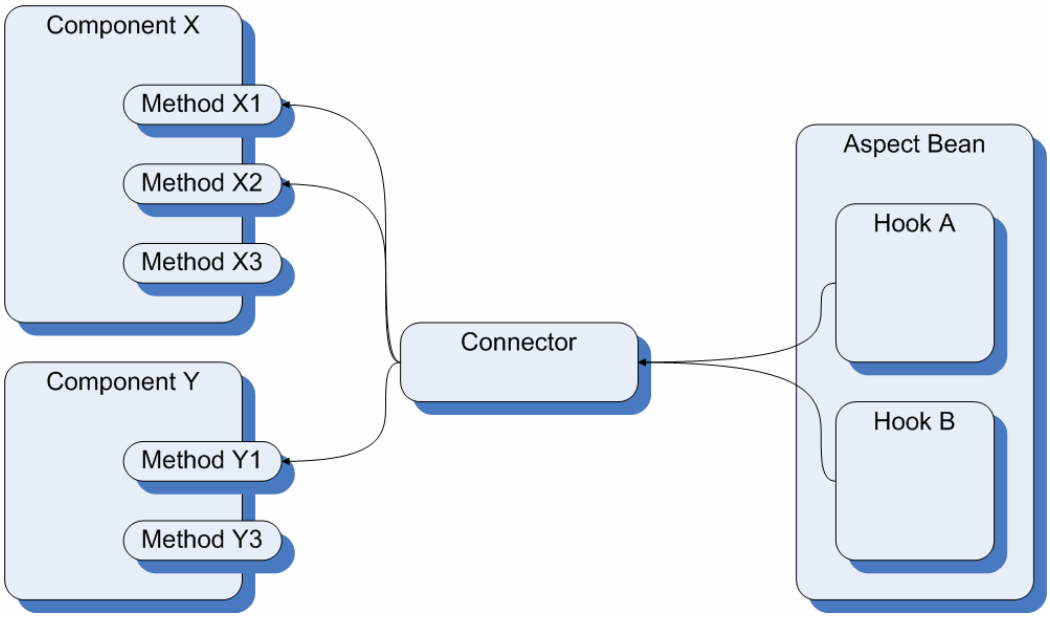

Figure 33: Schematic overview of JAsCo.

To make the JAsCo language operational, we introduce a new "aspect-enabled" component model. The JAsCo Beans component model is a backward compatible extension of the Java Beans component model where traps are already built-in. These traps are used to attach and detach aspects. As a result, JAsCo beans do not require any adaptation whatsoever to be subject to aspect application. The JAsCo component model enables run-time aspect addition and removal. In addition, aspects remain first class entities at run-time as they are not weaved and spread into the target components.

Figure 33 gives a schematic overview of the JAsCo approach. At the left-hand side, two normal Java Bean components are depicted which declare a number of methods and some events. At the right-hand side an aspect bean is shown that contains two hooks. A connector is then used to instantiate the abstract hooks to the concrete methods and events of the target component. Notice that the aspect bean is also able to declare normal methods and events. As such, the aspect bean can also be employed as a regular Java bean.

The next section explains the different features the JAsCo-language has to offer. The syntax of aspects and connectors is discussed using a small example. Section 4.3 introduces the JAsCo component model in more detail. Finally, section 4.4 proposes an Adaptive Programming extension to the JAsCo language.

4.2 Language

4.2.1 Introductory example

```
1  class AccessManager {
2
3     PermissionDb p_db = new PermissionDb();
4     User currentuser = null;
5     Vector listeners = new Vector();
6
7     boolean login(User user, String pass)  {
8        if(p_db.check_pass(user,pass)) {
9           currentuser = user;
10          return true;
11       }
12       else return false;
13    }
14    void logout() {
15       currentuser = null;
16    }
17
18    void addAccessManagerListener(AML listener) {
19       listeners.add(listener);
20    }
21    void removeAccesManagerListener(AML listener) {
22       listeners.remove(listener)
23    }
24    void fireAccessEvent(AccessEvent event) {
25       Iterator iter = listeners.iterator();
26       while(iter.hasNext()) {
27          AML listener = (AML) iter.next();
28          listener.accessEventOccured(event);
29       }
30    }
31
32    hook AccessControl {
33
34       AccessControl(method(..args)) {        When?
35          execute(method);
36       }
37
38       replace() {                            What?
39          if(p_db.check(currentuser))
40             return method(args);
41          else throw new AccessException();
42       }
43    }
44 }
```

Code Fragment 20: AccessManager aspect bean.

An example of a crosscutting concern is access control. Database systems for instance, need some control mechanism to manage user access to the data they hold. A similar concern could be stipulated in an ordinary application. Imagine a piece of software that runs on an operating system that allows only one user at the same time to be logged in. If users do not have the required permission, the access to some components of this system is denied.

Code Fragment 20 shows an access control aspect, specified using JAsCo. Notice that this aspect bean appears to be a normal Java Bean at first sight. Indeed, an aspect bean is able to declare normal fields and methods and even events (line 18-30 in Code Fragment 20). The crosscutting behavior itself is specified using the hook construct, which is a special kind of inner class. A hook specifies at least one constructor (line 34 till 36) and one behavior method (line 38 till 42), and is able to contain any number of ordinary Java class-members. A constructor of a hook specifies the condition on which the hook is triggered. When a hook is triggered, the hook behavior methods are executed. Parameters of a hook constructor are abstract method parameters that are bound to concrete methods or events in a connector. The constructor body specifies the condition itself on which the hook is triggered using these abstract method parameters. The constructor (line 34) of the `AccessControl` hook specifies that this hook can be deployed on every method which takes zero or more arguments as input. The constructor body (line 35) specifies that the hook is triggered whenever the method or event which is bound to the abstract method parameter `method` is executed.

The behavior methods of a hook are used to specify the various actions a hook needs to execute when hook is triggered. The `AccessControl` hook specifies only a replace method (line 38 till 42), which substitutes the behavior of the method that is about to execute when the hook is triggered. The replace method checks if the currently logged-in user has the proper access-permissions. This is done by querying the permissions database about the current user. When no problems are encountered, the original method is executed. Otherwise, an `AccessException` is thrown.

Deploying an aspect within an application is done by making use of connectors. Imagine that our application contains a printer component. Only people who have the appropriate print permissions, may access this component. Code Fragment 21 shows the connector that is used for instantiating the `AccessControl` hook upon the `Printer` component:

```
1  static connector PrintAccessControl {
2
3     AccessManager.AccessControl control =
4      new AccessManager.AccessControl(Printer.doJob(PrintJob));
5
6     control.replace();
7  }
```

Code Fragment 21: Connector that instantiates the AccesControl hook onto the doJob method of the Printer component.

A connector contains three kinds of constructs: one or more hook-instantiations (line 3-4), zero or more behavior method executions (line 6), and finally any number of regular Java constructs. The `PrintAccessControl` connector contains one hook instantiation `control`, which instantiates the `AccessControl` hook onto the `doJob` method that is defined in the `Printer` component interface. As such, the abstract method parameter `method` of this instance of the `AccessControl` hook is bound to the `doJob` method of the `Printer` component. Afterwards, the execution of the replace behavior method on the control hook is specified. To sum up, the connector of Code Fragment 21 declares that the replace method of the `AccessControl` hook should be executed, whenever the `doJob` method of the `Printer` component is executed. As a result, invoking the `doJob` method of the `Printer` component is restricted to users who have the required permissions.

A connector is also able to instantiate a hook onto the firing of an event, which is the main method of communication of the Java Beans component model. As a result, when the event is fired, the hook is triggered. Section 4.2.3.1.3 explains this in more detail.

4.2.2 Aspect Bean Syntax and Informal Semantics

In this section, the aspect bean syntax is discussed in detail. An aspect bean is able to contain one or more hooks that contain the crosscutting behavior itself. A hook is a participant of an aspect, and is used for specifying:

- when the normal execution of a method of a component should be "cut".
- what extra behavior should be executed at that precise moment in time.

For specifying the "when" part, a hook constructor and the isApplicable method are used. The "what" part is implemented in hook behavior methods.

4.2.2.1 Hook Constructor

A hook constructor is responsible for specifying in an abstract way "when" the hook's behavior should be triggered. In that sense, the hook constructor is similar to a kind of abstract pointcut. A hook constructor contains one or more abstract method parameters that denote the context where the hook can be instantiated upon. The abstract method parameters are bound to concrete methods or events when the hook is instantiated in a connector. Code Fragment 22 illustrates a hook constructor which declares three abstract method parameters. Method `m1` has to be bound to a method with two parameters: one of type `String` and one of type `int`. It is also possible to specify an abstract method parameter that can be bound by any method, using the `..` construct. Method `m2` in Code Fragment 22 is an example of this. The `..` construct can also be combined with specific types, but should always be the last argument type. For example, method `m3` specifies that it can be bound to any method that has as first argument type `PrintJob`[3]. Abstract method parameters have a scope that extends all over the hook. As a consequence, abstract method parameters can also be used in behavior methods. For example, in Code Fragment 20, the `method` abstract method parameter is used in the replace behavior method in order to call the original method bound to the `method` abstract method parameter.

The abstract method parameters are bound to concrete methods or events when the hook is instantiated in a connector. The binding mechanism is call-by-value, the abstract method parameter is bound to a reflective representation of the method. Call-by-name parameter passing is not applied in order to allow separate compilation. As JAsCo is an extension of Java, the scoping rules are also not altered. Also abstract method parameter invocations are lexically scoped in such a way that variable accesses in the method bound to the abstract method parameter are looked up in the lexical scope of the original method definition rather than in the lexical scope of the hook. For example, when `m1` in Code Fragment 22 is bound to a method `test` and the method body of `test` refers to a local variable `x`, then the `x` defined in the class that contains `test` is accessed and not an `x` defined in the aspect bean or hook. Notice that variable accesses inside the hook implementation (not the original implementation) will use the lexical scope of the hook to find the correct value.

The parameters of abstract method parameters are also qualified by parameter names in addition to types. This is because the parameters of an abstract method parameter are bound to the actual parameters that are passed to the method bound to the abstract method parameter. For example, the first parameter of method `m1` is named `s`. When the hook is triggered at run-time, the parameter `s` is bound to the first actual parameter passed to the method bound to `m1`. Parameters of abstract method parameters that are specified

[3] Notice that specifying application specific types like for instance PrintJob in a hook constructor, renders a hook less reusable. If possible, more generic or standard Java types are preferred.

using the `..` construct, are an exception to this rule. These parameters are bound to an array of all the remaining actual parameters that are passed to the method bound to the abstract method parameter at hand. For example, abstract method parameter `m3` specifies `..args2` as second parameter. The value of `arg2` is then an array that contains all the actual parameters except the first actual parameter where the method bound to abstract method parameter `m3` is executed with. Notice that there is no confusion possible when an abstract method parameter takes multiple arguments using the `..` syntax and when the first actual argument is an array. In this case, the parameter contains an array of which the first element is the actual argument, which is again an array.

Likewise to the abstract method parameters themselves, parameters of abstract method parameters have a scope that extends all over the hook. The parameter passing mechanism is call-by-value. As such, changing the parameter's value does not have an effect on the value of the original method argument. It is however possible to alter the arguments of which the method bound to the abstract method parameter is invoked with. See section 4.2.2.2 for a detailed explanation.

```
1  aHook(m1(String s, int i),m2(..args),m3(PrintJob pj, ..args2) {
2          execute(m1) && !(cflow(m2) || withincode(m3));
3  }
```

Code Fragment 22: An advanced hook constructor.

Specifying concrete types for parameters of abstract method parameters allows constraining the applicability of the hook to only methods of that signature. Attempting to bind the hook to an incompatible method signature in a connector results in a compile-time error. An incompatible method signature means either less or more parameters or types that are not equal to or not a subtype of the specified type. It is also possible to constrain the return type of an abstract method parameter. When no return type is specified, the abstract method parameter matches methods disregarding return type. When a return type is specified, the abstract method parameter can only be bound to methods that return that type or a subtype.

The constructor body specifies the triggering condition itself using the abstract method parameters. Every condition specification should at least specify the `execute` keyword[4]. This keyword states that the hook will be triggered each time the method bound to the abstract method parameter is executed. The `execute` keyword can be combined using the logical operators and (`&&`), not (`!`) and or (`||`) with two other keywords: `cflow` and `withincode`. The `cflow(x)` expression evaluates to true when the currently executing method is in the control flow of the method bound to abstract method parameter `x`. In other words, this means that when starting the execution of the currently executing method, the method bound to abstract method parameter `x` is not yet finished. Consequently, an activation record corresponding to the method bound to abstract method parameter `x` is still on the stack. The `withincode(x)` expression is more restrictive than `cflow` and obliges the method bound to `x` to be the direct caller of the currently executing method. In other words, an activation record corresponding to the method bound to `x` is the first activation record on the stack after the activation record of the currently executing method. For example, Code Fragment 22 specifies that `aHook` is triggered when the method bound to `m1` executes when it is outside the control flow of the method bound to `m2` and not directly called by the method bound to `m3`.

Notice that it is possible to specify more than one hook constructor to allow one hook to be instantiated onto several different abstract contexts. This increases the reusability of a hook. When a hook is instantiated in a connector, the correct hook constructor is resolved.

---

[4] Currently, only the *execute* is supported. In the future however, other keywords, like for example *call* or *after throwing* could be introduced.

4.2.2.2 Hook Behavior Methods

In order to specify the "what" part of a hook, behavior methods are used. Three kinds of behavior methods are available: *before*, *replace* and *after*. A before behavior method is able to specify behavior that should be executed before the original method that causes the hook to be triggered, is executed. Likewise, replace and after behavior methods specify behavior that executes respectively instead of and after the joinpoint where the hook is triggered. At least one behavior method has to be specified, otherwise the hook does not do anything. Notice that this is not enforced in the current JAsCo implementation. All behavior methods have no arguments. Before and after behavior methods do not have return types, the return type of replace is Object. Notice that this is not specified in a replace behavior method specification, see for example Code Fragment 21 (line 20). It is also possible to return primitive types like for example `int`, the JAsCo system automatically boxes primitive types to their object representation. As explained in the previous section, behavior methods are able to access abstract method parameters and parameters of abstract method parameters.

Before and after behavior are not able to influence the original method execution. Replace behavior methods are however able to do this. They can choose whether to invoke the original behavior and are also able to pass different arguments. This is done by employing abstract method parameters. As explained before, abstract method parameters of a hook constructor are bound to concrete methods in a connector and have a scope that extends over the entire hook. These abstract method parameters can be used to call the method bound to the abstract method parameter in two ways: using the original arguments or new arguments. Original arguments means that the method is called with the actual arguments that were passed to the concrete method that is bound to the abstract method parameter at hand. To realize this, just use the normal Java method invocation syntax and supply no arguments. Code Fragment 23, line 2, illustrates an example of this by invoking the method bound to the **`m1`** abstract method parameter of the constructor of Code Fragment 22. It is also possible to replace the original arguments with new arguments. This is done by calling the method and supplying the arguments. Code Fragment 23 line 1 shows an example where the method bound to **`m1`** is called with a newly specified string **`"test"`** and the original parameter **`i`**. Remember that parameters of abstract method parameters have a scope that extends all over the hook, so the argument **`i`** of **`m1`** is visible here.

```
1      m1("test",i);
2      m1();
```

Code Fragment 23: Invoking abstract method parameters of the hook constructor of Code Fragment 22.

It is sometimes required to just invoke the replaced method, independent of the concrete abstract method parameter that is bound to that replaced method. Therefore, JAsCo also supports AspectJ's `proceed` keyword. By specifying `proceed();` the replaced method is executed with the original arguments.

4.2.2.3 Inheritance

Inheritance of both aspect beans and hooks is supported. As a hook is a special kind of inner class, the inheritance semantics is equivalent to inheritance of inner classes. An aspect bean that extends another aspect bean inherits all the hooks declared in the parent aspect bean. Of course, the child aspect bean can also declare new hooks. It is also possible to define a new hook with the same name as a hook in a parent aspect bean. In that case, the new hook replaces the hook of the parent aspect bean. Code Fragment 24 illustrates an extended version of the `AccessManager` aspect bean of Code Fragment 20. The hook `TracingAccessControl` extends the `AccesControl` hook and adds a before behavior method that prints out some tracing information every time an access attempt is made. The original behavior method

and constructor of `AccessControl` are both inherited by the `TracingAccessControl` hook. As a result, when this hook is triggered, it first prints some tracing information and then verifies access.

```
1  class TracingAccessManager extends AccessManager {
2
3    hook TracingAccessControl extends AccessControl {
4
5      before() {
6        //print some tracing info
7      }
8    }
9 }
```

Code Fragment 24: Extended Access Manager aspect bean.

4.2.2.4 Other constructs

An aspect bean supports several other constructs that are useful in an aspect-oriented context. The following sections present:

- *calledobject* keyword: a way to refer to the object corresponding to the currently executed joinpoint,
- *isApplicable* method: is able to define an additional hook triggering condition in plain Java,
- *refinable* methods: a mechanism to defer implementation of certain behavior to the connector,
- *global* keyword: allows to refer to the aspect bean context from a hook,
- Multiple methods in the same aspect bean,
- Abstract method parameter reflection API.

4.2.2.4.1 The calledobject keyword

The critical reader might have noticed that the `AccessControl` hook of Code Fragment 20 checks access for given users globally without context information. In other words, for every part of the system where this instance of the `AccessControl` hook is applied to, the user is either granted access or refused. As a result, if one wants to realize a per component access policy, one has to instantiate an `AccessControl` hook with its own permissions database for each component in the system. This can be avoided when the component where the access attempt is made to is passed to the query in the permissions database. JAsCo allows referring to the object instance that executes the method that causes the hook to be triggered. This can be done using the `calledobject` keyword. Code Fragment 25 illustrates how the `calledobject` keyword could be used in the AccessControl example. The new replace method checks admission both on user id and on the component that is accessed (line 2).

```
1      replace() {
2        if(p_db.check(currentuser,calledobject.getClass()))
3            return method(args);
4        else throw new AccessException();
5      }
```

Code Fragment 25: Using the calledobject keyword to improve the AccessControl hook.

4.2.2.4.2 The isApplicable method

The `isApplicable` method is an extra construct that allows a more fine-grained control over when the hook should be triggered. Often, whether a hook is triggered or not depends on more than the conditions

that can be specified in a constructor. For example, a hook implementing an anniversary discount business rule is only triggered when the current date is the client's birthday. The `isApplicable` method allows specifying these additional conditions. In absence of such a construct, these conditions have to be tested in all the aspect behavior methods that are implemented. Furthermore, making this condition explicit by introducing a new keyword has the advantage that it allows more elaborated aspect combination strategies as the hook is only triggered when both the constructor body and the `isApplicable` method evaluate to true. In addition, introducing a new keyword allows optimizing this condition performance-wise in comparison to having it repeated in each hook behavior method. Code Fragment 26 illustrates an example of the `isApplicable` method. In this improved version of the `AccessControl` hook, the hook is only triggered when the user is not an administrator of the system because administrators are granted access to all components. When the hook is triggered, the replace behavior method inherited from the parent hook is executed.

```
1  class ImprovedAccessManager extends AccessManager {
2
3    hook ImprovedAccessControl extends AccessControl {
4
5      isApplicable() {
6        return !p_db.isAdmin(currentuser);
7      }
8    }
9  }
```

Code Fragment 26: AccessControl with administration check using the isApplicable method.

4.2.2.4.3 Refinable methods

```
1    hook AccessControl {
2
3      AccessControl(method(..args)) {
4        execute(method);
5      }
6
7      replace() {
8        if(p_db.check(currentuser)
9            return method(args);
10       else noAccessPolicy(currentuser);
11     }
12
13     abstract void noAccessPolicy(User user);
14
15   }
```

Code Fragment 27: Abstract method noAccessPolicy allows to customize the access policy in a connector.

It is possible to specify refinable methods in a hook by employing the abstract keyword. In contrary to a Java class, a hook containing abstract methods does not have to be declared abstract. The reason is that abstract methods can be provided with an implementation in a connector. In this way, it is possible to specify highly customizable hooks where some part of the logic is only known when instantiating the hook in a connector. As a result, one hook instantiated in different connectors can have varying behavior. Code Fragment 27 illustrates a new version of the *AccessControl* hook with a customizable policy for users that are denied access. An abstract method is introduced, named `noAccessPolicy` (line 13), that is called each time a user is rejected (line 10). This allows a connector to implement a specific rejection policy, for example blocking after three false attempts. Notice that it is also possible to provide abstract methods with

an implementation by overriding the method in a child hook. Abstract methods also allow a kind of aspectual polymorphism [MO03] as the specific implementation of the behavior is able to vary over the concrete types contained in the context the aspect bean is applied to.

JAsCo supports specifying default behavior for abstract methods in a hook. This can be achieved by implementing the abstract method instead of omitting the implementation. See Code Fragment 28 for an example. The default behavior is only used when the abstract method is not provided with an implementation in a connector or when the abstract method is not implemented in a child hook. Of course, one might wonder why the abstract keyword is there, since it is also possible to have an implementation. Why not allow every method to be redefined in a connector? The reason is performance. Methods that can be re-implemented in a connector require extra logic to fetch the implementation from the connector. This logic relies heavily on reflection, which demands a significant run-time overhead.

```
1      abstract void noAccessPolicy(User user) {
2           throw new AccessException();
3      }
```

Code Fragment 28: Providing an abstract method with default behavior.

Notice that it might be confusing to overload the abstract keyword with two different semantics in JAsCo, depending on whether the method is declared in the aspect bean or hook. Therefore, it is better to for example use the `refinable` keyword for refinable methods. For backward compatibility reasons, `abstract` is still used however. In the future, we plan to review the complete JAsCo syntax and semantics in order to clean up these issues.

4.2.2.4.4 Referring to the Aspect Bean

Regular Java inner classes have a lexical scope that includes their defining Java class. As such, it is possible to execute methods of the outer class from within the inner class. At the moment, JAsCo hooks do not have a lexical scope that includes the aspect bean. This is mainly a technical limitation of the current JAsCo implementation; it is the intention to provide this in the future. In order to allow hooks to read fields and call methods of the enclosing aspect bean, the `global` keyword is introduced[5]. For example, suppose the no-access policy of Code Fragment 27 has to log out every user that attempts to access a part of system where the user has no access permissions for. This can be achieved by calling the logout method of the enclosing aspect bean as illustrated in Code Fragment 29. Notice that the `global` keyword is still useful when the JAsCo implementation is improved to fully implement hooks as inner classes. In Java inner classes, it is not possible to refer to the enclosing Java class in order to for example pass it to a method invocation. JAsCo is able to support this using the `global` keyword.

```
1      void noAccessPolicy(User user) {
2           global.logout(user);
3      }
```

Code Fragment 29: Using the global keyword for accessing the enclosing aspect bean in which the hook is defined in order to log out the user at hand.

Hooks are meant to be semantically equivalent to Java inner classes. This means that the “this” keyword in a hook refers to the hook instance and not to the enclosing aspect bean instance.

---

[5] The global keyword is actually not very well chosen, because it gives the impression to refer to global entities like global variables, rather than to the enclosing aspect bean. Because JAsCo is already applied in practice in several projects, the global keyword is kept.

4.2.2.4.5 Specifying Multiple hooks in the same Aspect Bean

It is also possible to specify multiple hooks in the same aspect bean. Hooks defined in the same aspect bean are able to share the enclosing aspect bean context. As such, data and behavior can be shared between those hooks. For example, the `AccessManager` aspect bean expects that the login method is invoked explicitly onto the aspect bean. If the application already has a user management system, the application has to be aware of the fact that an aspect is present in order to invoke the login method on the aspect bean. It is possible to avoid this by declaring another hook that is responsible for 'capturing' the current user. As such, this additional hook can be instantiated on the already existing user management system in order to fetch the current user. Code Fragment 30 illustrates the `CapturingAccessManager` aspect bean that declares in addition to the `AccessControl` hook, a second hook named `CaptureUser`. This hook implements an after behavior method that fetches the current user and stores this information in the aspect bean. This information, namely the current active user, is then used in the `AccessControl` hook for verifying whether the currently logged-in user has access to the system.

```
1   class CapturingAccessManager {
2
3      PermissionDb p_db = new PermissionDb();
4      User currentuser = null;
5
6      void setUser(User user) {
7
8          currentuser =user;
9      }
10
11     hook CaptureUser {
12
13        CaptureUser(method(User user)) {
14          execute(method);
15        }
16
17        after() {
18          global.setUser(user);
19        }
20     }
21
22     hook AccessControl {
23
24         AccessControl(method(..args)) {
25           execute(method);
26         }
27
28         replace() {
29           if(global.p_db.check(global.currentuser))
30              return method(args);
31           else throw new AccessException();
32         }
33     }
34  }
```

Code Fragment 30: Specifying multiple hooks in the same aspect bean in order to capture the user at hand and to control access to the components in the system.

4.2.2.4.6 Reflection on abstract method parameters

As explained before, abstract method parameters of a hook constructor are bound to concrete methods in a connector and have a scope that extends over the entire hook. Apart from calling the bound method, the abstract method parameters support a reflection API, which is a subset of the normal Java Method

reflection API. The following methods can be invoked on every abstract method parameter in the current JAsCo implementation:

- `String getName()` : Returns the name of the method bound to this abstract method parameter.
- `String getClassName()` : Returns the full class name (with packages) is which the method bound to this abstract method parameter is defined.
- `int getModifiers()` : Returns an integer that represents the modifiers (private,static,synchronized, etc…) of this method. Use java.lang.reflect.Modifier to query which modifiers this integer represents.
- `Object[] getArgumentsArray()` : Returns the values of the actual parameters where the method bound to the abstract method parameter has been originally called with.
- `Class[] getArgumentTypes ()` : Returns the argument types of the method bound to the abstract method parameter.
- `Object getCalledObject()` : Returns the object instance where the method that is bound to the abstract method parameter is invoked upon.
- `Object invokeJAsCoMethod() throws Exception` : This method invokes the method bound to the abstract method parameter using the arguments that were originally passed to this method. The result is the same as calling the abstract method parameter directly without supplying arguments (see Code Fragment 23, line 2).
- `Object invoke(Object obj, Object[] args) throws Exception` : This method invokes the method bound to the abstract method parameter and allows to change both the receiver and the arguments of the method. The `obj` parameter is the receiver of the method. The `args` parameter represents an array with actual arguments for the method. Notice that both the receiver type as the number and types of the arguments have to be correct, otherwise an exception is thrown.

Code Fragment 31, line 2, illustrates how the abstract method parameter reflection API can be used to implement a logging hook. In this before behavior method (line 7-10), the name and class name of the method bound to the abstract method parameter `calledmethod` are printed on the standard output. As a result, every time the hook is triggered, some tracing information about the currently executing method is printed.

```
1  hook SimpleLogging {
2
3    SimpleLogging(calledmethod(..args)) {
4       execute(calledmethod);
5    }
6
7    before() {
8      System.out.println("Executing"+calledmethod.getName()+
9        " IN "+ calledmethod.getClassName());
10   }
11 }
```

Code Fragment 31: Using the abstract method parameter reflection API to print detailed tracing information.

4.2.3 Connector Syntax and Informal Semantics

A connector is responsible for at one hand instantiating a set of logically related hooks onto a concrete context and at the other hand thoroughly specifying how the instantiated hooks co-operate. For the latter, both precedence as combination strategies are supported.

4.2.3.1 Instantiating Hooks

In order to deploy a hook onto a given concrete context, the instantiation concept of object-orientation is used. Every abstract method parameter a hook constructor defines has to be bound to a concrete method by specifying a method signature in the connector. It is also possible to specify import declarations in a connector. As such, the method signatures do not have to contain full package names. Code Fragment 32 illustrates how a hook that contains the constructor of Code Fragment 22 can be instantiated.

```
1   import wsml.printing.*;
2
3   static connector TestConnector {
4      AnAspectBean.aHook testHook = new AnAspectBean.aHook(
5           void Classx.method1(java.lang.String, int),
6           int Classy.method2(float),
7           Object Printer.print(PrintJob)
8      );
9   }
```

Code Fragment 32: Instantiating a hook that specifies the constructor of Code Fragment 22.

4.2.3.1.1 Instantiating Hooks on multiple methods.

It is also possible to instantiate the same hook on several concrete methods at the same time. This can be achieved by specifying multiple method signatures delimited by commas and grouped by braces for a single abstract method parameter. In Code Fragment 33, the `AccessControl` hook Code Fragment 20 is instantiated on both the `startJuggling` and `stopJuggling` methods of the `Juggler` bean. There is an important difference when instantiating a hook on multiple methods using only one expression in comparison to employing several instantiation expressions: the former generates a single hook instance for all the methods while the latter generates separate instances. Instantiating a hook on multiple concrete methods as in Code Fragment 33 causes the same hook to be triggered for all the concrete methods. Instantiating a hook on a single method at a time causes a different hook instance to be triggered for each method.

```
1   static connector MultipleConnector {
2
3      AM.AccessControl control = new AM.AccessControl({
4         void sunw.demo.juggler.Juggler.startJuggling(),
5         void sunw.demo.juggler.Juggler.stopJuggling()
6      });
7   }
```

Code Fragment 33: Instantitaing the AccessControl hook on several methods.

4.2.3.1.2 Instantiating Hooks using wildcards.

JAsCo also supports wildcard expressions for specifying the method signatures in a hook instantiation. Both the star (`*`) and the question mark (`?`) wildcard expression are supported. The star wildcard expression matches with any finite number of characters (including none) while the question mark matches with only a single character. Characters are here all the possible characters that can be used in the Java

language. Instantiating the `AccessControl` hook on both the `startJuggling` and `stopJuggling` methods of the `Juggler` bean can thus also be realized using wildcards as depicted in Code Fragment 34. Notice that the example wildcard expression of Code Fragment 34 also matches the following method: `sunw.demo.juggler.JuggleD.Juggling()`. Of course, it is also possible to combine wildcard expressions with the multiple concrete methods construct of the previous paragraph.

```
1   static connector WildcardConnector {
2
3      AM.AccessControl control = new AM.AccessControl(
4         void sunw.demo.juggler.Juggle?.*Juggling()
5      );
6   }
```

Code Fragment 34: Using wildcard expressions to instantiate a hook on several methods.

4.2.3.1.3 Instantiating Hooks on events

In the Java Beans component model, components receive requests or information from other components by regular method invocations. For sending information or requests however, components fire events. JAsCo is able to intercept these events, by instantiating a hook with the `onevent` keyword. For example, if the `Printer` component throws an event when it finishes printing a document, this event can be intercepted by a hook in order to execute some appropriate action. Code Fragment 35 shows the instantiation of a logging hook which writes some statistics to file, when a printing job has finished. Currently JAsCo does not explicitly support vetoable events. It is possible to instantiate a hook on the firing of a vetoable event, but not on the acceptance or rejectance of such an event.

```
1  static connector EventConnector {
2
3     Logging.FileLogger logger = new
4        Logging.FileLogger(
5            onevent Printer.jobFinished(PrintEvent)
6        );
7  }
```

Code Fragment 35: Instantiating a hook on the firing of the jobFinished event of the Printer component.

4.2.3.1.4 Implementing abstract hook methods.

```
1   static connector ImplAbstractConnector {
2
3      AM.AccessControl control = new AM.AccessControl(* *.*(*)) {
4        void noAccessPolicy(User user) {
5               String str = "False attempt by :"+user.getName();
6               logger.setOuput(logger.ASCII);
7               logger.logToFile(str);
8               wsml.security.SecurityManager.banUser(user);
9        }
10     };
11  }
```

Code Fragment 36: Implementing the noAccessPolicy abstract method in a connector.

Abstract methods in a hook need to be provided with an implementation in a connector. The syntax for specifying an abstract method implementation is very similar to the syntax used for anonymous classes in Java. For example, suppose the no-access policy in the hook of Code Fragment 27 has to be customized to log every failed access attempt to a file and to ban in the future every user that tries to access a part of the

system where she/he has no access permissions for. Code Fragment 36 illustrates how this is implemented in JAsCo. The `AccessControl` hook is instantiated on all possible methods and events using a wildcard expression. Line 4 specifies the method signature of the abstract method that is implemented. Line 5 constructs the string that has to be logged by querying the username. Afterwards, the logger is initialized to log ASCII output (line 6) and the logged string is send to the logger (line 7). The last line (line 8) invokes the *banUser* method on the *SecurityManager* in order to ban the current user in the future. Notice that it is also possible to implement multiple abstract method parameters in a single hook instantiation expression.

4.2.3.1.5 Instantiating hooks on other hooks.

The previous example is in fact a very bad example because it hard codes the crosscutting logging concern in the connector. It is better to modularize the logging concern in a reusable hook and to instantiate it onto the `AccessControl` hook in a connector. Applying aspects on aspects is hardly ever supported in current aspect-oriented approaches (notable exception is EAOP [DFS02]). JAsCo however, does support aspects that interfere with other aspects. Code Fragment 37 shows how a logging hook can be applied onto the `AccessControl` hook. First the `AccessControl` hook is instantiated and the `noAccessPolicy` abstract method is implemented to ban the user when a false access attempt is made (line 3-7). Afterwards, the `FileLogger` hook is instantiated onto the `noAccessPolicy` abstract method of the `AccessControl` hook (line 9-10). As a result, when the `noAccessPolicy` method of the `AccessControl` hook is invoked, the `FileLogger` hook is triggered and some logging information is saved to file. Afterwards, the normal execution of the `noAccessPolicy` method resumes and the user is banned. Consequently, this connector has the same result as the connector specified in Code Fragment 36. The logging concern is however better modularized in comparison to Code Fragment 36.

```
1   static connector HooksOnHooksConnector {
2
3     AM.AccessControl control = new AM.AccessControl(* sy*.*(*)) {
4        void noAccessPolicy(User user) {
5               wsml.security.SecurityManager.banUser(user);
6        }
7     };
8
9     Logger.FileLogger logger = new
10        Logger.FileLogger(AM.AccessControl.noAccessPolicy(User));
11  }
```

Code Fragment 37: Applying the FileLogger hook onto the AccessControl hook.

It is crucial to be extra careful when instantiating hooks onto other hooks because this can easily lead to an infinite loop of two hooks causing each other to be triggered. Indeed, if the `AccessControl` hook is applied on the whole system as in Code Fragment 36, it would also affect the `FileLogger` hook. As a result, when the `noAccessPolicy` method is invoked by the `AccessControl` hook, the `FileLogger` hook is triggered. But when the `FileLogger` hook is triggered the `AccessControl` hook is at its turn triggered because it is applied on whole the system. Afterwards, the `AccessControl` hook invokes again the `noAccessPolicy` method and the application is stuck in an infinite loop. The current JAsCo implementation does not check for possible infinite loops at run-time. It is possible to check for potential infinite loops but the performance penalty will be high. The developer can however easily avoid an infinite loop caused by hook-on-hook instantiation by explicitly specifying that one of the hooks should not be triggered by the other hook. In Code Fragment 37 for example, the `AccessControl` hook

could use an extended constructor as depicted in Code Fragment 38. The second abstract method parameter has to be bound to the methods of the `FileLogger` hook in order to avoid an infinite loop.

```
1       AccessControl(method(..args), excluded(..args)) {
2         execute(method) && !cflow(excluded);
3       }
```

Code Fragment 38: Extending the AccessControl hook constructor to exclude certain parts of the system, notably other aspects.

4.2.3.1.6 Automatically creating multiple hook instances

Normally, when a hook is instantiated on several points of the application in one expression, there is only one hook instance created for all those points. If one wants to have different hook instances for different points in the application, one has to specify separate expressions for every point in the application where a unique hook instance is required. JAsCo also allows to automatically create multiple hook instances in a single hook instantiation expression. Currently, five constructs are supported `perall`, `perobject`, `perclass`, `permethod` and `percflow`. `Perall` creates a new hook instance each time the hook is triggered. This is useful when the hook has to perform some calculations and thereby store partial results that are not needed afterwards. `Perobject` generates one instance of the corresponding hook for each object instance where the currently executing method is invoked upon when the hook is triggered. In other words, for every matching object instance in the system, there is a separate instance of the hook. `Perclass` is similar to `perobject`, but then for classes. `Perclass` generates a new hook instance for every class type where the hook is triggered upon. `Permethod` means that there is a separate and unique instance for each method that causes the hook to be triggered when the method is executed. `Percflow` is similar to the `percflow` construct in AspectJ and instantiates a separate hook for every control flow through a joinpoint where the hook is triggered. Code Fragment 39 illustrates how `perobject` is used to instantiate a separate instance of the access control hook for every Juggler bean instance present in the application. `Permethod`, `perclass`, `perall` and `percflow` are specified in the same way as `perobject` in Code Fragment 39.

```
1   static connector PerobjectConnector {
2
3      perobject AM.AccessControl control = new AM.AccessControl(
4         void sunw.demo.juggler.Juggler.startJuggling()
5      );
6   }
```

Code Fragment 39: Automatically instantiating several instances of the same hook using the perobject keyword.

Several of the five per–keywords can be combined to realize an even more customizable hook instantiation. At the moment, the following combinations are supported: `perobject` with `permethod` or `percflow`; and `perclass` with `percflow`. The result of combining them is in fact the union of all the instances that would be normally generated when the keywords are applied separately. As a result combining `perall` with any other keyword does not make a lot of sense because this results in fact again in a `perall` behavior. Similarly, combining `permethod` and `perclass` generates the same as `permethod` alone. Code Fragment 40 shows an example of the combination of `perobject` and `permethod`. The result is that a new `AccessControl` instance is created for each method in each instance of the `Juggler` component. Calling for example the `startJuggling` method twice on the same `Juggler` instance causes the same unique hook instance to be triggered. However, calling this

method on another `Juggler` instance or calling the `stopJuggling` method on the first `Juggler` component results in another hook being triggered.

```
1   perobject permethod AM.AccessControl c = new AM.AccessControl(
2       void sunw.demo.juggler.Juggler.*()
3   );
```

Code Fragment 40: Combining the perobject and permethod keywords.

Notice that one should be cautious when using any of the five per-keywords because this potentially generates a lot of hook instances. As a result, these keywords require a lot of resources and slow down an application significantly!

4.2.3.1.7 Instantiating multiple hooks of the same aspect bean

It is possible to define more than one hook in a single aspect bean. If two or more of these hooks are instantiated in the same connector, they share the same aspect bean instance. As a result, changing data defined in the enclosing aspect bean in one hook is reflected in all the other hooks specified in that aspect bean. This is also the case when multiple instances of the same hook are created either manually or by employing the special keywords `perall`, `perobject`, `perclass`, `permethod` or `percflow` in a connector. Notice that if two hooks are instantiated in different connectors, the hooks each refer to a separate instance of the same aspect bean.

4.2.3.2 Executing Hook Behavior methods

Apart from specifying on which concrete points in the application a hook should be instantiated, a connector also allows to specify explicitly which behavior methods have to be executed when the hook is triggered. In addition to the before, replace and after behavior methods, an extra implicit behavior method, called default, can be employed. Specifying the default behavior method causes the execution of all the behavior methods that are specified in the hook. For example, specifying the replace behavior method of the `AccessControl` hook as in Code Fragment 21 or specifying the default behavior method as in Code Fragment 41 results in the same behavior, namely the replace behavior method is executed every time the hook is triggered. It is also possible to omit any behavior method execution specification, the default behavior method is then implicitly assumed.

```
1   static connector PrintAccessControl {
2
3     AccessManager.AccessControl control =
4      new AccessManager.AccessControl(Printer.doJob(PrintJob));
5
6     control.default();
7   }
```

Code Fragment 41: Default behavior method to execute the implemented behavior methods of a hook.

4.2.3.3 Precedence Strategies

A double motivation exists for permitting to specify behavior methods in the connector. The first advantage is that it enables advanced users of an aspect to tightly control the execution of the aspect behavior. The default method on the other hand provides an easy way for deploying an aspect within an application, without needing any knowledge about how the aspect behavior is executed. The second advantage of this approach is that it provides a partial solution for *the feature interaction* problem [PSC+01]. When multiple aspects are applied upon the same joinpoint of an application, some way is

needed to order the execution of their behaviors. JAsCo partly addresses this open issue in AOSD by the specification of the behavior method executions in the connector. All behavior methods of hooks that are triggered, are executed in the sequence specified in the connecter. If no explicit sequence is specified, they are executed in the order the hooks are instantiated. This means that all before methods are first executed in the sequence the corresponding hooks are instantiated. Afterwards, all replace methods are executed in the sequence the corresponding hooks are instantiated. Finally, all after methods are executed in the same sequence. The precedence of the behavior methods is enforced at run-time. If only a subset of the instantiated hooks are triggered, the sequence specified in the connector is still used for that subset. Behavior methods of other hooks are discarded. Notice that it is impossible to specify more than one type of behavior method in the same hook. In the current JAsCo implementation, two before methods for instance can not be implemented in the same hook. Executing the same type of behavior method for a given hook instance, results in a compile-time error.

```
1   static connector PrintAccessControl {
2
3      AccessManager.AccessControl control =
4           new AccessManager.AccessControl(Printer.*(*));
5
6      Logger.FileLogger logger =
7           new Logger.FileLogger(Printer.*(*));
8
9      LockingManager.LockControl lock =
10          new LockingManager.LockControl (Printer.*(*));
11
12     logger.before();
13     lock.before();
14     control.replace();
15     lock.after();
16     logger.after();
17  }
```

Code Fragment 42: Explicitly specifying precedence of several hooks.

For example, imagine that only one user at the same time may address the `Printer` component of the example in Code Fragment 21, and that the access to the printer still needs to be managed. In addition, some logging needs to be performed before a lock is requested and after the lock is released. Code Fragment 42 illustrates how the logging, access control and locking aspects can be instantiated simultaneously on the `Printer` component. JAsCo allows arranging the execution of the aspect behaviors by specifying the order in the connector body. Whenever mutual join points of the `control`, `logger` and `lock` hook instances are encountered, the sequence in which the behavior methods are specified in the connector is enforced. In the case of Code Fragment 42, first some logging is activated by invoking the before behavior method on the `logger` hook (line 12). Afterwards, the access to the `Printer` component is locked, by calling the *before* behavior method on the `lock` *hook* (line 13), so that no other user can access it. Next, the `AccessManager` checks if the user has the correct permissions to use the printer (line 14). Afterwards, the lock is freed (line 15), by calling the *after* behavior method, so that other users can access the `Printer` component again. Finally, some logging is executed again (line 16).

Notice that it is possible to write a connector where an after behavior method execution is specified earlier than a before behavior method execution. This does not result in the execution of the before behavior method after the after behavior method, which would be very counterintuitive. Instead, the before behavior methods are executed always before the execution of replaces and afters, but in the sequence specified in

the connector. For example, the sequence specified in Code Fragment 43 results in fact in the same behavior as the sequence in Code Fragment 42.

```
1      logger.before();
2      lock.after();
3      logger.after();
4      lock.before();
5      control.replace();
```

Code Fragment 43: Specifying before and after behavior methods in a weird sequence.

In comparison to AspectJ, JAsCo allows a more fine-grained control on the order in which aspects should be executed. In Code Fragment 42 for example, the sequence of the logger and locking hooks differs for the before and after behavior methods. This is not possible in most aspect-oriented approaches. In addition, JAsCo allows the precedence of aspects to vary over different applications as this is not hard coded into the aspects, but specified in the connectors.

```
1   static connector AroundConnector {
2
3      Hook1 hook1 = …
4      Hook2 hook2 = …
5      Hook3 hook3 = …
6
7      hook1.replace();
8      hook2.replace();
9      hook3.replace();
10  }
```

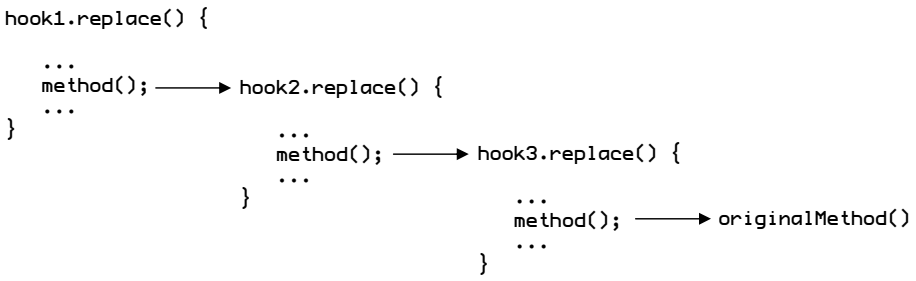

Figure 34: Chaining several replace behavior methods.

In the examples presented in the previous paragraphs, there is always a single replace behavior method execution specified. When multiple replace behavior methods are employed on the same joinpoint, the AspectJ around chaining approach is employed. This means that the firstly specified replace method is executed first. If this replace method invokes the original behavior, the second replace behavior method is executed instead of the replaced behavior and so on. When the last replace behavior method invokes the original behavior, then the original method is executed. Of course, this chain can be broken at any time when one of the replace behavior methods does not call the replaced behavior. Figure 34 illustrates an example of chaining several replace behavior methods. If `hook1` calls the original behavior in its replace method, the replace method of `hook2` is executed instead. Likewise, if `hook2` calls the original behavior in its replace method, the replace method of `hook3` is executed instead. The invocation of the original behavior in the last hook however, named `hook3`, does result in the execution of the original behavior.

One might wonder why a replace behavior method is named as such because it is in fact equivalent to the around advice in AspectJ. Originally however, replace behavior methods acted like a real replace and only one replace behavior method could be invoked per joinpoint. Experiments revealed that this is too restrictive in the sense that it limits the number of aspects that can be simultaneously applied. Because

JAsCo is already used in several projects and changing the name of a behavior method is a very drastic procedure, the name replace is kept.

4.2.3.4 Combination Strategies

Precedence strategies are a solution to some feature interaction problems, however other combinations of aspects require a more expressive way of declaring how they cooperate. For example, one might have to specify that when aspect A is triggered, aspect B can not be triggered. This problem could be solved by introducing an extra connector keyword *excludes* which specifies that aspect A excludes aspect B. However, other aspect combinations require additional keywords and it seems impossible to be able to define all possible combinations in advance. That's why we propose a more flexible and extensible system that allows to define a combination strategy using regular Java. A `CombinationStrategy` interface is introduced (see Code Fragment 44) that needs to be implemented by each concrete combination strategy. A JAsCo `CombinationStrategy` works like a filter on the list of hooks that are applicable at a certain point in the execution. Applicable hooks are hooks where the constructor and `isApplicable` method both evaluate to true.

```
1   interface CombinationStrategy {
2             public HookList validateCombinations(Hooklist);
3   }
```

Code Fragment 44: The CombinationStrategy interface.

The combination strategy interface can be implemented to realize the desired cooperation behavior. Each combination strategy needs to implement the `validateCombinations`-method, which filters the list of applicable hooks and possible modifies the behavior of individual hooks. For example, the combination strategy of Code Fragment 45 specifies an exclusion strategy. The combination strategy is initialized with two hooks in its constructor. The `validateCombinations` method specifies that the second hook is removed from the list of hooks that have to be triggered, whenever the first hook is present. As such, the second hook is not included in the hooks that are triggered and the behavior methods of the second hook are not executed whenever the first hook is present.

```
1   class ExcludeCombinationStrategy implements CombinationStrategy
2   {
3       private Object hookA, hookB;
4       ExcludeCombinationStrategy(Object a,Object b) {
5            hookA = a;
6            hookB = b;
7       }
8
9       HookList validateCombinations(Hooklist hlist) {
10
11           if (hlist.contains(hookA)) {
12                hlist.remove(hookB);
13           return hlist;
14      }
15  }
```

Code Fragment 45: Implementing an exclusion combination strategy.

The exclusion combination strategy has been used effectively in the context of business rules that are implemented using aspect-oriented technologies [CDS+03]. For example, take an online shop where a business rule is specified that gives a discount of 20% to frequent customers. In addition, another discount business rule specifies that a user receives a 5% discount on his/her birthday. The business policy is that the frequent customer discount may not be cumulated with other promotions. As a result, when the

frequent customer business rule is applicable, the birthday discount can not be applied. This collaboration can be implemented using the `ExcludeCombinationStrategy` of Code Fragment 45 as illustrated in the connector of Code Fragment 46. This connector instantiates two hooks on the checkout method of the online shop: one birthday discount hook (line 3) and one frequent customer hook (line 6). Afterwards, the behaviour methods of the hooks are specified (line 9-10). Finally, the exclude combination strategy is instantiated on both the frequent customer and the birthday hooks and added to the connector using the `addCombinationStrategy` keyword. The resulting application does not trigger the birthday discount hook when the frequent customer business rule is triggered. Notice that both hooks have to specify their triggering condition (frequent customer and birthday) in the `isApplicable` method (see section 4.2.2.4.2). Otherwise, the frequent customer hook would be always applicable and as a consequence the birthday discount is never executed.

```
1   static connector DiscountConnector {
2
3       Discounts.BirthdayDiscount birthday =
4            Discounts.BirthdayDiscount(* shop.checkout(*));
5
6       Discounts.FrequentDiscount frequent =
7            Discounts.FrequentDiscount(* shop.checkout(*));
8
9       birthday.replace();
10      frequentcustomer.replace();
11
12      addCombinationStrategy(
13         new ExcludeCombinationStrategy(frequent, birthday));
14  }
```

Code Fragment 46: Using the exclusion combination strategy to specify a discount combination.

It is also possible to define multiple combination strategies in the same connector. All these combination strategies are then merged using an approach similar to UNIX pipes. The sequence in which they are specified corresponds to the order in which they are employed in the pipeline. The first combination strategy receives the list of applicable hooks and filters them. The second combination strategy then receives this filtered list, performs its own filtering logic and passes the result on to the next combination strategy and so on. The hook list returned by the last combination strategy is then the list of hooks that have to be triggered at the current joinpoint.

```
1       HookList validateCombinations(Hooklist hlist) {
2
3            if (hlist.contains(hookA) && hlist.contains(hookB)) {
4                 hookB.setDiscount(hookB.getDiscoount() * 2);
5            return hlist;
6       }
```

Code Fragment 47: Adapting hook behavior depending on dynamic conditions using a combination strategy.

Normally, combination strategies function as a kind of filter on the list of applicable hooks at a certain joinpoint. However, combination strategies can also be employed to change some properties of the hooks depending on dynamic conditions. For example, suppose the birthday discount is meant to be increased instead of being not applicable when the user is a frequent customer. This can be easily achieved by the combination strategy of Code Fragment 47. In this `validateCombinations` method, the discount of `hookB` is doubled whenever the `hookA` is present in the list of applicable hooks at this joinpoint. Notice that when a hook is instantiated using one of the per-keywords (see section 4.2.3.1.6), multiple hook instances correspond to the same variable in a connector. However, only a single hook instance is able to

be applicable at each joinpoint. As such, combination strategies only work with that single hook instance. When properties of a hook are changed as in Code Fragment 47, this property is only changed in the single hook instance that is applicable at the current joinpoint.

4.2.3.5 Other constructs

A connector supports several other constructs that are useful in an aspect-oriented context. The following sections present:

- *static* keyword: allows to instantiate aspects on instance or class level,
- regular Java code in a connector,
- the run-time connector reflection API .

4.2.3.5.1 The static keyword

The attentive reader might have noticed that all connectors in the previous examples are preceded by the keyword `static`. This keyword states that the connector has to be applied on class level meaning that a hook is triggered regardless of the concrete object instance that executes the triggering method. Normally, this is the desired behavior for hooks that are instantiated in the connector. However, a connector can also be declared non-static, by omitting the `static` keyword. This causes the connector to only trigger hooks on those object instances that are explicitly registered with the connector at run-time. As a result, if the application does not register a single object instance, none of the hooks are able to be triggered, even if they are applicable on that context. In order to register object instances to a connector, the run-time connector API has to be used. See section 4.2.3.5.3 below for a more detailed explanation.

4.2.3.5.2 Java Code in a connector

It is also possible to include regular Java code in a connector in order to perform more elaborated calculations. For example, if the discount hooks of Code Fragment 46 have to be initialized with a discount, this can be specified in a connector as illustrated in Code Fragment 48. This connector invokes the `setDiscount` method (line 9-10) on both the birthday and frequent customer discount hooks to initialize the hooks with a concrete discount.

```
1   static connector DiscountConnector {
2
3       Discounts.BirthdayDiscount birthday =
4            Discounts.BirthdayDiscount(* shop.checkout(*));
5
6       Discounts.FrequentDiscount frequent =
7            Discounts.FrequentDiscount(* shop.checkout(*));
8
9       birthday.setDiscount(0.02);
10      frequent.setDiscount(0.20);
11      ...
```

Code Fragment 48: Initializing hooks in a connector.

In case the connector instantiates a hook with one of the per-keywords (see section 4.2.3.1.6), the hook instance variable does not refer to a single hook, but to a group of hooks. As a result, invoking a method on this variable invokes this method in fact on the whole group. For example, if the `birthday` discount of Code Fragment 48 would be instantiated using the `perobject` keyword, then the `setDiscount` method is invoked on every instance that is generated because of the perobject keyword.

Specifying other Java code apart from initializations of hooks and combination strategies is discouraged in the current JAsCo approach. The reason is that a connector should be solely used for instantiating hooks and specifying how they cooperate. All other logic has to be modularized in normal java beans and aspect beans!

4.2.3.5.3 Run-time connector API

JAsCo provides a standardized API for each connector at run-time. This API supports limited reflection like for example querying which hooks the connector has instantiated. In addition, two properties of the connector can be adapted: whether the connector is static or not and whether the connector is enabled or not. The following paragraphs explain the run-time connector API in more detail.

- `static Connector getConnector()` **:** A connector is implemented using the singleton pattern because only one instance for each connector is needed per application. This method retrieves the unique instance of the connector at hand where all the methods discussed in the following paragraphs can be invoked on.
- `void addInstance(Object object)` **:** This method registers an object instance to the connector. If the connector is non-static, it only triggers hooks on instances that are registered to the connector in this way. When the connector is static, it already triggers on all possible object instances and as a result this method has no direct observable effect. Obviously, the connector copes with this object instance when it is afterwards changed to non-static modus.
- `void removeInstance(Object object)` **:** This method is the counterpart of the method discussed in the pervious paragraph and removes object instances from the set of objects where the connector triggers hooks on in non-static mode.
- `Iterator getInstances()` **:** This method retrieves all object instances that are registered with the connector. Even when a connector is static, this method returns the list of currently registered object instances, although they are of no use as long as the connector is in static mode.
- `Iterator getHooks()` **:** This method allows to query all hooks that are instantiated by the connector. When the connector instantiates hooks using one of the per-keywords (see section 4.2.3.1.6), all the generated hook instances end up in the returned collection.
- `Iterator getMethods()` **:** Returns the list of methods in textual form where the hooks in the connector are instantiated onto. Methods that are specified using wildcards expressions are returned as such and not as a list of all the possible methods that match with the pattern.
- `void setEnabled(boolean bol)` **:** This method allows to enable and disable a connector at run-time. A disabled connector freezes all its hooks so that they are not triggered any longer. Although it is possible to compile and load connectors at run-time, this requires quite some overhead due to the compilation process. When the same connector has to be activated and de-activated often during the execution of the application at hand, it is better to enable and disable the connector using this method rather than compiling and removing the connector using the command-line tools.
- `boolean isEnabled()` **:** This method queries whether the connector is currently enabled or not.

- `void setAdaptOnClasses(boolean bol)` **:** This method allows to set the connector in static mode or in non-static mode. Invoke the method with `true` as parameter in order to change the connector to static mode. Calling the method with a `false` parameter, changes the connector to non-static mode. In non-static mode the hooks in the connector only trigger on instances explicitly registered with the connector. In static mode, the hooks trigger on all instances.
- `boolean adaptOnClasses()` **:** This method queries whether the connector is static or non-static. Returns `true` in case of a static connector and `false` in case of a non-static connector.

4.3 JAsCo Technology

4.3.1 Introduction

This section discusses the technological approach used to implement the JAsCo language. Let us recapitulate the requirements outlined in section 4.1.2 concerning the technology that is needed to realize the JAsCo approach. These requirements are:

1. not dependent on source-code
2. no invasive changes of target components
3. aspects should remain first class at run-time
4. dynamic application and removal of aspects at run-time

Obviously, the first requirement excludes source-code weaving as employed by for example AspectJ. The second requirement excludes in fact any sort of traditional weaving including byte code weaving that is for example used in HyperJ because traditional byte code weaving invasively changes the target components when applying aspects. The third requirement is not a fundamental issue, because even source-code weaving keeps aspects first-class if properly implemented. Nevertheless, it is a constraint that has to be fulfilled. The fourth requirement demands a run-time infrastructure to support run-time aspect management and likewise to the second requirement excludes any sort of traditional weaving. To sum up, the technology used for JAsCo may not use traditional weaving and has to provide an extensive run-time infrastructure. Indisputably, the biggest challenge is finding a replacement for traditional weaving in order to intervene with the normal execution of the application at hand. Despite the limitations caused by the above requirements, several different possibilities exist to establish such an interception system. The possibilities considered in this dissertation are:

- Developing a new Java virtual machine or modifying an existing open-source implementation.
- Using the debugging interface of the Sun's Java Virtual Machine
- Decorate every component with a wrapper
- Or, use a new component model where explicit traps are already built-in.

Due to the enormous task, developing a new aspect-enabled Java virtual machine is not an option for our research. Modifying an existing Java virtual machine like for example Jikes [BFG02, BGH03] from IBM however, is feasible. The idea is that this modified virtual machine is able to notify an aspect framework of events where aspects are interested in. In addition, the aspects are able to change the behavior of the currently executing method and even change some other properties like for example the execution sequence or data. Because a specialized virtual machine is used, this solution requires the least run-time overhead for applying aspects. Furthermore, this approach is by far the most powerful as there is no technological limitation at what the aspects can describe. However, employing a specialized virtual machine also has several drawbacks. First of all, an obvious drawback consists of the lack of portability of the approach. Currently, there are plenty of different virtual machines for diverse platforms and application domains. Therefore, forcing to employ a specialized virtual machine in order to apply aspects, seriously limits the approach. Another problem is that currently no open-source virtual machine implementations exist that are able to compete both feature-wise and performance-wise with commercial virtual machines. As a consequence, while the modified aspect-enabled virtual machine only imposes a small performance

overhead in comparison with the original open-source version, a much bigger difference will be noticeable with commercial virtual machines. The last drawback is that such a drastic change as employing a new virtual machine stands out against the philosophy of JAsCo to stay as close as possible to the regular Java approach.

Another solution consists of using the Java Virtual Machine Debugging Interface (JVMDI) to be notified of every event in the application at hand. The debugger API is included in all the standard Java virtual machines of Sun and other vendors and allows intercepting every imaginable event in the target application. The biggest advantage of this approach in comparison with the other approaches is that it is the most flexible and portable approach. No new virtual machine has to be used and the system works with every Java application. A serious drawback of this approach is however performance. Because employing the debugging interface means that the application has to be run in debugging mode, a serious performance overhead is imposed on the entire application. Even if only a very small portion of the application is affected by aspects, the entire application suffers from a considerable performance penalty! Another drawback is that the adaptations that aspects are able to describe are limited to those adaptations that the JVMDI offers. In addition, specialized Java Virtual Machines do not always support the full range of JVMDI features. In particular the embedded virtual machines typically only offer a small subset of debugging features.

The third possibility for implementing JAsCo consists of deploying wrappers around every component in the system. These wrappers are used to intercept all messages to the components. The wrapper approach has the huge advantage that it does not require any changes to the virtual machine or target components. As a result, the wrapper approach should be less complex in comparison to the other possibilities and the easiest to implement. However, deploying a wrapper to every component in the system does impose a considerable performance overhead due to the extra level of indirection. This is a waste of resources if only a very small part of the system is affected by aspects. However, there exist systems like for example Gilgul [CSC01] that allow to dynamically apply wrappers and so avoiding the need for applying wrappers to all components. In addition, Gilgul solves the issue concerning object identity that arises when a wrapper is dynamically applied to a component. However, Gilgul employs a modified version of the Java virtual machine. As such, this approach also inherits the drawbacks of using a specialized virtual machine. The main drawback however of implementing aspects as wrappers is that it is impossible to make sure that the wrapper is not bypassed. For example, a component could pass itself to another component in order to expect a callback. To avoid this, the wrapper has to check every passed argument. If the component is detected, the component is replaced with the wrapper. However, detecting this is not always possible. A component could for example pass a part of the component to the other component. As such, this subcomponent is not the same as the component itself and the wrapper is not able to detect this and replace the subcomponent. Another drawback of the wrapper approach consists of the fact that wrappers are external to the Java bean. As such, wrappers are not able to access nor adapt non-public methods or fields of the Java bean. All the other approaches outlined here, do allow accessing the internals of a Java bean.

The final possibility to implement the JAsCo language consists of introducing a new component model where the traps that allow aspect addition and removal are already built-in. The component model can be made backward compatible in such a way that these new components can be used as regular Java Beans. Ideally, new components are shipped using this new component model. In this way, attaching and removing aspects to components implemented in the new component model does not require any adaptation whatsoever to the target beans. Of course, expecting all component developers to develop using this new component model is rather utopic. However, it is possible to automatically transform a regular Java bean into a component of the new component model at application startup time by using advanced

byte-code adaptation techniques that insert the traps. This way, run-time application and removal of aspects is still possible with regular Java beans. However, the issues related to QOS guarantees or encrypted components remain unsolved for normal Java Beans. Another drawback of this approach is that it requires a performance overhead to execute the built-in traps. Because the traps are on a higher level compared to the adapted virtual machine, the performance penalty of this approach is larger. However, in comparison to using the debugging interface the performance overhead is expected to be smaller because only the parts of the system are equipped with traps.

We eventually choose for introducing a new component model because it is the approach that realizes the best middle ground between staying as close as possible to the regular Java concepts, imposing a minimal performance overhead and achieving the highest flexibility and portability while satisfying the requirements outlined in the beginning of this section. The next section describes the approach in more detail.

4.3.2 New Component Model

The JAsCo beans component model is a backward compatible extension of the Java beans component model where the traps that enable aspect interaction are already built-in. Every trap refers to the JAsCo run-time infrastructure that manages the registered connectors and aspect beans. Figure 35 illustrates the run-time infrastructure schematically. The central connector registry serves as the main addressing point for all JAsCo entities and contains a registry of connectors and instantiated hooks. The connector registry is notified when a trap has been reached or when a connector has been loaded. As such, the database of registered connectors and hooks is updated dynamically.

The left-hand side of Figure 35 shows the JAsCo bean *comp1*. All methods of *comp1* are equipped with traps, so that when a method is called, the execution is deferred to the connector registry. The main method of communication of Java Beans is event posting, so throwing an event also reschedules execution to the connector registry. When a trap is reached, the connector registry looks up all connectors that registered for that particular method or event. Connectors that are declared non-static and thus only trigger on specific instances are only taken into account when the object instance of the current joinpoint is explicitly registered with the connector. Every valid connector on its turn calculates which hooks are applicable on the current joinpoint in the application by evaluating the hook constructors and calling the *isApplicable* methods. Hooks that are not applicable are filtered from the list of applicable hooks. Afterwards, the applicable hook list of each valid connector is filtered by all combination strategies that are defined in the connector. All the resulting hook lists are then merged to form a single list of hooks that are allowed to be triggered. Finally, the before, replace and after behavior methods are executed on the resulting list of applicable hooks as specified in the section 4.2.2.

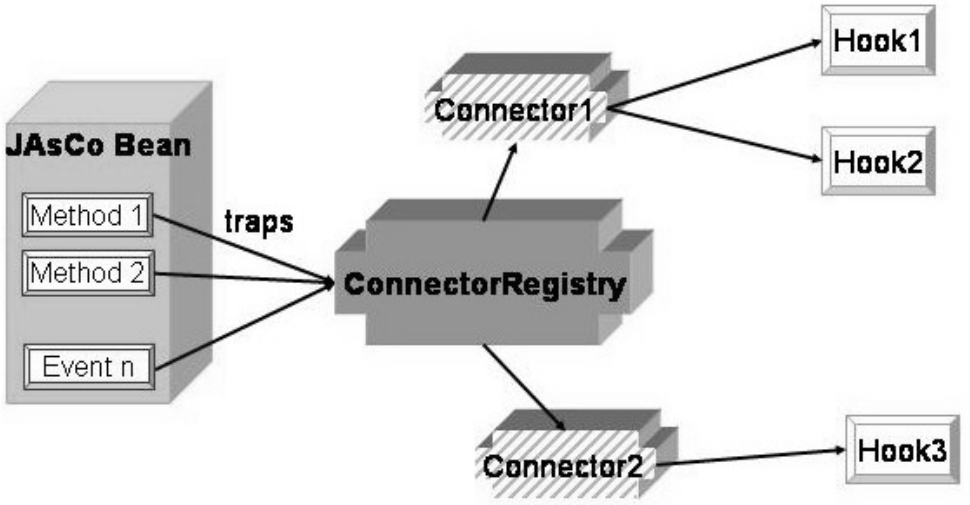

Figure 35: Schematic overview of the JAsCo run-time infrastructure.

Notice that the sequence in which connectors are processed is not predictable. The idea is that hooks that are logically related are contained inside one connector. Precedence and combination strategies can then be used to manage the cooperation between these logically related hooks. As such, connectors are orthogonal entities and the sequence in which they are processed does not matter. Experiments however revealed that this is not always possible in practice. Sometimes hooks have to be instantiated in different connectors while they are still logically related. This is for example the case when new hooks have to be added to the system at run-time. The only option consists of instantiating these hooks in a new connector. Therefore, in order to manage connector combinations and in particular connector precedence, an extra concept named a connector combination strategy has been recently introduced. Section 4.3.3 discusses connector combination strategies in more detail.

While the advantages of having explicit connectors at compile-time are obvious, one might question why we keep this extra level of indirection at run-time. The idea is that connectors serve as collections of related aspects. We want to add, remove or edit the behavior of these aspects at the same time. For example, if we want to alter a connector to only trigger on a certain instance of a JAsCo bean and not on other instances, we can now do this easily by accessing the connector. Otherwise, we have to manually notify all the affected aspects of this change. In addition, this would require that the aspects have public methods for adapting their behavior other than those defined in the aspect's source code, what could be confusing. In short, it is possible to get rid of an explicit connector at run-time to gain efficiency, but we would end up with a polluted model where the connector's logic is spread over different entities.

Another possible problem of a central connector registry is distribution. A single central connector registry in a highly distributed environment causes an important performance bottleneck. It is however possible to solve this by (partially) replicating the connector registry over several well placed nodes in the network. Generally, the cost of synchronization for new connectors that are added and removed will be much less than the cost of executing all traps through a central connector registry.

Our approach is very flexible to support unanticipated run-time changes. Connectors can be easily loaded and un-loaded at run-time. The connector registry detects whether connectors are removed or added to the system and takes appropriate actions. We also support the instantiation of a hook on expressions that contain wildcards. These wildcard expressions are matched at run-time. Consequently, when a new component is added to an application, it is automatically affected by all aspects that were declared using

wildcards and turn out to be applicable on the new component. This is not possible in most other dynamic AOP solutions. In addition, run-time wildcard matching makes the compilation of a connector that instantiates a wildcard aspect very fast, this in comparison to approaches that resolve these wildcards at compile-time. On the other hand, matching the wildcard expressions at run-time degrades the run-time performance.

4.3.3 Connector Registry API

```
1   interface ConnectorCombinationStrategy {
2       public Vector validateCombinations(Vector,RuntimeContext);
3   }
```

Code Fragment 49: Connector combination strategy interface.

The connector registry introduced in the previous section has a public interface that allows to enroll connector combination strategies. Connector combination strategies can be used to control the execution sequence of connectors. Likewise to normal combination strategies, connector combination strategies can be implemented in plain Java and allow to filter the list of all connectors at each joinpoint encountered. Code Fragment 49 illustrates the `ConnectorCombinationStrategy` interface. By implementing the `validateCombinations` method, the programmer is able to filter the list of connectors available at this joinpoint. The RuntimeContext parameter can be used to do some reflection about the current joinpoint. The Vector of connectors returned is used to find all hooks that are have to be triggered. When deleting a connector from the list, the hooks that are instantiated in this connector are not being triggered any longer. Deleting a connector from the list does not cause this connector to become permanently removed from the system, it is only removed from the list of connectors at the current joinpoint.

```
1   class Rearranger implements ConnectorCombinationStrategy {
2
3         public Vector validateCombinations  (Vector l,
4             RuntimeContext c) {
5
6            Collections.sort(l,new Comparator() {
7                public int compare(Object o1, Object o2) {
8                    String s1 =  o1.getClass().getName();
9                    String s2 =  o2.getClass().getName();
10                   return s1.compareTo(s2);
11               }
12           });
13           return l;
14        }
15  }
```

Code Fragment 50: Implementing a ConnectorCombinationStrategy in order to re-arrange connector sequence.

By implementing the `ConnectorCombinationStrategy`, arranging the connector sequence is possible. In addition, it is also possible to change some properties of the connectors using the run-time connector API. Code Fragment 49 illustrates a connector combination strategy that changes the sequence of the supplied connectors. The sequence of the connectors is changed to the alphabetic order of their names.

In order to enroll the connector combination strategies, the connector registry run-time API has to be used. Apart from adding and removing connector combination strategies, a limited form of reflection is supported. The run-time connector registry API supports the following methods:

- `static void addConnectorCombinationStrategy( ConnectorCombinationStrategy strategy)` : This method adds a connector combination strategy to the connector registry. Connector combination strategies are executed in the sequence they are added. As such, the connector combination strategy that is added latest, filters the hooks returned from the previous connector combination strategy.
- `static void removeConnectorCombinationStrategy( ConnectorCombinationStrategy strategy)` : This method removes a connector combination strategy from the connector registry.
- `static Iterator getConnectors()` : This method returns an iterator over the list of connectors. The connectors can be accessed and some properties can be changed using the connector run-time API.
- `static void addConnectorRegistryListener( ConnectorRegistryListener l)` : This method adds an observer to the connector registry. This observer is notified each time:
  - A new connector is detected in the system.
  - A connector is removed from the system.
  - A connector combination strategy is added.
  - A connector combination strategy is removed.
- `static void removeConnectorRegistryListener( ConnectorRegistryListener l)` : This method removes the observer `l` from the connector registry.

4.4 Adaptive Programming and JAsCo

The JAsCo language presented so far is focused on describing aspects in the traditional AspectJ meaning. Adaptive Programming allows modularizing a different kind of crosscutting concerns using Adaptive Visitors. This section presents an extension of the JAsCo language that recuperates ideas from Adaptive Programming. The JAsCo Adaptive Programming implementation allows regular aspect beans to function as adaptive visitors. As such, aspect beans are able to implement both traditional AspectJ aspects and adaptive visitors, which increases their reusability even more. Aspect beans are instantiated as adaptive visitors onto a given traversal strategy in a special kind of connectors, named traversal connectors. In addition, traversal connectors are able to specify explicit precedence and combination strategies between cooperating adaptive visitors represented as aspect beans.

4.4.1 Introduction

Adaptive Programming [Lie96, LOO01] aims at isolating crosscutting concerns that originate from scattering an operation that involves several participants among these participants in order to adhere to the Law of Demeter. As explained in section 2.3.2.3, Adaptive Programming solves a very different kind of crosscutting concerns in comparison to other AOP approaches. In fact, Adaptive Programming is complementary to most other AOP approaches and can thus be employed simultaneously. An extension for AspectJ, called DAJ [SL02, LL02], has been recently introduced. DAJ combines the Adaptive Programming ideas with the AspectJ language. The Adaptive Programming ideas are applicable to the JAsCo approach as well. However, the ideas of JAsCo regarding independent aspects and expressive combinations of aspects and components, are also valuable for Adaptive Programming in the context of component-based software engineering. To motivate the possible contributions of the JAsCo ideas to current Adaptive Programming technologies, consider the following example:

```
1 class DataStorePersistence extends Visitor {
2
3    int i = 0;
4
5   public void before(DataStore store) {
6       if(isChangedSinceLastVisit(store)) {
7         FileOutputStream fw = new FileOutputStream("state"+i++);
8         ObjectOutputStream writer=new ObjectOutputStream(fw);
9         writer.writeObject(store.getData());
10        writer.close();
11      }
12   }
13
14   public boolean isChangedSinceLastVisit(DataStore store) {
15       //returns true if store is changed since the last visit
16   }
17 }
```

Code Fragment 51: DataStoreSerializer Adaptive Visitor that allows to serialize each visited data store on file.

The example of Code Fragment 51 illustrates a `DataStorePersistence` adaptive visitor implemented using the DJ library [OL01]. The `DataStorePersistence` adaptive visitor allows taking an incremental backup of the data stored in every visited `DataStore`. The visitor implements a before advice that fetches the data from each store and saves it to a file when the state of the store is changed

since the last visit. If the store is not changed since the last visit, the backup behavior is not executed. When this adaptive visitor is applied to an application, the visitor traverses the entire application and before visiting an object of type `DataStore` that is changed since the last backup, the data held in the store is written to file. As such, an effective incremental backup can be taken of the state of all the `DataStore` objects contained within an application. Code Fragment 52 demonstrates how the Adaptive Visitor of Code Fragment 51 can be used to backup all `DataStore` objects starting from the root system object of the application. Notice that using the adaptive visitor, the backup method does not need to hard-code the relations between the components. As a result, the `DataStorePersistence` visitor remains applicable when additional `DataStore` instances are added to the system or when the relationships between the system components change completely.

```
1 public void backup() {
2
3   ClassGraph cg = new ClassGraph(“system”);
4   Strategy sg = new Strategy("from system.Root to *");
5   TraversalGraph tg = new TraversalGraph(sg, cg);
6   tg.traverse(mySystemRoot,new DataStorePersistence());
7 }
```

Code Fragment 52: Instantiating a DataStoreSerializer in order to traverse the system for taking a backup of the state of the application.

In the context of component-based software engineering, a component has to be independent from other components in order to achieve a loosely coupled system. As such, a component can not rely on one specific component to realize its behavior. Translating this requirement to Adaptive Programming means that the adaptive visitors need to be completely independent from the components they visit. By the very nature of Adaptive Programming, they are already independent of the architecture of the component-based application at hand. However, adaptive visitors still refer to specific component types and specific component APIs, rendering the visitor not as reusable as required by CBSE. In Code Fragment 51 for instance, the `DataStore` type and the `getData` method are hard coded into the adaptive visitor. As a result, it is not possible to apply the same Adaptive Visitor onto a different context. In JAsCo however, aspect beans are abstract and reusable entities which do not depend onto specific component types in order to function properly. As a consequence, integrating the aspect independence idea into Adaptive Programming contributes to achieving a higher reusability and flexibility of adaptive visitors.

Another area where JAsCo ideas contribute to Adaptive Programming consists of the expressive combinations of aspects that JAsCo allows to describe. In the current Adaptive Programming implementations, there is only limited support to combine several adaptive visitors in such a way that they visit the same traversal strategy together. For example, suppose that in addition to the backup behavior, a log has to be kept of every object saved to file. This can only be achieved by adding some logging behavior to the `DataStorePersistence` adaptive visitor. As such, the logging concern is tangled with the backup concern. A visitor that implements logging behavior already exists in the DJ library, namely the `TraceVisitor`. Therefore, the tangled logging concern can be solved if a strong combination mechanism is available that allows to specify that the `TraceVisitor` and the `DataStorePersistence` visitor jointly visit the same traversal. However, the `TraceVisitor` can only be triggered whenever the `DataStorePersistence` visitor saves the visited object to file. Current Adaptive Programming implementations cannot specify such expressive combinations. For example, the DJ library allows combining several visitors on the same traversal strategy, but the composition mechanism is limited to specifying the precedence. As such, the JAsCo ideas concerning

expressive combinations of aspects using precedence and combination strategies are also applicable into the context of Adaptive Programming.

In the next sections, an extension to the JAsCo language is proposed, that recuperates the ideas of Adaptive Programming into the JAsCo language. The Adaptive Programming extension to JAsCo stays as close as possible to the original JAsCo concepts, while offering support for Adaptive Programming into a component-based context.

4.4.2 Employing Aspect Beans as Adaptive Visitors

An adaptive visitor is in fact very similar to a set of related advices as it is able to group several before, after and replace methods that have to be executed when the corresponding component type is visited. Therefore, it seems natural to employ a regular JAsCo aspect bean as a kind of abstract and loosely coupled adaptive visitor. As such, aspect beans are able to describe both traditional aspects and traversal oriented aspects, which increases the reusability of aspect beans even more. Code Fragment 53 illustrates the aspect bean that implements the backup behavior introduced in the previous section. The aspect bean contains one hook, namely the `Backup` hook that implements the crosscutting backup behavior. The constructor of the hook expects one abstract method parameter, namely `triggeringmethod`. The hook is triggered when the method bound to the `triggeringmethod` abstract method parameter is executed. An `isApplicable` method defines an additional condition on which the hook is triggered, namely that the currently visited object is changed since it was last visited. If so, the hook has to be triggered. Similar to the adaptive visitor of Code Fragment 51, one before method is implemented that serializes the visited object to file using the abstract method `getDataMethod`. The `getDataMethod` is responsible for fetching the data from the called object and has to be implemented in a connector or child hook. Thus, the aspect bean does not refer to a specific component type or API. As such, the aspect bean remains completely independent and reusable.

```
1  class DataPersistence {
2
3    hook Backup {
4      int i = 0;
5
6      Backup(triggeringmethod(..args)) {
7        execute(triggeringmethod);
8      }
9
10     isApplicable() {
11      //returns true if calledobject is changed since last visit
12     }
13
14     before() {
15       FileOutputStream fw = new FileOutputStream ("state"+i++);
16       ObjectOutputStream writer=new ObjectOutputStream(fw);
17       writer.writeObject(getDataMethod(calledobject));
18       writer.close();
19     }
20
21     public abstract Object getDataMethod(Object context);
20   }
21 }
```

Code Fragment 53: DataPersistence aspect bean that specifies a reusable backup aspect.

```
1 connector PersistanceConnector {
2
3   DataPersistence.Backup hook = new
4     DataPersistence.Backup(* DataStore.set*(*)) {
5       public Object getDataMethod(Object calledobject) {
6         DataStore store = (DataStore) calledobject;
7         return store.getData();
8     }
9   };
10
11  hook.before();
12 }
```

Code Fragment 54: Instantiating the DataPersistenceAspectBean in a regular JAsCo connector.

The `DataPersistence` aspect bean can be used as a traditional JAsCo aspect as illustrated by the connector of Code Fragment 54. This connector instantiates the `Backup` hook on every method of the `DataStore` class which name starts with `set`. As a result, the `triggeringmethod` abstract method parameter is bound to every method of the `DataStore` class which name starts with `set`. Every time the state of a `DataStore` object instance is altered using a method which name starts with `set`, the hook is triggered and a backup is taken. The `getDataMethod` abstract method of the hook is implemented by invoking the `getData` method on the called object. Notice that casting the called object to a `DataStore` is safe because the hook is only instantiated on objects of type `DataStore`.

In order to instantiate an aspect bean as an adaptive visitor, a new kind of connector is introduced, namely a *traversal connector*. A traversal connector specifies a traversal strategy and instantiates hooks as adaptive visitors on specific components within the specified traversal. Code Fragment 55 illustrates an example traversal connector that instantiates the aspect bean of Code Fragment 53 on the *from system.Root to* * traversal strategy (line 1). The `visiting` keyword allows declaring on which specific classes encountered within the traversal, the hook has to be triggered. In this example, the `Backup` hook is only triggered on `DataStore` objects. The result of applying the traversal connector of Code Fragment 55 is the following: "traverse the object structure as specified by the traversal strategy *from system.Root to* * and invoke the before behavior method of the `Backup` hook each time a `DataStore` object is visited during the traversal". Likewise to a regular connector, the `getDataMethod` is implemented in order to fetch the data of the visited objects.

```
1 traversalconnector BackupTraversal("from system.Root to *") {
2
3   DataPersistenceAspectBean.Backup hook = new
4     DataPersistenceAspectBean.Backup(visiting DataStore) {
5        public void getDataMethod(Object visitedobject) {
6         DataStore store = (DataStore) visitedobject;
7         return store.getData();
8       }
9   };
10
11  hook.before();
12 }
```

Code Fragment 55: BackupTraversal connector .

The main difference between aspects in the "AspectJ sense" and Adaptive Visitors consists in the invocation. Traditional aspects are invoked implicitly whenever the current joinpoint matches the pointcut specification of the aspect. For example, in the connector of Code Fragment 54, the backup hook is triggered every time a method which name starts with `set` is executed on a `DataStore` object. Traversal strategies however, need to be invoked explicitly in order to start the traversal. For example, Code

Fragment 56 illustrates how the traversal specified in Code Fragment 55 is invoked. The `mySystemRoot` is an instance of the `system.Root` class from where the traversal has to start.

```
1 public void backup(system.Root mySystemRoot) {
2      BackupTraversal myBackup = BackupTraversal.getTraversal();
3      myBackup.traverse(mySystemRoot);
4 }
```

Code Fragment 56: Invoking the BackupTraversal connector of Code Fragment 55.

The DJ library is employed behind the scenes in order to implement the behavior described by a traversal connector. Chapter 6 contains a detailed explanation of the traversal connector compilation process. As such, the backup behavior realized using JAsCo aspect beans and traversal connectors, is the same as the behavior established using the DJ library in Code Fragment 52. The involved adaptive visitors, which are represented as JAsCo aspect beans, are however now truly reusable and independent of specific component types.

```
1   class SearchBean {
2
3      Object searchObject;
4      List visitednodes = new List();
5
6      Object getSearchObject() {
7         return searchObject;
8      }
9      void setSearchObject(Object o) {
10        searchobject=o;
11     }
12     List getResultingPath() {
13        return visitednodes;
14     }
15
16     hook BuildPath {
17
18       BuildPath(visitingmethod(..args)) {
19         execute(visitingmethod);
20       }
21
22       replace() {
23
24         if(calledobject==global.getSearchObject())
25            ;
26         else {
27            global.visitednodes.add(calledobject);
28            visitingmethod();
29         }
30       }
31     }
32  }
```

Code Fragment 57: Search aspect bean that allows to build a path of objects visited in order to reach a specific object.

One might wonder to which method the abstract method parameter `triggeringmethod` of the hook of Code Fragment 53 is bound. Indeed, the connector of Code Fragment 55 specifies that this abstract method parameter is bound to the "visiting DataStore" concept, which is not a concrete method signature. However, the result of the "visiting DataStore" declaration is that all `DataStore` objects encountered in the system are visited. The visiting behavior itself can be perceived as a method execution and is also implemented as such in the DJ library. As a result, the method bound to the `triggeringmethod`

abstract method parameter corresponds to the implicit method that implements the visiting behavior itself. This allows the aspect bean to effectively manage the visiting process itself in a hook. When a hook implements a replace method, the hook might decide to stop the visiting process by not invoking the abstract method parameter which is bound to the implicit visiting method. This is illustrated by the aspect bean of Code Fragment 57. This aspect bean searches a specific object and builds a path of objects visited while traversing the object structure. The hook `BuildPath` specifies a constructor which expects one abstract method parameter, named `visitingmethod`, that has to be bound to the visiting declaration in a connector. In addition, the hook implements a replace behavior method that is responsible for building the list of visited nodes. When the object to search for has been reached, the traversal stops because the method bound to the abstract method parameter `visitingmethod` is not invoked. Otherwise, the current visited object is added to the list of visited nodes and the traversal is continued by invoking the method bound to `visitingmethod`. Notice that the semantics of the `calledobject` keyword is also changed when the hook is used as an adaptive visitor. Instead of referring to the object where the currently executing method has been invoked upon, the `calledobject` keyword refers to the currently visited object. This is the logical behavior because if the traversal behavior itself is perceived as a method, the visiting of an object can be perceived as the execution of that method on that object.

Code Fragment 58 illustrates a traversal connector that instantiates the `BuildPath` hook on all the classes in the system using a wildcard. The resulting traversal starts at an instance of the class `system.Root` and visits every reachable object from that instance. When the object that has to be found is located, the traversal is stopped.

```
1 traversalconnector SearchTraversal("from system.Root to *") {
2
3       SearchBean.BuildPath builder = new
4                 SearchBean.BuildPath(visiting *);
5
6       builder.replace();
7 }
```

Code Fragment 58: Instantiating the BuildPath hook on all the classes within the system.

4.4.3 Advanced features of JAsCo Traversal Connectors

4.4.3.1 Precedence and Combination Strategies

An aspect bean, employed as adaptive visitor, is completely independent of concrete components and other aspect beans. A strong mechanism is provided in order to combine them. Similar to regular JAsCo connectors, traversal connectors are also able to explicitly declare precedence and combination strategies. This allows specifying several adaptive visitors onto a single traversal strategy as a complex combination of several cooperating aspect beans. The following paragraphs illustrate how the JAsCo precedence and combination strategies can be used in the context of Adaptive Programming.

When two or more aspect beans are combined in one traversal connector specification, it is important to be able to explicitly declare precedence of the involved beans. Code Fragment 59 illustrates a traversal connector that instantiates both the `Backup` hook and the `FileLogger` hook introduced in section 4.2.3.3. Likewise to the example of Code Fragment 55, the `Backup` hook is instantiated on the visiting of class `DataStore`. The `FileLogger` hook is instantiated on all the classes which are visited during the traversal using a wildcard. The traversal connector allows to explicitly control precedence of both hooks when they both visit the same object (line 11-12). In this case, the before behavior method of the logger hook has to be triggered prior to triggering the before behavior method of the backup hook.

```
1   traversalconnector BackupTraversal("from system.Root to *") {
2
3     DataPersistenceAspectBean.Backup backup = new
4      DataPersistenceAspectBean.Backup(visiting  DataStore) {
5         public void getDataMethod(Object context) {
6            DataStore store = (DataStore) context;
7            return store.getData();
8         }
9     };
10
11    Logger.FileLogger logger = new
12         Logger.FileLogger(visiting *);
13
14    logger.before();
15    backup.before();
16  }
```

Code Fragment 59: Specifying a precedence strategy in a traversal connector.

However, suppose the collaboration between the Backup hook and the FileLogger hook changes so that a log is kept of all objects that have been saved to file. Remember that the Backup hook does not save objects of which the state is not altered since the last visit. As such, the FileLogger hook can only be triggered when the Backup hook is triggered. For these kinds of collaborations among hooks, combination strategies can be employed.

Code Fragment 60 illustrates a twin combination strategy that takes two hooks as input in its constructor (line 4-7). The validateCombinations method specifies that the second hook (hookB) is only kept in the list of hooks when the first hook (hookA) occurs in the list of hooks. As such, the second hook can only be triggered when the first hook is also triggered. Using this "twin" combination strategy, the traversal connector of Code Fragment 59 can be extended to only allow the logging hook to be triggered when the backup hook is also triggered. This is illustrated by Code Fragment 61. The TwinCombinationStrategy is instantiated with both the backup and logger hooks. Likewise to combination strategies in regular JAsCo connectors, the TwinCombinationStrategy is added to the traversal connector using the addCombinationStrategy keyword.

```
1   class TwinCombinationStrategy implements CombinationStrategy
2   {
3       private Object hookA, hookB;
4       TwinCombinationStrategy (Object a,Object b) {
5            hookA = a;
6            hookB = b;
7       }
8
9       HookList validateCombinations(Hooklist hlist) {
10
11           if (!hlist.contains(hookA)) {
12                hlist.remove(hookB);
13           return hlist;
14      }
15  }
```

Code Fragment 60: Twin combination strategy that makes sure that hookB is only triggered when hookA is also triggered.

```
1   traversalconnector BackupTraversal("from system.Root to *") {
2
3     DataPersistenceAspectBean.Backup backup = new
4       DataPersistenceAspectBean.Backup(visiting  DataStore) {
5        public void getDataMethod(Object context) {
6          DataStore store = (DataStore) context;
7          return store.getData();
8        }
9      };
10
11     Logger.FileLogger logger = new
12         Logger.FileLogger(visiting *);
13
14     logger.before();
15     backup.before();
16
17     TwinCombinationStrategy twin = new
18          TwinCombinationStrategy(backup,logger);
19     addCombinationStrategy(twin);
20  }
```

Code Fragment 61: Specifying a combination strategy in a traversal connector.

4.4.3.2 Traversal Strategies

The traversal strategies specified in a traversal connector are passed directly to the DJ library when the traversal connector is compiled. As such, the traversal strategies supported by JAsCo traversal connectors are identical to those supported by the DJ library. Examples of more advanced keywords for traversal strategies include:

- `bypassing:` The bypassing keyword allows to denote classes that may not be visited during the traversal. For example, in the traversal strategy *"from A bypassing C to B"*, an instance of the class C can not be visited in order to traverse starting from an instance of class A and reaching an instance of class B. In other words, when considering the class hierarchy as a class graph, all paths from class A to B that contain the class C are not visited. An exception is thrown if such a traversal is not possible.
- `via:` The via keyword denotes the opposite of the bypassing keyword, namely that the class corresponding to that keyword has to be visited during the traversal. For example, in the traversal strategy *"from A via C to B"*, an instance of class C has to be visited in order to traverse from an instance of class A to an instance of class B. In other words, when considering the class hierarchy as a class graph, all paths from class A to B that do not contain the class C are not visited. An exception is thrown if such a traversal is not possible.
- `->Start,field,End:` This expression can be combined with the `via` or `bypassing` keywords to specify a relation between two classes instead of a single class. In other words, when considering the class hierarchy as a graph, the pattern specifies the edge labeled `field` from class `Start` to class `End`. For example, in the traversal strategy *"from A via ->C,f,* to B"*, an instance of class C has to be visited in order to traverse from an instance of class A to an instance of class B. In addition, after visiting class C, the traversal can only continue by following the field f. The wildcard denotes that by following the field f any target class may be reached.

4.4.4 Concluding remarks

Employing independent aspect beans and expressive combinations improves on current Adaptive Programming implementations in the context of component-based software engineering. Adaptive visitors implemented as aspect beans are now truly reusable as no context information is hard coded. Aspect beans can even be reused both as aspects instantiated in regular JAsCo connectors and as adaptive visitors instantiated in traversal connectors. In addition, adaptive visitors implemented as aspect beans can easily be combined in traversal connectors in order to visit the same traversal as specified by the common traversal strategy. The traversal connector syntax is very close to the original JAsCo syntax making the Adaptive Programming extension to JAsCo quite logical and easy to learn from the JAsCo point of view. However, a drawback of this approach is that the JAsCo syntax and keywords are not always intuitive from the viewpoint of Adaptive Programming. Indeed, keywords often have a different semantics when the hook is used as an adaptive visitor. The `calledobject` keyword for example refers to the visited object instead of a called object when the hook is used as an adaptive visitor, making the keyword less intuitive. In addition, the constructor body of a hook is in fact ignored when employing this hook as an adaptive visitor. This could be solved by changing the semantics of the hook constructor keywords to be useful in a traversal specification. For example, *Cflow* could denote that the corresponding class has to be visited prior to the currently visited class. Likewise, *withincode* can be used to specify that the corresponding class is the previously visited class. Changing the semantics of a hook constructor is not a completely satisfying solution as it causes confusion. As such, aspect beans become somewhat less readable and maintainable.

Chapter 5 Invasive Composition Adapters

In this chapter, the invasive composition adapter model is presented. An invasive composition adapter is an enhanced version of a regular composition adapter with a JAsCo aspect bean implementation. The first section recapitulates the need for an invasive composition adapter. The following section presents the invasive composition adapter model in more detail using an example. Afterwards, an algorithm for applying an invasive composition adapter is introduced. Finally, an extension to the PacoSuite documentation is proposed in order to unleash the full power of the aspect bean implementation.

5.1 Introduction

In the previous chapter, the JAsCo language is introduced that recuperates aspect-oriented ideas in the component-based world. The original motivation for the JAsCo language consists of enhancing the composition adapter model explained in Chapter 3. A regular composition adapter is limited to describing protocol transformations like for example re-routing or ignoring messages. As such, aspects that require adaptations to the behavior of the components themselves cannot be described. To solve this limitation, we propose to integrate aspect beans into PacoSuite. Aspect beans are able to alter the implementation of the public interface of a component, so by integrating aspect beans, aspects that require those kinds of adaptations can also be represented in PacoSuite.

An aspect bean has to be documented in a similar way as the other PacoSuite entities. The PacoSuite documentation is limited to the syntactic and synchronization (protocol) levels of component documentation [BJPW99]. This is because PacoSuite only checks compatibility of components based on the external protocol. As such, the documentation of an aspect bean has to describe the external protocol of the aspect bean, meaning the interaction with other components. In addition to the normal interaction, the documentation has to describe the protocol adaptations caused to the other components. Indeed, the aspect bean implementation might change the external protocol of other components. As such, the documentation of these components is not consistent any longer with the adapted behavior of the components. In order to be able to still verify component compatibility of these components with a composition pattern, these protocol adaptations have to be documented. A regular composition adapter is ideally suited to describe protocol transformations while a component usage scenario is able to specify the normal interaction of a component. As such, the documentation of an aspect bean is in fact a combination of a normal component usage scenario and a composition adapter.

In order to integrate an aspect bean within PacoSuite, we propose an invasive composition adapter. An invasive composition adapter consists of a special kind of composition adapter and an aspect bean implementation in the JAsCo language. The aspect bean implementation is able to alter the internal behavior of the components, while the special composition adapter servers as the documentation and is able to describe protocol transformations. This is visualized in Figure 36. The right hand side of Figure 36 represents an invasive composition adapter, which consists of two parts: a composition adapter (upper part of hexagon) and an aspect bean implementation (lower part of hexagon). The left hand side illustrates a composition pattern (CP) and three components (COMP1-3). The composition pattern is affected by the composition adapter part of the invasive composition adapter while the components are altered by the aspect bean implementation of the invasive composition adapter. Notice that invasive composition adapters are not fully invasive because the adaptations of the aspect bean implementation are deliberately limited to the public interface of a component.

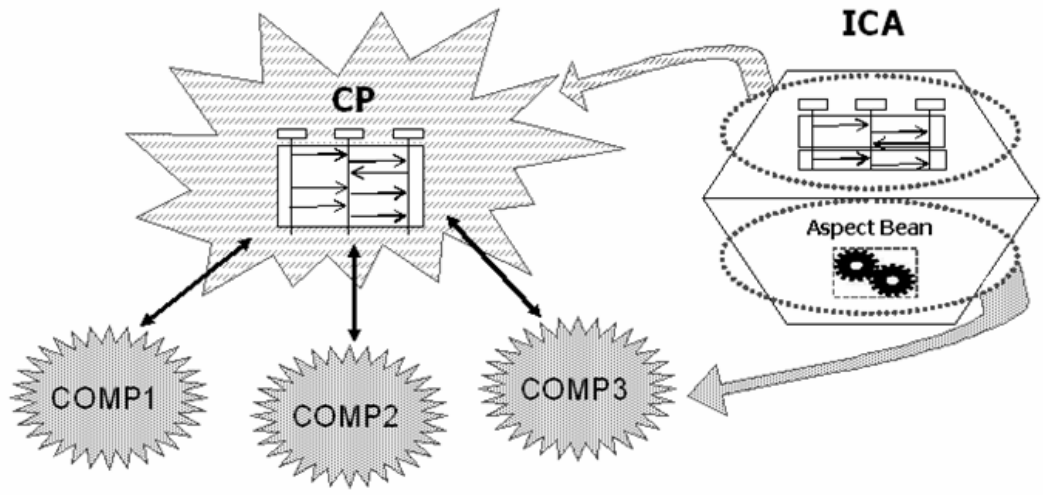

Figure 36: Visualization of an invasive composition adapter. The left hand side illustrates a composition pattern (CP) and three components. The right hand side shows an invasive composition adapter which adapts both the composition pattern and components.

The next section introduces the invasive composition adapter model using the timestamping example introduced in Chapter 3. Afterwards, section 5.3 explains the algorithms required for automatically inserting the protocol transformations described by an invasive composition adapter. In addition, an algorithm is introduced that allows generating a JAsCo connector given a composition pattern and filled-in roles. The generated connector instantiates the hooks defined in the aspect bean implementation of the invasive composition adapter onto the correct components.

5.2 Documentation

To illustrate the invasive composition adapter model, consider again the timestamping concern. The goal of this concern consists of being able to verify timing contracts at run-time. In order to achieve this, every component in the system has to include timestamping behavior. As such, the timestamping concern crosscuts the whole system. In Chapter 3, the timestamping concern is successfully modularized by a composition adapter. The composition adapter approach has however the disadvantage that the accuracy of the timestamps is rather bad, because the timestamp is only taken after a re-routing process. When considering network delays and scheduling priorities the accuracy of the timestamp is even worse. In order to achieve a correct timestamp, the components themselves have to be altered to take a timestamp just before executing a method or firing an event. Therefore, an invasive composition adapter which contains an implementation in the JAsCo language is employed as an improved timestamping composition adapter.

5.2.1 Aspect Bean implementation

Code Fragment 62 illustrates the `DynamicTimer` aspect bean that implements the timestamping concern. The `DynamicTimer` aspect bean allows registering interested observers for timing events (line 5 to 7). These observers are notified of every timestamp the aspect bean takes. The aspect bean contains one hook, named `TimeStamp`, which contains the crosscutting behavior itself. The constructor of the `TimeStamp` hook (line 25 to 27) specifies that the hook is able to be triggered on the execution of any method that takes zero or more arguments. The before behavior method of the `TimeStamp` hook takes a timestamp and assigns the timestamp to an instance variable of the hook. The after behavior method is responsible for notifying all the observers that a timestamp has been taken. An observer of this aspect bean could for example be a component that verifies timing contracts on the basis of the supplied timestamps.

```
1   class DynamicTimer {
2
3      Vector listeners = new Vector();
4
5      public void addTimeListener(TimeListener listener) {
6          listeners.add(listener);
7      }
8
9      public void removeTimeListener(TimeListener listener) {
10         listeners.remove(listener);
11     }
12
13     protected void fireTimeStampTakenEvent(Method m, long t) {
14         Iterator listenersI = listeners.iterator();
15         while(listenersI.hasNext()) {
16            TimeListener l = (TimeListener)listenersI.next();
17            l.timeStampTaken(m,t);
18         }
19     }
20
21     hook TimeStamp {
22
23        private long timestamp;
24
25        TimeStamp(timedmethod(..args)) {
26           execute(timedmethod);
27        }
28
29        before() {
30            long timestamp= System.currentTimeMillis();
31        }
32
33        after()  {
34            fireTimeStampTakenEvent(timedmethod,timestamp);
35        }
36    }
37 }
```

Code Fragment 62: DynamicTimer aspect bean that modularizes the timing concern.

The `TimeStamp` hook can be instantiated onto the `Juggler` and `JButton` beans in order to time the `startJuggling` and `stopJuggling` methods of the Juggler and the `actionPerformed` event of the `JButton` as illustrated by Code Fragment 63. As a consequence, the `timedmethod` abstract method parameter of the constructor of the `TimeStamp` hook is bound to both the `startJuggling` and `stopJuggling` methods and the `actionPerformed` event. The result of compiling this connector is that a timestamp is taken before every execution of either the `startJuggling` and `stopJuggling` methods and before firing an `actionPerformed` event. After the execution of the `startJuggling` and `stopJuggling` methods or the firing of the `actionPerformed` event, a `timeStampTaken` event is fired containing the timestamp and the timed method. Notice that the component composer does not have to deal with a JAsCo connector; an invasive composition adapter merely consists out of an MSC documentation and an aspect bean implementation. The JAsCo connector is automatically generated from a complete composition as explained in the next section.

```
1   connector TimeConnector {
2
3     DynamicTimer.TimeStamp timer = new DynamicTimer.TimeStamp (
4     {
5          void Juggler.startJuggling(),
6          void Juggler.stopJuggling(),
7          onevent JButton.actionPerformed(ActionEvent)
8     });
9
10     timer.before();
11     timer.after();
12  }
```

Code Fragment 63: Connector for instantiating the TimeStamp hook of Code Fragment 62 onto the startJuggling and stopJuggling methods of the Juggler bean and on the actionPerformed event of the JButton bean.

Notice that the `TimeStamp` hook of Code Fragment 62 is a very simplistic timestamping implementation. The before behavior method overwrites the instance variable `timestamp` for each method execution. Whenever a hook is instantiated on a recursive method, the timestamp of the previous recursion step is replaced by the timestamp of the current recursion step. As such, the timestamps that are sent to the observers in the after behavior method are all equal to the timestamp of the last recursive execution of the method, which is not the desired behavior. This problem also occurs with two methods invoking each other; the timestamp of both methods corresponds of the timestamp of the method invoked last. In addition, the implementation is not thread safe as all threads receive the timestamp generated by the thread executed last. In order to avoid this problem, the TimeStamp hook can be instantiated using the *perall* keyword explained in section 4.2.3.1.6. As such, a different instance of the TimeStamp hook is created for every joinpoint. Another solution consists of implementing a more elaborated version of the `TimeStamp` hook that uses a stack to solve the recursion issue and employs a map containing timestamps for each thread in order to achieve thread safety. In order to keep the example as simple as possible, this enhanced implementation is not illustrated here. However, the correct version of the DynamicTimer aspect bean is presented in Appendix B.

5.2.2 Invasive Composition Adapter Model

5.2.2.1 Invasive Composition Adapter Context Part

The documentation of a composition adapter has to be altered in order to deal with the implementation of the aspect bean in JAsCo. Figure 37 illustrates the invasive composition adapter that documents the `DynamicTimer` aspect bean of Code Fragment 62. An invasive composition adapter MSC still consists of two parts: a context part and an adapter part. Likewise to a regular composition adapter, the context part specifies where the composition adapter transformations should be applied. For example, the invasive composition adapter of Figure 37 specifies that the adapter part has to be inserted on every SIGNAL between certain source and destination roles. In addition, the messages in the context part of an invasive composition adapter are able to contain *hook mappings*. A hook mapping consists of an abstract method parameter name specified in a hook's constructor qualified with the name of the hook. The semantics of a hook mapping is that the hook abstract method parameter specified in the hook mapping will be bound to the message that contains the hook mapping. Hooks are of course not directly instantiated onto those messages in the context part because these are abstract protocol messages. However, eventually components are mapped onto the roles of the context part. As such, the hook abstract method parameter specified in a hook mapping is bound to those messages of the components that match the corresponding

message of the context part. For example, the invasive composition adapter of Figure 37 specifies that the `timedmethod` abstract method parameter of the `TimeStamp` hook has to be bound to all the messages defined by components that match with the SIGNAL primitive.

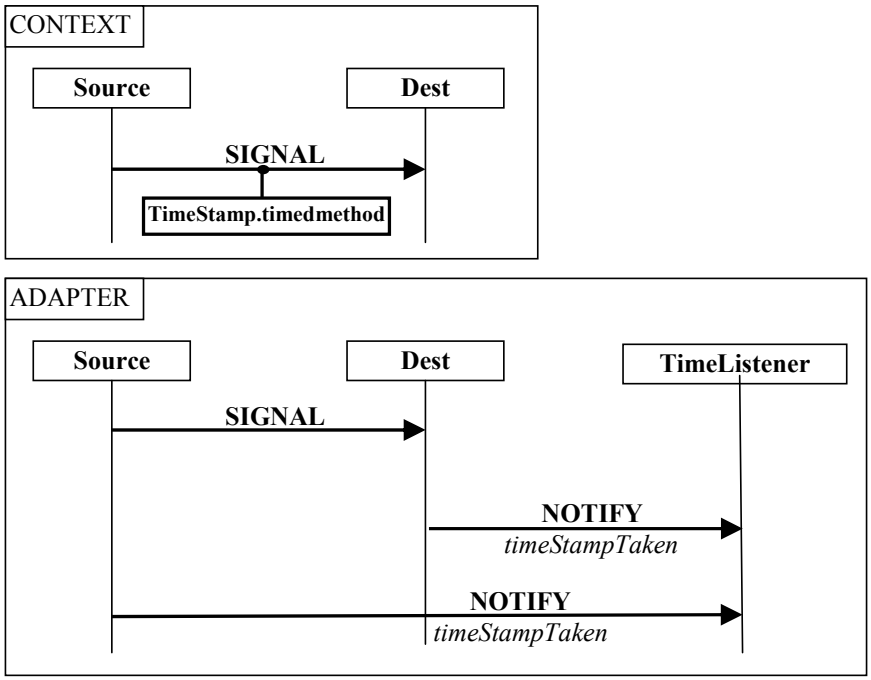

Figure 37: Invasive composition adapter model for the DynamicTimer Aspect Bean of Code Fragment 62.

Notice that a message in the context part of an invasive composition adapter corresponds to at least two messages defined in component usage scenarios if the complete composition is valid. This is because components are mapped on both the starting role as target role of a message contained in the context part. As such, the message matches with the firing of the event of the component mapped on the starting role and with the execution of one ore more methods in the component mapped on the target role. If this message in the context part contains a hook mapping, then the corresponding hook abstract method parameter can be bound to both the fired event of the component mapped on the starting role and the executed method(s) of the component mapped on the target role. For example, suppose the *JButton* component (see Figure 7, page 52) is mapped upon the *Source* role of the invasive composition adapter Figure 37 and the *Juggler* component (see Figure 2, page 42) is mapped upon the *Dest* role. The SIGNAL primitive then matches with pairs of messages, for example *actionPerformed/startJuggling*. For this pair, the timestamp hook is bound to both the firing of the *actionPerformed* event and the execution of the *startJuggling* method. This is of course not always the desired behavior. One might want to limit the context to which the hook is instantiated upon, to only those messages executed by the component mapped onto the target role of the context message. It is possible to specify this graphically by connecting the hook mapping to the target role of a context part message instead of connecting the hook mapping to the middle of the message. This is illustrated by Figure 38 .

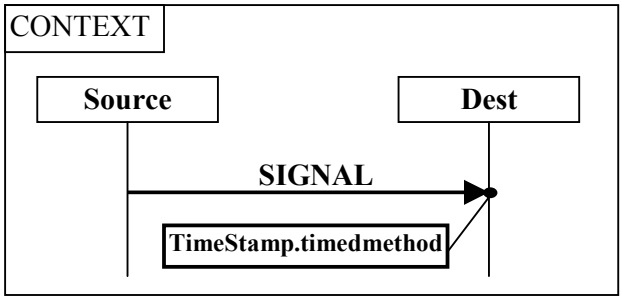

Figure 38: Context part that contains a hook mapping to the target role for the SIGNAL primitive.

Of course, it is also possible to limit the context to which the hook is instantiated upon, to the event fired by the component mapped onto the starting role of the context part message. This is illustrated by Figure 39. Notice that in the Java Beans component model, a component sends messages to other components by firing events. Therefore, messages of the component mapped onto the starting role of a context part message are always implemented by the firing of an event. Of course, this might change when applying the invasive composition adapter model onto another component model.

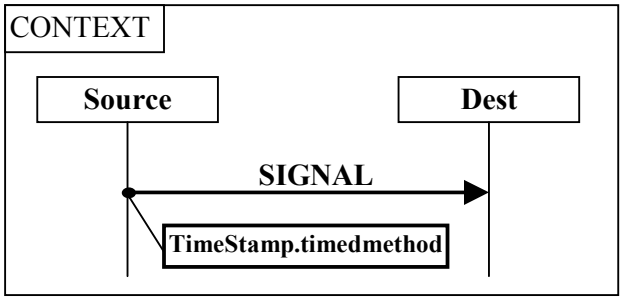

Figure 39: Context part message SIGNAL that contains a hook mapping that is limited to the starting role of the message.

Notice that the distinction made between the firing of an event and the execution of the methods as a consequence of the event is not the same as the difference between the *call* and *execution* keywords in AspectJ. The call keyword denotes the invocation of a method and the execution keyword denotes the execution of the same method. The important difference between those keywords is the context visible to the aspect applied to either the call or execution of a method. Indeed, when applying an aspect on the call of a method, only the context of the caller class is accessible while when applying the aspect on the execution of a method, only the context of the callee is accessible. In the invasive composition adapter case, the hook mappings are really applied to different methods or events, instead of either the call or execution of the same method. Both those methods and events have also a corresponding call and execution. As such, the distinction between call and execution can also be made, but then in the aspect bean implementation of the invasive composition adapter. The current aspect bean implementation in JAsCo however, only allows to instantiate hooks onto the execution of a method or event.

It is possible to omit the abstract method parameter in a hook mapping when the constructor of the hook defines only one abstract method parameter. In this case, it is unambiguous which abstract method parameter has to be bound to the component messages that match the corresponding context part message. As such, the abstract method parameter can be resolved automatically. Messages contained in the context part are able to contain multiple hook mappings when the same message has to be bound to several hook constructor abstract method parameters. In addition, if the aspect bean implementation contains several hooks, multiple hooks might be part of hook mappings defined in the context part. It is also possible to

map a single hook abstract method parameter on several messages in the context part. As a result, this abstract method parameter is bound to all the messages that match one of the context part messages.

5.2.2.2 Invasive Composition Adapter Adapter Part

The adapter part of an invasive composition adapter documents the concrete protocol transformations that occur as a result of the application of the aspect bean implementation. In case of the invasive composition adapter of Figure 37, the SIGNAL primitive is still sent in the same way. However, the `DynamicTimer` aspect bean specifies that a timestamp has to be taken before an adapted method is executed (see Code Fragment 62, line 29 till 31). This behavior is not documented in the composition adapter as it is internal to the aspect bean and no communication with other components is involved. Remember that the PacoSuite approach only verifies compatibility on the protocol level. As such, behavior that does not result in component communication is not relevant for verifying compatibility and to generate glue-code. The `DynamicTimer` aspect bean does however implement an after behavior method (see Code Fragment 62, line 33 till 35) that causes a `timeStampTaken` event to be fired (see Code Fragment 62, line 17). This behavior has to be documented in the composition adapter as it requires communication with other components. Therefore, the adapter part specifies that both the *Source* and *Dest* roles notify a *TimeListener* participant after the original method is executed. Messages that are sent or received as a result of the changes described by a JAsCo aspect bean require an implementation mapping because these messages are part of the concrete protocol of the aspect bean. In Code Fragment 62, the NOTIFY messages are implemented by firing the `timeStampTaken` event. The implementation mapping of the NOTIFY primitive to `timeStampTaken` is required to be able to generate glue-code that will call the correct method of the component that is mapped on the *TimeListener* role as a result of firing of the `timeStampTaken` event. Notice that the component that is mapped onto the *TimeListener* role does not have to understand the `timeStampTaken` event. Glue-code that translates the `timeStampTaken` event into one or more methods of the mapped component can be automatically generated using the documentation of Figure 37.

Notice that the aspect bean implementation of the invasive composition adapter can only alter the implementation of public methods and events. An aspect bean is not able to define new methods or events. For example, the aspect bean implementation of the invasive composition adapter of Figure 37 is not able to add an additional event *timestampTaken* to the component mapped onto the *Source* role, although it is documented as such. In reality however, it is the aspect bean itself that fires the *timestampTaken* event (see Code Fragment 62, line 17) instead of the components mapped onto the *Source* and *Dest* roles. However, this is not specified as such in an invasive composition adapter in order to make the model easier to understand. The first version of an invasive composition adapter included an additional role for representing the aspect bean implementation. This approach proved to be not as intuitive as there is no apparent relation between this role and the context part roles. In case of the timestamping composition adapter for example, the role representing the aspect bean would send a NOTIFY event out of the blue without a clear connection with the protocol between the context part roles. Omitting the aspect bean role is not a problem because all communication that is caused by the aspect bean contains implementation mappings. Therefore, the PacoSuite tool is able to resolve which messages are in fact sent or received by the aspect bean. In the remainder of this chapter, we assume that the external behavior of the components is indeed altered by the aspect bean.

5.2.3 Applying an invasive composition adapter

In order to apply an invasive composition adapter onto a given component composition, each invasive composition adapter context part role needs to be mapped onto a unique composition pattern role. Therefore, when the composition pattern contains fewer roles than the invasive composition adapter context part, the invasive composition adapter can never be applied. Similar, to a regular composition adapter, it is possible that the context part does not match with the composition pattern. As such, the application of this invasive composition adapter on the composition pattern at hand is invalid.

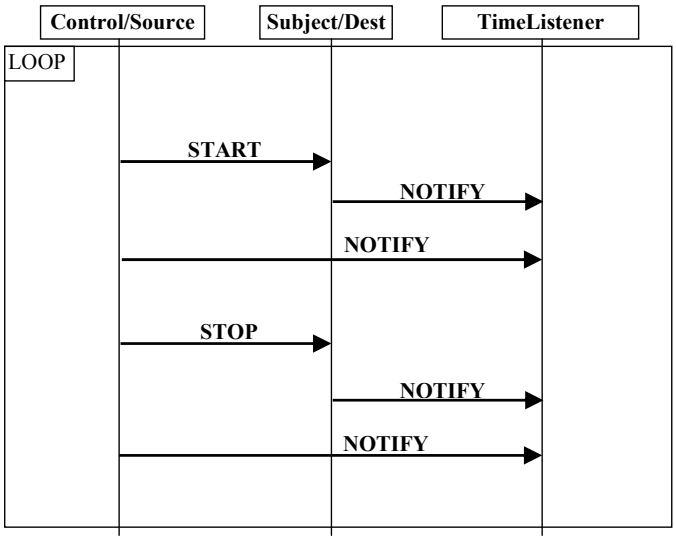

Figure 40: Resulting composition pattern after applying the invasive composition adapter of Figure 37 onto TogglerControl composition pattern of Figure 4 (page 44).

For example, suppose the invasive composition adapter Figure 37 is applied upon the ToggleControl composition pattern of Figure 4 (see page 44). The *Source* role of the invasive composition adapter is mapped upon the *Control* role. Likewise, the *Dest* role is mapped upon the *Subject* role. The resulting composition pattern is altered so that two NOTIFY primitives are sent after each START and STOP. This is illustrated by Figure 40. Notice that mapping the *Source* role onto the *Subject* role and the *Dest* role onto the *Control* role results in an invalid application of the invasive composition adapter because the context part does not match with the composition pattern.

After applying the invasive composition adapter, components can be mapped upon the roles of the composition pattern and invasive composition adapter. From this component mapping, a connector is generated that instantiated the hooks defined in the invasive composition adapter onto the concrete component context. The concrete component context consists of messages defined in component usage scenarios that match with context part messages that contain one or more hook mappings. Only messages of components mapped upon context part roles are able to match context part messages. As such, only those components are subject for aspect application.

For example, suppose that after applying the *DynamicTimer* invasive composition adapter onto the *ToggleControl* composition pattern, the *JButton* component is mapped onto the combined *Control/Source* role. In addition, the *Juggler* component is mapped upon the combined *Subject/Dest* role and a component that logs all received events to file is mapped upon the *TimeListener* role. Only the *JButton* and *Juggler* components are mapped upon context part roles and are thus subject for aspect application. The context part of the *DynamicTimer* invasive composition adapter matches with all messages of the involved

components because it specifies a single SIGNAL primitive that is the top-most primitive in the primitive hierarchy. As such, all the messages of the *JButton* and *Juggler* components are employed to instantiate the *TimeStamp* hook upon. The usage scenario of the *JButton* component defines a single message SIGNAL that is implemented by the *actionPerformed* event (see Figure 7, page 52). The *Juggler* component defines two messages implemented by *startJuggling* and *stopJuggling* (see Figure 2, page 42). As such, the TimeStamp hook has to be instantiated on the *startJuggling* and *stopJuggling* methods of the Juggler bean and on the firing of the *actionPerformed* event of the JButton bean. This connector is illustrated by Code Fragment 63. Notice that although the aspect bean of the *DynamicTimer* invasive composition adapter is instantiated on all component communication, it is also possible to define a more fine-grained instantiation context for the aspect bean implementation. For example, if the context part contains a full protocol, the aspect bean is only instantiated onto those messages that match the protocol history defined by the context part.

The aspect bean implementation of the invasive composition adapter possibly alters the external protocol of the involved components. As such, the documentation of these components is not consistent any longer with their behavior. Therefore, the documentation has to be altered to incorporate the additional logic introduced by the invasive composition adapter. Suppose the Juggler component is mapped onto the *Dest* role of the timestamping invasive composition adapter of Figure 37. The Juggler component is then altered by the aspect bean implementation to fire a NOTIFY event after the *startJuggling* and *stopJuggling* methods (see also the connector of Code Fragment 63). The component usage scenario consistent with the adapted *Juggler* component is illustrated by Figure 41. Notice that only one NOTIFY message is included in this adapted version because this is the only adaptation that is relevant for the external protocol of the *Juggler* component.

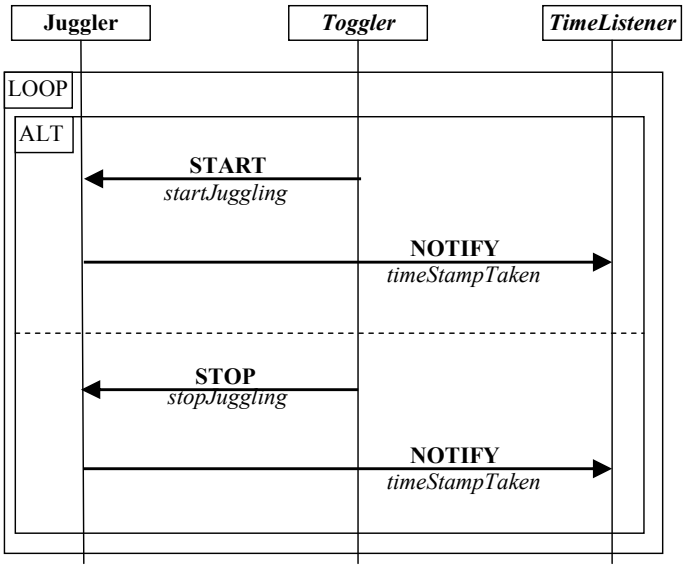

Figure 41: Usage scenario of the Juggler component enhanced with a timestamping aspect bean.

Notice that the difference in behavior between the timestamping invasive composition adapter and timestamping regular composition adapter of Figure 18 consists of both the moment when the timestamps are taken and the number of timestamps taken. Using the regular composition adapter, a timestamp is taking by re-routing a message to a timestamping role. As such, one timestamp is taken in between the sending of the message by the source component (firing of an event) and the receiving of the message by the destination component (execution of a method). Using the timestamping invasive composition adapter, the timestamps are taken inside the components just before the message is sent and received. As such, for

every context part message, twice the number of timestamps are generated in comparison with the regular composition adapter approach.

5.2.4 Concluding remarks

Similar to a regular composition adapter, an invasive composition adapter is able to describe a full protocol in its context part using the ALT, OPT and LOOP MSC constructs. The adapter part is also able to contain any of the MSC constructs. In addition it is possible to specify an empty adapter part when the aspect bean merely deletes the original behavior. Furthermore, the primitives contained in the adapter part of an invasive composition adapter are downcasted to the concrete primitives that match the context part. Likewise to the regular composition adapters, invasive composition adapter messages are able to specify primitive names in order to clearly specify to which context part primitive, an adapter part primitive belongs.

The invasive composition adapter model inherits all the contributions of the composition adapter model while achieving the additional expressiveness of the aspect bean implementation. The invasive composition adapter model is also able to specify protocol-based aspect triggering conditions. This means that a full protocol history can be specified that has to occur before the aspect bean implementation is triggered. Most other AOSD approaches do not allow this kind of protocol-based triggering conditions for aspects.

5.3 Automatically applying an Invasive Composition Adapter

In order to automatically apply an invasive composition adapter onto a composition pattern with filled-in roles, a three-step algorithm is developed. The first step focuses on building the global state-machine that represents the complete composition. The protocol changes described by the invasive composition adapter have to be applied to the composition pattern. In addition, the protocol changes have to be applied to the component usage scenarios to make them consistent with the altered component behavior. In the next step, a JAsCo connector is generated that instantiates the aspect bean implementation of the invasive composition adapter onto the concrete component context. Finally, glue-code needs to be generated for the complete composition that simulates the global state machine built in the first step. The glue-code has to be slightly altered with respect to the glue-code generated in the regular PacoSuite approach [Wyd01] in order to be able to activate and deactivate the generated connectors depending on the protocol history. The next sections explain this three-step algorithm in more detail.

5.3.1 Step 1: Building the global state machine

The goal of this step consists of building the global state machine of the complete composition. The complete composition consists of a composition pattern where an invasive composition adapter is applied upon and components that fill the roles of the composition pattern and invasive composition adapter. When the context part of the invasive composition adapter does not match the composition pattern, the application of the invasive composition adapter is not valid and the algorithm fails.

In order to build the global state machine, the protocol transformations described by the invasive composition adapter have to be applied to the composition pattern. Because an invasive composition adapter is very similar to a regular composition adapter, the algorithm for applying a regular composition adapter elucidated in section 3.3 can be reused. The invasive composition adapter does however contain some extra information that is not required for applying the protocol transformations of a regular composition adapter. The additional information consists of the hook mappings of context part messages and the implementation mappings of adapter part messages sent as a result of the aspect bean implementation of the invasive composition adapter. This information has nothing to do with the abstract protocol transformations documented by a composition adapter. As a result, the composition adapter application algorithm has to ignore the extra information, which is easily accomplished.

The composition pattern is however not the only artifact affected by the invasive composition adapter. Indeed, the aspect bean implementation of the invasive composition adapter is able to change the mapped components. These changes possibly alter the external protocol of the components, rendering the component usage scenario documentation inconsistent with the adapted behavior of the components. Therefore, the component usage scenarios have to be altered by applying the protocol transformations specified in the invasive composition adapter.

Definition 20: Incremental check operator inccheck

Let:

- S_P be the composition pattern P represented as a DFA
- S_C be the component C represented as a DFA
- r be a role of the composition pattern P

Then **inccheck**(S_P,S_C,r) is the state machine resulting from incrementally checking whether the component C can function as role r of P [Wyd01]

In order to clearly define the invasive composition adapter application algorithm, two helper definitions are needed. The first helper definition (see Definition 20) defines the inccheck operator as the incremental checking algorithm elucidated in [Wyd01, VW01]. The operator takes as input a composition pattern DFA, a component usage scenario DFA and a role of the composition pattern where the component is mapped upon. The inccheck operator returns the state machine as calculated by the incremental checking algorithm. The state machine contains in fact the combined behavior of the component and the composition pattern. If this state machine is empty, meaning that it does not contain any start-stop paths, the component is incompatible with the selected role of the composition pattern. The algorithm is incremental because the resulting state machine can again be used as input composition pattern DFA for building the combined behavior with another component mapped to a different role in the composition pattern.

Definition 21: Composition Adapter Application operator CAA

Let:

- A is an (invasive) composition adapter
- P is a composition pattern or component usage scenario
- M is a valid role mapping

Then **CAA**(A,P,M) is the state machine resulting from applying the composition adapter application algorithm defined in section 3.3 with A, P and M as input

The second helper definition (see Definition 21) defines the operator CAA as the composition adapter application algorithm elucidated in section 3.3. This operator takes as input a composition adapter A, a composition pattern or component usage scenario P and a set of role mappings M. The algorithm returns the state machine that represents the transformed composition pattern or component usage scenario P as specified by the protocol transformations of the composition adapter A. The CAA operator fails when the context part of the composition adapter does not match with the composition pattern. Notice that in the original composition adapter application algorithm, P is defined to be a composition pattern. A component usage scenario can however also serve as a target for this algorithm when the extra information, namely the implementation mappings, is ignored. Also notice that an invasive composition adapter can also be used as a composition adapter when ignoring the extra information.

Algorithm 7: Building the global state machine

Let:

1. P be the composition pattern
2. I be the invasive composition adapter; I_C is its context part and I_A is its adapter part
3. M_I be a valid role mapping between roles(I_C) and roles(P)
4. K be the set of components mapped onto the roles of P and I
5. M_K be valid mappings of components onto composition pattern roles or adapter part roles; $M_K \subset (K \times (roles(P) \cup roles(I_A)))$
6. M_E^C be the mappings of component usage scenario roles of $C \in K$ onto context part roles; $M_E^C \subset (roles(C) \times roles(I_C))$

Then the algorithm is as follows:

7. S := CAA(I,P,M_I)
8. foreach (C,p) $\in$ M_K
9. if $\exists$r:(r,p)$\in M_I$
10. I' = $proj_r$(I)
11. S_C = CAA(I',C,M_E^C)
12. else
13. S_C is the DFA representation of C
14. end if
15. S := inccheck(S,S_C,p)
16. end foreach
17. result = S

The algorithm to build the global state machine is specified in Algorithm 7. In short, the algorithm first applies the invasive composition adapter onto the composition pattern. When the context part does not match with the composition pattern, the algorithm fails. Otherwise, the adaptations specified by the invasive composition adapter are inserted into the composition pattern. Next, the algorithm iterates over all components and builds a global state machine starting from the transformed composition pattern DFA. When the component at hand is mapped onto a context part role, the adaptations described by the invasive composition adapter are inserted into that component usage scenario. When the component is not mapped onto a context part role, it is not affected by the aspect bean implementation. As such, the usage scenario of this component can be immediately translated to a DFA. Afterwards, the original or adapted component DFA is combined with the global state machine.

The algorithm starts with a complete component composition (line 1-6). This means:

- A composition pattern P (line 1).
- An invasive composition adapter I (line 2).
- A valid role mapping M_I between the invasive composition adapter context part roles and the roles of the composition pattern (line 3). This mapping can easily be specified by the component composer by visually dragging invasive composition adapter roles onto composition pattern

roles. It is also possible to automatically find a possible role mapping using the algorithm proposed in section 3.4.2.

- A set of components `K` used in the composition (line 4). A component is used in the composition if the component is mapped onto one or more roles of the composition pattern and/or invasive composition adapter.
- A set of mappings M_K from a component used in the composition onto a role of the composition pattern and/or the adapter part of an invasive composition adapter (line 5). These mappings can easily be specified by the component composer by visually dragging components onto a role. Notice that components are never directly mapped onto context part roles, they are mapped onto the composition pattern role which is on its turn mapped onto a context part role of the invasive composition adapter.
- For each component `C` used in the composition, a role mapping M_E^C from the roles of the usage scenario of the component onto the roles of the invasive composition adapter context part. Some role mappings might be empty if the corresponding component is not mapped onto a role of the context part. These role mappings can be easily calculated by the environment participants resolving algorithm proposed in [Wyd01].

In the first step of the algorithm, the protocol transformations described by the invasive composition adapter are inserted into the composition pattern. This can be easily accomplished by employing the composition adapter application operator `CAA` (line 7). When the context part of the invasive composition adapter does not match with the composition pattern, the `CAA` operator fails and as such the invasive composition adapter cannot be applied. Afterwards, the algorithm iterates over all pairs (C,p) contained in the mapping set M_K (line 8). Components that are not mapped upon context part roles are simply converted to their DFA representation (line 9 and 13). This is because components that are not mapped upon context part roles are not adapted by the aspect bean implementation. As such, the component usage scenario remains identical. For components that are mapped onto context part roles, meaning that the protocol transformations described by the invasive composition adapter are applicable, an adapted DFA has to be computed. Therefore, the projection of the invasive composition adapter to the context part role where the component is mapped upon is taken (line 10). This is required because the invasive composition adapter describes a full protocol transformation of several cooperating roles. Only the protocol transformations that deal with the role where the component is mapped upon have to be taken into account. The other transformations are irrelevant for the component at hand. Computing the projection of an invasive composition adapter consists of taking the projection of both the context and adapter parts. Computing the projection of single MSCs can be easily accomplished (see Figure 5, page 47). Next, the protocol transformations described by the invasive composition adapter are inserted into the component using the `CAA` operator (line 11). In the last step of the iteration, the computed component DFA is combined with the resulting composition DFA using the `inccheck` operator. The resulting state machine is then the combined composition up till now. After iterating over all components, the result consists of the last state machine computed by the `inccheck` operator. This state machine describes the adapted protocol of the composition pattern and the adapted or original protocol of the components. As such, the combined global state machine has been computed.

Notice that it is possible that the inccheck operator fails and that an empty state machine is returned. When this occurs, the composition is not valid. As such, when an empty state machine is encountered as result of applying the inccheck, the global algorithm fails.

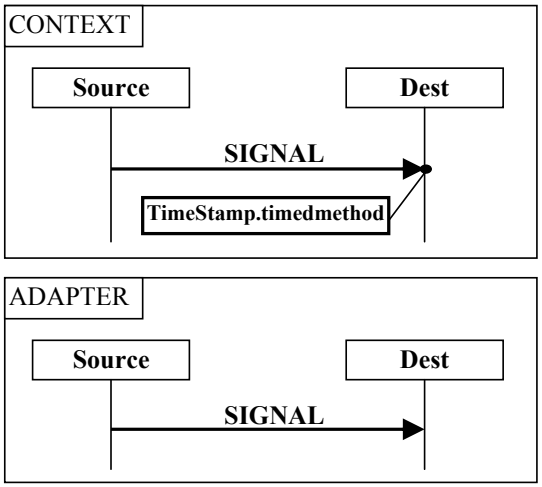

Figure 42: Result after applying the projection of the invasive composition adapter that contains the context part of Figure 38 to the Source role.

The algorithm is still open for improvement performance-wise. It is for example perfectly possible that after taking the projection of the invasive composition adapter to a certain context part role, the resulting invasive composition adapter does not define a protocol transformation any longer. For example, after taking the projection to the *Source* role of the invasive composition adapter that contains the context part Figure 38, the resulting adapter part equals the context part as illustrated by Figure 42. As such, applying this composition adapter to a usage scenario of a component renders again the same usage scenario. Because the composition adapter application algorithm is an expensive algorithm, it is better to check whether the composition adapter still describes an adaptation before applying the algorithm.

5.3.2 Step 2: Generating the connector

In this step the connector is generated that instantiates the hooks defined in the aspect bean implementation of the invasive composition adapter onto the concrete component context. The concrete component context consists in this case of all concrete component methods and events that match with messages of the context part containing a hook mapping. For example, the hook H of the invasive composition adapter of Figure 43 has to be instantiated on every method or event that matches with the C primitive. However, only those methods and events are taken into account that are preceded by a message that matches with the B primitive. As such, a protocol history is defined. For example, suppose one maps the component COMP of Figure 44 onto the P1 role of the context part of Figure 43. In that case, the *method* abstract method parameter of hook H has to be bound to the method *doTheC*. The *method* abstract method parameter is not bound to the *doTheC2* method because the second C primitive is not directly preceded by a B primitive and as such does not match with the context part.

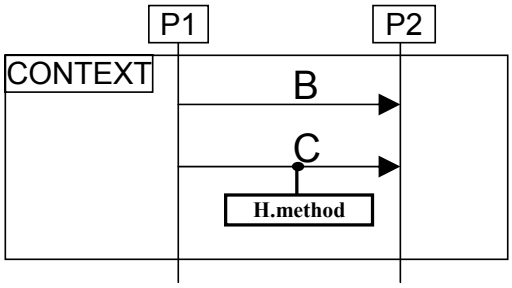

Figure 43: context part of an invasive composition adapter.

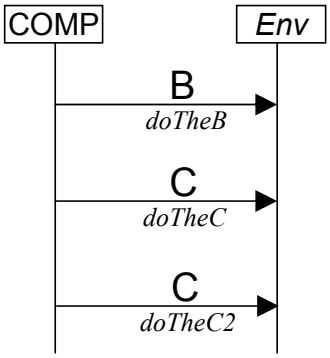

Figure 44: Component usage scenario of component COMP.

In order to find all matching methods or events, an algorithm very similar to the algorithm that searches all paths matching with the context part of a composition adapter can be used (see section 3.3.2). Indeed, we have to match the context part with the component usage scenario and remember the methods or events that match with context part messages that contain a hook mapping. This can be easily accomplished by slightly altering the first step of the composition adapter application algorithm. The composition adapter application algorithm computes the intersection of the context part with the component automaton for every state of this latter automaton. Every transition label of both automata contains the MSC message that corresponds to this transition and when computing the intersection, the transition labels are compared. As such, finding all methods or events is easily accomplished by altering this algorithm to store all methods or events that match with a message that contains a hook mapping. From this stored mapping, the generation of the connector is trivial.

Notice that in the previous step, which builds the global state machine, the composition adapter application algorithm is already computed for all the components mapped on context part roles. As such, the algorithm that matches the context part with the usage scenario of the components is already performed. Therefore, it suffices that in the previous step, the composition adapter application algorithm is altered to remember the methods and events that match with context part messages containing hook mappings. These mappings can then be used in this step to generate the connector. As such, the connector generation process only requires a minimal overhead. After the connector is generated, the JAsCo tools can be employed to compile the connector.

5.3.3 Step 3: Generating the glue-code

In the last step, the glue-code that simulates the global state-machine is generated. This glue-code translates the syntactical incompatibilities of the involved components and constrains the component behavior as specified by the composition pattern and invasive composition adapter. The glue-code

generation process does need to be altered slightly in comparison with the original PacoSuite algorithm. This is because the hooks defined in the aspect bean implementation of the invasive composition adapter can only be triggered after the protocol history defined in the context part is encountered. JAsCo does however not allow specifying a protocol history condition. For example, the hook H of the invasive composition adapter of Figure 43 can only be triggered after a B-C abstract protocol. The generated connector instantiates the hook H on the implementation of the C primitive. When both the B and C are implemented by the same method, the hook H will be triggered on both the B and C primitives, which is not the desired behavior. Therefore, the glue-code generation process has to be altered to explicitly enable and disable the connector that instantiates the aspect bean whenever the hook is able to be triggered.

Code Fragment 64 illustrates a part of the glue-code that is generated for a composition containing the invasive composition adapter of Figure 43. Notice that this glue-code is simplified for enhancing the readability. The reader is referred to Chapter 7 for fragments of generated glue-code. The fragment starts with fetching the next state for the given incoming method. Depending on the fetched state, the correct outgoing method has to be executed. For the first state (`STATE_1`), the generated connector named `TheConnector` is disabled (line 7). This is because this first method corresponds to the B abstract primitive and the hook is not allowed to trigger on this method. After receiving the second method however, the connector is enabled (line 14) because the protocol history matches the protocol history of the C primitive in the context part of Figure 43.

```
1   PacoState nextState = getStatemachine().doTransition(method);
2   if(nextState == null) return;
3
4   switch((int) currentState.getId()) {
5
6         case STATE_1: {
7               TheConnector.getConnector().setEnabled(false);
8
9               //do parameter translation
10              //call the correct component(s)
11              break;
12              }
13        case STATE_2: {
14              TheConnector.getConnector().setEnabled(true);
15
16              //do parameter translation
17              //call the correct component(s)
18              break;
19              }
20        ...
```

Code Fragment 64: Fragment of glue-code that implements the global state-machine. Notice that the connector is explicitly enabled and disabled.

5.3.4 Stacking several invasive composition adapters

The algorithm described in the previous sections only copes with one invasive composition adapter that is applied onto the composition pattern. It is however also possible to apply several invasive composition adapters onto the same composition pattern. This is not a problem though. The algorithm described in the previous sections needs to be altered slightly:

- In the first step, instead of a single composition adapter application, the invasive composition adapter transformations are applied sequentially to the composition pattern and to the components. As such, the global state machine is built.
- The connector generation process then simply generates a connector per invasive composition adapter instead of a single connector. For controlling the sequence of the invasive composition adapters, a connector combination strategy is employed.
- In the last step, the glue-code enables and disables multiple connectors instead of a single connector.

One might wonder why a separate connector is generated for each invasive composition adapter instead of just instantiating all the hooks in one connector. The reason is that the hooks have to be enabled and disabled is the glue-code in order to implement protocol history conditions. Enabling and disabling individual hooks is not possible in the current JAsCo implementation, only groups of hooks instantiated in the same connector can be enabled and disabled. Therefore, the hooks are instantiated in separate connectors so that the glue-code can control which hooks are able to be triggered by controlling which connectors are enabled.

5.3.5 Concluding Remarks

The algorithm elucidated in previous sections allows to easily apply an invasive composition adapter onto a given component composition. The context part can again be exploited to verify whether the invasive composition adapter is applicable. The transformations described by the invasive composition adapter can be automatically inserted into the composition pattern and component usage scenarios In addition, automatically generating glue-code and verifying compatibility of components with a composition pattern enhanced by an invasive composition adapter are still possible. Therefore, an invasive composition adapter also complies with the requirements for integrating aspect-oriented ideas into PacoSuite elucidated in section 3.1. As such, component composition in PacoSuite still does not require in-depth technological knowledge of the involved components or aspect beans. The high abstraction level of PacoSuite is thus maintained. In fact, for the component composer, it remains completely transparent whether she/he is employing an invasive composition adapter or regular composition adapter. For both entities, the changes can be automatically applied without bothering the component composer.

5.4 Extending the model

The invasive composition adapter model allows to instantiate aspect beans on the external protocol of components. The aspect beans themselves are however able to describe adaptations of the public interface of a component, meaning public methods and events. It is possible that the public interface of a component is invoked from within the component. Aspect beans are able to intercept this kind of behavior. The invasive composition adapter documentation is however not able to cope with this internal usage of the public interface of a component. As such, the aspect bean implementation of the invasive composition adapter can not be instantiated on those internal invocations. While limiting the documentation of PacoSuite entities to the external protocol is logical when considering compatibility checking and glue-code generation, it does not allow describing all adaptations the JAsCo aspect beans support. This is a limitation of the current invasive composition adapter model. In order to exploit the full power of the aspect bean implementation, it has to be possible to instantiate aspect beans on invocations from within a component.

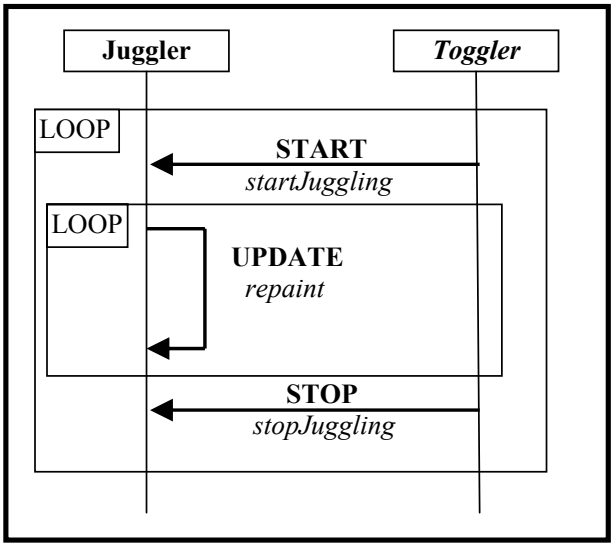

Figure 45: Component usage scenario of the Juggler component where self-invocations are documented.

In order to support self-invocations in the PacoSuite approach, component usage scenarios have to be enhanced with extra syntax. Self-invocations are specified in the same way as normal message sents in component usage scenarios. They are also labeled with both a primitive type and concrete implementation mapping. Figure 45 illustrates a component usage scenario of the Juggler component where a self-invocation is documented. After receiving a START message, the Juggler component is continuously sending an UPDATE message to itself in order to repaint the juggling animation. The composition pattern documentation does not have to be augmented with this self-invocation syntax because a composition pattern only documents an abstract protocol between components. Self-invocations are not relevant for verifying compatibility with components and to generate glue-code. As a result, the algorithms have to be adapted in order to ignore the self-invocations, which is easy to accomplish.

5.4.1 First approach

A first approach for allowing invasive composition adapters to specify aspect beans on self- invocations consists of adding this extra self-invocation syntax to the context part of an invasive composition adapter. As such, the context part of an invasive composition adapter is able to specify that the invasive composition adapter has to apply on a given self-invocation. Figure 46 specifies an alternative version of the timestamping invasive composition adapter making use of this self-invocation syntax. This timestamping invasive composition adapter is still implemented by the *DynamicTimer* aspect bean of Code Fragment 62. The context part of the invasive composition adapter specifies that it is applicable on any SIGNAL sent by a component to itself. In addition, the *TimeStamp* hook is instantiated on all the concrete messages that match with this SIGNAL self-invocation. The adapter part specifies that the SIGNAL is still sent in the same way. Afterwards, the aspect bean implementation makes sure that a *TimeListener* role is notified by firing a *timeStampTaken* event.

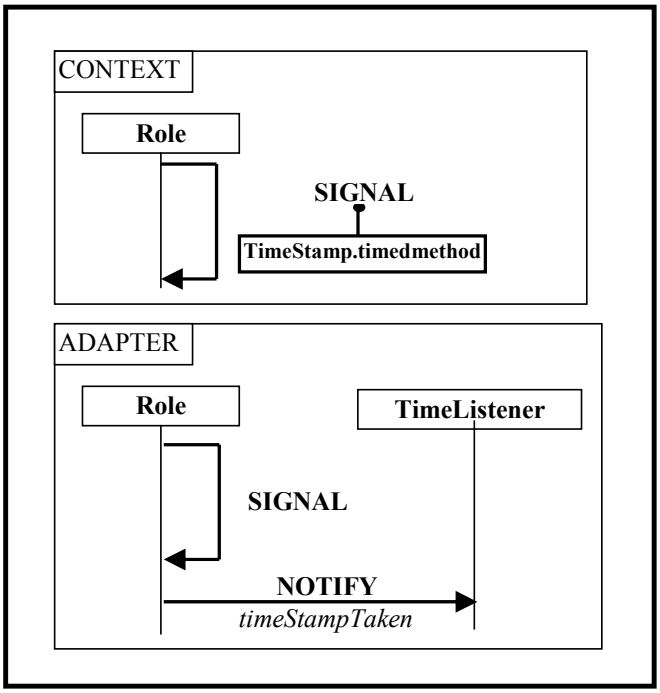

Figure 46: Invasive composition adapter that specifies a self-invocation in its context part.

The invasive composition adapter of Figure 46 can be used to time the self-invocations of the Juggler component. Therefore, the Juggler component has to be mapped onto the *Role* role of the invasive composition adapter. As a result, a connector is generated that instantiates the *TimeStamp* hook onto the *repaint* method of the Juggler component. As such, for each invocation of the *repaint* method, a timestamp is taken.

The drawback of allowing an invasive composition adapter to declare explicit self-invocations consists in the lack of reusability of the involved invasive composition adapter. When explicitly declaring a self-invocation, the context part does not match with inter-component message sents. One has to create a new invasive composition adapter for every combination of inter – and intra communication types with concrete messages defined in the context part. As such, it is impossible to reuse an invasive composition adapter for both inter-component as intra-component message sents. For example, the invasive composition adapter of Figure 46 documents in fact the same timestamping behavior as the invasive composition adapter of Figure 37. Using this current approach, it is impossible to specify a generic

timestamping invasive composition adapter that is reusable for both inter and intra component communication, apart from explicitly enumerating all possible combinations in several alternatives.

5.4.2 Improving the first approach

In order to allow reusing an invasive composition adapter for both inter-component and intra-component communication, a different solution has to be found than specifying explicit self-invocations. A better solution is to allow mapping several invasive composition adapter context part roles onto the same composition pattern role. The mapped roles become effectively merged and all communication between the roles is considered as intra-component communication. As such, it is possible to avoid explicitly specifying self-invocations in an invasive composition adapter. The invasive composition adapter is now reusable for both inter-component and intra-component communication. For example, it is possible to apply the invasive composition adapter of Figure 37 onto the *ToggleControl* composition pattern of Figure 4 (page 44) in such a way that the *Source* and *Dest* roles are both applied onto the *Subject* role. Afterwards, the *Juggler* component can be mapped onto the combined *Source* and *Dest* roles. The UPDATE self-invocation of the Juggler component matches with the SIGNAL defined in the context part as this SIGNAL is now also considered as a self-invocation. As a result, a connector is generated that instantiates the *TimeStamp* hook on the *repaint* method. In order to take timestamps for the START and STOP messages received by the Juggler component, an additional timestamping composition adapter has to applied that maps the *Source* and *Dest* roles respectively onto the *Control* and *Subject* roles of the composition pattern. This is possible because invasive composition adapters can be stacked to allow a combined transformation.

The attentive reader might remember that mapping several composition adapter roles onto the same composition pattern role is explicitly forbidden in Chapter 3. In addition, the algorithm to search for all possible role mappings only finds role mappings where no two context roles are applied onto the same composition pattern role. This is a logical constraint in the context of a regular composition adapter. A regular composition adapter only deals with the external protocol. As a result, when merging the composition adapter roles, the context specification is lost. As such, it is always possible to apply a composition adapter by merging all its roles to one composition pattern role. In the context of an invasive composition adapter, which is able to specify transformations of the internals of a component, merging context part roles is not a problem any longer. This is because the communication between merged roles is not lost; instead it is regarded as intra-component communication. In addition, because the components specify the self-invocations explicitly, it is still possible to verify whether the merged role application of the invasive composition adapter makes sense. As such, mapping several context roles on the same composition pattern role is not a meaningless role mapping any longer.

Chapter 6 Tool Support

In this chapter, the implementation of tool support is discussed in more detail. All of the tools are implemented in the Java programming language and are thus cross-platform. The next section discusses the tools implemented for the JAsCo language. Section 6.2 introduces the PacoSuite tool suite and more specifically the extensions made to implement composition adapter support. In addition, the invasive composition adapter implementation and the integration with the JAsCo tools is presented

6.1 JAsCo

The JAsCo implementation consists mainly out of five command-line tools, a graphical tool and the run-time infrastructure. The six tools employed in JAsCo are *beanTransformer* for transforming Java Beans to JAsCo beans with built-in traps, *compileAspect* for compiling aspects, *compileConnector* for compiling connectors, *removeConnector* for removing and unloading a connector, *compileTraversal* for compiling traversal connectors and finally the GUI tool *introspect* that allows to introspect a running system. The JAsCo run-time infrastructure runs in the background of any JAsCo enabled application and manages the dynamic features of the JAsCo system like for example loading connectors at run-time or executing combination strategies. The complete JAsCo system is a pure Java application and consists out of 360 classes. The 360 classes contain in total 24533 lines of code without counting comments and blank lines.

6.1.1 Tool Overview

6.1.1.1 BeanTransformer

The beanTransformer tool is undoubtly the core tool in the JAsCo tool suite. This tool takes as input a Java bean in binary form and transforms it in such a way that for every public method and event, a trap is inserted. At run-time, the traps inquire the connector registry for hooks that are registered on this trap. If no hooks are registered, the normal execution continues, otherwise the hooks' behavior is executed at the appropriate moment (i.e. before, after, replace). Attaching a trap to a method is quite straightforward. The following steps are required to attach a trap to a method:

1. Copy the implementation of the method to an unexisting private method
2. Replace the behavior of the original method by the logic of the trap.

As such, the trap is executed instead of the original method. The original method is however still available, but under a different name. Notice that the original behavior cannot be invoked from outside the Java bean because the original methods are changed to private methods.

Although events are also implemented using method invocations, attaching a trap on the firing of an event is somewhat trickier. The problem is that the Java Beans convention does not standardize the name of the method that causes the event to fire. The word "fire" followed by the event name is often used for the name of the firing method. All the Java Swing components for instance, use that strategy. In general however, it is even impossible to rely on the existence of a method that fires the event. Fortunately, the name of methods that add or remove an event listener is standardized. Therefore, the following strategy is used: when the fire method exists, the trap is attached to this method using the approach outlined above; otherwise the standardized methods for adding and removing listeners are patched in such a way that the JAsCo system is in control of firing the event. This means that the following steps are required for each event X the java bean defines:

1. If the method *fireX* exists, goto step 9.
2. An instance variable *listenersX* containing a collection of listeners is added to the Java bean. The *listenersX* is not already an instance variable of this Java bean or superclasses of this Java bean.
3. The standardized method *addXListener* for adding listeners is renamed to an unexisting private method named *addXListenerOriginal*.

4. The standardized method *removeXListener* for removing listeners is renamed to an unexisting private method named *removeXListenerOriginal*.
5. The implementation of the standardized method *addXListener* for adding listeners is replaced to add the listener to the *listenersX* collection.
6. The implementation of the standardized method *removeXListener* for removing listeners is replaced to remove the listener from the *listenersX* collection.
7. A private fire method named *fireX* is generated. This method invokes the event X on every listener contained in the *listenersX* collection.
8. The following behavior is added to all constructors of the Java bean:
   8.1. Create a listener for the event X. This listener invokes the *fireX* method in order to fire the event to all registered listeners.
   8.2. Add that listener to the bean by invoking the *addXListenerOriginal* method.
9. The *fireX* method is trapped in the same way as a normal method is trapped. The trap is however aware of the fact that this method corresponds in fact to the firing of an event.

Notice this approach is not correct when the *fireX* method exists for a given event X, but implements a completely different behavior than firing the event. To be sure, it is better to always patch the standardized add and remove methods of event listeners. This approach does however require an additional overhead because events are not fired directly, but in two steps. First the event is fired by the Java bean and is then catched by the JAsCo system and fired again. In addition, naming a method *fireX* with X an event name and not implementing the firing logic is very unlogical. Therefore, the possibly incorrect approach presented above is used per default by the *beanTransformer* tool. It is however possible to force the *beanTransformer* to patch all add and remove listeners in case there exists fire methods that no not implement the firing logic of the corresponding event.

The transformations explained in the previous paragraphs do not change the public interface of the Java bean by adding or removing public methods or events. As such, the transformed Java bean can not be distinguished from the original bean by observing its public interface. As a result, it is for example possible to employ the transformed Java bean in visual component composition environments in the same way as the original bean. It is also not possible to bypass the execution of the traps, because the original add and remove methods of the given event are not invocable from outside the Java bean.

The beanTransformer tool also allows to explicitly protect one or more methods or events of the bean from aspect application. These methods or events are not equipped with a trap. As such, no aspect interference is ever possible. This is useful for enhancing performance or when a method is really crucial and no aspect interference can be allowed. Of course, when abusing this feature by shielding too much of the Java bean, the expressiveness of possible aspects that crosscut this bean is seriously limited.

Technically, the transformation process employs byte-code manipulation techniques using a custom-made library named JBE [Suv02]. The JBE library uses BCEL [DZ03] to transform Java byte codes to human readable Jasmin assembler code and Jasmin [MD03] is used to do the transformation the other way around, i.e. from Jasmin to a Java class file. The main advantage of the JBE library is that it allows injecting plain Java code into a class file. This in contrast to most other byte-code manipulation libraries like for example

Javassist [Chi00] or JMangler[KCA01]. The JBE library takes care of compiling the inserted Java source code to Java byte code. This simplifies implementing the transformation process greatly.

6.1.1.2 CompileAspect

The compileAspect tool is a command-line tool that takes an aspect bean source file as input and compiles it to binary form suitable to run in a Java virtual machine. The compileAspect tool works in two phases: first the aspect bean is translated to several java classes and afterwards these java classes are compiled. In fact, the compileAspect tool is a cross-compiler from JAsCo to Java as it compiles the JAsCo aspect bean language to classes in the Java language. The compileAspect tool generates one Java class for the aspect bean itself. Each hook defined in the aspect bean is also translated to a separate class in the Java language. The standard java compiler (javac) is then used to compile all the generated classes to the Java binary format.

It is also possible to equip an aspect bean and its hooks with traps in order to allow applying aspects on aspects. This is however not the default behavior, because this feature easily causes infinite loops when employed carelessly as explained in 4.2.3.1.5. Therefore, an optional flag passed to the compileAspect tool defines whether or not traps are inserted in the aspect bean itself.

6.1.1.3 CompileConnector

The *compileConnector* tool compiles a connector to a Java byte code representation. Similar to the compileAspect tool, the compileConnector tool first compiles the connector to a Java class representing the connector. Afterwards, the connector is compiled to Java byte code using the standard Java compiler. Notice that connectors are not able to contain a package declaration and are always compiled to the package “connector”. In order to activate this connector in an application, it has to be in the classpath of the application. When the connector is compiled, it is detected by the run-time infrastructure and dynamically loaded in all running applications that have the connector in their classpath. As such, all the hooks instantiated in the connector are able to be triggered in that application.

Optionally, the compileConnector tool allows inserting traps at all components where a hook is instantiated upon in the connector. This is done by invoking the beanTransformer tool explained above for all those components. As such, the JAsCo approach can also be used as a traditional byte-code weaving approach where the components are adapted at the moment an aspect is applied.

6.1.1.4 RemoveConnector

The *removeConnector* tool allows removing a compiled connector. If this connector is already used in an application, the run-time infrastructure in that application detects the removal and dynamically unloads the connector. As such, all the hooks defined in the connector cannot be triggered any longer.

6.1.1.5 CompileTraversal

The *compileTraversal* tool allows compiling a traversal connector to a binary Java class representation. Likewise to the compileAspect and compileConnector tools, the compileTraversal tool first translates the traversal connector to a Java class implementing the traversal connector. Afterwards, the traversal connector is compiled to the Java binary format using the standard Java compiler. The generated class representing the traversal connector employs in fact the DJ[OL01] library in order to achieve the traversals. The generated class contains an inner class that implements the DJ visitor interface. This inner class takes care of executing the correct hooks at every object instance visited. The correct hooks are computed by dynamically invoking the *isApplicable* methods for each hook. Afterwards, all the applicable

hooks are filtered by applying all combination strategies defined in the traversal connector. Finally, the hooks that are left are triggered in the sequence defined in the traversal connector. When invoking the traversal connector, the visitor is instantiated onto the traversal strategy specified in the traversal connector. The DJ library takes care of the traversal process itself.

6.1.1.6 Introspect

The *introspect* tool is a GUI tool that allows to introspect which connectors are loaded for a given classpath. When new connectors are dynamically added, this is reflected in the introspect tool. In addition, the tool shows all the hooks that are instantiated in a connector. The computed bindings of abstract method parameters to concrete method signatures can also be inspected. Besides pure non-intrusive introspection, the introspect tool allows to change some properties of the connectors. For example, the connectors can be dynamically disabled or enabled and they can be changed from non-static to static or vice-versa. As such, this tool effectively aids debugging JAsCo applications. Figure 47 illustrates a screenshot of the introspect tool. At the left-hand side, the connectors loaded at the given classpath are depicted in a tree structure. The children of the connectors are the hooks. Children of hooks are all the concrete method signatures bound to one ore more abstract method parameters. Currently, there are three connectors loaded: *basicConnector*, *Logging* and *testConnector*. *BasicConnector* and *Logging* each instantiate one hook while *testConnector* instantiates two hooks. All hooks are instantiated on the complete system using wildcards. The right-hand side allows editing some properties of the node selected in the connector tree. The currently selected node is the *Logging* connector and this connector is enabled and static.

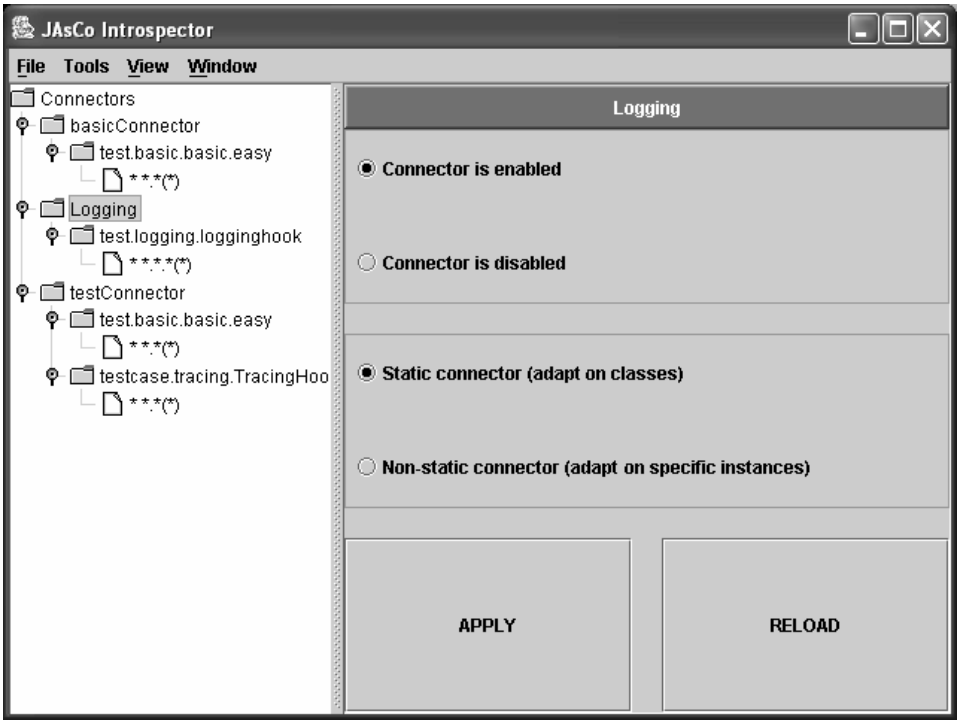

Figure 47: Screenshot of the Introspect tool.

The introspect tool also functions as a simple IDE because it is able to compile aspects and connectors using visual wizards. Transforming Java beans is also supported by the introspect tool. The introspect tool is however by no means meant to be a fully operational development environment. For this purpose, a plug-in for the Eclipse framework [Ecl03] is currently under development. This plug-in allows integrated JAsCo development using visual wizards and templates.

6.1.1.7 Tools API

It is also possible to invoke the JAsCo tools directly from any Java program using the JAsCo tools API. This tools API is implemented in the JAsCo class and supports the following methods:

- `compileAspect(String anAspectFile)` **:** This method compiles the aspect bean defined in the *anAspectFile* file.
- `compileConnector(String aConnectorFile)` **:** This method compiles the connector defined in the *aConnectorFile* file.
- `compileTraversal(String aTraversalFile)` **:** This method compiles the traversal connector defined in the *aTraversalFile* file.
- `removeConnectorInOutputDir(String aConnectorFile)` **:** This method removes the connector defined it the *aConnectorFile* file.
- `removeConnectorInClassPath(String aConnector)` **:** This method removes all occurrences of the connector named *aConnector* that can be found in the current classpath.
- `transformBeanInOutputDir(String javaBean,Vector excluded)` **:** This method transforms the bean denoted by the name *javaBean* which is located in the output directory. The *excluded* parameter is a list of methods or events where no traps may be attached.
- `transformBeanInOutputDirInNewVM(String javaBean,Vector excl)` **:** This method is equivalent to the previous method except that the transformation process runs on a different virtual machine. The method blocks until the transformation process is finished. This is useful when the transformed bean has to be employed in the current virtual machine. As the transformation process loads the untransformed bean for introspection purposes and a Java bean cannot be replaced at run-time in the current JVM specification (1.4), the transformation is never visible in the current virtual machine. Therefore, when transforming the bean in another JVM, the transformed bean can be used in the current JVM if it is not yet loaded by the application.
- `createIntrospectorWindow()` **:** This method creates a new introspection window for the given classpath.

Apart from the methods outlined above, the tools API supports a number of methods that allow to dynamically change the classpath. In addition, an output directory can be specified where all the entities generated by the methods explained above are placed.

6.1.2 Run-time infrastructure

The JAsCo run-time infrastructure is responsible for managing the loading and unloading of connectors and computing which hooks are applicable at a given trap. The run-time infrastructure consists mainly of the ConnectorRegistry class, which is already introduced in section 4.3.2. The ConnectorRegistry is queried each time a trap is executed and returns the list of applicable hooks. The ConnectorRegistry makes sure that the combination strategies and connector combination strategies are executed to filter the list of applicable hooks. In order to detect whether a connector has been placed or removed in the classpath of the application, the ConnectorRegistry scans the complete classpath of the application at fixed time intervals. The default value for the time interval is one second, but it can be altered to be less or more if required. As such, the ConnectorRegistry detects when connectors are placed in the classpath of the application. This is of course not the most optimal solution performance-wise for detecting whether connectors are added or

removed. In order to improve the performance, it is possible to disable the classpath scanning. As such, remotely loading aspects into the application is not possible anymore. From within the application, the tools API can however still be employed to compile and load a connector in this application. It is also possible to specify a connector loadpath instead of the classpath. As such, the run-time infrastructure only scans the loadpath. This is mainly useful in case a large classpapth is specified. The performance overhead is decreased when only a limited connector loadpath has to be scanned instead of the long classpath.

Another solution consists of running the connector registry as a web service or just making the connector registry listen to a network port. The compileConnector and removeConnector tools are then able to communicate whether changes have occurred. The problem with this last solution is that it is less flexible and more error-prone. When several applications are running, several connector registries have to be active. As such, a centralized service has to be available for finding the connector registries that have the directory where the connector is placed in their classpath. This means that a fixed service has to run on the machine where JAsCo applications have to be executed. Apart from the fact that this is a hassle, it is not a 100% pure Java solution as implementing and deploying such a service is operating system dependent. Therefore, the simple scanning solution is chosen for now.

The run-time infrastructure is also responsible for another issue, namely managing properties of connectors over several virtual machines. It is possible that the same connector is loaded in several virtual machines. This is for example the case when a connector is loaded in the introspect tool and by an application that has the connector in its classpath. It has to be possible to alter properties of the connector from the visual introspect tool in such a way that the connector that is loaded in the other virtual machine is affected. This is of course not the default behavior for property changes, but for the particular case of the introspect tool it is required. Otherwise, the connector is for example enabled using the introspect tool, but because the application runs in a different virtual machine, the connector loaded by the application is not altered. Therefore, a separate property file can be generated that contains property values for a connector. Currently, two properties are supported, namely whether the connector is enabled or not and whether the connector is static or not. When a property file is present, the properties of a connector are changed as dictated by the property file. Adding, altering or deleting property files is detected in the same process as adding or removing connectors is detected. Therefore, when a property file is changed at run-time, this is detected by all the applications that have the corresponding connector in their classpath. As such, the property of the connector is altered in all of the applications. When no property files are used in the system, only a minimal performance overhead is required as it suffices to check whether a property file exists for all the loaded connectors.

6.1.3 Performance assessment

One of the biggest issues of dynamic AOP consists of the high performance overhead required for the dynamic adaptiveness of applications. JAsCo is in fact one of the most dynamic approaches as it allows to specify hook triggering conditions that have to be evaluated at run-time (*isApplicable* construct). In addition, combination strategies are able to take run-time values in consideration for filtering hooks. As such, combination strategies have to be computed at run-time degrading the run-time performance. When implementing the JAsCo language, very little attention has been paid to performance considerations. Instead, we systematically choose for maximum flexibility and having a high-quality model. That these choices have a negative effect on performance is no surprise. To prove that our approach is still functional in practice we perform a performance experiment that applies JAsCo to a real-life application, namely PacoSuite, and the overhead of the built-in traps is evaluated.

6.1.3.1 Real-life system equipped with traps

An interesting experiment consists of evaluating the performance of a real-life system that is equipped with traps. Indeed, one might wonder whether it is feasible to equip a complete system with traps without downgrading the performance too badly. In addition, most of the time aspects are not applied to the whole system, but only to a small part of the system. The traps are however applied to the overall system. When the overhead becomes too large, the approach might be unfeasible in practice.

Table 2: Performance evaluation of the transformed version of PacoSuite equipped with traps in comparison to the regular PacoSuite system.

Experiment	Generating glue-code for the encrypted Photolab composition	Checking the encrypted Photolab composition
Without traps	**2814ms**	**206ms**
With traps	**2896ms**	**4763ms**
Overhead of traps	**2,90%**	**2212%**
Number of traps	**2.877**	**220.787**
Overhead per 1000 traps	**28,50ms**	**20,64ms**

As benchmarking application, the PacoSuite visual component composition environment is chosen. PacoSuite consists of 1200 classes and is thus sufficiently large to be able to be used as a realistic benchmark application for JAsCo. The details of the PacoSuite implementation are discussed in section 6.2. For this benchmark, all classes of the PacoSuite application are equipped with traps using the *beanTransformer* tool. The benchmark consists of two experiments and the results are illustrated by Table 2. The time values in Table 2 are average times over ten executions of the experiment where the two highest and two lowest values are discarded for calculating the average. In the first experiment, the time needed for generating glue-code for the encrypted Photolab composition discussed in Chapter 7 is measured. This composition consists of two composition patterns, four composition adapters and fourteen components. This experiment contains a mixture of several kinds of operations, namely computing several DFA algorithms, I/O access to generate glue-code and starting an external compiler. Because only the first part, namely the DFA algorithms, are very resource intensive in the sense that a lot of short methods have to be invoked in order to compute the algorithms, the total overhead is quite good. The trapped PacoSuite only runs 2,90% slower than the non-trapped PacoSuite on our test system[6]. During the computation, 2877 traps have been encountered, with an average computation overhead of 28,50ms per thousand traps.

In order to evaluate the overhead of using JAsCo when a lot of traps are encountered, another experiment is conducted. This experiment evaluates the time needed for validating the same encrypted Photolab composition. As explained before, checking the validity of a complete composition is very expensive for a trapped system because a lot of method invocations are involved and thus a lot of traps are encountered. In

[6] Pentium-M 2Ghz, 1GB RAM, Windows XP SP1, JDK 1.4.1_02

this particular example, 220787 traps are encountered. Although the overhead per thousand traps is smaller in comparison to the previous experiment, the total performance overhead of the trapped system is quite large now. In relative terms, the trapped PacoSuite runs 2212% slower than the original PacoSuite for this experiment, which is of course unacceptable. However, one has to keep in mind that this is the worst-case scenario. When less traps are encountered like for instance in the first experiment, the overhead is acceptable for most application domains. Nevertheless, the performance overhead of using JAsCo is way too large when confronted with a lot of traps.

6.1.3.2 Jutta: Improving the current JAsCo implementation

Notice that the implementation of JAsCo is still in prototype phase and that not a lot of attention has been paid to performance optimizations. The main reason for the high overhead of traps is that the whole JAsCo system is in fact an aspect interpreter. JAsCo evaluates for each trap which hooks are applicable for that trap. When no connectors are added or removed, the set of applicable hooks remains unchanged for every trap. As such, when the same trap is encountered a lot, the same logic for finding the appropriate aspects and executing their behavior, is computed over and over again. Therefore, a huge performance gain can be realized when the combined aspect behavior could be somehow compiled and cached for traps that are used often. Of course, this compilation process requires some time, but when the trap is executed a lot, this pays of. In fact, this strategy is similar to just-in-time compilers used for modern virtual machines and therefore our approach is named *Jutta* (just-in-time combined aspect compilation). State-of-the-art just-in-time compilers employ a so called inline caching approach to optimize method execution [HCU91]. The optimisation results from the observation that most code that sends a message to an object always sends messages to objects of the *same class*. Therefore the message send can be compiled to a direct call to the appropriate method of that class. That is, the class of the object that receives a message is cached *inline* in the code that sends the message.

The first application of inline caching for AOP is introduced in the Wool Dynamic AOP system proposed by Sato et al. [SCT03]. Wool supports two different weaving techniques: using breakpoints for trapping the application or invasively inserting the aspect behavior inside the target classes. Employing the breakpoints is in fact similar to our trapped approach. The traps are however not inserted explicitly; the Java debugging system takes care of trapping the whole application. Likewise to our traps, using the debugging interface requires a fixed overhead per encountered trap. Invasively inserting the aspects into the target components at the other hand, does require a substantial overhead the first time the trap is executed. Succeeding executions of the traps are however a lot faster because the aspects are already in place. The Wool system allows aspects to be in control of the decision whether they have to be invasively inserted or not for a given trap. As such, aspects that will be executed a lot can be dynamically inserted in the target application. These aspects are thus compiled into the target application at run-time, improving the performance of dynamic AOP greatly for aspects that are executed a lot. The technology used by Wool does however require the virtual machine to run in debugging mode. As a result, a global overhead for the complete application is experienced. The JAsCo technology uses explicit traps and thus avoids this global overhead. Furthermore, the Java debugging interface is only supported on a limited number of virtual machines and as such the approach is less portable as explained in section 4.3.1. In addition, wool aspects are quite limited in their dynamism in comparison to JAsCo hooks. Wool aspects can not define run-time conditions, wildcards that are resolved at run-time, dynamic combination strategies etcetera. As such, the ideas of Wool cannot be straightforwardly recuperated into JAsCo.

In order to recuperate and extend the ideas of Wool in JAsCo, we first need to evaluate which behavior needs to be executed for each trap. The combined behavior for several hooks at a given trap consists in fact

of three steps. The first step evaluates which hooks are applicable for a given trap using the constructors of the hooks and the *isApplicable* methods if defined. The second step applies the available combination strategies to the list of applicable hooks. Finally, the third step consists of executing the behavior methods of all the applicable hooks in the appropriate sequence. Caching the result of the first step of a hook for each trap is not always achievable because it is possible that the first step has to be re-evaluated for every execution of a given trap. For example, when a hook defines a *cflow* condition in its constructor, this constructor has to be re-evaluated for every execution of a trap. However, the entire constructor does not have to be re-evaluated. In this case, only the result of the *cflow* condition is able to change over different executions of the trap. As such, partial evaluation techniques can be used to cache a partially evaluated constructor. In addition, for the particular *cflow* construct, it is sometimes possible to statically analyze whether the condition might ever be true or not by examining the call graph of an application. This technique is elucidated in [SD03].

In the second step, combination strategies are applied to the set of hooks that are tested to be applicable at the given trap. Of course, when no combination strategies are present, caching this behavior is straightforward. In general however, caching the result of the combined behavior of all combination strategies for a given trap is not possible. This is because a combination strategy might depend on dynamic values to compute the list of applicable hooks. As such, combination strategies have to be recomputed for every execution of a given trap. Some combination strategies do however not depend on dynamic values and always render the same result for a given input set of hooks. As such, these combination strategies do not have to be recomputed for every execution of a trap. Currently, there is however no way to find out whether a combination strategy depends on dynamic values or not. But we could for example introduce a keyword or property that defines that a combination strategy has to be recomputed for every execution of a trap. When combination strategies do not have that property, they can be optimized by caching the resulting combination.

In the last step, the sequence of the hooks is determined as specified by the connector and the behavior methods of applicable hooks are executed. This combined behavior can be easily cached because it does not change for every execution of a given trap. Notice that the result of the execution of the hook behavior methods is not cached, merely the combined hook behavior method invocations are compiled and cached.

The JAsCo approach is however a dynamic AOP approach. As such, the cached behavior for a given trap might become invalid in some cases. This happens when a connector is added that instantiates a hook that is applicable on that trap or when a connector is removed that contains an applicable hook for the trap. In addition, it is possible to change properties of a connector dynamically so that the applicable context of the instantiated hooks is altered. The Jutta system has to be able to cope with these issues.

```
1  public void executeJoinpoint(Joinpoint p) {
2
3     hook0._Jasco_initialze(p);
4     hook1._Jasco_initialze(p);
5     hook2._Jasco_initialze(p);
6     hook1.before();
7     hook2.before();
8     hook0.replace();
9  }
```

Code Fragment 65: Java counterpart of the cached combined aspectual behavior at a joinpoint.

In order to assess whether caching compiled aspect behavior would be feasible and effective for optimizing JAsCo, the JAsCo implementation is extended with a prototype Jutta system. This system allows storing combined behavior of several hooks for each trap. In order to generate fragments of combined behavior,

the byte-code manipulation library Javassist [CN03] is employed. Using Javassist, a Java byte code class representation is generated on the fly, without requiring a compilation step. Code Fragment 65 illustrates the Java counterpart of an example cached joinpoint behavior execution. The code fragment first initializes all the hooks with the current joinpoint and then executes only those advices that are defined in the connector in the correct sequence.

The current Jutta implementation is however still an experimental prototype and does not allow caching combined behavior when hooks depend on dynamic values for deciding whether they are applicable. This is for example the case with the *cflow* condition in a hook constructor or an *isApplicable* method. In those cases, the current Jutta system falls back to the interpreter. As such, the JAsCo-Jutta implementation is still correct, but does not experience a performance gain when employing these kinds of hooks.

Table 3: Performance evaluation of a first optimization to JAsCo. The transformed version of PacoSuite equipped with traps and compared to the regular PacoSuite system.

JAsCo-Jutta	Generating glue-code for the encrypted Photolab composition	Checking the encrypted Photolab composition
Without traps	**1086ms**	**207ms**
With traps	**1107ms**	**216ms**
Overhead of traps	**1,92%**	**4,19%**
Number of traps	**1.123**	**220.787**
Overhead per 1000 traps	**18,55ms**	**0,039ms**

To evaluate this more intelligent JAsCo system, the two experiments of the previous section are repeated. Table 3 illustrates the results. The time values in Table 3 are average times over ten executions of the experiment where the two highest and two lowest values are discarded for calculating the average. For the first experiment, namely generating glue-code for the encrypted photolab composition, the overhead is decreased from 2,90% to 1,92% using the enhanced JAsCo implementation. The performance gain of the Jutta system does not come into play a lot because this experiment consists of a mixture of several kinds of operations and as such only a limited number of traps are executed. In the second experiment however, a lot of traps are encountered. As such, the currently implemented Jutta system performs maximally. The overhead for executing the 220787 traps is 4,19% in comparison to the non-trapped PacoSuite. The extended JAsCo implementation improves greatly over the original JAsCo implementation as the overhead is shrinked from a gigantic 2212% to an acceptable 4,19%. Notice that this overhead is in a worst case scenario where a lot of traps are encountered. When fewer traps are encountered, the overhead is a lot smaller as illustrated by the first experiment of the previous section.

In order to better asses the maximum overhead of the extended JAsCo system, another experiment is done (see Table 4 for the results). In this experiment, a composition for a Scrabble game is validated by the PacoSuite tool. The Scrabble composition consists of six composition patterns, nine composition adapters and thirteen components. Over three million traps are encountered when validating the Scrabble composition. The trapped PacoSuite only runs 3,95% slower than the original PacoSuite. Merely 0,014ms

are required to execute thousand traps in this experiment. As such, it is safe to say that the overhead of a trapped application versus a non-trapped application is in most cases maximum 5%. This is an acceptable result for most application domains. Of course, the Jutta system performs maximally when aspects are applied, because Jutta compiles combined aspectual behavior, which is in case that no aspects are applied quite trivial. Therefore, the next section presents a performance evalutation of JAsCo/Jutta and compares it with several state-of-the-art dynamic AOP solutions available today.

Table 4: Performance evaluation of a first optimization to JAsCo. The transformed version of PacoSuite equipped with traps and compared to the regular PacoSuite system.

JAsCo-Jutta	Scrabble Composition
Without traps	1097ms
With traps	1140ms
Overhead of traps	3,95%
Number of traps	3.034.216
Overhead per 1000 traps	0,014ms

6.1.3.3 Comparison with other approaches

In order to assess the performance of the JAsCo-Jutta dynamic AOP system, we compare the performance with other state-of-the-art dynamic AOP systems. For this end, we employ two benchmark applications: PacoSuite and the JAC [PDFS01] benchmark. The PacoSuite benchmark is meant as a real-life performance evaluation and consists of loading and validating a component composition. The JAC benchmark at the other hand is a benchmark application employed by the JAC team that is a pure synthetic benchmark which involves invoking several methods with different signatures. The following dynamic AOP approaches are tested and compared: JAsCo-Jutta, JBoss/AOP [BCF+03, FR03], PROSE [PAG03], JAC and AspectWerkz [BV03]. Notice that this selection is not meant as a comprehensive overview of all existing dynamic AOP systems. We merely selected those systems that are publically available and seemed stable enough in our opinion. Nevertheless, this selection is a good overview of current dynamic AOP approaches.

In the first experiment, we execute both benchmarks without any aspects applied. The benchmark applications are first executed a couple of times without timing the execution time in order to give the JVM the opportunity to optimize the code. Also, some approaches install traps that enable aspect interference at load-time. The overhead of this process is thus not included in the timings. The timings are average values over 40 different executions of the benchmark application. Table 5 illustrates the results of the performance comparison without any aspects applied on our test system [7]. JAsCo-Jutta is one of the most optimal dynamic AOP systems when no aspects are applied. Notice that the higher overhead of AspectWerkz in the PacoSuite benchmark is due to that AspectWerkz also traps field access, while the other approaches do not allow placing traps at fields. The JAC bench consists out of method invocations

[7] Pentium 4 2.66Ghz, 1GB RAM, Mandrake Linux 9.2, JDK 1.4.

only, so there AspectWerkz performs better. The PacoSuite application does not run with JAC, so sadly enough no results are obtained for JAC in the PacoSuite benchmark. JAC experiences a higher overhead than other approaches because JAC employs the Java reflection API, which is not performant at all.

Table 5: Comparing performance of JAsCo with other AOP approaches, no aspects applied.

No aspects test	**JAC bench** 40000 traps	**PacoSuite bench** 210200 traps
Without AOP	**10ms**	**547ms**
JAsCo-Jutta 0.4	**10ms**	**554ms**
JBoss/AOP 4.0	**10ms**	**571ms**
AspectWerkz 0.9	**14ms**	**621ms**
Prose 1.1.2	**14ms**	**609ms**
JAC 0.11	**620ms**	**-**

Table 6: Executing the JAC bench with an around advice applied to the complete system. * means that the overhead is not measurable.

One around advice	**JAC bench** 40000 joinpoints	**Overhead per aspect execution**
AspectJ	**10ms**	*
JAsCo-Jutta 0.4	**19ms**	**0,225 µs**
JBoss/AOP 4.0	**48ms**	**0,950 µs**
AspectWerkz 0.9	**42ms**	**0,800 µs**
Prose 1.1.2	**43523ms**[8]	**1088 µs**
JAC 0.11	**711ms**	**17,56 µs**
JAsCo without Jutta	**42301ms**	**1057 µs**

[8] The PROSE system does not allow an around advice, so we applied a single before advice instead. Theoretically, a before advice can be executed faster because no around advice chain has to be built. Still, a very high overhead is experienced, which is probably due to a non-optimized implementation.

Of course, the real value of a caching system only appears when we apply aspects. Therefore, the same benchmarks are executed again with a single aspect applied to the complete system using wildcards. The aspect contains the corresponding equivalent of an AspectJ around advice that merely executes the original behavior and increases a counter. The counter increasing is done to avoid that some intelligent approaches would detect an empty advice and would never execute the actual advice. This is of course a valuable strategy in general, but now we want to assess the overhead of aspect execution, so the aspect's advice has to be executed. Table 6 depicts the result of applying a single around advice to all method invocations of the JAC benchmark. As utopic reference performance, the AspectJ performance of an around advice is incorporated. The performance of JAsCo without the Jutta system is also included in order to prove the gigantic performance gain of the Jutta system. As Table 6 shows, the JAsCo Jutta system induces a performance gain of more than 70% in comparison to the closest competitors AspectWerkz and JBoss/AOP. The performance gain in comparison with other dynamic AOP systems (including JAsCo without Jutta) is even more than 98%!

We also ran the PacoSuite benchmark with a single around aspect applied. Table 7 illustrates the results, which are in line with the results of Table 6. AspectWerkz does suffer a higher considerably higher overhead than with the JAC benchmark. This is probably due to the additional traps for field access. Without an intelligent caching system, the system has to query whether the aspect is applicable for all these traps. Notice that JAC and PROSE are omitted because we failed to get an around advice working on this benchmark application.

Table 7: Executing the PacoSuite bench with an around advice applied to the complete system.

No aspects test	**PacoSuite bench** 210200 joinpoints	**Overhead per aspect execution**
AspectJ	**553ms**	**0,0285 µs**
JAsCo-Jutta 0.4	**667ms**	**0,571 µs**
JBoss/AOP 4.0	**787ms**[9]	**1,15 µs**
AspectWerkz 0.9	**3268ms**[9]	**15,6 µs**

6.1.3.4 Discussion

This is of course only a first prototype implementation of the Jutta system, but the results already suggest that when implementing the full caching system as outlined above, a huge performance gain can be expected in comparison to the original JAsCo implementation. In addition, the implementation of JAsCo is not yet mature and only in prototype stage. As JAsCo is currently used in several industrial and academic

[9] The actual result with AspectWerkz was 5040ms and JBoss/AOP 885ms, but because they both also trap private methods all other approaches do not, a higher overhead is experienced as the around advice is executed for all public and private methods. There is also no way to apply an aspect only on all public methods. Therefore, the overhead for the execution of only public methods is computed from the overhead per around execution times the number of public methods. Notice that this was not an issue in the JAC benchmark because it only consists of public methods.

research projects, the implementation will become more and more mature. As such, performance bottlenecks can be identified and the performance of the JAsCo system is enhanced. Nevertheless, the JAsCo/Jutta system outperforms all tested dynamic AOP systems including some approaches that are advertised as production quality software (JBoss/AOP and AspectWerkz). Furthermore, the JAsCo approach is one of the most dynamic AOP approaches when compared to the tested systems. The overhead for allowing to remotely insert aspects by scanning the classpath for example, is only experienced by JAsCo, but still included in the benchmark results.

The Jutta system is not only applicable to JAsCo, the ideas can be recuperated in any other dynamic AOP approach regardless of which technology they use for intercepting the program execution. Therefore, we plan to decouple the Jutta system from JAsCo and as such achieve a general dynamic AOP optimizer. In the long term, the best approach to support dynamic AOP or even regular AOP consists of dedicated aspect-oriented virtual machines. Indeed, preprocessing, load-time trap insertion or employing the debugging interface of a virtual machine are all solutions that are feasible on the short-term, but are quite cumbersome and error-prone in comparison with a dedicated execution environment. The Jutta system and ideas are however still applicable for such dedicated virtual machines.

The main limitation of the Jutta system is the overhead required for generating the JIT-compiled cached behavior. Currently, this overhead is around 10ms for every combined aspect behavior fragment that has to be generated. For most application domains, this is an acceptable overhead. Furthermore, the optimized code fragment is only generated when the joinpoint is executed the first time. As such, for joinpoints that are not executed, no overhead is experienced. The Jutta system also stores all code fragments generated for a given hook combination. As such, when at another joinpoint, the same hook combination is applicable, the overhead for generating the combined hook behavior code fragment is avoided. In addition, the Jutta system includes a set of pre-defined typical combined aspectual behaviors. For those combined aspectual behaviors, the generation overhead is avoided.

6.2 PacoSuite

The PacoSuite tool suite consists of two main visual tools, namely PacoDoc and PacoWire. PacoDoc allows visually editing, loading and saving composition patterns, component usage scenarios, composition adapters and invasive composition adapters. PacoWire is the actual component composition environment that allows to visually wire the components, composition patterns and (invasive) composition adapters. The wired composition can be automatically validated using the algorithms described in this dissertation and glue-code is automatically generated.

The PacoSuite tool suite is in fact an extended version of the PacoSuite environment developed in the context of Bart Wydaeghe's PhD. The extension to both PacoDoc and PacoWire consists mainly of support for composition adapters and invasive composition adapters. The complete PacoSuite implementation consists of 1202 classes and 34436 lines of code without counting comments and blank lines whereas the original PacoSuite consists of 718 classes and 20971 lines of code.

6.2.1 PacoDoc

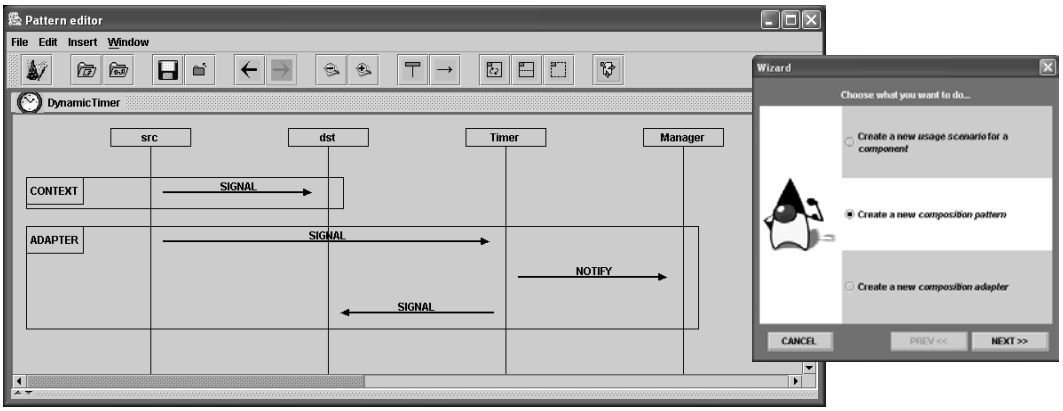

Figure 48: Screenshots of PacoDoc showing the DynamicTimer composition adapter and a visual wizard.

The PacoDoc tool is extended in order to be able to create, edit, save and load composition adapters and invasive composition adapters. Likewise to component usage scenarios and composition patterns, (invasive) composition adapters are saved to an XML format. Figure 48 illustrates two screenshots of the PacoDoc tool. The left screenshot illustrates the DynamicTimer composition adapter introduced in Chapter 3. The right screenshot shows a visual wizard that allows to easily create the PacoSuite entities by following the wizard. In addition, the wizard allows generating skeleton documentation.

6.2.2 PacoWire

PacoWire is our actual component composition environment that allows creating an application by visually assembling components, composition patterns and composition adapters. The following sections explain the extensions made to PacoWire in order to support composition adapters and invasive composition adapters implemented in JAsCo.

6.2.2.1 Composition Adapter Support

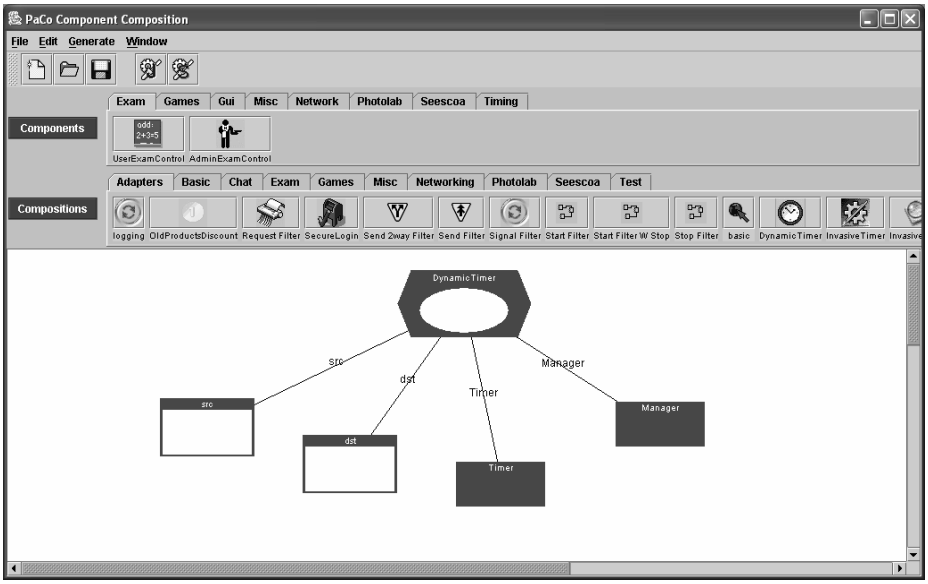

Figure 49: DynamcTimer composition adapter dragged on the composition canvas of pacowire.

Likewise to component usage scenarios and composition patterns, it is possible to visually drag composition adapters to the composition canvas. A composition adapter is represented by a hexagonal shape connected to several squares representing the roles of the composition adapter. Figure 49 illustrates a screenshot of the DynamicTimer composition adapter represented in the PacoWire tool. The hexagonal shape has an oval gap inside where a composition pattern can be filled in. Context part roles also contain a square gap where composition pattern roles can be mapped upon. Remember that only roles of the context part have to be mapped upon roles of a composition pattern. Adapter part roles are newly introduced roles. Mapping a composition pattern onto a composition adapter amounts to visually dragging the composition pattern on the composition adapter. The composition adapter context part roles are automatically matched with composition pattern roles using the algorithm elucidated in section 3.4. When several different role mappings are possible, a dialog is popped up where the component composer is able to select the appropriate role mapping out of all the possible role mappings. It is of course also possible to disable the automatic role mappings. As such, the composition adapter context part roles have to be manually mapped onto the composition pattern roles by dragging the roles. If the application of the composition adapter is invalid, an error dialog is shown and the drag of the composition pattern onto the composition adapter or vice versa is refused.

When a composition adapter is applied on a composition pattern, components can be mapped upon the composition pattern roles, context part roles merged with composition pattern roles and adapter part roles. This is illustrated by Figure 50 where the JButton component is about to be dragged onto the merged Control/Src role. Likewise to mapping components onto regular composition pattern roles, the drag-and-drop action is refused when the component is detected to be incompatible with the selected role. After all roles are filled, the composition is checked as a whole and the regular PacoSuite glue-code generation process can start.

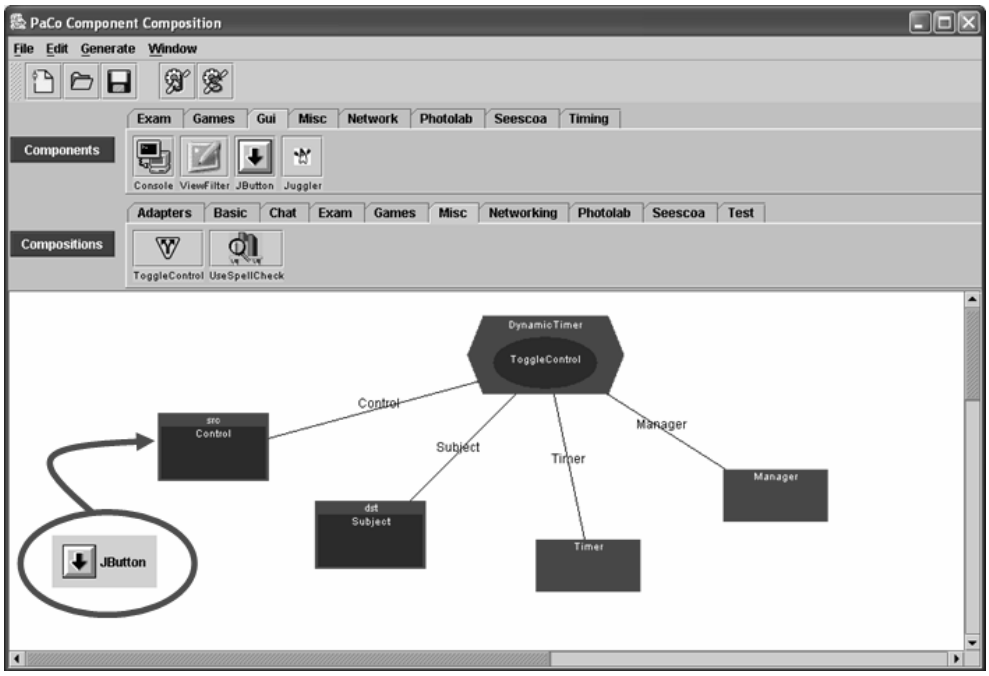

Figure 50: Dragging the JButton component onto the combined Control/Src role.

6.2.2.2 Invasive Composition Adapter Support

An invasive composition adapter is represented on the composition canvas in the same way as a regular composition adapter. Figure 51 illustrates a screenshot of the PacoWire tool where the InvasiveTimer invasive composition adapter introduced in Chapter 5 is dragged onto the composition canvas. An invasive composition adapter can also be dragged upon a composition pattern and afterwards components are visually mapped onto the available roles. When all the roles are filled, the composition can be checked as a whole and the glue-code generation process can start as outlined in section 5.3. During the glue-code generation process, a connector in the JAsCo language is generated for every invasive composition adapter involved. The compileConnector tool introduced in section 6.1.1.3 is then employed to compile the generated connectors. Afterwards, the glue-code itself is generated and the application generation is finished. Notice that for a user of the PacoWire application, it is completely transparent whether an invasive or regular composition adapter is used. An invasive composition adapter is represented in the same way on the composition canvas and the JAsCo tools are executed behind the scenes.

There is however one noticeable difference between an invasive composition adapter and a regular composition adapter. This is because an invasive composition adapter is implemented by a JAsCo aspect bean. JAsCo aspect beans are backward compatible to Java beans and are thus also able to define bean properties. For example, a bean property for the InvasiveTimer invasive composition adapter could be the starting offset for counting timestamps. Bean properties of a regular Java bean component can be altered using a visual wizard. It is also possible to alter bean properties of an invasive composition adapter using a similar wizard.

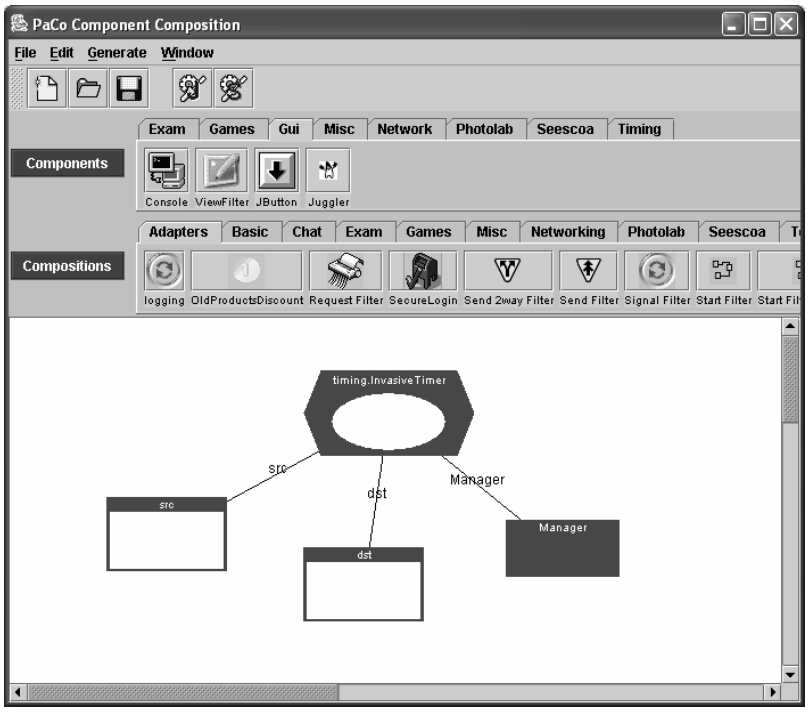

Figure 51: InvasiveTimer invasive composition adapter shown on the composition canvas.

The JAsCo approach requires that all components are transformed to insert the traps that allow aspect interference. When no traps are present, the aspect bean implementation of an invasive composition adapter is not able to realize its adaptations. Therefore, the PacoWire tool makes sure that every Java bean which is not yet trapped and where an aspect bean possibly applies to, is transformed while generating glue-code. Applying this transformation process does of course significantly slow down the glue-code generation process. Therefore, it is possible to transform a Java bean component when it is added to the PacoSuite library.

6.2.2.3 Stacking composition adapters

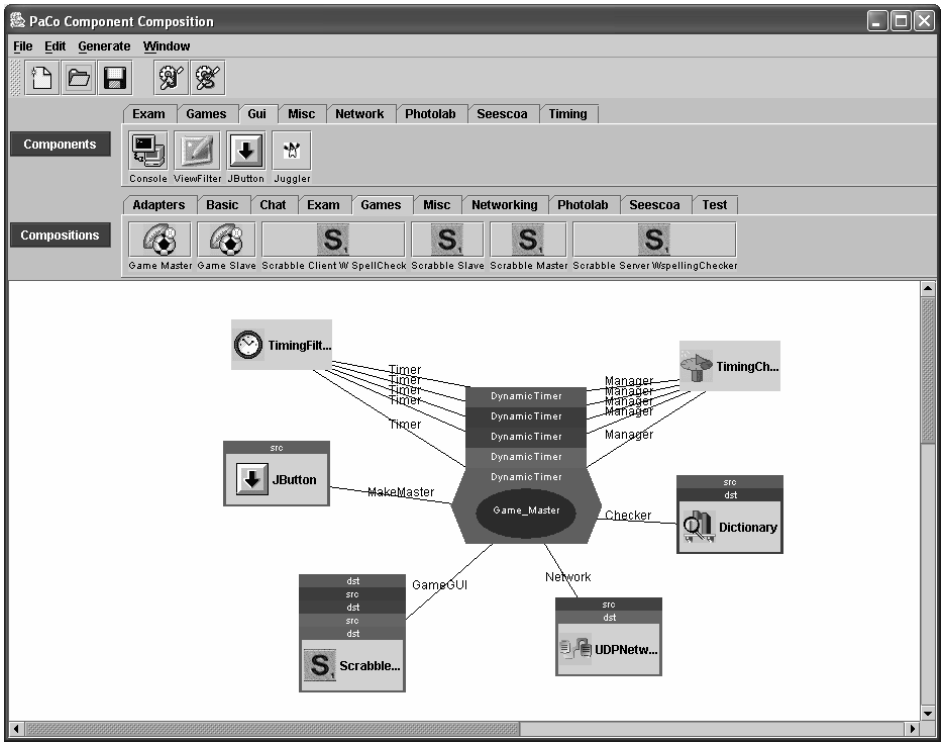

Figure 52: Stacking several DynamicTimer composition adapters onto the same composition pattern.

It is also possible to stack multiple (invasive) composition adapters onto the same composition pattern. The composition adapters are inserted in the sequence the component composer specifies. In order to apply several composition adapters on the same composition pattern, it suffices to drag all the composition adapters onto the composition pattern. Only the first composition adapter keeps its original representation. The subsequent composition adapters are shrinked in order to enhance clarity. This is shown by Figure 52 where several instances of the DynamicTimer composition adapter are applied onto the Game_Master composition pattern in order to time all communication between the roles of the Game_Master composition pattern. The first instance of the DynamicTimer composition adapter is still represented as a hexagonal shape while the other instances are shrinked to a small rectangular shape labeled with their type. The composition adapter context part roles are shrinked in a similar way. Every composition adapter instance is colored differently in order to be able to recognize which role belongs to which composition adapter. Notice that it is of course also possible to apply instances of several different composition adapters onto the same composition pattern. Also notice that the roles are already filled with concrete components in Figure 52.

Chapter 7 Case Study

In this chapter, a case study is introduced as a validation of the approach proposed in this dissertation. The case study implements a larger and non-trivial example in comparison to the timestamping concern employed as a running example throughout this thesis. Furthermore, it can be argued that using an invasive composition adapter for specifying timing constraints validation is not really necessary. Indeed, a regular composition adapter is also able to describe this concern, only the accuracy of the timestamps differs. Therefore, a more elaborated example is discussed. This case study consists of an e-commerce application that is visually wired using the PacoSuite tool. Several crosscutting concerns are identified that can be easily modularized using (invasive) composition adapters. The following sections discuss the case study in more detail. The next section describes the PhotoLab application as it is created in the original PacoSuite approach using composition patterns and components. Sections 7.2, 7.3, 7.4 and 7.5 describe four crosscutting concerns that are modularized as (invasive) composition adapters.

7.1 Wiring the PhotoLab application in PacoSuite

The case study at hand consists of a distributed e-commerce application for printing digital pictures. The e-commerce application consists of two sub-applications: a client and a server. The client application requires users to login by providing a username and password. The login information is sent to the server for validation. When the user has the appropriate permissions, the user is able to select several digital pictures stored on the host computer. In addition, it is possible to define the print format and the number of prints for a given picture. The client application validates whether the selected pictures are indeed pictures encoded in a supported format (JPEG, PNG and GIF). In addition, the quality of the picture with respect to the selected print format is analyzed. When the selected picture file is not supported or when the quality is beneath a certain threshold, the picture is not accepted. When the user has selected all the pictures, they are uploaded to the server together with the print size and number of prints information. The server calculates the price for this order, bills it onto the credit card of the user and returns a receipt.

The original PhotoLab application consists out of two composition patterns and ten components. The two composition patterns represent the two sub-applications of the PhotoLab system, namely the client and server application. The next sections introduce both the client and server application.

7.1.1 Client Application

The *ECommerceClient* composition pattern describes the protocol required for the client application. This composition pattern is in fact a generic e-commerce composition pattern rather than a specific digital printing composition pattern. Figure 53 illustrates the ECommerceClient composition pattern. This composition pattern consists of four roles: the *Graphical User Interface (GUI)* role, the *Network* role, the *Launcher* role and the *Checker* role. The GUI role represents the user interface of the online shop which interacts with the user. The client application communicates with the server application through the Network role. The Launcher role is responsible for launching the application. Finally, the Checker role is responsible for validating the pictures selected by the user. In the PhotoLab example, this means checking whether the selected file is a valid picture and whether the quality is good enough for the selected print format.

The composition pattern specifies the following interaction: First, the Launcher role is responsible for starting the application by sending a START primitive. As a result, the component mapped on the GUI role lauches a login dialog. Afterwards, the user interface sends the login information of the user to the Network role by sending a SEND message. The server then checks the login and password of the user and returns the result using a PERFORM primitive. In case of false login information, the application stops. Otherwise, the GUI interface for selecting the pictures to print is started. Afterwards, the user interface requires the Checker role to check the validity of the supplied user data. This is done by sending continuous REQUEST-ANSWER messages between the GUI and Checker roles. When the user accepts the order, it is sent over the network using a SEND primitive and the network returns the receipt by sending a PERFORM primitive.

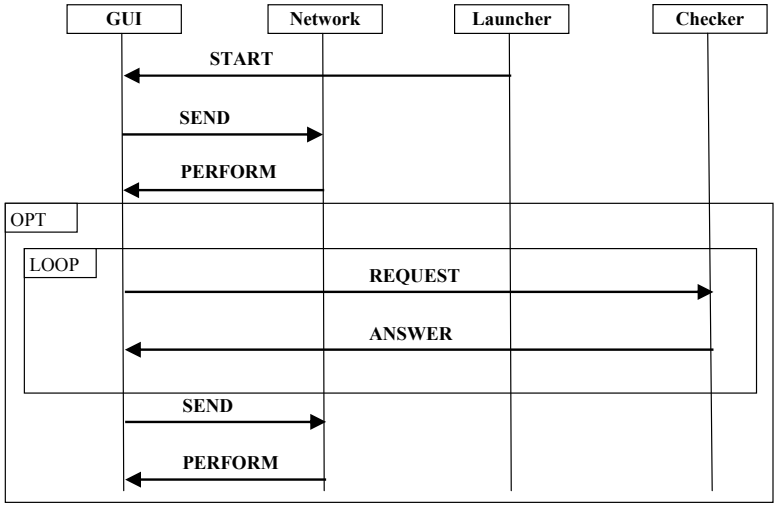

Figure 53: ECommerceClient composition pattern.

In order to instantiate the ECommerceClient composition pattern, components have to be mapped onto the roles of the composition pattern. The following four components are employed in this composition: JButton, PhotoRenderer, UDPNetwork and PhotoLabGUI. The JButton component is already discussed in the previous chapters (see Figure 7, page 52). The PhotoRenderer component allows visualizing a given picture and is able to validate the quality of a picture. Figure 54 illustrates a usage scenario of the PhotoRenderer component. This usage scenario specifies two alternatives in a loop. In the first alternative, the PhotoRenderer component expects a NOTIFY message implemented by the *preview* method. As a result, the PhotoRenderer component visualizes the picture sent. In the second alternative, the PhotoRenderer expects a REQUEST implemented by the *checkPictures* method. This message contains a set of pictures that have to be validated. Afterwards, an ANSWER message is sent, implemented by the *notifyValidateResult* containing the validation results.

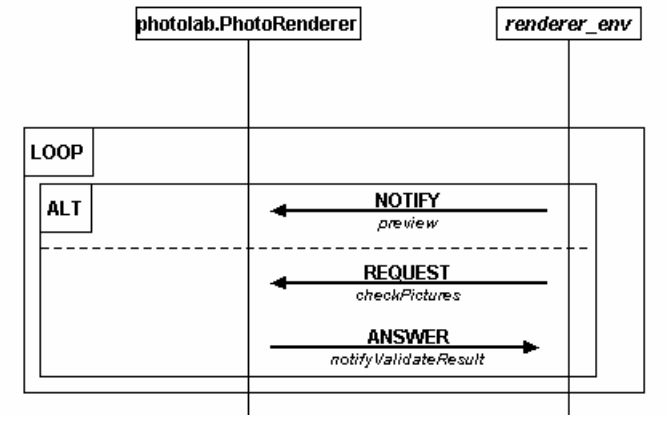

Figure 54: Usage Scenario of the PhotoRenderer component.

The UDPNetwork component is a networking component that allows sending and receiving message using the UDP/IP network protocol (see Figure 55). The component either expects a SEND message implemented by the *sendPacket* method in order to send a UDP packet or sends a PERFORM message implemented by a *receive* event when a packet has been received.

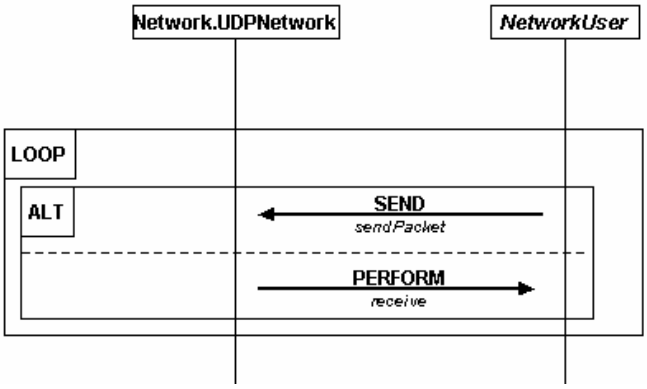

Figure 55: Usage scenario of the UDPNetwork component.

Figure 56 illustrates a usage scenario of the PhotoLabGUI component. This component represents the user interface itself and cooperates with all previously presented components. The component first expects a START message implemented by the *showLogin*. The result is that the PhotoLabGUI component shows a login dialog box. When the user accepts a login and password, the login and password are sent to the Network environment participant using a SEND primitive implemented by the *communicate* event. Afterwards, the component expects a PERFORM message implemented by the *loginResult* method containing the result of the login. When the login fails, the login dialog box is shown again and the user is able to correct the supplied login and password combination. When the login succeeds, the component shows a user interface that allows selecting digital pictures, specifying a print format and supplying the number of prints. It is possible to visualize the pictures by sending a NOTIFY message implemented by the *notifyPreviewRequest* event to the *Verifier* environment participant. In addition, the quality of the supplied pictures can be validated by sending a REQUEST message implemented by the *notifyValidateRequest* event to the same *Verifier* environment participant. The *Verifier* environment participant returns the validation result by sending an ANSWER message implemented by the *buyProducts* method. When all pictures are selected, the component sends the result to the Network environment participant using a SEND primitive implemented by the *communicate* event. Afterwards, the component expects a PERFORM primitive implemented by the *submitReport* method containing the resulting receipt.

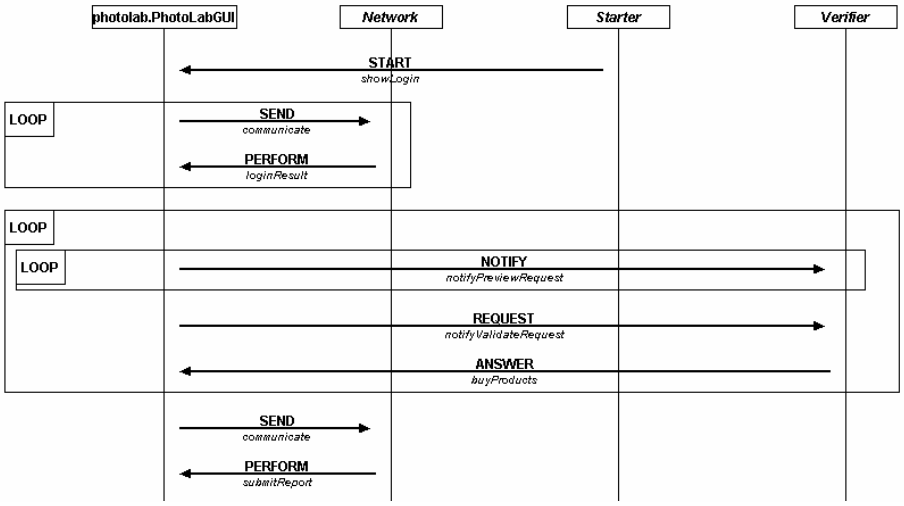

Figure 56: Usage Scenario of the PhotoLabGUI component.

The four components above can be used to fill the roles of the ECommerceClient composition pattern. Figure 57 illustrates a screenshot of the ECommerceClient composition pattern in the PacoSuite tool with filled-in roles. The oval with squares connected to it via lines is the representation of a composition pattern in the PacoSuite tool. Each square represents a role of the composition pattern and the name of the role is specified on both the square and the line that connects it to the composition pattern (see Launcher role in Figure 57). Components are also represented by squares containing an icon and the name of the component. In the screenshot of Figure 57, four components are present on the canvas. The PhotoLabGUI component is mapped onto the GUI role, the UDPNetwork component is mapped onto the Network role and the PhotoRenderer component is mapped onto the Checker role. The JButton component still needs to be dragged onto the remaining Launcher role. Afterwards, glue-code is generated that translates the syntactical incompatibilities of the involved components. For example, the JButton component fires an *actionPerformed* event that needs to be translated into the invocation of the *showLogin* method on the PhotoLabGUI component. In addition, the glue-code forces the components to behave as specified by the composition pattern by ignoring all unexpected messages.

Notice that the four components do not offer the exact functionality as requested by the composition pattern. The PhotoRenderer component for instance allows to visualize pictures by sending a NOTIFY primitive. This behavior is however not employed in the ECommerceClient composition pattern. Allowing a component to offer more functionality than what the composition pattern expects or vice versa is one of the main benefits of the original PacoSuite approach because it contributes to achieving a highly reusable components and composition patterns.

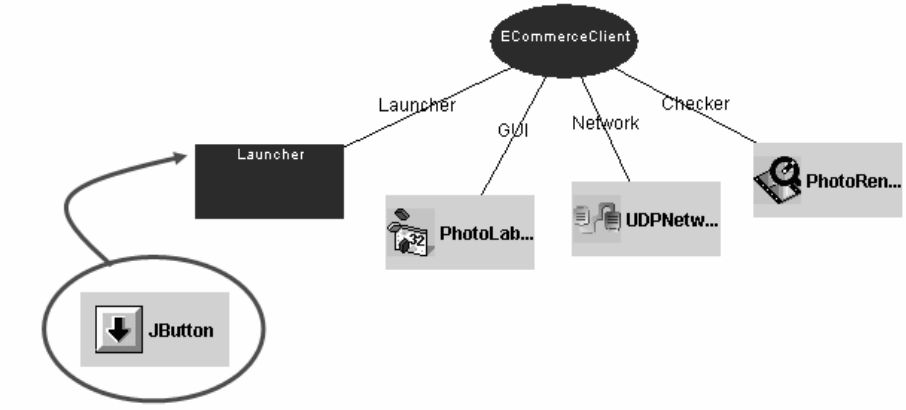

Figure 57: Mapping the JButton, PhotoLabGUI, Network and PhotoRenderer components onto the ECommerceClient composition pattern.

7.1.2 Server Application

Likewise to the client application, the server application also consists of one composition pattern, namely *ECommerceServer* (see Figure 58). This composition pattern realizes the protocol required for a generic e-commerce server. The ECommerceServer composition pattern contains six roles: *Logix*, *Network*, *UserDB*, *ProductDB*, *Accountant* and *Toggler*. The Logix role represents the main orchestrating role in this collaboration and realizes the server logic by communicating with the other roles. The Network is used to communicate with the client application. The UserDB and ProductDB roles represent two databases that contain information about respectively the users and products of the e-commerce application. The Accountant role is responsible for computing the price for a given order and for billing the user. The

Toggler role is used to start and stop the e-commerce application. The application starts with a START signal sent by the Toggler role. Afterwards, the Logix role expects a PERFORM message from the Network role containing the username and password of a given user. The Logix role verifies the username and password by querying the UserDB role and the result is sent to the Network role using a SEND primitive. When the user supplies a correct password, the Logix role expects a PERFORM signal from the Network role containing the order of the user. The logix role then sends a REQUEST primitive to the Accountant role to compute the price of the order. The Accountant accumulates the total price by fetching the product information from the ProductDB role and bills the total amount to the user. Afterwards, the Accountant role sends an ANSWER primitive to the Logix role containing the receipt. The receipt is then forwarded to the Network role by the Logix role. Afterwards, either a STOP message is sent by the Toggler role or a new user is able to login by sending a PERFORM message.

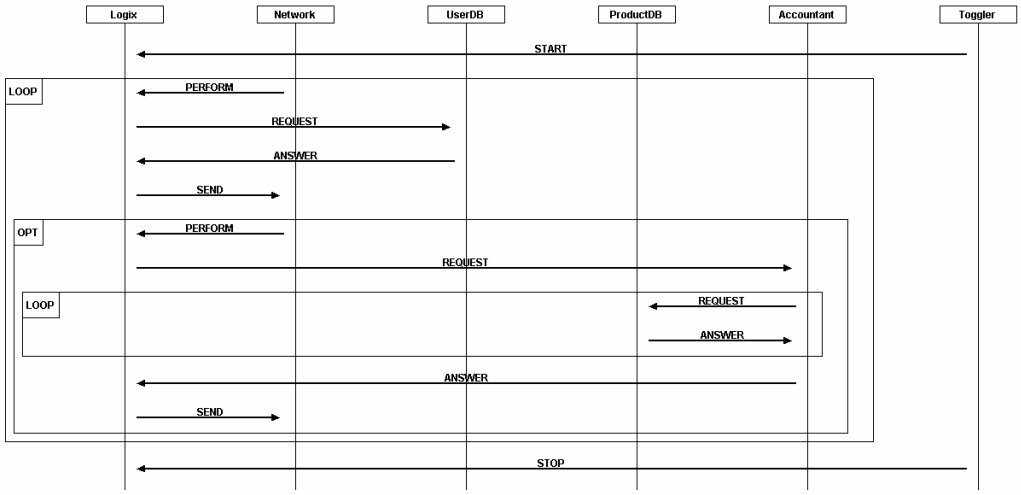

Figure 58: Generic eECommerceServer composition pattern.

The six roles of the ECommerceServer composition pattern have to be filled by six components. These components are two SimpleDatabase components, an Accountancy component, a JButton component, an UDPNetwork component and a PhotoLab component. The JButton and UDPNetwork components are already discussed in the previous section. The SimpleDatabase component is as the name already suggests a primitive database component. This component persistently stores its information using the file system and supports a simple query syntax. Figure 59 illustrates a usage scenario of the SimpleDatabase component. This usage scenario documents that the SimpleDatabase component either expects a query using the REQUEST message implemented by the *requestData* method or expects a SET message implemented by the *setData* method in order to alter the database contents. After receiving a REQUEST message, the component sends an ANSWER implemented by the *dataResult* method.

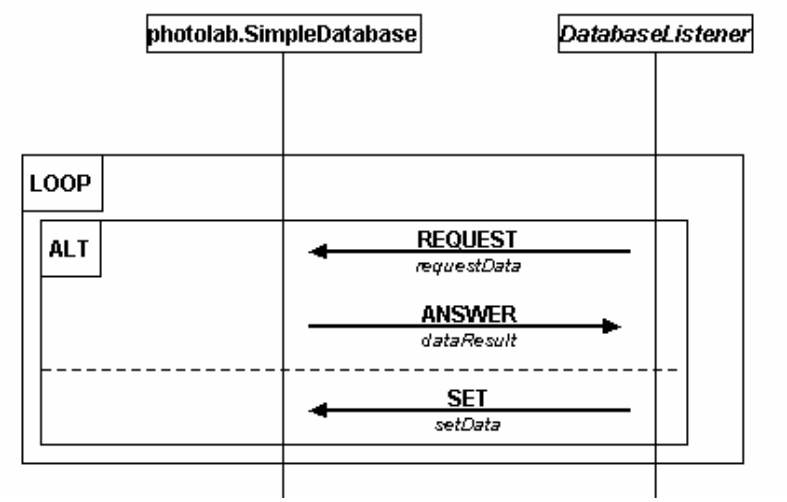

Figure 59: Usage scenario of the SimpleDatabase component.

The Accountancy component is responsible for computing the total price of an order and to bill the user. Figure 60 illustrates a usage scenario of the Accountancy component. The component first expects a REQUEST message implemented by the *checkout* method containing the order of the user. Afterwards, the component accumulates the total price of an order by querying a product database. This is done by continuously sending a REQUEST message implemented by the *getPrice* event to the *ProductDB* environment participant. Afterwards, the *ProductDB* submits the result of the query by sending an ANSWER message implemented by the *submitPrice* event. When the total price is computed, the Accountancy component bills the user and sends a receipt to the *accountancyListener* environment participant by sending an ANSWER message implemented by the *checkedOut* event. Subsequently, the component is ready to receive another request.

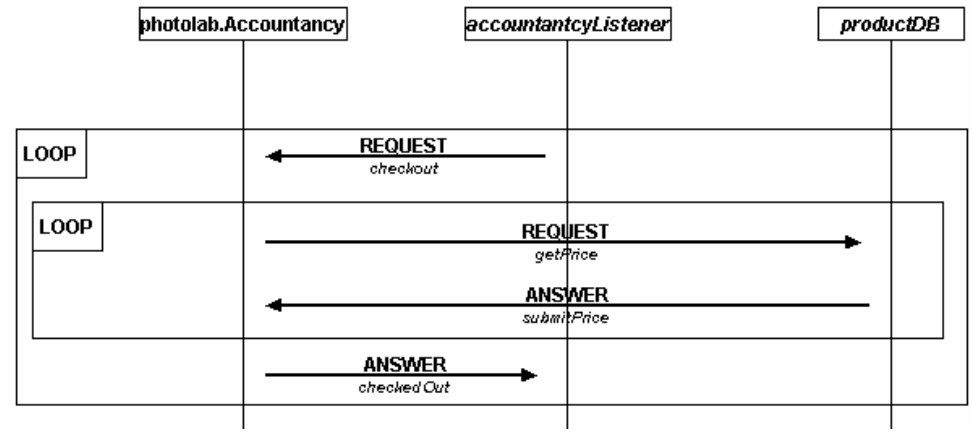

Figure 60: Usage scenario of the Accountancy component.

The final component involved in the server application is the PhotoLab component. This component communicates with the client application through the network and as such drives the collaboration of the server application. A usage scenario of this component is illustrated by Figure 61. This usage scenario specifies that the PhotoLab component first expects a START message implemented by the *startServer* method. Afterwards, the PhotoLab component is ready to communicate with the client application. The first message the component expects is a PERFORM message implemented by the login method containing the login and password of a user. The PhotoLab component validates the password by querying

the product database. This is achieved by sending a REQUEST message implemented by the *getPassword* event. The *ProductDB* role returns the resulting password by sending an ANSWER message implemented by the *validateCustomer* method. The result is then sent to the client application. When the password is not valid, the PhotoLab component expects another login attempt by sending the PERFORM message implemented by the *login* method. Otherwise, the PhotoLab component expects a PERFORM message implemented by the *buyProducts* method containing the order of the user. Subsequently, the PhotoLab component sends a REQUEST implemented by the *calcPrice* event to the *Accountant* environment participant for computing the total price of the order. Afterwards, the PhotoLab component expects an ANSWER message implemented by the *submitReport* method containing the resulting receipt. The receipt is then sent to the client application by sending a SEND message implemented by the *communicate* event through the *Network* environment participant. The PhotoLab component is now ready to receive another login request or a STOP message implemented by *stopServer* in order to stop the application.

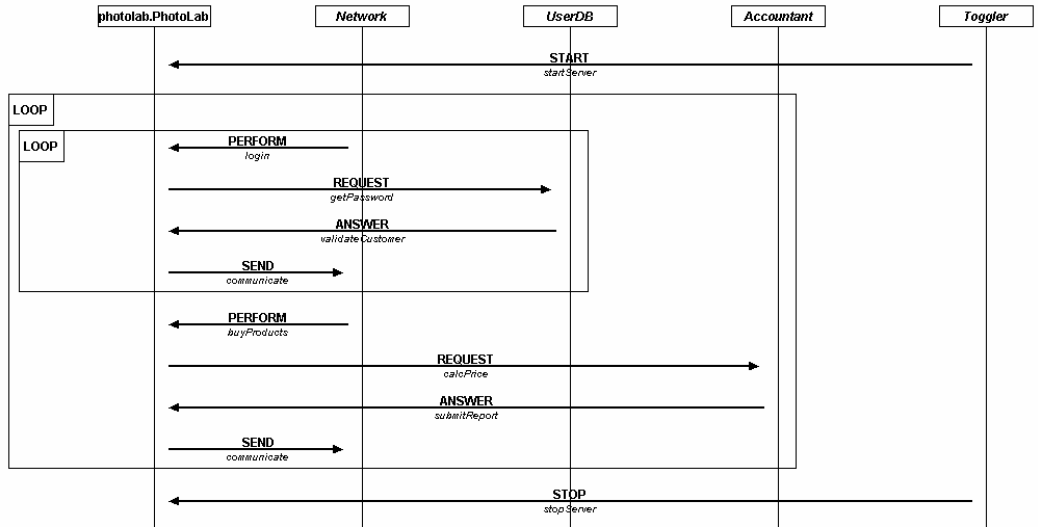

Figure 61: Usage scenario of the PhotoLab component.

The ECommerceServer composition pattern roles can now be filled with the six components presented above as illustrated by Figure 62: the Toggler role is filled by the JButton component, the PhotoLab component is mapped onto the Logix role, the UDPNetwork component is mapped upon the Network role, the UserDB and ProductDB roles are each filled by a separate SimpleDatabase component and the Accountant role is filled by the Accountancy component. Glue-code is generated automatically from this complete composition and as such the server application is visually wired.

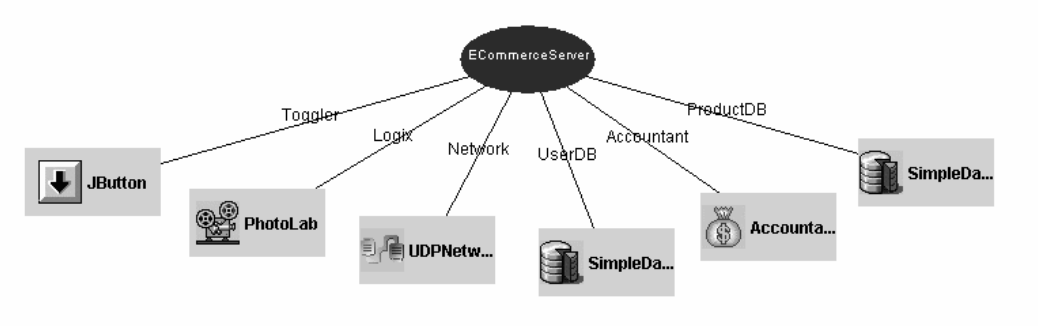

Figure 62:ECommerceServer composition wired in the PacoWire application.

Combining the client and server application renders the complete PhotoLab application. The PhotoLab application only contains its basic functionality. Several additional concerns that need to be integrated into this application can not be nicely separated into a single component. The next sections aim at modularizing these crosscutting concerns by (invasive) composition adapters.

7.2 Timing the PhotoRenderer component

In this section, the InvasiveTimer invasive composition adapter introduced in Chapter 5 is applied to the PhotoLab application in order to illustrate the reusability of composition adapters. Remember that the InvasiveTimer invasive composition adapter modularizes the timestamping concern by invasively inserting timestamp logic in the target components. The InvasiveTimer invasive composition adapter is used in the PhotoLab client application for timing the PhotoRenderer component. The PhotoRenderer component allows verifying the quality of a given picture. This is however a complex process and might require a lot of time. In order to verify whether the component performs as promised by the vendor, the communication from and to this component is timed by applying the InvasiveTimer composition adapter.

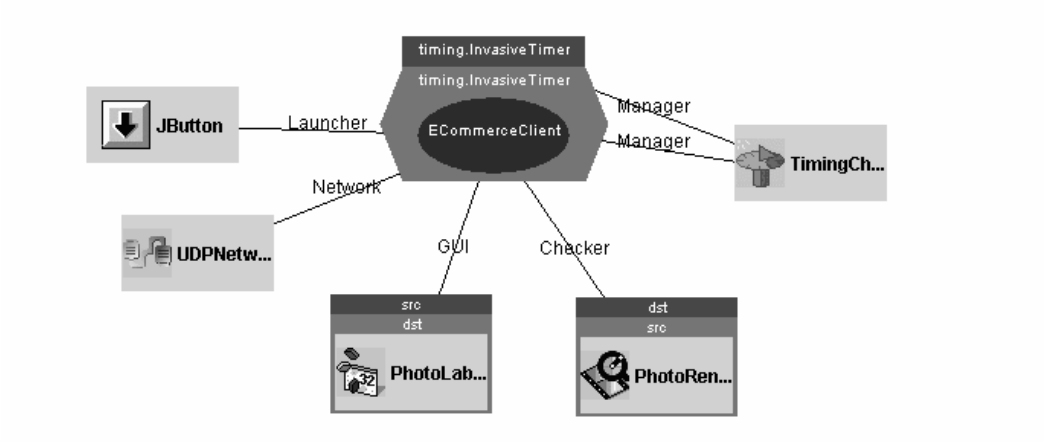

Figure 63: InvasiveTimer invasive composition adapter applied to the ECommerceClient composition pattern in order to timestamp the communication to and from the PhotoRenderer component.

Figure 63 illustrates a screenshot of PacoWire where the InvasiveTimer invasive composition adapter is applied to the ECommerceClient composition pattern. The PhotoRenderer component only communicates with the PhotoLabGUI component. As such, it suffices to apply two InvasiveTimer invasive composition adapters. The *src* and *dst* context part roles of the first InvasiveTimer instance is applied respectively on the PhotoLabGUI and the PhotoRenderer components. As a result, the communication from the PhotoLabGUI component to the PhotoRenderer component is timed. In order to time the communication in the other direction, namely from the PhotoRenderer to the PhotoLabGUI component, the *src* and *dst* context part roles of the second InvasiveTimer instance are mapped upon respectively the PhotoRenderer and the PhotoLabGUI components. The result is that a connector is generated for each InvasiveTimer instance that applies the TimeStamp hook on the concrete methods and events. Code Fragment 66 illustrates the connector that is generated for the second InvasiveTimer invasive composition adapter that times all communication from the PhotoRenderer component to the PhotoLabGUI component. This communication consists of only one ANSWER message (see Figure 53) that is implemented by the *buyProducts* method in the PhotoLabGUI component and by the *notifyValidateResult* event of the PhotoRenderer component. As such, the connector instantiates the TimeStamp hook on both the *buyProducts* method and *notifyValidateResult* event. The result is that the timestamping logic is inserted at both joinpoints.

```
1   connector InvasiveTimer$914Connector {
2     timing.InvasiveTimer.TimingHook hook1 = new
3       timing.InvasiveTimer.TimingHook({
4         void photolab.PhotoLabGUI.buyProducts(java.lang.String),
5         onevent photolab.PhotoRenderer.notifyValidateResult
6            (java.lang.String)
7       }
8     );
9
10    hook1.default();
11  }
```

Code Fragment 66: One of the two connectors generated by PacoWire from the composition of Figure 63.

Notice that the connector is declared non-static, this means that the connector only triggers on instances that have been explicitly registered with the connector. Several instances of the same component type might be used in the composition. Some of those instances might be mapped on the invasive composition adapter context part roles while others are not. As such, the TimeStamp hook can be only triggered on those component instances that have been mapped upon context part roles of the InvasiveTimer invasive composition adapter. The glue-code is responsible for explicitly registering all component instances that are mapped upon the context part roles with the correct connectors. The glue-code generated from the composition of Figure 63 is depicted in Appendix C. In addition, Appendix C also illustrates the connector generated for the other InvasiveTimer instance.

7.3 Encryption aspect

The following two crosscutting concerns we identified are in fact security concerns. Several security related concerns have already been recognized in literature as good candidates for aspects [VDD01]. Typical examples of such crosscutting security concerns are access control and confidentiality. The confidentiality concern is also applicable to the PhotoLab case study as the user has to supply a login and password over the network to the server. Currently, all information (including login information) is sent over the network in plain ASCII. When this network is the World Wide Web, it is highly desirable that the sensitive information is encrypted. However, the encryption concern is not the main functionality of the PhotoLab application and integrating this concern within the PhotoLab application requires tangling this concern with the main concerns of the application. In addition, the encryption concern involves several participants of the PhotoLab application. As such, this concern has to be spread and duplicated among these participants in order to realize encryption. The overhead of encryption might for example not be required when the application works over a private network connection. When the concern is spread over several participants, it is very difficult to edit or remove this concern.

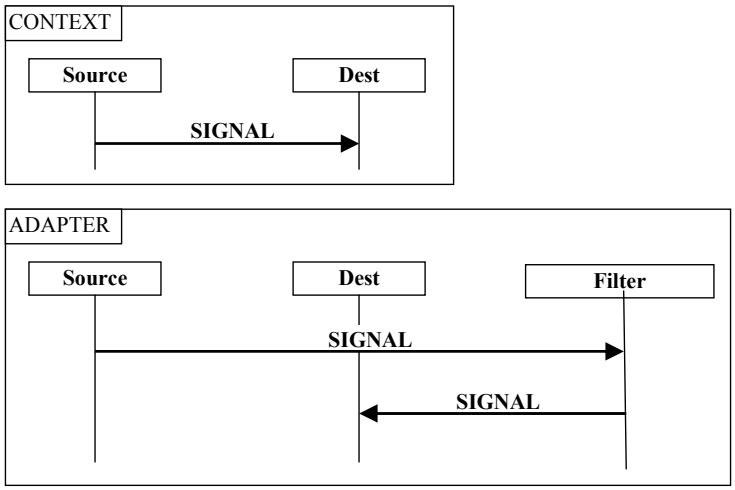

Figure 64: Generic re-routing composition adapter that can be used for modularizing the encryption concern.

The encryption concern can in fact be easily modularized by a regular composition adapter that re-routes all data sent to and from the network through an encryption component. An invasive composition adapter could also be employed, but is not really necessary in this case as the internal logic of the components themselves does not need to be adapted. Of course, when the communication channel between the components of the client or server application is already unsafe because for example a certain insecure operating system is used, an invasive composition adapter is required in order to insert the encryption logic into the components themselves. Otherwise, a generic re-routing composition adapter already performs the job.

Figure 64 illustrates the regular composition adapter that implements a generic re-routing logic. This composition adapter can be used to realize the encryption concern by applying the composition adapter four times as illustrated by Figure 65. One instance is applied at the client application in order to re-route the communication from the PhotoLabGUI component to the Network component through an encryption component. This can be done by mapping the *Source* role onto the *GUI* role and the *Dest* role onto the

Network role. Afterwards, the PhotoLabGUI component is mapped onto the combined *Source/GUI* role. The UDPNetwork component is mapped onto the combined *Dest/Network* role. The Filter role is filled by an additional encryption component. As such, the messages sent from the PhotoLabGUI component are re-routed through the Encryption component. The Encryption component then takes care of encrypting the data. In order to decrypt the communication from the Network component to the PhotoLabGUI component, another instance of this composition adapter is inserted. This is done by applying the rerouting composition adapter the other way around, namely by mapping the *Source* and *Dest* roles onto respectively the *Network* and *GUI* roles. The PhotoLabGUI and UDPNetwork components are again mapped onto their appropriate roles and the Filter role is then filled by a decryption component. Likewise, two instances of this composition adapter are applied at the ECommerceServer composition pattern in order to decrypt and encrypt communication from and to the network role.

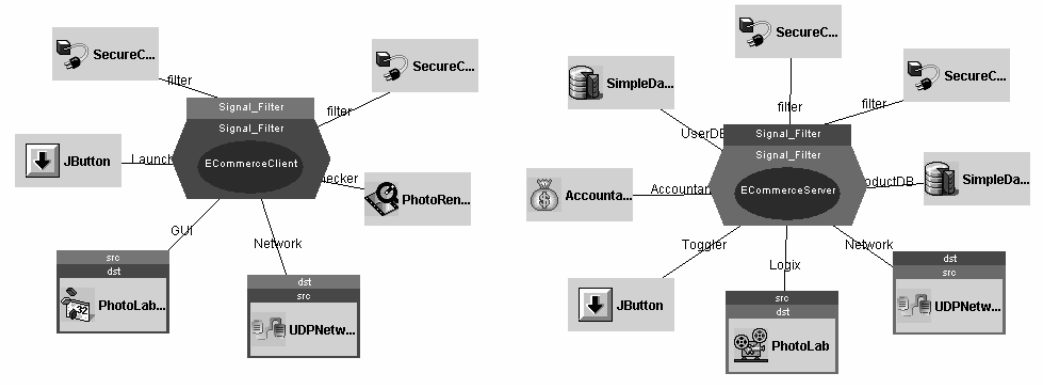

Figure 65: Applying the SignalFilter composition adapter of Figure 64 onto the ECommerceServer and ECommerceClient composition patterns in order to encrypt and decrypt communication over the network.

Instead of a re-routing solution, an invasive composition adapter allows encrypting and decrypting data inside the components before the data is sent to the network or received by the network. The invasive composition adapter that realizes the encryption concern is depicted in Figure 66. This invasive composition adapter specifies in its context part a single SIGNAL primitive between certain *Safe* and *UnSafe* roles. All messages sent from the *Safe* role that match this SIGNAL primitive are used to instantiate the Encryption hook upon. The adapter part specifies that the SIGNAL is still sent in the same way. The Encryption hook implements a replace behavior method that is responsible for encrypting the data prior to firing the replaced event. As such, the component mapped upon the *Safe* role is adapted so that all the events fired by the component contain encrypted data.

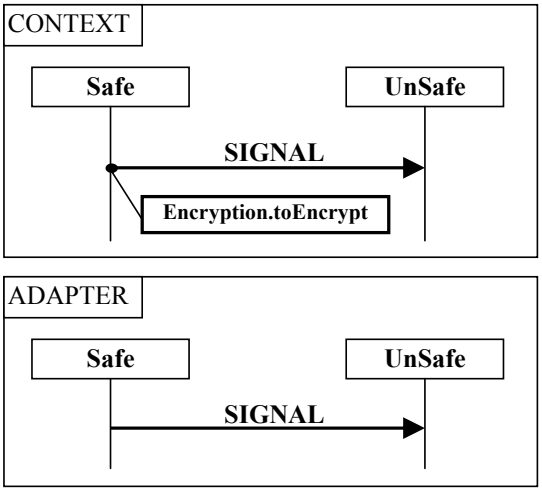

Figure 66: Encryption invasive composition adapter.

This invasive composition adapter is applied to the ECommerceClient composition pattern by mapping the *Safe* and *UnSafe* roles respectively onto the *GUI* and *Network* roles. As such, all events fired by the PhotoLabGUI component, mapped onto the *GUI* role, to the UDPNetwork component, mapped onto the *Network* role, contain encrypted data. The invasive composition adapter is also applied at the ECommerceServer composition pattern by mapping the *Safe* and *UnSafe* roles respectively onto the *Logix* and *Network* roles. As such, all events fired by the PhotoLab component, mapped onto the *Logix* role, and are received by the UDPNetwork component, mapped onto the *Network* role, contain encrypted data.

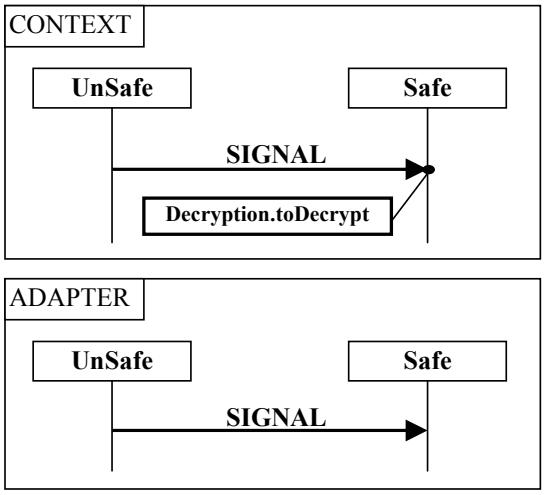

Figure 67: Decryption invasive composition adapter.

Of course, in order to keep the application running, the corresponding components also need to be adapted to decrypt data received from a network component. Figure 67 illustrates the decryption invasive composition adapter. This invasive composition adapter specifies a single SIGNAL primitive in its context part. The messages matching the SIGNAL primitive received by the component mapped onto the *Safe* role are used to instantiate the Decryption hook upon. As such, these messages are adapted to decrypt the data received prior to executing the original behavior. The adapter part specifies that the SIGNAL is still sent in the same way because the external protocol of the component mapped upon the *Safe* role is unchanged.

The application of this invasive composition adapter onto the PhotoLab application is illustrated in Figure 68. This invasive composition adapter can be applied onto the ECommerceClient composition pattern in

order to decrypt the messages sent from the Network role. To this end, the *UnSafe* and *Safe* roles have to be mapped respectively onto the *Network* and *GUI* roles. Afterwards, the PhotoLabGUI component can be mapped onto the combined *Safe/GUI* role. Likewise, the UDPNetwork component is mapped onto the *UnSafe/Network* role. As a result, a connector is generated that instantiates the Decryption hook onto all methods invoked onto the PhotoLabGUI component by the UDPNetwork component. The decryption behavior is thus inserted at those methods and the data is decrypted prior to executing the original behavior. Likewise, the decryption invasive composition adapter also has to be applied upon the ECommerceServer composition pattern in order to decrypt the data received by the PhotoLab component from the UDPNetwork component.

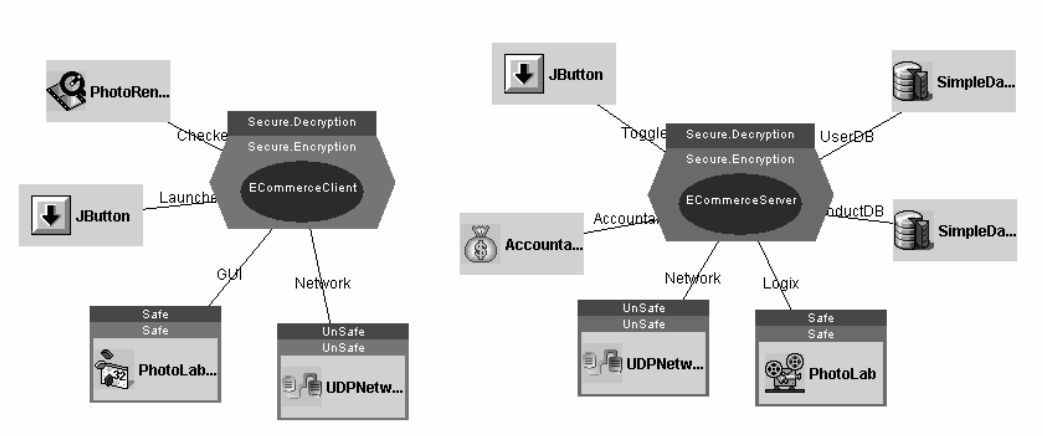

Figure 68: PhotoLab client and server composition patterns enhanced with the encryption and decryption invasive composition adapters.

Adapting the encryption/decryption algorithm becomes very easy because the encryption/decryption logic is nicely modularized by an (invasive) composition adapter. Likewise, deleting the encryption/decryption logic is as simple as deleting the involved (invasive) composition adapters. The encryption (invasive) composition adapter solution is also perfectly reusable in other applications because of the abstract protocol specification. The same encryption and decryption invasive composition adapters are for example also used in an internet chat application.

Notice that the encryption aspect has to be applied on both the client and server application in order to be useful. Indeed, only applying the encryption aspect onto the server application causes encrypted data to be sent to the client application that does not know how to cope with this. PacoSuite is not able to check whether the combined behavior of the server and client application makes sense, which is a drawback of the current approach. This is because there is communication between components among different compositions that bypasses the glue-code through the network. No protocol description exists that describes this protocol between the components. Therefore, the protocol is “invisible” for PacoSuite. This issue could be solved by introducing an extra concept for describing this kind of invisible protocol.

7.4 SecureLogin aspect

Another security concern that needs to be integrated into the PhotoLab application are anti-hacking measures. Encrypting the login and password information is a first step in securing a network-based application from malicious users. However, it is still perfectly possible to obtain the password of a particular user in a brute-force way by iterating over all possible passwords. The regular PhotoLab application does not deal with false login attempts. A good security measurement consists of blocking user accounts after *x* number of false attempts for a certain period of time. This logic is however not the core functionality of the application and the overhead of the protection might not always be desirable. In addition, the secure login concern is tangled with the Logix role of ECommerceServer composition pattern. Indeed, the component mapped onto the Logix role needs to be able to decide when a user becomes blocked and send the appropriate information back to the client application.

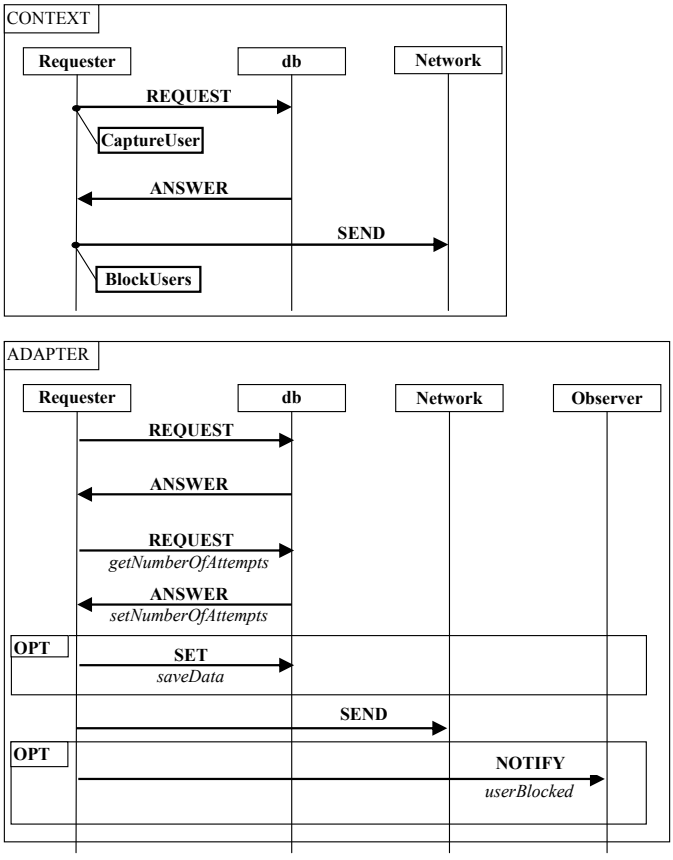

Figure 69: SecureLogin Invasive Composition adapter that realizes blocking after a predefined number of false login attempts.

In order to modularize the secure login concern, we model this concern as an invasive composition adapter. This allows us to reuse this concern in different applications without hard-coding and scattering this logic among the involved components or composition patterns. The SecureLogin invasive composition adapter is depicted in Figure 69. The invasive composition adapter is implemented by an aspect bean which contains two hooks: *captureUser* and *blockUsers*. The *captureUser* hook is responsible for capturing the user at hand. The *blockUsers* hook takes care of implementing the actual blocking decision.

In addition, the *blockUsers* hook alters the message returned to the client application to a message stating that the user is blocked if necessary. The full implementation of the aspect bean is depicted in Appendix E.

The context part of the invasive composition adapter of Figure 69 specifies that it is applicable on a REQUEST-ANSWER-SEND protocol. The *captureUser* hook is instantiated on the source of the REQUEST message and the *blockUsers* hook on the source of the SEND message. The adapter part specifies a quite drastic change of this protocol. After the first REQUEST-ANSWER, an additional REQUEST-ANSWER is sent by the adaptations described in the *blockUsers* hook. This is required for querying the UserDB about the number of false logins attempted by the user that has been captured by the *captureUser* hook. When the current attempt is a false attempt, this number is updated by invoking a SET message on the UserDB. When the maximum number of false attempts has been reached for the current user, a message containing a blocking notice is sent to the client application by sending a SEND message to the Network role. Otherwise, the normal login result is returned by sending the SEND message to the Network role. When a user has been blocked, an additional event is fired by the Requester role in order to inform interested observers. This event can be captured by a security user interface so that a human security manager might decide to unblock this account in case of a mistake or to take further actions in case a hacking attempt is detected. It is however also possible to just map a statistics component or an artificial intelligence security agent component on this role as long as they adhere to the specified abstract protocol. Notice that the messages that have been added by the aspect bean implementation contain implementation mappings. This is required for being able to adapt the usage scenarios of the components at hand in order to be consistent with the newly introduced protocol.

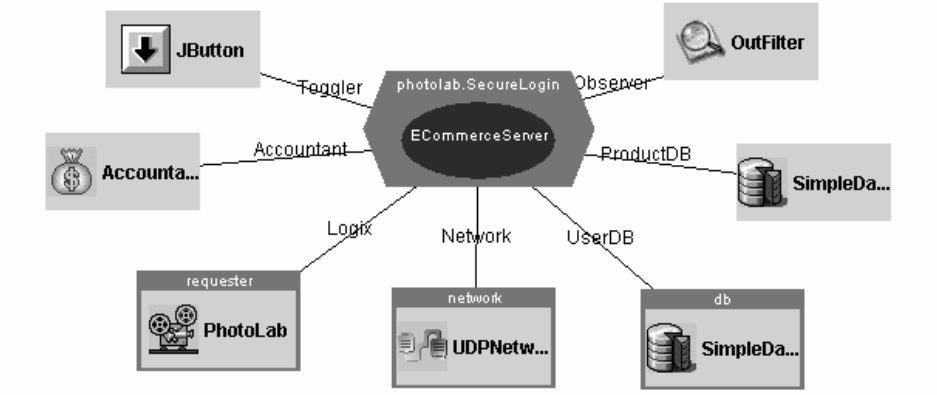

Figure 70: Applying the SecureLogin invasive composition adapter onto the ECommerceServer composition pattern.

Figure 70 illustrates how the SecureLogin invasive composition adapter is applied onto the ECommerceServer composition pattern. The *requester* role is mapped onto the *Logix* role, the *network* role is mapped onto the *network* role and the *db* role is mapped onto the *UserDB* role. As such, the invasive composition adapter logic is inserted into the PhotoLab component and SimpleDatabase components in order to implement the blocking strategy. Removing the blocking strategy is as simple as deleting the invasive composition adapter from the composition.

Currently, the SecureLogin aspect bean implements a rather simple approach for deciding when to block a user. A user is blocked after a predefined number of false attempts accumulated over the entire usage history of that user. This is not always the desired behavior as everybody mistypes her/his password once in a while. Therefore, a more intelligent blocking decision approach has to be implemented that only counts the false attempts made in for example the last ten minutes. Furthermore, in order to decrease the

administration hassle of manually unblocking users, a more intelligent blocking strategy is required that for example only blocks a user for a couple of minutes. This waiting time could gradually be increased for each additional blocking for that user in a certain time period. Implementing both an enhanced blocking decision and blocking strategy is however very easy to perform because the SecureLogin aspect is cleanly modularized. As such, the only module that has to be adapted is the aspect bean implementation of the invasive composition adapter. In contrary, adapting the blocking decision and blocking strategy is a cumbersome task when the SecureLogin aspect is scattered over the different involved components. Likewise, deleting the SecureLogin aspect from the composition involves manually adapting several components and the composition pattern when the aspect is not nicely encapsulated by an invasive composition adapter.

7.5 Obsolete product discount aspect

The following concern that cannot be easily modularized in the PhotoLab application consists of a discount for obsolete products. In this case, the obsolete product is a photo paper format that is no longer in use. The obsolete product discount is in fact an example of a business rule. A business rule is a statement that defines or constraints some aspect of the business. It is intended to assert business structure or to control the behavior of the business [Ros97]. A business rule often applies to different concrete places in the target application. As such, the code for triggering the business rule and for coping with the result of the business rule has to be duplicated and spread over these places. In addition, both the business rule logic and the places in the application where the business rule applies to tend to change a lot. Business rules have already been identified as crosscutting concerns in literature. Ossher et al. [OT01] modularize business rules as hyperslices. Cibrán et al. [CDJ03] argue that aspect-oriented programming is an ideal paradigm for modularizing a business rule linking layer.

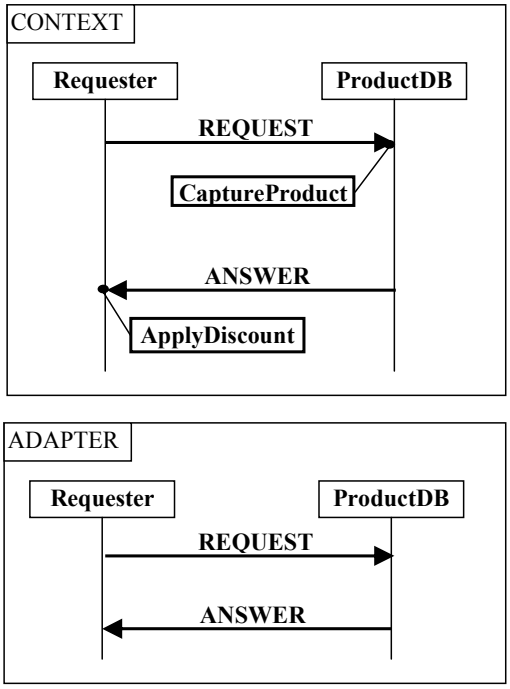

Figure 71: Obsolete product discount invasive composition adapter.

In the photolab context, the obsolete product discount is in fact tangled with the Accountancy component because this component has to be able to cope with obsolete products and compute a discount for obsolete products. In addition, the obsolete product discount logic is spread among two participants of the ECommerceServer composition pattern, namely the Accountancy and ProductDB roles. Furthermore, it has to be possible to flexibly insert, adapt and delete this discount. Therefore, the photolab discount is modularized as an invasive composition adapter. Figure 71 illustrates the obsolete product discount invasive composition adapter. This composition adapter is implemented by an aspect bean which declares two hooks: *CaptureProduct* and *ApplyDiscount*. The implementation of the aspect bean is depicted in Appendix D. The context part declares that this invasive composition adapter is applicable on a consecutive REQUEST and ANSWER. Notice that a different hook is mapped on both the primitives of the context part. The *CaptureProduct* hook is responsible for capturing all relevant information of the price request of a certain product. In addition, the *CaptureProduct* hook is responsible for verifying whether the

current product is obsolete or not. As such, this hook decides whether the discount is applicable or not. The *ApplyDiscount* acts on the answer of the request and changes the result if the product is considered obsolete. The adapter part of the invasive composition adapter declares that the REQUEST and ANSWER are sent in the same way as before. Indeed, this invasive composition adapter only changes the interior behavior of the component that is mapped onto the Requester role.

Figure 72 illustrates the application of the obsolete product discount invasive composition adapter onto the ECommerceServer composition pattern. The *Requester* and *ProductDB* roles of the invasive composition adapter are mapped respectively onto the *Logix* and *ProductDB* roles of the ECommerceServer composition pattern. Afterwards the Accountancy and SimpleDatabase components have been mapped onto these combined roles. As a consequence, the behavior of the accountancy component is altered in order to compute a discount for obsolete products. The SimpleDatabase component instance that is mapped onto the *ProducDB* role is altered in order to cope with obsolete product information. Notice that the second SimpleDatabase component instance which is mapped onto the *UserDB* role is not affected by the aspect bean implementation of the invasive composition adapter. This is because the JAsCo language allows specifying aspects on instance level by employing non-static connectors. As such, the hooks are triggered on only those instances of a certain component type that are explicitly registered with the connector.

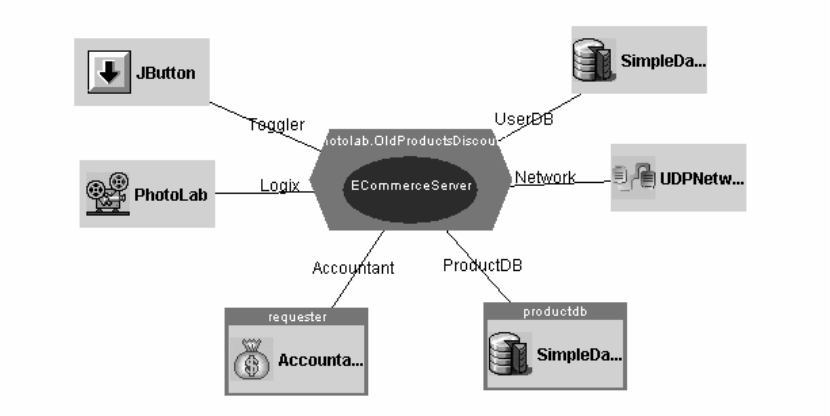

Figure 72: Obsolete product discount invasive composition adapter applied to the ECommerceServer composition.

Using an invasive composition adapter implemented by a JAsCo aspect bean to modularize the obsolete product discount business rule, allows to easily change the rule if necessary. The business management might for instance decide that all users that buy an obsolete product receive a small discount over their total bill. Coping with these changed requirements is very easy as the obsolete product discount logic is modularized into one aspect bean. Instead of altering the returned price for an obsolete product, the total price is changed when an obsolete product is bought. In addition, the obsolete product discount can be easily removed when the obsolete product is sold out.

7.6 Inserting all aspects

The previous sections introduce several concerns of the PhotoLab application that are successfully modularized using invasive composition adapters implemented by aspect beans. The invasive composition adapters are however always applied in isolation. It is of course also possible to apply all aspects at the same time by stacking the invasive composition adapters. This is illustrated in Figure 73. Notice that stacking several (invasive) composition adapters onto the same composition pattern does not always lead to a valid composition even though all the composition adapters are valid in isolation. This is because composition adapters might conflict. One composition adapter might for example remove or alter the protocol required for another composition adapter to be applicable. As such, even when these composition adapters are valid in isolation, combining them results in an invalid composition. The PacoSuite tool is however able to detect this by employing the composition adapter application algorithm defined in 3.3. Indeed, the first step of this algorithm fails when the context part of a composition adapter is not available any longer. As such, the component composer is notified of composition adapter conflicts.

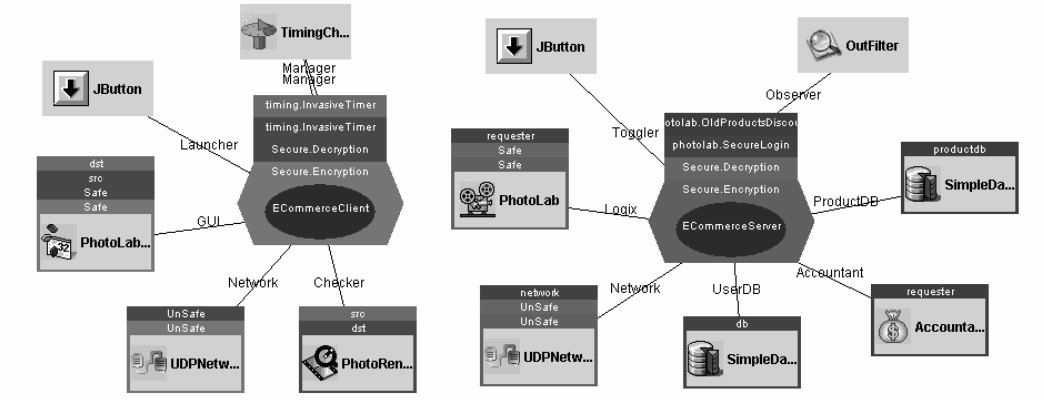

Figure 73: Stacking the invasiveTimer, encryption/decryption, obsolete product discount and SecureLogin invasive composition adapters onto the PhotoLab application.

Chapter 8 Conclusions

This chapter presents the conclusions of the dissertation. The first section summarizes the approach elucidated in the dissertation. Afterwards, the main contributions of the dissertation are recapitulated. Section 0 evaluates the applicability of the approach and finally section 0 discusses the limitations and future work.

8.1 Summary of the dissertation

The goal of this dissertation consists of integrating aspect-oriented and component-based software engineering and as such combining the advantages of both aspect-oriented and component-based ideas. The context in which this research is conducted consists of the PacoSuite component-based methodology developed in Bart Wydaeghe's PhD [Wyd01]. We present a composition adapter in order to modularize crosscutting concerns in PacoSuite. A composition adapter describes transformations of a composition pattern in an abstract way independent of a specific component API. We introduce an algorithm that allows to verify whether a composition adapter is applicable onto a given composition pattern. Furthermore, the adaptations described by the composition adapter can be automatically inserted into the composition pattern. Composition adapters are an ideal means for describing protocol transformations. Aspects that require other adaptations, like for example internal adaptations to the components, can however not be represented by a composition adapter. Therefore, the JAsCo aspect-oriented programming language tailored for component-based software engineering is introduced.

JAsCo is an extension of the Java language and introduces two main additional concepts, namely aspect beans and connectors. Aspect beans are based on regular Java beans and allow describing crosscutting behavior in a special kind of inner class, called a hook. Hooks specify the context on which they are applicable in an abstract way. The concrete deployment information for hooks is specified in a connector by instantiating the hook onto a specific context. Connectors are also able to declare explicit precedence and combination strategies in order to manage the cooperation of several aspect beans and components. These precedence and combination strategies are a partial solution to the so-called feature interaction problem [PSC+01], as they allow preventing possible conflicts. JAsCo also supports aspectual polymorphism [MO03] by employing abstract methods that are implemented in a connector. The JAsCo technology introduces a new component model where traps that allow dynamic aspect attachment and removal are already built-in. Furthermore, both aspect beans and connectors remain first class entities at run-time. We also propose an extension of the JAsCo language that recuperates ideas from Adaptive Programming. JAsCo aspect beans are able to represent both traditional aspects and adaptive visitors, which increases the reusability and versatility of aspect beans even more. We show that the JAsCo contributions concerning reusable aspects and an explicit aspect composition mechanism are also applicable to adaptive programming.

JAsCo aspect beans are represented in the PacoSuite approach by employing an enhanced version of a composition adapter, called an invasive composition adapter. Invasive composition adapters document the protocol transformations caused by the adaptations of the aspect bean implementation. Likewise to a regular composition adapter, an invasive composition adapter can be applied onto a given component composition. The protocol changes described by the invasive composition adapter are automatically inserted into the component's usage scenarios and composition pattern. Furthermore, a JAsCo connector is generated automatically that instantiates the hooks contained in the aspect bean onto the correct component context. It is still possible to automatically generate glue-code and check compatibility of adapted components with the composition pattern. As such, an invasive composition adapter is able to realize all the benefits of a regular composition adapter with the added expressiveness of the JAsCo implementation.

As a proof of concept of the approach, an extension of the PacoSuite visual component composition environment has been developed. This extended tool suite allows to visually apply a regular or an invasive composition adapter onto a given component context using the algorithms presented in this dissertation. The JAsCo language is also implemented by a set of tools, compilers and a run-time environment. The

current JAsCo implementation (0.4.x) supports almost all the features presented in the dissertation. Using the tool support, the ideas presented by this dissertation are applied onto a distributed e-commerce application for printing digital pictures. We identify several crosscutting and tangled concerns in this application and successfully modularize these concerns as invasive composition adapters.

8.2 Summary of the contributions

- We present a composition adapter that is an ideal means for describing aspects that require basic protocol transformations. Composition adapters allow specifying a novel triggering condition based upon the protocol history. We propose an algorithm that allows to automatically match a composition adapter with a given composition pattern. Furthermore, we propose an algorithm that allows to automatically insert the adaptations specified by a composition adapter.
- We present an invasive composition adapter implemented by a JAsCo aspect bean in order to combine both protocol adaptations and invasive adaptations to the components. We present an algorithm that allows to automatically apply an invasive composition adapter onto a given component composition.
- We implement support for both regular and invasive composition adapters in PacoSuite and perform a case study to evaluate the approach.
- We present the JAsCo AOP language aimed at component-based software engineering. JAsCo allows specifying abstract and reusable aspect beans that are instantiated onto a concrete context in a connector.
- We introduce explicit precedence and combination strategies that are able to control the collaboration among several cooperating aspect beans.
- We present an Adaptive Programming extension of JAsCo that allows aspect beans to be instantiated as adaptive visitors in traversal connectors. Traversal connectors are also able to specify explicit precedence and combination strategies that control the collaboration among several cooperating adaptive visitors represented as aspect beans.
- The JAsCo technology allows aspect beans and connectors to remain first class entities at run-time and enables dynamic aspect addition and removal. Furthermore, we propose the Jutta dynamic AOP just-in-time compiler that makes JAsCo-Jutta the most optimal dynamic AOP system performance-wise.

8.3 Applicability

Composition adapters are employed successfully in two large-scale research projects. In the SEESCOA project[10] (Software Engineering for Embedded Systems using a Component-Oriented Approach), an approach is developed for verifying Quality Of Service (QOS) guarantees of a component composition both statically and dynamically. Ideally, QOS guarantees can be verified statically using formal proofs. However, in most cases, the QOS guarantees have to validated dynamically by executing and testing the resulting application. In order to avoid hard-coding QOS verification checks into the components, composition adapters are employed that re-route all relevant messages to verification components. As such, this concern is nicely modularized and can be easily removed when the application goes into production.

Composition adapters are also employed in another research project, called ADAPSIS[11] (Adaptation of IP Services based on Profiles). In this project, one of the research topics consists of developing an approach for mining end-user profiles starting from existing component-based applications. The main problem is that these applications are generally not designed for allowing end-user profile mining. The profile mining concern is in fact a crosscutting concern and has been encapsulated by a composition adapter [WVPW02]. In addition, the component-based applications have to be adapted so that they are able to dynamically customize their behavior depending on the mined profile of the current end-user. This concern requires invasive changes to the components themselves in order to alter their behavior. Invasive composition adapters are employed to implement this concern effectively.

In the case study performed in this dissertation, JAsCo is only evaluated as an implementation mechanism for invasive composition adapters. The JAsCo language and ideas are however also applicable without the PacoSuite methodology. This dissertation does not explicitly validate the JAsCo aspect-oriented ideas outside the PacoSuite methodology. However, JAsCo is currently employed successfully in several research and industrial projects, which already validate the general applicability of JAsCo. JAsCo is for example used in a research line at SSEL where aspect-oriented programming is investigated for modularizing domain knowledge [CDJ03]. More concretely, aspect-oriented technologies are employed to improve the integration of business rules with object-oriented software. Current approaches that support business rules at the implementation level only separate the business rules themselves and not the code that links them to the core application. This code crosscuts the core application and aspect-oriented programming is an ideal means for linking business rules to a core application. In this research, different aspect-oriented technologies, like for instance AspectJ [KHH+00], HyperJ [OT00], SOUL/AOP [GB03] and JAsCo, are assessed with respect to the suitability for modularizing business rules. The JAsCo ideas concerning expressive combinations and prioritization between aspects prove to be a major improvement over other AOP approaches for modularizing business rule links [CDS+03,Lam03].

The Web Services Management Layer (WSML) is another convincing application of the features provided by the JAsCo approach. The Web Services Management Layer is developed in the context of the MOSAIC

---

[10] The SEESCOA project is funded by the IWT, Belgium. Main project partners are the universities of Leuven (KUL), Ghent (UGent), Limburg (LUC) and Brussels (VUB) and a user commission consisting of six Flemish-based companies.

[11] ADAPSIS is partly funded by the IWT, Flanders (Belgium), partners are the University of Brussels (VUB), Alcatel Belgium and Data4s Future Technologies.

project[12]. The Web Services Management Layer is motivated by the observation that current approaches for the integration of web services hard-wire their references into client applications, affecting adaptability and reusability. Moreover, no management support is provided, which is fundamental for achieving robustness [Szy01]. To solve these problems, the Web Services Management Layer [CV03, VC03] is proposed as an intermediate layer in between the application and the world of web services. The Web Services Management Layer allows decoupling web services from the core application, realizing the concept of *just-in-time integration* of services. As a result, multiple services or compositions of services can be used to provide the same functionality. In addition, the Web Services Management Layer supports service selection policies and client-side service management concerns. Martin et al. [MATH03] already identified the enormous potential benefits of aspect-orientation for Web Services. The Web Services Management Layer relies heavily on JAsCo AOP in order to avoid tangling the application code with service related management code. JAsCo aspect beans are for example employed to implement dynamic selection policies based on price, availability and performance [VCV+04]. JAsCo aspect beans also modularize client-side management concerns like monitoring, billing and transaction management. The main advantages of the JAsCo approach in the context of the Web Services Management Layer are the explicit and elaborate combinations that can be specified between interacting aspects. Indeed, because the WSML consists of a multitude of cooperating aspects, being able to manage their combined behavior is crucial. Furthermore, the dynamic property of JAsCo is exploited to achieve run-time service integration and selection.

There are also several other projects that aim at recuperating JAsCo ideas to other component models and platforms besides Java beans. For example, a JAsCo inspired AOP language targeted at the .NET platform is recently introduced [VVSJ03]. JAsCo.NET inherits the contributions of the original JAsCo approach, namely independent and reusable aspects beans and connectors that allow to explicitly manage aspect compositions. Similar to JAsCo, JAsCo.NET allows dynamic aspect application and removal. In addition, the JAsCo.NET language provides support for instantiating aspects on Web Service interactions. A first implementation of the JAsCo.NET language has been recently published. In addition to JAsCo.NET, a JAsCo run-time architecture (named JAsCoME) optimized for embedded devices is being developed in the context of the CoDAMoS[13] (Context-Driven Adaptation of Mobile Services) project. CoDAMoS is a strategic fundamental research project aimed at solving a set of key challenges in the area of Ambient Intelligence, where personal devices will form an extension of each user's environment, running mobile services adapted to the user and his context. In order to achieve this dynamic adaptiveness, the CoDAMoS project plans at employing JAsCoME dynamic AOP. Another research line in the CoDAMoS project aims for generating blueprints of components out of a conceptual design. The development and composition of components at a conceptual level has the advantage that no early commitment is made to a specific architecture and/or technology for implementation and the components are maximally configurable at the moment of their application. A modeling framework, called CoCompose [Wag03,WJ03], is proposed, which employs Model Driven Development (MDD) [MM03] in order to allow step-wise refinement from a

[12] MOSAIC is partly funded by the IWT, Flanders (Belgium), partners are the University of Brussels (VUB) and Alcatel Belgium.

[13] The CoDAMoS project is funded by the IWT, Belgium. Main project partners are the universities of Leuven (KUL), Ghent (UGent), Limburg (LUC) and Brussels (VUB) and a user commission consisting of nineteen Flemish-based companies.

high-level design to the lowest level. In recent experiments, JAsCo is integrated within CoCompose to be able to refine CoCompose conceptual designs to JAsCo entities.

8.4 Limitations and Future Work

8.4.1 Continuations

One of the possible problems with the composition adapter approach consists of the performance of the composition adapter application algorithm. The worst-case complexity of this algorithm is of exponential nature and might lead to state explosions. Attie and Lorenz have recently proposed an approach for verifying model-based composition without leading to state explosions [AL03]. They explicitly acknowledge that this approach allows improving the efficiency of our algorithm for verifying compatibility of a certain component with a role of a composition pattern. Whether this approach is also applicable to improve the efficiency of the composition adapter application algorithm, remains to be investigated.

Another limitation consists of the domain dependency of the composition adapter specification. This is due to the PacoSuite primitives employed to document the messages sent between the various roles. It is possible to agree on such a set of semantic primitives for a limited application domain. However, it is unfeasible to come up and agree on a general set of semantic primitives. To avoid the domain dependency, one could just forget about the semantic primitives and only document components with their API. This is of course at the expense of the reusability and genericity of the composition adapter model. The composition adapter contributions concerning an explicit protocol history condition and automatic protocol transformations are however still valid. As such, the composition adapter model is also applicable outside the context of PacoSuite.

It is possible to stack several composition adapters onto the same composition pattern. This might however lead to conflicts because for example a composition adapter might remove the context required for another composition adapter. These conflicts can be detected using the composition adapter application algorithm. However, it would be interesting to allow an automatic resolution of conflicts. For example, when changing the precedence of two conflicting composition adapters, the end result might be valid. It is a major contribution for solving the so-called feature interaction problem when we are able to deduce these solutions from a set of conflicting composition adapters. The PacoWire tool includes a preliminary implementation of such an automatic conflict detection approach which is proposed in [VSJ03b]. A complete and sound approach is however subject to further research.

The JAsCo AOP language is a dynamic AOP approach and is as such subject to a run-time performance overhead. In section 6.1.3.2, an improvement of the JAsCo run-time infrastructure is proposed through *just-in-time combined aspect compilation*. The idea is that the compiled combined behavior of all aspects is cached for every trap or groups of similar traps in the system. We already implemented a first version of this approach and illustrated that a huge performance gain is realized. A full implementation of this idea requires further research.

The JAsCo approach deliberately restricts the set of joinpoints on which JAsCo is able to interfere to the public interface of a component. As such, aspects do not rely on the shielded internals of a component and a loosely coupled system is achieved. The drawback is that aspect beans cannot be instantiated onto for example the setting of a private field. This is however not a fundamental issue, as the language and tools can be easily extended to support these joinpoints if this is considered absolutely necessary.

JAsCo is designed in a bottom-up fashion by incrementally adding new features to the language. In some cases, the semantics is not always well defined. Furthermore, due to backward compatibility reasons, some

language elements are not captured by the most optimal syntax. In the future, we plan to first define a clean semantics for the JAsCo language and redesign the language as optimally as possible according to the semantics.

8.4.2 Further research directions

The JAsCo approach allows implementing aspect beans as an extended version of a Java bean. An aspect bean is backward compatible with a Java bean and can for example be used in visual component composition environments. As such, aspect beans are in fact also regular components. Experience with JAsCo has proven that the other way around, namely unifying components and aspects is also a valuable contribution. This idea stems from the observation that current aspect-oriented approaches (including JAsCo) introduce an additional module construct for representing a concern combined with a composition mechanism. The developer is forced to choose a specific representation, such as an aspect or component, for every concern. Akşit et al. [AM01] already motivated that this gives rise to problems when evolving the software, because changing the representation of a concern can have a deep impact on the software architecture. It is possible that a certain concern is initially perfectly modularized by a normal component. However, the requirements of the application might change over time and as such the concern that is modularized as a component turns out to be crosscutting. The concern then has to be refactored into an aspect, which is a cumbersome and error-prone task. The main cause of this problem consists of the additional module construct (the aspect) introduced by aspect-oriented technologies in order to modularize a crosscutting concern. Inherently, the behavior of these concerns is not different from the behavior of non-crosscutting concerns, only the composition mechanism differs. When introducing a separate aspect module, the composition mechanism is in fact tangled with the behavior of the concern itself. As a result, other composition mechanisms are inherently ruled out. Davy Suvee is currently pursuing this promising research topic and aims at a unified architecture for both components and aspects. He proposes a new programming language, called FuseJ [Suv03,SVW04], which recuperates aspect-oriented ideas and allows implementing all concerns as Java beans. In addition, FuseJ provides a strong composition mechanism that is able to describe both aspect-oriented and component-based compositions. As such, new composition mechanisms are made as unobtrusive as possible, e.g. when representing a software element as a method, it should be possible to compose this method with others as if it were an advice. This allows for postponing the choice for a specific composition mechanism and also enables employing the new composition mechanism on existing software.

Joinpoints in current aspect-oriented approaches are often inadequate in their expressiveness. Aspects can only be applied onto a limited set of possible joinpoint types as for example method executions. Furthermore, components typically communicate in a very specific and standardized way depending on the component model at hand. Current joinpoint models do however not support declaring joinpoints on this kind of communication. JAsCo already introduces an AspectJ inspired joinpoint model for Java Beans. Other component models are however not supported. It would be interesting to investigate more complex communication schemes as for example vetoable events or other more advanced publish/subscribe protocols. Another problem with current joinpoint models is that they have only limited support for specifying dynamic conditions for aspect application. AspectJ and JAsCo capture most dynamic conditions in an “if” construct, which is essentially expressed in the base language. These conditions are thus not captured by the joinpoint model itself, which makes intelligent weaving on the basis of these dynamic conditions impossible. Another huge problem with current joinpoint models is that they offer very limited robustness with respect to evolution. Joinpoint descriptions are often too tightly coupled with concrete code artifacts. As a result, when the application evolves, all joinpoint descriptions have to be manually

altered [HOU03]. Component-based software engineering advocates a very loose coupling between modules, which is thus violated by current joinpoint models. An even worse problem is that joinpoints often deteriorate to an uncomprehensible list of application artifacts after successive refactorings [TKVV04]. In order to solve the problems identified above, more expressive joinpoint models are required that are able to describe joinpoints more precisely, which are able to explicitly capture dynamic conditions and allow to describe joinpoints in a more abstract and "intentional" way in order to accommodate evolution.

As a direct consequence of the more expressive and intentional joinpoint models, aspect weaving becomes a lot harder. It will not be straightforward to identify where in the static program code and when during the runtime of the program, joinpoints are reached. Furthermore, when dynamic conditions are captured explicitly in the joinpoint model, it becomes possible to weave these dynamic joinpoints in a more intelligent way. The advices of the aspect are only attached to those static "shadow joinpoints" [MK03] where the dynamic joinpoints possibly occur. An important limitation of current weaving techniques applied in AspectJ is that aspects cannot be added or removed at run-time; the application needs to be stopped, recompiled and restarted in order to change the aspectual behavior. JAsCo already allows to dynamically add and remove aspects by introducing a new component model where traps that enable aspect interference are built-in. However, when we introduce more expressive and precise joinpoint models, a wealth of traps need to be added in order to allow dynamic aspect weaving and unweaving, which generates an unacceptable overhead. The Java virtual machine recently introduced a new functionality (named HotSwap) that allows to dynamically alter the byte code of a class at run-time. This technology could be exploited in order to insert traps only at those joinpoints where aspects are applicable. When aspects are added or removed at run-time, the required traps can be dynamically added or removed. We have already conducted a first experiment that proves the feasibility of this approach [VS04b]. Another possible approach to realize dynamic AOP consists of custom "aspect-enabled" virtual machines. Such a virtual machine offers an aspect-oriented architecture that allows adding and removing aspects dynamically. We are convinced that in the long term, this is the best solution to support aspect-oriented software development. Indeed, preprocessing, load-time trap insertion or employing the debugging interface (HotSwap) of a virtual machine are all solutions that are feasible on the short-term, but are quite cumbersome and error-prone in comparison with a dedicated execution environment [BHMO04].

Appendix A Glossary

.NET	.NET framework introduced by Microsoft for allowing interoperability between different programming languages. .NET borrows a lot of features from Java and improves Java on some levels.
ADT	Abstract Data Type
AOP	Aspect-Oriented Programming, see Chapter 2, page 54.
AOSD	Aspect-Oriented Software Development, see Chapter 2, page 54.
AP	Adaptive Programming, a methodology that allows structure shy traversals of object oriented software. See section 2.3.2.3, page 58.
API	A set of exposed functions with defined parameter and return types. Components typically offer one ore more APIs in order to exploit its functionality.
CBSD	Component-Based Software Development, see Chapter 2, page 38.
CBSE	Component-Based Software Engineering, see Chapter 2, page 38.
CCM	Corba Component Model. Specification available at: http://www.omg.org.
CORBA	Common Object Request Broker Architecture**.** Specification available at: http://www.omg.org.
DJ	Library realization of Adaptive Programming. See section 2.3.2.3, page 58.
EJB	Enterprise Java Beans is the component model of J2EE.
DFA	Deterministic Finite Automaton, more information in [HMU01].
IDE	Integrated Development Environment. Eclipse or JBuilder are examples of an IDE.
J2EE	Java 2 Enterprise Edition is a Java platform specification designed for mainframe-scale computing typical of large enterprises. EJB is a part of J2EE.
JDI	Java Debugging Interface, a 100% Java API for intercepting the program execution in another Sun Java virtual machine. See also JVMDI.
JIT	Just In Time compiler, a technique for improving the performance of interpreted languages that allows to compile (parts of) the program to native code before it starts running.
JVMDI	Java Virtual Machine Debugging Interface, native API for intercepting the program execution in the Sun Java virtual machine. See also JDI.
MDSOC	Multi-Dimensional Separation of Cconcerns
MSC	Message Sequence Chart. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, September 1993.
NFA	Non-deterministic Finite Automaton, see also DFA. More information in [HMU01].
OOSD	Object Oriented Software Development

XML Extensible Markup Language

Appendix B Correct DynamcTimer implementation

```
1   class DynamicTimer {
2
3      Vector listeners = new Vector();
4
5      public void addTimeListener(TimeListener listener) {
6          listeners.add(listener);
7      }
8
9      public void removeTimeListener(TimeListener listener) {
10         listeners.remove(listener);
11     }
12
13     protected void fireTimeStampTakenEvent(Method m, Long t) {
14         Iterator listenersI = listeners.iterator();
15         while(listenersI.hasNext()) {
16            TimeListener l = (TimeListener)listenersI.next();
17            l.timeStampTaken(m,t);
18         }
19     }
20
21     hook TimeStamp {
22
23        private HashMap timestamps;
24
25        TimeStamp(timedmethod(..args)) {
26           execute(timedmethod);
27        }
28
29        before() {
30            Long timestamp= new Long(System.currentTimeMillis());
31            Thread current = Thread.currentThread();
32            Stack stack = (Stack) timestamps.get(current);
33            if(stack==null) {
34                stack=new Stack();
35                timestamps.put(current, stack);
36            }
37            stack.push(timestamp);
38        }
39
40        after()  {
41            Thread current = Thread.currentThread();
42            Stack stack = (Stack) timestamps.get(current);
43            Long timestamp = (Long) stack.pop();
44            fireTimeStampTakenEvent(timedmethod,timestamp);
45        }
46    }
47  }
```

Appendix C Timing the PhotoLab client application

This appendix shows an example of connectors and glue-code that have been generated by PacoWire. The example composition is the timed PhotoLab client application introduced in section 7.2.

```
connector InvasiveTimer$914Connector {
   timing.InvasiveTimer.TimingHook hook1 = new
     timing.InvasiveTimer.TimingHook({
       void photolab.PhotoLabGUI.buyProducts(java.lang.String),
       onevent photolab.PhotoRenderer.notifyValidateResult
          (java.lang.String),
      }
    );

    hook1.default();
}
```

```
connector InvasiveTimer$922Connector {
   timing.InvasiveTimer.TimingHook hook1 = new
      timing.InvasiveTimer.TimingHook({
       void photolab.PhotoRenderer.checkPictures
         (java.util.Vector),
       onevent photolab.PhotoLabGUI.notifyValidateRequest
         (java.util.Vector)
      }
   );

   hook1.default();
}
```

```
1  package testadapter;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7  import javax.swing.border.*;
8  import pacowire.util.*;
9
10 public class TestApp extends AbstractFrame  {
11
12    private PhotolabClient$955 myPhotolabClient$955;
13
14    public  TestApp()  {
15
16       // Init Frame
17       super("Demo",200,300,true);
18
19       // Instantiate all distinct components of all patterns
20       // and set all options right
21       timing.InvasiveTimer invasiveTimer1077 =
22          Connector.InvasiveTimer$914Connector.getConnector().InvasiveTimer;
23       timing.InvasiveTimer invasiveTimer1269 =
24          Connector.InvasiveTimer$922Connector.getConnector().InvasiveTimer;
25       timing.InvasiveTimer invasiveTimer1433 =
26          Connector.InvasiveTimer$914Connector.getConnector().InvasiveTimer;
27       photolab.PhotoLabGUI photoLabGUI1467 = new photolab.PhotoLabGUI();
28       photolab.PhotoRenderer photoRenderer1524 = new photolab.PhotoRenderer();
29       Network.UDPNetwork uDPNetwork1566 = new Network.UDPNetwork();
30       javax.swing.JButton jButton1614 = new javax.swing.JButton();
31       timing.TimingChecker timingChecker1650 = new timing.TimingChecker();
32
33       //Setting all connectors on instance only
34       //and adding the correct instances to each connector
35       Connector.InvasiveTimer$914Connector.getConnector().setAdaptOnClasses(false);
36       Connector.InvasiveTimer$914Connector.getConnector().
37          addInstance(photoRenderer1524);
38       Connector.InvasiveTimer$914Connector.getConnector().
39          addInstance(photoLabGUI1467);
40       Connector.InvasiveTimer$922Connector.getConnector().setAdaptOnClasses(false);
41       Connector.InvasiveTimer$922Connector.getConnector().
42          addInstance(photoLabGUI1467);
43       Connector.InvasiveTimer$922Connector.getConnector().
44          addInstance(photoRenderer1524);
45
46       // Instantiate the generated composition class for each composition
47       myPhotolabClient$955 = new PhotolabClient$955(this,timingChecker1650,
48       photoLabGUI1467, uDPNetwork1566, invasiveTimer1433, jButton1614,
49       photoRenderer1524, invasiveTimer1269);
50
51        // Subscribe patterns as listeners
52       timingChecker1650.addTimingCheckerListener(myPhotolabClient$955);
53       photoLabGUI1467.addPhotoLabGUIListener(myPhotolabClient$955);
54       uDPNetwork1566.addUDPNetworkListener(myPhotolabClient$955);
55       invasiveTimer1433.addTimingObserver(myPhotolabClient$955);
56       jButton1614.addActionListener(myPhotolabClient$955);
57       photoRenderer1524.addPhotoRendererListener(myPhotolabClient$955);
58       invasiveTimer1269.addTimingObserver(myPhotolabClient$955);
59
60        // Start all patterns
61        myPhotolabClient$955.startComposition();
62
63        // GUI things
64        getContentPane().setLayout(new FlowLayout());
65        JPanel tmp = null;
66        tmp = new JPanel();
67        tmp.setBorder(new TitledBorder("Launcher"));
68        tmp.add(jButton1614);
69        getContentPane().add(tmp);
70
71        setVisible(true);
72        pack();
73     }
74
75     public static void main(String[] args)  {
76
77        TestApp myTestApp = new TestApp();
78     }
79
80 }
```

```
1  package testadapter;
2
3  import java.util.*;
4  import pacowire.automata.*;
5  import pacowire.util.*;
6  import Common.*;
7  import java.lang.*;
8  import photolab.*;
9  import java.awt.event.*;
10
11 public class PhotolabClient$955 implements timing.TimingCheckerListener,
12   photolab.PhotoLabGUIListener, Network.UDPNetworkListener, timing.TimingObserver,
13   java.awt.event.ActionListener, photolab.PhotoRendererListener {
14
15    private TestApp main;
16    private timing.TimingChecker timingChecker1650 = null;
17    private photolab.PhotoLabGUI photoLabGUI1467 = null;
18    private Network.UDPNetwork uDPNetwork1566 = null;
19    private timing.InvasiveTimer invasiveTimer1433 = null;
20    private javax.swing.JButton jButton1614 = null;
21    private photolab.PhotoRenderer photoRenderer1524 = null;
22    private timing.InvasiveTimer invasiveTimer1269 = null;
23    private boolean isInitialised = false;
24    private Queue queue = new Queue();
25    private PacoStateMachine stateMachine;
26    protected static final int STATE_1 = 1;
27    protected static final int STATE_2 = 2;
28    protected static final int STATE_3 = 3;
29    protected static final int STATE_4 = 4;
30    protected static final int STATE_5 = 5;
31    protected static final int STATE_31 = 31;
32    protected static final int STATE_6 = 6;
33    protected static final int STATE_25 = 25;
34    protected static final int STATE_26 = 26;
35    protected static final int STATE_30 = 30;
36    protected static final int STATE_27 = 27;
37    protected static final int STATE_28 = 28;
38    protected static final int STATE_29 = 29;
39    protected static final int STATE_22 = 22;
40    protected static final int STATE_11 = 11;
41    protected static final int STATE_24 = 24;
42    protected static final int STATE_12 = 12;
43    protected static final int STATE_17 = 17;
44    protected static final int STATE_18 = 18;
45    protected static final int STATE_23 = 23;
46    protected static final int STATE_19 = 19;
47    protected static final int STATE_20 = 20;
48    protected static final int STATE_21 = 21;
49    protected static final int STATE_13 = 13;
50    protected static final int STATE_14 = 14;
51    protected static final int STATE_15 = 15;
52    protected static final int STATE_16 = 16;
53    protected static final int STATE_7 = 7;
54    protected static final int STATE_8 = 8;
55    protected static final int STATE_9 = 9;
56    protected static final int STATE_10 = 10;
57
58    public  PhotolabClient$955(TestApp main, timing.TimingChecker timingChecker1650,
59      photolab.PhotoLabGUI photoLabGUI1467, Network.UDPNetwork uDPNetwork1566,
60      timing.InvasiveTimer invasiveTimer1433, javax.swing.JButton jButton1614,
61      photolab.PhotoRenderer photoRenderer1524, timing.InvasiveTimer invasiveTimer1269)
62    {
63       this.main = main;
64       this.timingChecker1650 = timingChecker1650;
65       this.photoLabGUI1467 = photoLabGUI1467;
66       this.uDPNetwork1566 = uDPNetwork1566;
67       this.invasiveTimer1433 = invasiveTimer1433;
68       this.jButton1614 = jButton1614;
69       this.photoRenderer1524 = photoRenderer1524;
70       this.invasiveTimer1269 = invasiveTimer1269;
71       initialiseStateMachine();
72    }
73    public void constraintViolated(timing.ConstraintViolationEvent par1)  {
74       Vector parameters = new Vector();
75       parameters.addElement(par1);
76       Object src = null;
77       processEvent("constraintViolated", src, parameters);
78    }
79    public void communicate(java.lang.String par1)  {
80       Vector parameters = new Vector();
```

```
81       parameters.addElement(par1);
82       Object src = null;
83       processEvent("communicate", src, parameters);
84    }
85    public void notifyPreviewRequest(photolab.PreviewRequestEvent par1)  {
86       Vector parameters = new Vector();
87       parameters.addElement(par1);
88       Object src = null;
89       processEvent("notifyPreviewRequest", src, parameters);
90    }
91    public void notifyValidateRequest(java.util.Vector par1)  {
92       Vector parameters = new Vector();
93       parameters.addElement(par1);
94       Object src = null;
95       processEvent("notifyValidateRequest", src, parameters);
96
97    }
98    public void listeningStarted(Common.PacoEvent par1)  {
99       Vector parameters = new Vector();
100      parameters.addElement(par1);
101      Object src = ((java.util.EventObject) par1).getSource();
102      processEvent("listeningStarted", src, parameters);
103   }
104   public void listeningStopped(Common.PacoEvent par1)  {
105      Vector parameters = new Vector();
106      parameters.addElement(par1);
107      Object src = ((java.util.EventObject) par1).getSource();
108      processEvent("listeningStopped", src, parameters);
109   }
110   public void receive(Common.PacoEvent par1)  {
111      Vector parameters = new Vector();
112      parameters.addElement(par1);
113      Object src = ((java.util.EventObject) par1).getSource();
114      processEvent("receive", src, parameters);
115   }
116   public void timeStampTaken(java.lang.Long par1, java.lang.Object par2)  {
117      Vector parameters = new Vector();
118      parameters.addElement(par1);
119      parameters.addElement(par2);
120      Object src = null;
121      processEvent("timeStampTaken", src, parameters);
122   }
123   public void actionPerformed(java.awt.event.ActionEvent par1)  {
124      Vector parameters = new Vector();
125      parameters.addElement(par1);
126      Object src = ((java.util.EventObject) par1).getSource();
127      processEvent("actionPerformed", src, parameters);
128   }
129   public void notifyValidateResult(java.lang.String par1)  {
130      Vector parameters = new Vector();
131      parameters.addElement(par1);
132      Object src = null;
133      processEvent("notifyValidateResult", src, parameters);
134   }
135   protected boolean isInitialised()  {
136      return isInitialised;
137   }
138
139   protected void setInitialised(boolean bool)  {
140      this.isInitialised = bool;
141   }
142   public void processEvent(String name, Object src, Vector parameters)  {
143      if(!isInitialised())
144         addEventToQueue(name,src,parameters);
145      else executeMappingCode(getStatemachine().getCurrentState(),name, src,
146            parameters);
147   }
148   protected PacoStateMachine getStatemachine()  {
149      return stateMachine;
150   }
151   protected void addEventToQueue(String name, Object src, Vector parameters)  {
152      Vector tmp = new Vector();
153      tmp.add(name);
154      tmp.add(src);
155      tmp.add(parameters);
156      queue.enqueue(tmp);
157   }
158   public void startComposition()  {
159      setInitialised(true);
160      while(!queue.isEmpty()) {
```

```
161         Vector next = (Vector) queue.dequeue();
162         String name = next.elementAt(0).toString();
163         Object src  = (Object) next.elementAt(1);
164         Vector params = (Vector) next.elementAt(2);
165         executeMappingCode(getStatemachine().getCurrentState(),name,src, params);
166      }
167   }
168   protected void initialiseStateMachine()  {
169      PacoState State_1 = new PacoState(PhotolabClient$955.STATE_1);
170      State_1.setIsStartState(true);
171      PacoState State_2 = new PacoState(PhotolabClient$955.STATE_2);
172      PacoState State_3 = new PacoState(PhotolabClient$955.STATE_3);
173      PacoState State_4 = new PacoState(PhotolabClient$955.STATE_4);
174      PacoState State_5 = new PacoState(PhotolabClient$955.STATE_5);
175      PacoState State_31 = new PacoState(PhotolabClient$955.STATE_31);
176      State_31.setIsStopState(true);
177      State_5.addNextState("receive", State_31);
178      State_4.addNextState("communicate", State_5);
179      PacoState State_6 = new PacoState(PhotolabClient$955.STATE_6);
180      PacoState State_25 = new PacoState(PhotolabClient$955.STATE_25);
181      PacoState State_26 = new PacoState(PhotolabClient$955.STATE_26);
182      PacoState State_30 = new PacoState(PhotolabClient$955.STATE_30);
183      State_30.addNextState("notifyPreviewRequest", State_26);
184      PacoState State_27 = new PacoState(PhotolabClient$955.STATE_27);
185      PacoState State_28 = new PacoState(PhotolabClient$955.STATE_28);
186      PacoState State_29 = new PacoState(PhotolabClient$955.STATE_29);
187      PacoState State_22 = new PacoState(PhotolabClient$955.STATE_22);
188      PacoState State_11 = new PacoState(PhotolabClient$955.STATE_11);
189      PacoState State_24 = new PacoState(PhotolabClient$955.STATE_24);
190      State_24.setIsStopState(true);
191      State_11.addNextState("receive", State_24);
192      State_22.addNextState("communicate", State_11);
193      PacoState State_12 = new PacoState(PhotolabClient$955.STATE_12);
194      PacoState State_17 = new PacoState(PhotolabClient$955.STATE_17);
195      PacoState State_18 = new PacoState(PhotolabClient$955.STATE_18);
196      PacoState State_23 = new PacoState(PhotolabClient$955.STATE_23);
197      State_23.addNextState("notifyPreviewRequest", State_18);
198      PacoState State_19 = new PacoState(PhotolabClient$955.STATE_19);
199      PacoState State_20 = new PacoState(PhotolabClient$955.STATE_20);
200      PacoState State_21 = new PacoState(PhotolabClient$955.STATE_21);
201      State_21.addNextState("timeStampTaken", State_22);
202      State_20.addNextState("notifyValidateResult", State_21);
203      State_19.addNextState("timeStampTaken", State_20);
204      State_23.addNextState("notifyValidateRequest", State_19);
205      State_18.addNextState("timeStampTaken", State_23);
206      State_17.addNextState("notifyPreviewRequest", State_18);
207      State_17.addNextState("notifyValidateRequest", State_19);
208      State_12.addNextState("timeStampTaken", State_17);
209      State_22.addNextState("notifyPreviewRequest", State_12);
210      PacoState State_13 = new PacoState(PhotolabClient$955.STATE_13);
211      PacoState State_14 = new PacoState(PhotolabClient$955.STATE_14);
212      PacoState State_15 = new PacoState(PhotolabClient$955.STATE_15);
213      PacoState State_16 = new PacoState(PhotolabClient$955.STATE_16);
214      State_16.addNextState("communicate", State_11);
215      State_16.addNextState("notifyPreviewRequest", State_12);
216      State_16.addNextState("notifyValidateRequest", State_13);
217      State_15.addNextState("timeStampTaken", State_16);
218      State_14.addNextState("notifyValidateResult", State_15);
219      State_13.addNextState("timeStampTaken", State_14);
220      State_22.addNextState("notifyValidateRequest", State_13);
221      State_29.addNextState("timeStampTaken", State_22);
222      State_28.addNextState("notifyValidateResult", State_29);
223      State_27.addNextState("timeStampTaken", State_28);
224      State_30.addNextState("notifyValidateRequest", State_27);
225      State_26.addNextState("timeStampTaken", State_30);
226      State_25.addNextState("notifyPreviewRequest", State_26);
227      State_25.addNextState("notifyValidateRequest", State_27);
228      State_6.addNextState("timeStampTaken", State_25);
229      State_4.addNextState("notifyPreviewRequest", State_6);
230      PacoState State_7 = new PacoState(PhotolabClient$955.STATE_7);
231      PacoState State_8 = new PacoState(PhotolabClient$955.STATE_8);
232      PacoState State_9 = new PacoState(PhotolabClient$955.STATE_9);
233      PacoState State_10 = new PacoState(PhotolabClient$955.STATE_10);
234      State_10.addNextState("communicate", State_11);
235      State_10.addNextState("notifyPreviewRequest", State_12);
236      State_10.addNextState("notifyValidateRequest", State_13);
237      State_9.addNextState("timeStampTaken", State_10);
238      State_8.addNextState("notifyValidateResult", State_9);
239      State_7.addNextState("timeStampTaken", State_8);
240      State_4.addNextState("notifyValidateRequest", State_7);
```

```
241      State_3.addNextState("receive", State_4);
242      State_2.addNextState("communicate", State_3);
243      State_1.addNextState("actionPerformed", State_2);
244      stateMachine = new PacoStateMachine(State_1);
245   }
246   protected void executeMappingCode(PacoState currentState, String name, Object src,
247     Vector parameters)
248   {
249      PacoState nextState = getStatemachine().doTransition(name);
250      if(nextState == null) return;
251      switch((int) currentState.getId()) {
252         case STATE_16: {
253            if (name.equals("communicate") && (src == null|| src ==
254                  photoLabGUI1467)){
255               String inpar0 = (String) parameters.elementAt(0);
256               String outpar0 = (String) inpar0;
257               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
258               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
259               uDPNetwork1566.sendPacket(outpar0);
260            }
261            else if (name.equals("notifyPreviewRequest") && (src == null|| src ==
262                 photoLabGUI1467)){
263               PreviewRequestEvent inpar0 = (PreviewRequestEvent)
264                 parameters.elementAt(0);
265               java.lang.String outpar0 = inpar0.getFile();
266               java.lang.Float outpar1 = inpar0.getWidth();
267               java.lang.Float outpar2 = inpar0.getHeight();
268               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
269               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
270               photoRenderer1524.preview(outpar0, outpar1, outpar2);
271            }
272            else if (name.equals("notifyValidateRequest") && (src == null|| src ==
273                 photoLabGUI1467)){
274               Vector inpar0 = (Vector) parameters.elementAt(0);
275               Vector outpar0 = (Vector) inpar0;
276               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
277               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
278               photoRenderer1524.checkPictures(outpar0);
279            }
280            break;
281         }
282         case STATE_15: {
283            if (name.equals("timeStampTaken") && (src == null|| src ==
284                 invasiveTimer1433)){
285               Long inpar0 = (Long) parameters.elementAt(0);
286               Object inpar1 = (Object) parameters.elementAt(1);
287               java.lang.String outpar0 = "ANSWER";
288               long outpar1=inpar0.longValue();
289               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
290               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
291               timingChecker1650.checkTiming(outpar0, outpar1);
292            }
293            break;
294         }
295         case STATE_14: {
296            if (name.equals("notifyValidateResult") && (src == null|| src ==
297                 photoRenderer1524)){
298               String inpar0 = (String) parameters.elementAt(0);
299               String outpar0 = (String) inpar0;
300               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(true);
301               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
302               photoLabGUI1467.buyProducts(outpar0);
303            }
304            break;
305         }
306         case STATE_13: {
307            if (name.equals("timeStampTaken") && (src == null|| src ==
308                 invasiveTimer1269)){
309               Long inpar0 = (Long) parameters.elementAt(0);
310               Object inpar1 = (Object) parameters.elementAt(1);
311               java.lang.String outpar0 = "REQUEST";
312               long outpar1=inpar0.longValue();
313               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
314               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
315               timingChecker1650.checkTiming(outpar0, outpar1);
316            }
317            break;
318         }
319         case STATE_12: {
320            if (name.equals("timeStampTaken") && (src == null|| src ==
```

```
321                invasiveTimer1269)){
322             Long inpar0 = (Long) parameters.elementAt(0);
323             Object inpar1 = (Object) parameters.elementAt(1);
324             java.lang.String outpar0 = "NOTIFY";
325             long outpar1=inpar0.longValue();
326             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
327             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
328             timingChecker1650.checkTiming(outpar0, outpar1);
329          }
330          break;
331       }
332       case STATE_11: {
333          if (name.equals("receive") && (src == null|| src == uDPNetwork1566)){
334             PacoEvent inpar0 = (PacoEvent) parameters.elementAt(0);
335             java.lang.String outpar0 = inpar0.getParam();
336             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
337             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
338             photoLabGUI1467.submitReport(outpar0);
339          }
340          break;
341       }
342       case STATE_10: {
343          if (name.equals("communicate") && (src == null|| src ==
344                photoLabGUI1467)){
345             String inpar0 = (String) parameters.elementAt(0);
346             String outpar0 = (String) inpar0;
347             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
348             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
349             uDPNetwork1566.sendPacket(outpar0);
350          }
351          else if (name.equals("notifyPreviewRequest") && (src == null|| src ==
352                photoLabGUI1467)){
353             PreviewRequestEvent inpar0 = (PreviewRequestEvent)
354               parameters.elementAt(0);
355             java.lang.String outpar0 = inpar0.getFile();
356             java.lang.Float outpar1 = inpar0.getWidth();
357             java.lang.Float outpar2 = inpar0.getHeight();
358             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
359             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
360             photoRenderer1524.preview(outpar0, outpar1, outpar2);
361          }
362          else if (name.equals("notifyValidateRequest") && (src == null|| src ==
363                photoLabGUI1467)){
364             Vector inpar0 = (Vector) parameters.elementAt(0);
365             Vector outpar0 = (Vector) inpar0;
366             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
367             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
368             photoRenderer1524.checkPictures(outpar0);
369          }
370          break;
371       }
372       case STATE_29: {
373          if (name.equals("timeStampTaken") && (src == null|| src ==
374                invasiveTimer1433)){
375             Long inpar0 = (Long) parameters.elementAt(0);
376             Object inpar1 = (Object) parameters.elementAt(1);
377             java.lang.String outpar0 = "ANSWER";
378             long outpar1=inpar0.longValue();
379             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
380             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
381             timingChecker1650.checkTiming(outpar0, outpar1);
382          }
383          break;
384       }
385       case STATE_28: {
386          if (name.equals("notifyValidateResult") && (src == null|| src ==
387                photoRenderer1524)){
388             String inpar0 = (String) parameters.elementAt(0);
389             String outpar0 = (String) inpar0;
390             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(true);
391             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
392             photoLabGUI1467.buyProducts(outpar0);
393          }
394          break;
395       }
396       case STATE_27: {
397          if (name.equals("timeStampTaken") && (src == null|| src ==
398                invasiveTimer1269)){
399             Long inpar0 = (Long) parameters.elementAt(0);
400             Object inpar1 = (Object) parameters.elementAt(1);
```

```
401             java.lang.String outpar0 = "REQUEST";
402             long outpar1=inpar0.longValue();
403             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
404             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
405             timingChecker1650.checkTiming(outpar0, outpar1);
406          }
407          break;
408       }
409       case STATE_26: {
410          if (name.equals("timeStampTaken") && (src == null|| src ==
411                invasiveTimer1269)){
412             Long inpar0 = (Long) parameters.elementAt(0);
413             Object inpar1 = (Object) parameters.elementAt(1);
414             java.lang.String outpar0 = "NOTIFY";
415             long outpar1=inpar0.longValue();
416             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
417             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
418             timingChecker1650.checkTiming(outpar0, outpar1);
419          }
420          break;
421       }
422       case STATE_25: {
423          if (name.equals("notifyPreviewRequest") && (src == null|| src ==
424                photoLabGUI1467)){
425             PreviewRequestEvent inpar0 = (PreviewRequestEvent)
426                parameters.elementAt(0);
427             java.lang.String outpar0 = inpar0.getFile();
428             java.lang.Float outpar1 = inpar0.getWidth();
429             java.lang.Float outpar2 = inpar0.getHeight();
430             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
431             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
432             photoRenderer1524.preview(outpar0, outpar1, outpar2);
433          }
434          else if (name.equals("notifyValidateRequest") && (src == null|| src ==
435                photoLabGUI1467)){
436             Vector inpar0 = (Vector) parameters.elementAt(0);
437             Vector outpar0 = (Vector) inpar0;
438             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
439             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
440             photoRenderer1524.checkPictures(outpar0);
441          }
442          break;
443       }
444       case STATE_23: {
445          if (name.equals("notifyPreviewRequest") && (src == null|| src ==
446                photoLabGUI1467)){
447             PreviewRequestEvent inpar0 = (PreviewRequestEvent)
448               parameters.elementAt(0);
449             java.lang.String outpar0 = inpar0.getFile();
450             java.lang.Float outpar1 = inpar0.getWidth();
451             java.lang.Float outpar2 = inpar0.getHeight();
452             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
453             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
454             photoRenderer1524.preview(outpar0, outpar1, outpar2);
455          }
456          else if (name.equals("notifyValidateRequest") && (src == null|| src ==
457                photoLabGUI1467)){
458             Vector inpar0 = (Vector) parameters.elementAt(0);
459             Vector outpar0 = (Vector) inpar0;
460             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
461             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
462             photoRenderer1524.checkPictures(outpar0);
463          }
464          break;
465       }
466       case STATE_22: {
467          if (name.equals("communicate") && (src == null|| src ==
468                photoLabGUI1467)){
469             String inpar0 = (String) parameters.elementAt(0);
470             String outpar0 = (String) inpar0;
471             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
472             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
473             uDPNetwork1566.sendPacket(outpar0);
474          }
475          else if (name.equals("notifyPreviewRequest") && (src == null|| src ==
476                photoLabGUI1467)){
477             PreviewRequestEvent inpar0 = (PreviewRequestEvent)
478                parameters.elementAt(0);
479             java.lang.String outpar0 = inpar0.getFile();
480             java.lang.Float outpar1 = inpar0.getWidth();
```

```
481             java.lang.Float outpar2 = inpar0.getHeight();
482             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
483             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
484             photoRenderer1524.preview(outpar0, outpar1, outpar2);
485          }
486          else if (name.equals("notifyValidateRequest") && (src == null|| src ==
487               photoLabGUI1467)){
488             Vector inpar0 = (Vector) parameters.elementAt(0);
489             Vector outpar0 = (Vector) inpar0;
490             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
491             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
492             photoRenderer1524.checkPictures(outpar0);
493          }
494          break;
495       }
496       case STATE_21: {
497          if (name.equals("timeStampTaken") && (src == null|| src ==
498               invasiveTimer1433)){
499             Long inpar0 = (Long) parameters.elementAt(0);
500             Object inpar1 = (Object) parameters.elementAt(1);
501             java.lang.String outpar0 = "ANSWER";
502             long outpar1=inpar0.longValue();
503             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
504             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
505             timingChecker1650.checkTiming(outpar0, outpar1);
506          }
507          break;
508       }
509       case STATE_20: {
510          if (name.equals("notifyValidateResult") && (src == null|| src ==
511               photoRenderer1524)){
512             String inpar0 = (String) parameters.elementAt(0);
513             String outpar0 = (String) inpar0;
514             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(true);
515             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
516             photoLabGUI1467.buyProducts(outpar0);
517          }
518          break;
519       }
520       case STATE_9: {
521          if (name.equals("timeStampTaken") && (src == null|| src ==
522               invasiveTimer1433)){
523             Long inpar0 = (Long) parameters.elementAt(0);
524             Object inpar1 = (Object) parameters.elementAt(1);
525             java.lang.String outpar0 = "ANSWER";
526             long outpar1=inpar0.longValue();
527             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
528             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
529             timingChecker1650.checkTiming(outpar0, outpar1);
530          }
531          break;
532       }
533       case STATE_8: {
534          if (name.equals("notifyValidateResult") && (src == null|| src ==
535               photoRenderer1524)){
536             String inpar0 = (String) parameters.elementAt(0);
537             String outpar0 = (String) inpar0;
538             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(true);
539             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
540             photoLabGUI1467.buyProducts(outpar0);
541          }
542          break;
543       }
544       case STATE_7: {
545          if (name.equals("timeStampTaken") && (src == null|| src ==
546               invasiveTimer1269)){
547             Long inpar0 = (Long) parameters.elementAt(0);
548             Object inpar1 = (Object) parameters.elementAt(1);
549             java.lang.String outpar0 = "REQUEST";
550             long outpar1=inpar0.longValue();
551             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
552             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
553             timingChecker1650.checkTiming(outpar0, outpar1);
554          }
555          break;
556       }
557       case STATE_6: {
558          if (name.equals("timeStampTaken") && (src == null|| src ==
559               invasiveTimer1269)){
560             Long inpar0 = (Long) parameters.elementAt(0);
```

```
561             Object inpar1 = (Object) parameters.elementAt(1);
562             java.lang.String outpar0 = "NOTIFY";
563             long outpar1=inpar0.longValue();
564             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
565             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
566             timingChecker1650.checkTiming(outpar0, outpar1);
567          }
568          break;
569       }
570       case STATE_30: {
571          if (name.equals("notifyPreviewRequest") && (src == null|| src ==
572               photoLabGUI1467)){
573             PreviewRequestEvent inpar0 = (PreviewRequestEvent)
574                parameters.elementAt(0);
575             java.lang.String outpar0 = inpar0.getFile();
576             java.lang.Float outpar1 = inpar0.getWidth();
577             java.lang.Float outpar2 = inpar0.getHeight();
578             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
579             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
580             photoRenderer1524.preview(outpar0, outpar1, outpar2);
581          }
582          else if (name.equals("notifyValidateRequest") && (src == null|| src ==
583               photoLabGUI1467)){
584             Vector inpar0 = (Vector) parameters.elementAt(0);
585             Vector outpar0 = (Vector) inpar0;
586             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
587             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
588             photoRenderer1524.checkPictures(outpar0);
589          }
590          break;
591       }
592       case STATE_5: {
593          if (name.equals("receive") && (src == null|| src == uDPNetwork1566)){
594             PacoEvent inpar0 = (PacoEvent) parameters.elementAt(0);
595             java.lang.String outpar0 = inpar0.getParam();
596             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
597             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
598             photoLabGUI1467.submitReport(outpar0);
599          }
600          break;
601       }
602       case STATE_4: {
603          if (name.equals("communicate") && (src == null|| src ==
604               photoLabGUI1467)){
605             String inpar0 = (String) parameters.elementAt(0);
606             String outpar0 = (String) inpar0;
607             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
608             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
609             uDPNetwork1566.sendPacket(outpar0);
610          }
611          else if (name.equals("notifyPreviewRequest") && (src == null|| src ==
612               photoLabGUI1467)){
613             PreviewRequestEvent inpar0 = (PreviewRequestEvent)
614               parameters.elementAt(0);
615             java.lang.String outpar0 = inpar0.getFile();
616             java.lang.Float outpar1 = inpar0.getWidth();
617             java.lang.Float outpar2 = inpar0.getHeight();
618             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
619             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
620             photoRenderer1524.preview(outpar0, outpar1, outpar2);
621          }
622          else if (name.equals("notifyValidateRequest") && (src == null|| src ==
623               photoLabGUI1467)){
624             Vector inpar0 = (Vector) parameters.elementAt(0);
625             Vector outpar0 = (Vector) inpar0;
626             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
627             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
628             photoRenderer1524.checkPictures(outpar0);
629          }
630          break;
631       }
632       case STATE_3: {
633          if (name.equals("receive") && (src == null|| src == uDPNetwork1566)){
634             PacoEvent inpar0 = (PacoEvent) parameters.elementAt(0);
635             java.lang.String outpar0 = inpar0.getParam();
636             Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
637             Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
638             photoLabGUI1467.loginResult(outpar0);
639          }
640          break;
```

```
641         }
642         case STATE_2: {
643            if (name.equals("communicate") && (src == null|| src ==
644                 photoLabGUI1467)){
645               String inpar0 = (String) parameters.elementAt(0);
646               String outpar0 = (String) inpar0;
647               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
648               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
649               uDPNetwork1566.sendPacket(outpar0);
650            }
651            break;
652         }
653         case STATE_1: {
654            if (name.equals("actionPerformed") && (src == null|| src ==
655                 jButton1614)){
656               ActionEvent inpar0 = (ActionEvent) parameters.elementAt(0);
657               boolean outpar0 = true;
658               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
659               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
660               photoLabGUI1467.showLogin();
661            }
662            break;
663         }
664         case STATE_19: {
665            if (name.equals("timeStampTaken") && (src == null|| src ==
666                 invasiveTimer1269)){
667               Long inpar0 = (Long) parameters.elementAt(0);
668               Object inpar1 = (Object) parameters.elementAt(1);
669               java.lang.String outpar0 = "REQUEST";
670               long outpar1=inpar0.longValue();
671               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
672               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
673               timingChecker1650.checkTiming(outpar0, outpar1);
674            }
675            break;
676         }
677         case STATE_18: {
678            if (name.equals("timeStampTaken") && (src == null|| src ==
679                 invasiveTimer1269)){
680               Long inpar0 = (Long) parameters.elementAt(0);
681               Object inpar1 = (Object) parameters.elementAt(1);
682               java.lang.String outpar0 = "NOTIFY";
683               long outpar1=inpar0.longValue();
684               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
685               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(false);
686               timingChecker1650.checkTiming(outpar0, outpar1);
687            }
688            break;
689         }
690         case STATE_17: {
691            if (name.equals("notifyPreviewRequest") && (src == null|| src ==
692                  photoLabGUI1467)){
693               PreviewRequestEvent inpar0 = (PreviewRequestEvent)
694                  parameters.elementAt(0);
695               java.lang.String outpar0 = inpar0.getFile();
696               java.lang.Float outpar1 = inpar0.getWidth();
697               java.lang.Float outpar2 = inpar0.getHeight();
698               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
699               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
700               photoRenderer1524.preview(outpar0, outpar1, outpar2);
701            }
702            else if (name.equals("notifyValidateRequest") && (src == null|| src ==
703                 photoLabGUI1467)){
704               Vector inpar0 = (Vector) parameters.elementAt(0);
705               Vector outpar0 = (Vector) inpar0;
706
707               Connector.InvasiveTimer$914Connector.getConnector().setEnabled(false);
708               Connector.InvasiveTimer$922Connector.getConnector().setEnabled(true);
709               photoRenderer1524.checkPictures(outpar0);
710            }
711            break;
712         }
713         default: {
714            break;
715         }
716      }
717   }
718}
719}
```

Appendix D Obsolete Product Discount Aspect Bean

```
1  package photolab;
2
3  class ObsoleteProductsDiscount {
4
5       private boolean doApplyDiscount;
6       private float discountPerc = 0.1f;
7
8       public float getDiscount() {
9
10              return discountPerc;
11      }
12      public void setDiscount(float f) {
13
14              discountPerc=f;
15      }
16      public boolean doApplyDiscount() {
17
18              return doApplyDiscount;
19      }
20      public void setDoApplyDiscount(boolean b) {
21
22              doApplyDiscount=b;
23      }
24
25
26      hook CaptureProduct {
27
28              CaptureProduct(method(String productreq)) {
29                      execute(method);
30
31              }
32
33              before() {
34                 int i = productreq.indexOf(" ");
35                 String productname=productreq.substring(0,i);
36                 String res = ((photolab.SimpleDatabase)
37                 calledobject).getData(productname+" old");
38                 global.setDoApplyDiscount(new Boolean(res).booleanValue());
39              }
40      }
41
42      hook ApplyDiscount {
43
44              ApplyDiscount(method(float price)) {
45                      execute(method);
46
47              }
48
49              isApplicable() {
50
51                      return global.doApplyDiscount();
52              }
53
54              replace() {
55                      price-=price*global.getDiscount();
56                      return method(price);
57              }
58      }
59 }
```

Appendix E SecureLogin Aspect Bean

```
1  package photolab;
2
3  class SecureLogin {
4
5       private int blockTreshold = 2;
6       public void setBlockTreshold(int i) {
7
8               blockTreshold=i;
9       }
10      public int getBlockTreshold() {
11
12              return blockTreshold;
13      }
14      private String userName = "";
15      public void setUserName(String name) {
16
17              userName=name;
18      }
19      public String getUserName() {
20
21              return userName;
22      }
23      private int noattempts = 0;
24      public void setNumberOfAttempts(int i) {
25
26              noattempts=i;
27      }
28      public int getNumberOfAttempts() {
29
30              return noattempts;
31      }
32      private Vector listeners = new Vector();
33      public void addDataPersistancyListener(DataPersistancyListener l) {
34              listeners.add(l);
35      }
36      public void removeDataPersistancyListener(DataPersistancyListener l) {
37              listeners.remove(l);
38      }
39      public void fireDataPersistancyRequest(String key, String value) {
40              Iterator i = listeners.iterator();
41              while(i.hasNext()) {
42                  DataPersistancyListener l = (DataPersistancyListener ) i.next();
43                  l.saveData(key,value);
44              }
45      }
46      public void fireNOERequest(String user) {
47              Iterator i = listeners.iterator();
48              while(i.hasNext()) {
49                  DataPersistancyListener l = (DataPersistancyListener ) i.next();
50                  l.getNumberOfAttempts(key);
51              }
52      }
53
54      hook CaptureUser {
55              CaptureUser(method(String userreq)) {
56                      execute(method);
57
58              }
59              before() {
60                  int i = userreq.indexOf(" ");
61                  String username=userreq.substring(0,i);
62                  global.setUserName(username);
63              }
64      }
65
66      hook BlockUsers {
67
68              BlockUsers(method(String validationresult)) {
```

```
69                    execute(method);
70             }
71
72             replace() {
73                global.fireNOERequest(global.getUserName()+" wronglogins");
74                if(!validationresult.equals("")) {
75                   global.setNumberOfAttempts(global.getNumberOfAttempts()+1);
76                   global.fireDataPersistancyRequest(
77                         global.getUserName()+" wronglogins",
78                         global.getNumberOfAttempts()+""
79                   );
80                }
81
82                if(global.getNumberOfAttempts()>global.getBlockTreshold())
83                    return method("Account is blocked due to excessive false logins");
84                else return method(validationresult);
85             }
86
87      }
88
89 }
```

Bibliography

[AM01] Akşit, M. and Marcelloni, F. *Deferring Elimination of Design Alternatives in Object-Oriented Methods.* Concurrency and Computation: Practice and Experience, 13(14), 2001, pp. 1247-1279.

[AG97] Allen, R. and Garlan, D. *A Formal Basis for Architectural Connection.* ACM Transactions on Software Engineering and Methodology, vol. 6, no. 3, pp. 213-249, 1997.

[AL03] Attie, P. and Lorenz, D.H. *Correctness of Model-based Component Composition without State Explosion.* In proceedings of ECOOP Workshop on Correctness of Model-based Software Composition, July 2003.

[Aßm03] Aßmann, U. *Invasive Software Composition*. Springer, ISBN 3-540-44385-1, 2003.

[BFG02] Bacon, D., Fink, S. and Grove, D. *Space- and Time-Efficient Implementation of the Java Object Model.* In proceedings of European Conference on Object-Oriented Programming (ECOOP 2002), Malaga Spain, June 2002.

[BFM93] Bellinzona, R., Fugini, M. G. and de Mey V. *Reuse of Specifications and Designs in a Development Information System.* In Proceedings IFIP WG 8.1 Working Conference on Information System Development Process, 1993.

[BD62] Bellman R.E. and Dreyfus S.E. *Applied Dynamical Programming.* Princeton University Press, 1962.

[BA01] Bergmans, L. and Aksit, M. *Composing Crosscutting Concerns Using Composition Filters*. Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[BAT01] Bergmans, L., Aksit, M., and Tekinerdogan, B. *Aspect Composition Using Composition Filters*. In Software Architectures and Component Technology: The State of the Art in Research and Practice, Aksit M. (Ed.), Kluwer Academic Publishers, pp. 357 - 382, ISBN 0-7923-7576-9, October 2001.

[BJPW99] Beugnard, A., Jezequel, J.-M., Plouzeau, N. and Watkins, D. *Making Components Contract Aware.* In IEEE Software Vol. 32(7), pp. 38-45, 1999.

[BGH03] Blackburn, S., Grove, D. and Hind, M. Jikes Java Virtual Machine. Available at: http://www-124.ibm.com/developerworks/oss/jikesrvm/index.shtml

[BHMO04] Bockisch, D., Haupt, M. Mezini, M. and Ostermann, K. *Virtual Machine Support for Dynamic Join Points.* In Proceedings of International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, UK, March 2004.

[BV03] Bonér, J. and Vasseur A. *AspectWerkz: is a dynamic, lightweight and high-performant AOP/AOSD framework for Java.* Available at: http://aspectwerkz.codehaus.org/

[BMD02] Brichau, J., Mens, K. and De Volder, K. *Building Composable Aspect-specific Languages with Logic Metaprogramming.* In Proceedings of GPCE 2002. Pittsburgh, USA, October 2002.

[BDD00] Brichau, J., De Meuter, W. and De Volder, K. *Jumping Aspects*. In Proceedings of Workshop on Aspects and Dimensions of Concerns at ECOOP, 2000.

[BW96] Brown, A.W. and Wallnau K.C. *Engineering of Component-Based Systems*. Proceedings of the 2nd IEEE International Conference on Complex Computer Systems, October 1996.

[BCF+03] Burke, B., Chau, A., Fleury, M., Brock, A., Godwin, A. and Gliebe, H. *JBoss Aspect Oriented Programming*. Available at: www.jboss.org/developers/projects/jboss/aop.jsp

[Chi00] Chiba, S. *Load-time Structural Reflection in Java.* In proceedings of ECOOP 2000 -- Object-Oriented Programming, LNCS 1850, Springer Verlag, pp 313-336, 2000.

[CN03] Chiba, S. and Nishizawa, M. *An Easy-to-Use Toolkit for Efficient Java Bytecode Translators.* Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp.364-376, Springer-Verlag, 2003.

[CDJ03] Cibran, M., D'Hondt, M. and Jonckers, V. *Aspect-Oriented Programming for Connecting Business Rules.* In Proceedings of the 6th International Conference on Business Information Systems (BIS'03). Colorado Springs, USA, June 2003.

[CDS+03] Cibrán, M., D'Hondt, M., Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo for Linking Business Rules to Object-Oriented Software.* In Proceedings of CSITeA'03 International Conference. Rio De Janeiro, Brazil, June 2003.

[CV03] Cibrán M.A. and Verheecke B. *Modularizing Client-Side Web Service Management Aspects.* In Proceedings of Second Nordic Conference on Web Services (NCWS'03), Vaxjo, Sweden, November 2003.

[CSC01] Costanza, P., Stiemerling, O. and Cremers, A. *Object Identity and Dynamic Recomposition of Components.* In Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 38, 12-14 März 2001, Zürich, Switzerland, IEEE Computer Society Press, 2001.

[CN86] Cox, B. and Novobilski, A.J. *Object-oriented Programming, An Evolutionary Approach*. Addison Wesley Publishing Company, Second Edition 1986.

[CE00] Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, Reading, Massachusetts, USA, 2000.

[DZ03] Dahm, K. and van Zyl, J. *Byte Code Engineering Library (BCEL).* Available at: http://jakarta.apache.org/bcel/index.html.

[DVD02] De Win, B., Vanhaute, B. and De Decker, B. *How aspect-oriented programming can help to build secure software.* Informatica vol.26(2), 2002, pp. 141-149.

[DLVW98] De Pauw, W., Lorenz, D., Vlissides, J. and Wegman, M. *Execution patterns in object-oriented visualization.* In Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS). Santa Fe, USA, April 1998.

[Dij72] Dijkstra, E.W. *The Humble Programmer.* In Commununications ACM 15, 10: 859–866, 1972.

[DFS02] Douence, R., Fradet, P. and Südholt, M. *A framework for the detection and resolution of aspect interactions.* In Proc. of the ACMSIGPLAN/SIGSOFT Conf. on Generative Programming and Component Engineering (GPCE), October 2002.

[DMS01] Douence, R., Motelet, O. and Südholt, M. *A formal definition of crosscuts.* In Proc. of the 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns, volume 2192 of LNCS. Springer Verlag, September 2001.

[DS02] Douence, R. and Südholt, M. *A model and a tool for Event-Based AOP.* Technical report no. 02/11/INFO, École des Mines de Nantes, December 2002.

[DW99] D'Souza, D. and Willis A. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, Reading. MA, 1999.

[DEM02] Duclos, F., Estublier, J. and Morat, P. *Describing and Using Non Functional Aspects in Component Based Applications.* In Proceedings of the 1st international conference on Aspect-oriented software development, Enschede The Netherlands, April 2002.

[DK92] Dunn, M. F. and Knight, J. C. *Certification of Reusable Software Parts.* CS-93-41, available at: http://citeseer.nj.nec.com/michael93certification.html, 1992.

[Ecl03] Eclipse IDE framework. More information available at: http://www.eclipse.org/.

[Far02] Farías, A. *A component model with explicit protocols.* PhD dissertation, Ecole Des Mines De Nantes, December 2003.

[Fil00] Filman, R.E. *Applying aspect-oriented programming to intelligent systems.* Position paper at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns, Cannes France, June 2000.

[FBLL02] Filman, R., E., Barrett, S., Lee, D. and Linden, T. *Inserting Ilities by Controlling Communications.* Communications of the ACM, Vol. 45, No 1, pp. 116-122, January 2002.

[Fle03] Fleury, M et al. JBoss application server. http://www.jboss.org/overview

[FR03] Fleury, M and Reverbel, F. *The JBoss Extensible Server.* In Proceedings of Middleware 2003 Int Conference, Rio de Janeiro, Brazil, LNCS(2672), January 2003.

[GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, ISBN 0-201-63361-2, 1995.

[GS93] Garlan, D. and Shaw, M. *An Introduction to Software Architecture.* Advances in Software Engineering and Knowledge Engineering Volume 2, pp 1-39. New York, NY: World Scientific Press, 1993.

[GB03] Gybels, K. and Brichau, J. *Arranging language features for more robust pattern-based crosscuts.* In Proceedings of international conference on aspect-oriented software development (AOSD), Boston, USA, ISBN 1-58113-660-9, ACM Press, March 2003.

[HC02] Hanenberg, S. and Costanza, P., *Connecting Aspects in AspectJ: Strategies vs. Patterns*, First Workshop on Aspects, Components, and Patterns for Infrastructure Software at 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, April 23, 2002.

[HOU03] Hanenberg, S., Oberschulte, C. and Unland, R. *Refactoring of aspect-oriented software.* In proceedings of Net.ObjectDays 2003, Erfurt, Germany, September 2003.

[HO93] Harrison, W., and Ossher, H. *Subject-Oriented Programming - A Critique of Pure Objects*. Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, September 1993.

[HC01] Heineman, G., T. and Councill, W., T. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, ISBN 0-201-70485-4, 2001.

[HM01] Herrmann, S. and Mezzini, M. *Combining Composition Styles in the Evolvable Language LAC*. In Proceedings of workshop on advanced separation of concerns in software engineering at ICSE, Toronto, Canada, May 2001.

[Hoa85] Hoare, C. A. R., *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol03] Holub, A. *Why extends is evil.* In JavaWorld, August 2003. Available at: http://www.javaworld.com/

[HCU91] Hölzle, U., Chambers, G. and Ungar, D. *Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches.* In Proceedings of ECOOP 1991 international conference, Springer Verlag, June 1991.

[HMU01] Hopcroft, J. E., Motwani, R., and Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*, Second ed. 2001.

[IFC96] Ierusalimschy, R., de Figueiredo, L. H. and Celes, W. *Lua-an extensible extension language*. Software: Practice & Experience, Volume 26(6), pp 635-652, 1996.

[KHH+00] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W., G. *An overview of AspectJ.* In Proceedings of ECOOP'2000, SpringerVerlag, 2000.

[KLM+97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin J.. *Aspect-oriented programming.* In Proceedings of ECOOP'97, pp. 220–242, SpringerVerlag, 1997.

[KPRS00] Klaeren, H., Pulvermuller, E., Rashid, A. and Speck, A. *Aspect Composition applying the Design by Contract Principle.* In Proceedings of the GCSE 2000, Second International Symposium on Generative and Component-Based Software Engineering. Erfurt, Germany, 2000.

[KCA01] Kniesel G., Costanza, P. and Austermann, M. *JMangler - A Framework for Load-Time Transformation of Java Class Files.* IEEE Workshop on Source Code Analysis and Manipulation (SCAM), collocated with International Conference on Software Maintenance (ICSM), November 2001.

[Koo95] Koopmans, P. *On the design and realization of the Sina compiler.* MSc. thesis, Dept. of Computer Science, University of Twente, 1995.

[Lam03] Lambrechts, G. *Het gebruik van JAsCo voor linken van onafhankelijke business-regels in een Object-georienteerde applicatie.* MSc. Thesis, Vrije Universiteit Brussel, Belgium, 2003.

[Law96] Lawson, L. G. *Dynamic Programming Based Learning Methods for Continuous Time Processes Using Neural Architectures.* Clemson University Department of Mathematical Sciences. Gneural Gnome Publishing, Inc, 1996.

[Lie96] Lieberherr, K. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. Available at: www.ccs.neu.edu/research/demeter.

[LH89] Lieberherr, K. and Holland, I. *Assuring Good Style for Object-Oriented Programs*. IEEE Software, pages 38-48., September 1989.

[LL02] Lieberherr, K. and Lorenz, D. *Coupling Aspect-Oriented and Adaptive Programming.* Book chapter under review. Available at: http://www.ccs.neu.edu/research/demeter/biblio/AOPAP.html

[LLM99] Lieberherr, K., Lorenz, D. and Mezini, M. *Programming with Aspectual Components.* Northeastern University Technical Report, NU-CSS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[LO97] Lieberherr, K and Orleans, D. *Preventive Program Maintenance in Demeter/Java* (Research Demonstration). In Proceedings of International Conference of Software Engineering (ICSE), pp. 604-605, 1997.

[LOO01] Lieberherr, K., Orleans, D. and Ovlinger, J. *Aspect-Oriented Programming with Adaptive Methods*. Communications of the ACM, Vol. 44, No. 10, October 2001.

[LLO03] Lieberherr, K., Lorenz, D. and Ovlinger, J. *Aspectual Collaborations: Combining Modules and Aspects.* British Computer Society Journal (Special issue on AOP), Vol 45(5), pp. 542-565, September 2003.

[LH00] Ludwig, A. and Heuzeroth, D. *Metaprogramming in the Large.* In Proceedings of international conference on generic programming and component engineering (GPCE), LNCS 2177, September 2000.

[MATH03] Martin, J., Arsanjani, A., Tarr, P. and Hailpern, B. *Web Services: Promises and Compromises.* Queue (ACM). Vol. 1. No. 1. Pages 48-58. 2003.

[MK03] Masuhara, H. and Kiczales, G. *Modeling Crosscutting in Aspect-Oriented Mechanisms.* In proceedings of ECOOP international conference, Darmstadt, Germany, July 2003.

[MFH01] McDirmid, S., Flatt, M and Hsieh, W. *Jiazzi: New Age Components for Old Fashioned Java.* In the proceedings of OOPSLA 2001 international conference, October 2001.

[MH03] McDirmid, S. and Hsieh. W. *Aspect-Oriented Programming in Jiazzi*. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, March 17-21, 2003.

[MT00] Medvidovic, N. and Taylor, R. *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transaction on Software Engineering, 26 (1), January 2000.

[Mey99] Meyer, B. *Onto components*. In IEEE Computer, Volume 32, January 1999.

[MD03] Meyer J and Downing, T. *Java Virtual Machine*. O'Reilly Press, ISBN 1-56592-194-1, March 1997.

[ML98] Mezini, M and Lieberherr, K. Adaptive plug-and-play components for evolutionary software development. In C. Chambers (editor) Object-Oriented Programming Systems Languages and Applications Conference in Special Issue of SIGPLAN Notices, Vancouver, October 1998.

[MO02] Mezini, M. and Ostermann, K. *Integrating Independent Components with On-Demand Remodularisation.* Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOSPLA), Seattle, Washington, USA, November 4-8, 2002.

[MO03] Mezini, M and Ostermann, K. *Conquering Aspects with Caesar.* In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 90-100, March 17-21, 2003.

[MS98] Mikhajlov, L. and Sekerinski, E. *A study of the fragile base class problem*. Lecture Notes in Computer Science, 1445:355–382, 1998.

[MM03] Miller, J. and Mukerji, J. *MDA Guide.* Object Management Group, version 1.0., 2003.

[OL01] Orleans, D. and Lieberherr, K. *DJ: Dynamic Adaptive Programming in Java.* In Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001

[OT00] Ossher, H. and Tarr, P. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

[OT01] Ossher, H. and Tarr, P. *Using multidimensional separation of concerns to (re)shape evolving software.* Communications of the ACM, Vol. 44, No. 10, pp. 43-50, October 2001.

[Ovl04] Ovlinger, J. *Combining Aspects and Modules*. PhD dissertation, Northeastern University, January 2004.

[Par72] Parnas, D., L. *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM, vol. 15, no. 12, December 1972.

[PDFS01] Pawlak, R., Duchien, L., Florin, G. and Seinturier, L. *JAC : a Flexible Solution for Aspect Oriented Programming in Java.* In Proceedings of Reflection 2001, Kyoto, Japan, September 2001.

[PSDF01] Pawlak, R., Seinturier, L, Duchien, L. and Florin, G. *Dynamic wrappers: handling the composition issue with JAC.* In proceedings of TOOLS-USA 2001, IEEE, Santa-Barbara CA, USA, pages 56-65, August 2001.

[PFFT03] Pinto, M., Fuentes, L. Fayad M. E. and Troya, J.M. J.M. *Separation of Coordination in a Dynamic Aspect-Oriented Framework*. In proceedings of the 1st International Conference on Aspect-Oriented Software Development, April 2002, The Netherlands.

[PFT03] Pinto, M., Fuentes, L. and Troya, J.M. *DAOP-ADL: an architecture description language for dynamic component and aspect-based development.* Proceedings of the second international conference on Generative programming and component engineering. LNCS, Erfurt, Germany, September 2003.

[PAG03] Popovici, A., Alonso, G. and Gross T. *Just in Time Aspects: Efficient Dynamic Weaving for Java.* In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, Boston, USA, pp 100-109, ACM Press, 2003

[PSC+01] Pulvermüller, E., Speck, A., Coplien, J.O., D'Hondt, M. and De Meuter, W. Proceedings of Workshop on "feature interaction in composed systems" at ECOOP 2001. Available at: http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/

[RFS+00] Reid, A., Flatt, M., Stoller, L., Lepreau, J. and Eide, E. *Knit: Component Composition for Systems Software.* In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), October 2000.

[Reu03] Reussner, R. H.. *Automatic Component Protocol Adaptation with the CoCoNut Tool Suite*. In proceedings of Future Generation Computer Systems, Elsevier, The Netherlands, 2003.

[Ros97] Ross, R. *The Business Rule Book: Classifying, Defining and Modeling Rules, Second Edition*. Business Rule Solutions, LLC, 1997.

[Sal01] Salinas, P. *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*. MSc. Thesis, Vrije Universiteit Brussel, Belgium, 2001.

[SCT03] Sato, Y., Chiba, S. and Tatsubori, M. *A Selective, Just-In-Time Aspect Weaver.* In Proceedings of 2nd International Conference on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp.189-208, Springer-Verlag, 2003.

[SVSJ03] Schollaert, P., Vanderperren, W., Suvee, D. and Jonckers, V. *Online reconfiguration of component based applications in PacoSuite.* Software Composition workshop @ ETAPS 2003, Warshaw, Poland, April 2003. Also published in Electronic Notes in Theoretical Computer Science, Vol 82(5), 2003.

[SD03] Serini, D. and De Moor, O. *Static analysis of aspects.* In proceedings of the 2nd international conference on Aspect-oriented software development (AOSD), USA, pp 30-39, ISBN 1-58113-660-9, ACM Press, March 2003.

[SG96] Shaw M. and Garlan D. *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, ISBN: 0-13-182957-2, 1996.

[Sho97] Short, K. *Component Based Development and Object Modeling.* Texas Instruments Software, 1997.

[SL02] Sung J. and Lieberherr, K. *DAJ: A Case Study of Extending AspectJ*. Northeastern University Technical Report NU-CCS-02-16, 2002. Available at: http://www.ccs.neu.edu/research/demeter/papers/publications.html

[Suv02] Suvee, D. *Java Byte code editor and library*. Apprenticeship assignment at SSEL, December 2002. Available at: http://ssel.vub.ac.be/Members/dsuvee/jbe/index.htm

[Suv03] Suvee, D. *FuseJ: Achieving a Symbiosis between Aspects and Components.* In proceedings of the 5th GPCE Young Researchers Workshop. Erfurt, Germany, September 2003.

[SVW04] Suvee, D., Vanderperren, W. and Wagelaar, D. *There are no Aspects.* In Proceedings of SC 2004, Barcelona, Spain, April 2004.

[SVJ03] Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development.* In Proceedings of international conference on aspect-oriented software development (AOSD), Boston, USA, pp 21-29, ISBN 1-58113-660-9, ACM Press, March 2003.

[Szy96] Szyperski, C. *Independently extensible systems. Software engineering potential and challenges.* Proceedings of the 19th Australasian Computer Science Conference Melbourne, Australia, February 1996.

[Szy98] Szyperski, C. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, ISBN 0-201-17888-5, 1998.

[Szy01] Szyperski, C. *Components and Web Services.* Beyond Objects column, Software Development. Vol. 9, No. 8, August 2001.

[THO+00] Tarr, P., Harrison, W., Ossher, H., Finkelstein, A., Nuseibeh, B. and Perry, D. *Workshop report of the workshop on Multi-Dimensional Separation of Concerns in Software Engineering.* In proceedings of international conference of on Software Engineering (ICSE), ACM Press, 2000.

[TOHS99] Tarr, P., Ossher, H., Harrison, W., Sutton, S., M. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In proceedings of international conference on Software Engineering (ICSE), ACM Press, 1999.

[TKVV04] Tourwe, T., Kellens, A., Vanderperren, W. and Vannieuwenhuyse, F. *Inductively Generated Pointcuts to Support Refactoring to Aspects*. In proceedings of SPLAT workshop at AOSD, Lancaster, UK, March 2004.

[Van01] Vanderperren, W. *Applying aspect-oriented programming ideas in a component based context: Composition Adapters.* In Proceedings of NetObjectDays 2001, Erfurt, Germany, pp. 201-206, ISBN 3-00-008419-3, September 2001.

[Van02a] Vanderperren, W. *A pattern based approach to separate tangled concerns in component based development.* In Proceedings of ACP4IS workshop at AOSD 2002, Enschede, The Netherlands, April 2002.

[Van02b] Vanderperren, W. *Localizing crosscutting concerns in visual component based development.* In Proceedings of Software Engineering Research and Practice International Conference (SERP), Las Vegas, USA, pp. 138-145, ISBN 1-892512-99-8, CSREA Press, June 2002.

[VS03] Vanderperren, W. and Suvee, D. *Integrating aspect-oriented ideas into component based software development.* Dutch-Belgian AOSD worksop, Enschede, The Netherlands, January 2002.

[VS04a] Vanderperren, W. and Suvee, D. *JAsCoAP: Adaptive Programming for Component-Based Software Engineering.* In proceedings of Third International AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (Short Paper), Lancaster, UK, March 2004.

[VS04b] Vanderperren, W. and Suvee, D. *Optimizing JAsCo dynamic AOP through HotSwap and Jutta.* In proceedings of Dynamic Aspects Workshop (DAW), Lancaster, UK, March 2004.

[VSJ03a] Vanderperren, W., Suvee, D. and Jonckers, V. *Invasive Composition Adapters: an aspect-oriented approach for visual component-based development.* In proceedings of the ACP4IS workshop at AOSD 2003, March 2003.

[VSJ03b] Vanderperren, W., Suvee, D. and Jonckers, V. *Automatic Feature Interaction Analysis in PacoSuite.* In Proceedings of SCI 2003 International Conference, USA, pp. 382-387, ISBN 980-6560-01-9, July 2003.

[VSJ03c] Vanderperren, W., Suvee, D. and Jonckers, V. *Combining AOSD and CBSD in PacoSuite through Invasive Composition Adapters and JAsCo.* In Proceedings of Node 2003 international conference, Erfurt, Germany, pp. 36-50, ISBN 3-9808628-2-8, September 2003.

[VSJ05] Vanderperren, W., Suvee, D. and Jonckers, V. *PacoSuite&JAsCo: Combining Aspect-Oriented and Component-Based Software Engineering.* Special Session of the International Journal on SoftwareTools for Technology Transfer (STTT), Springer-Verlag, 2005.

[VSVC04] Vanderperren, W., Suvee, D., Verheecke, B. and Cibran, M.A. JAsCo&WSML: *AOP for Component-Based Software Engineering applied to a Web Services Management Layer.* Formal Research Demo at AOSD 2004, Lancaster, UK, March 2004.

[VSWJ03] Vanderperren, W., Suvee, D., Wydaeghe, B. and Jonckers, V. *PacoSuite & JAsCo: A visual component composition environment with advanced aspect separation features.* In Proceedings of International Conference on fundamentals of Software Engineering (FASE), Warshaw, Poland, pp. 166-169, ISBN 3-540-00899-3, Lecture Notes in Computer Science, Vol 2621, April 2003.

[VW01] Vanderperren, W. and Wydaeghe, B. *Towards a New Component Composition Process.* In Proceedings of ECBS 2001 Int Conf, Washington, USA, April 2001.

[VW02] Vanderperren, W. and Wydaeghe, B. *Separating concerns in a high-level component-based context.* EasyComp workshop @ ETAPS 2002, Grénoble, France, April 2002. Also published in Electronic Notes in Theoretical Computer Science, Vol 65(4), ISBN 0-4441-2216, 2002.

[VDD01] Vanhaute, B., De Win, B. and De Decker B. *Building Frameworks in AspectJ.* Workshop on Advanced Separation of Concerns (L. Bergmans, M. Glandrup, J. Brichau and S. Clarke, eds.), pp. 1-6, 2001.

[VC03] Verheecke, B. and Cibran, M. A. *AOP for Dynamic Configuration and Management of Web services in Client-Applications.* In Proceedings of 2003 International Conference on Web Services - Europe (ICWS'03-Europe), Erfurt, Germany, September 2003.

[VCV+04] Verheecke, B., Cibran, M., Vanderperren, W., Suvee, D. and Jonckers, V. *AOP for Dynamic Configuration and Management of Web Services.* International Journal of Web Services Research (JWSR), 1(3), 25-41, July-Sept 2004.

[VVSJ03] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V. *JAsCo.NET: Unraveling Crosscutting Concerns in .NET Web Services.* In Proceedings of Second Nordic Conference on Web Services NCWS'03, Vaxjo, Sweden., November 2003.

[Wag03] Wagelaar, D. *Towards a Context-Driven Development Framework for Ambient Intelligence.* In Proceedings of ICDCS'2004, Tokyo, Japan, March 2004.

[WJ03] Wagelaar, D. and Jonckers, V. *A Concept-Based Approach to Software Design.* In proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003), ISBN/ISSN: 0-88986-394-6, Marina del Rey, USA, November 2003.

[Wic99] Wichman, J., C. *The Development of a Preprocessor to Facilitate Composition Filters in the Java Language*. MSc. thesis, Dept. of Computer Science, University of Twente, 1999.

[Wyd01] Wydaeghe, B. *PACOSUITE: Component Composition Based on Composition Patterns and Usage Scenarios*. PhD dissertation, VUB, November 2001.

[WV01] Wydaeghe, B. and Vanderperren, W. *Visual Component Composition Using Composition Patterns.* In Proceedings of Tools 2001 Int Conf, Santa Barbara, USA, pp. 120-129, ISBN 0-7695-1251-8, IEEE Computer Society, July 2001.

[WVPW02] Wydaeghe, B., Vanderperren, W., Pijpops, T. and Westerhuis, F. *Adapsis: Adaptation of IP Services Based on Profiles.* SSEL technical report, May 2002.