Semantics Preservation in Model-based Composition

by Jon Oldevik

THESIS

submitted to Department of Informatics
Faculty of Mathematics and Natural Sciences,
University of Oslo
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (PhD)

December 2009

© Jon Oldevik, 2010

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 938

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Inger Sandved Anfinsen.
Printed in Norway: AiT e-dit AS.

Produced in co-operation with Unipub.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

Separation of concerns is an important factor in the development of complex software systems. Being able to reason about system concerns in isolation and compose them to a whole are key elements for succeeding with the specification and implementation of such systems. In software product line engineering, management of features is an essential activity in the product development process. Features represent concerns, or parts of concerns, which are composed into products.

The mechanisms supporting separation of concerns become increasingly more sophisticated, e.g. through aspect-oriented tools and techniques featuring flexible composition of crosscutting concerns in programming and modelling.

Composition of concerns do not come without challenges; in current aspect-oriented composition languages, the syntactic nature of pointcut expressions leads to vulnerable relationships between aspects and base systems. When multiple concerns are involved, in the form of an aspect-oriented design, a product line feature design, or other, these may be in conflict with each other. In many cases, the state-of-the-art composition technologies leave little control for constraining the effects imposed on a system by compositions.

This thesis defines theoretical and practical solutions for detecting and solving conflicting or problematic situations when composing software systems. First, it gives a definition for semantics preservation for sequence diagram aspect composition, which is a tool that helps ensuring consistent application of scenario – or trace-based – aspects even if the base model changes. Second, it defines mechanisms for composing and analysing product line features, which help toward ensuring consistency of feature composition by addressing confluence and conflict situations. Third, it defines a technique for defining and associating composition contracts with models. It allows constraining the model composition by pre- and post-conditions, which gives increased control over changes that can be imposed by model composition, e.g. by aspects.

These aspects all contribute toward the overall goal of the thesis – *semantics preservation of systems during model composition.*

Acknowledgements

The work represented by this thesis was made possible thanks to the people at and the funding from Department of Informatics, University of Oslo and SINTEF. The work has been done in the context of the SWAT project (Semantics-preserving Weaving - Advancing the Technology), funded by the Norwegian Research Council (project number 167172/V30).

I extend enormous gratitude toward my principle adviser, Øystein Haugen, who guided me safely through the process in a collaborative and constructive manner, and to my secondary advisers, Stein Krogdahl and Birger Møller-Pedersen for their support and guidance in the process. I also extend gratitude toward my fellow scholars on the SWAT project, Roy Grønmo, Frederik Sørensen, and Eyvind W. Axelsen for participating in discussions and collaboration along the way.

I want to thank Bjørn Skjellaug, the research director of my department in SINTEF ICT, Cooperative and Trusted Systems, for encouragement during the initiation and fulfilment of this PhD, and my colleagues at SINTEF for their valuable collaboration in projects and discussions.

I am highly grateful for the courtesy shown by Ingolf H. Krüger and his research team at University of California, San Diego (UCSD), by giving me the opportunity for an inspirational year of collaboration.

Most of all, thanks to my loved ones, Hege, Ellef and Eva, who have encouraged me unconditionally along the way, and ensured a good balance of the constituents of life.

Contents

List of Figures

Part I

Overview

Chapter 1

Introduction

An important feature in software engineering is the ability to separate concerns during development, i.e. being able to conceptually focus on *one* aspect of the system while ignoring others, and then being able to *compose* these concerns to a coherent whole. Languages and methodologies provide developers with tools aiding the separation of concerns (SoC) process.

In this thesis, we address how composition – and model composition in particular – is used for providing SoC in software engineering, and how semantics of models is influenced when they are subject to composition. Our goal is *semantics preservation* in the context of composition.

As system complexity is continuously increasing, so are the demands for flexible languages and tools for system design, implementation, and run-time. This is partly addressed by static and dynamic composition mechanisms that can be used to compose systems from concerns. These can for example be applied in the context of service-oriented, web-based, or embedded systems to support service composition, adaption, and configuration. There is, however, a trade-off between flexibility and control; with increased flexibility of composition mechanisms, comes an increased risk of losing control over the effects they have on the system.

Aspect-orientation has become a popular mechanism for concern composition, as seen in AspectJ[1], the enterprise Java framework Spring[2], or in enterprise service bus (ESB) platforms such as Mule[3]. In aspect-orientation, composition is commonly specified by combinations of selection mechanisms called *pointcut* and insertion directives called *advice*. *Pointcuts* specify *where* in a base specification or implementation new elements, such as code, shall be composed, or inserted. *Advice* defines what the new elements to be inserted are, e.g. *crosscutting* transactional code. *Crosscutting* means that the same *concern* (e.g. transaction) is required many places in the system specification or implementation, e.g. there may be many methods that use the same transaction code.

With such flexible composition mechanisms, it can be difficult to manage and control what happens to existing models or code, and there is a risk that assumptions made by the original developers are broken.

In this thesis, we give a definition of semantics preservation in the context of aspect-oriented composition of sequence diagrams. This definition provides a way of reasoning

[1] http://www.eclipse.org/aspectj/
[2] http://www.springsource.org/
[3] http://www.mulesource.org

about compositions of behaviours and their ability to preserve semantics. The definition is also applied to trace-based aspect at the programming level, giving a mechanism for reasoning about semantics preservation of aspect compositions in Java. This approach to semantics preservation is one way of controlling change imposed by aspect compositions. We also provide a more general approach by means of *composition contracts*. Composition contracts use the ideas from Design by Contract (DbC) [123] by associating contracts with models. These act as extended interfaces that define what kind of modifications that compositions are allowed to make.

Feature composition is an inherent property of product line engineering (PLE). In PLE, products are commonly specified in terms of their *features*, which represent functional or non-functional properties of the product. Some of these features are *common* for all products in the product line, while others are *variable*, i.e. they occur only in some products. The product definition process involves a selection – or resolution – of the set of features that shall be part of a product. The feature resolutions result in transformation – or composition – of features in the context of the product model.

The set of features in a product line may have dependencies between them; we say that they *interact*. Some feature interactions are conflicts, which means that the features cannot be part of the same product configuration. Other interactions may require a certain ordering of the feature resolutions. We address how conflicting feature interactions between can be detected and avoided by analysing *confluence properties* of the feature composition. Confluence theory stems from *term rewriting* [165], and implies that transformations, or feature compositions, can be applied *order-independent* and yield the *same result*.

Together, our contributions provide technologies that can partake in making SoC with composition less error prone and easier to control for the developer. Figure 1.1 illustrates the contributions of the thesis.

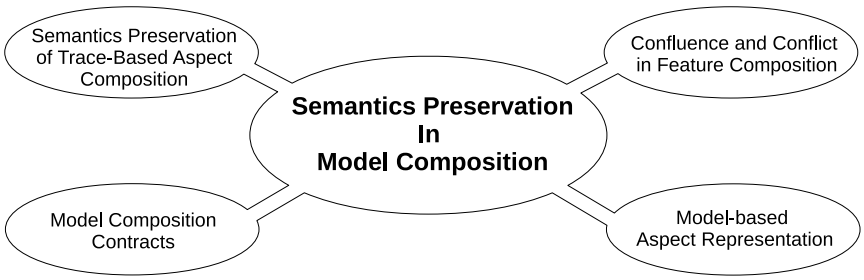


Figure 1.1: *Thesis contribution overview*

The work in this thesis has been done in the context of the SWAT[4] project. SWAT is a research project that focuses on how mechanism for software composition can be improved by taking into account semantics, and how composition can be safeguarded against changes that might break the system or change it in incomprehensible ways.

No single technology is likely to provide the optimal and general solution for any particular problem, i.e. there is no *silver bullet* for software engineering problems [29]. Rather than searching for the silver bullet, software engineers should continue to invest

[4]Semantics-preserving Weaving - Advancing the Technology, Norwegian Research Council project 167172/V30

in a technology tool box with specialised tools and methods tailored for specific needs. The contributions of this thesis provide pieces of this tool box.

1.1 Contribution Overview and Motivation

Existing software engineering techniques, such as aspect-oriented programming and modelling, provide SoC by allowing developers to construct software by composing concerns statically or dynamically to provide complete systems. The process of composition, however, may be error prone, and lead to less than optimal systems, systems that do not work as expected, or even non-working systems.

In order to meet this challenge, the system and its specification – and its stakeholders – would benefit from mechanisms that help protecting the system from unintentional modifications. Of course, one cannot expect to foresee all future changes for a system, but one can hope to establish a system specification with a certain level of trust that its semantics – or part of its semantics – will be stable throughout the system lifetime. We call this notion of trust *semantics preservation*.

We say that a system specification is *semantics preserving with respect to composition* if selected semantic characteristics of that specification are kept during composition. The characteristics that scope semantics preservation may vary between semantic domains – defined by languages – and the usage of those languages in application domains. For example, semantics preservation for sequence diagrams with respect to aspect composition may be defined based on the set of *traces* they represent and that events in these may not be removed.

Our work on semantics preservation in model composition is addressed through these complementary focus areas:

- definition and application of semantics preservation for compositions of trace-based specifications, such as sequence diagrams,
- consistent feature compositions supported by confluence and conflict analysis,
- composition contracts that govern modifications imposed to models by compositions, and
- representation mechanisms for aspect-oriented models.

The challenges in these areas are detailed further below.

1.1.1 Semantics Preservation of Trace-Based Aspect Composition

UML Sequence Diagrams [134] and Message Sequence Charts (MSC) [91] are languages for specifying interactions between roles, or components, in a system using message passing between system roles. Sequence diagrams represent behavioural scenarios and are often used to specify the important, typical, and prohibited behaviours of a system. They are excellent for specifying inter-component protocols or service interactions in service-oriented systems or detailing of requirements.

Using aspect-oriented mechanisms for SoC at the sequence diagram level to modularise crosscutting behaviours can have many usage areas, such as providing error detection and mitigation, security, or data conversion.

When a sequence diagram is modified by an aspect, the behaviours – and hence the semantics – it specifies will change. In many cases, however, it is desirable to preserve whole or parts of the original semantics. Figure 1.2 illustrates – with a car crash detection scenario – how we address semantics preservation in the context of sequence diagram aspect composition: *when a sequence diagram is modified, can we know the effect existing aspects have on that modified model?*

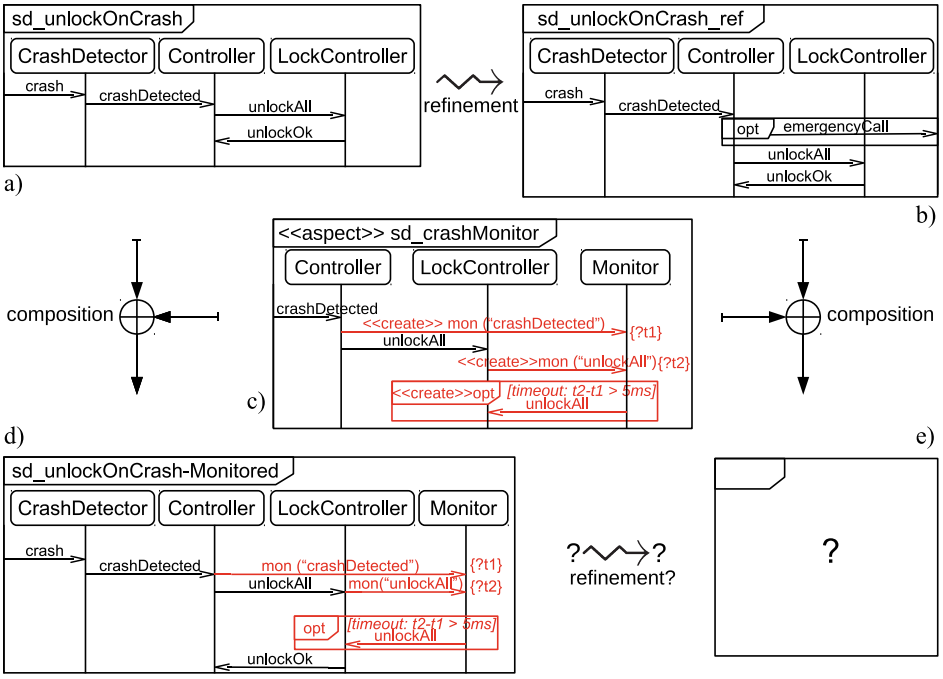


Figure 1.2: *A crash detection scenario is defined in a) and refined in b). Will the crash monitoring aspect in c) have the intended effect on the modified scenario?*

The crash detection monitor aspect in Figure 1.2 c) looks for the message *crashDetected* followed by *unlockAll* and inserts a *mon* call to a *monitor* for each of these. The monitor times the incoming events, and in case of a timeout, guarded by the expression *[t2-t1 > 5ms]*, the monitor triggers an *unlockAll* on the *LockController*.

For the example in Figure 1.2, the aspect *sd_crashMonitor* in c) will have an effect on the sequence diagram *sd_unlockOnCrash* in a), since the message sequence (or trace) defined by the aspect pointcut will be matched. The effect is illustrated in d). In the refinement *sd_unlockOnCrash_ref* in b), however, it is not obvious if there will be a match; it depends on the matching semantics used. Hence, the effect of the aspect composition is uncertain, as illustrated in e). Matching based on the syntactic elements of the diagram will not find a match, while matching based on the semantics of the diagram will find a match. Even if the aspect has an effect, the relationship between

the composed models in d) and e) is uncertain. *Will the refinement relationship between the base models in a) and b) also hold between the composed models in d) and e)?*

The problem arises when a system specification or implementation is modified in some way; we cannot know if existing aspects will still have the intended effect after the change or if semantic relationships such as refinement will hold after a composition.

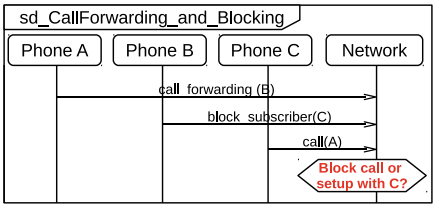
To this end, we describe an approach for reasoning about semantics preservation of sequence diagrams during aspect composition; we give a definition for semantics preservation of sequence diagram aspects and show how different sequence diagram aspect approaches meet this definition. We map these ideas to trace-based semantics in Java and show how this definition of semantics preservation can be applied in the context of trace-based aspects in Java.

1.1.2 Confluence and Conflict in Feature Composition

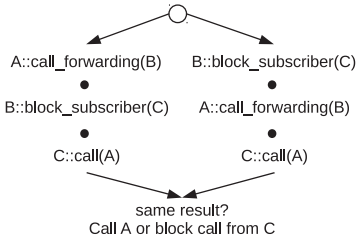
The features of a software product represent its functional or quality characteristics. It can be further related to system-level artifacts, e.g. models, architectural or design elements, or code.

The construction of a product from a product line requires selection of the features that define the product. A composition of the associated system level artifacts has to be done in order to build a *product configuration*.

Since features may depend on other features or be conflicting, the order by which features are *resolved* and *composed* may be important for ensuring that valid product configurations are defined. Being able to detect potential inconsistencies in a resolution scenario e.g. by analysing confluence of feature resolution transformations, is valuable for increasing the quality of the product line construction process. Figure 1.3 illustrates the problem using an example from the telecommunication domain, where the features *call_forwarding* and *block_subscriber* interact and cause a potential conflict situation: *should the call from C to A, which is forwarded to B who blocks C, be blocked or forwarded?*



(a) Feature interaction conflict in a telecommunication scenario.



(b) Feature composition confluence

Figure 1.3: *Conflict between call forwarding and call blocking features in a telecommunication network 1.3(a), and illustration of confluence 1.3(b)*

Confluence provides a kind of measure for the compositional process with respect to SoC: As separated concerns establish *independence* between them, they can be analysed and understood in separation from the rest of the system; this may in turn facilitate *local reasoning* of concerns. *Confluent composition* systems imply *independence* between individual composition steps, i.e. between composition of concerns, or

features. This implies that the involved composition steps – or feature resolutions – can be done order-independently. Hence, ensuring confluence implies that concerns are well separated and can be freely composed.

Another example that illustrates the importance of detecting potential conflicts is shown in Figure 1.4. Here, the conflict is due to the mechanical effects of the system (the car): the cruise control feature attempts to accelerate the car to maintain speed, whereas the Electronic Stability Program (ESP), potentially at the same time, tries to decelerate the car after detecting a skid. They both affect the *speed* of the car, but with contradicting goals.

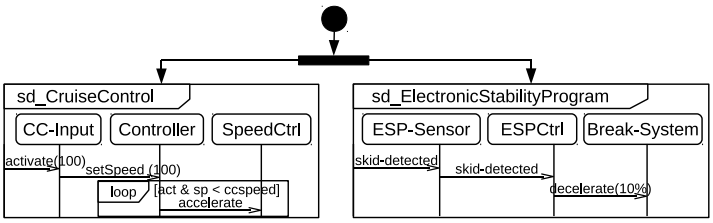


Figure 1.4: *Conflicts between cruise control and ESP features.*

We address the confluence of product line feature composition and show how potential conflicts between features can be analysed.

1.1.3 Model Composition Contracts

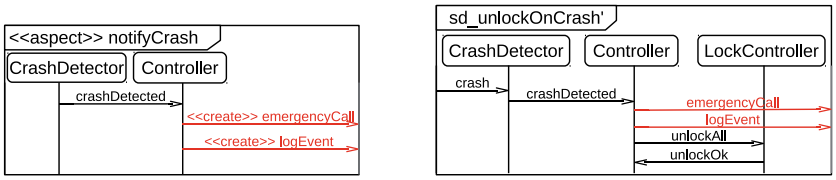
State-of-the-art aspect-oriented mechanisms are powerful and flexible. They allow concerns to be modularised in aspects and help removing *tangling* and *scattering*, i.e. the spreading of concerns throughout the system specification or implementation. The use of these mechanisms may, however, have problematic side-effects that reduce system robustness: *fragile pointcuts* is one problem that makes the modularised aspects vulnerable to base model evolution; *aspect interaction* is another problem, which require careful consideration of the ordering of aspect compositions.

An orthogonal problem is that the aspect-oriented mechanisms are hard to control from the point of view of the base system developer, since there is little or no control over what kind of changes an aspect can impose on the base model. For example, in AspectJ, an aspect can override private methods using *around advice*, which may be in opposition to the assumptions made by the base system owner.

Figure 1.5 illustrates how the *car crash scenario* (from Figure 1.2a) has been modified by an aspect (Figure 1.5(a)), which leads to undesired behaviour that would be advantageous to prevent (Figure 1.5(b)), since new behaviour is inserted that may delay the unlocking of the doors.

If we are able to specify constraints on the effects that aspects can impose on a base model or program and enforce these constraints during composition, we can gain increased confidence that compositions produce reliable systems that work as expected.

We introduce an approach called *Composition by Contract (CbC)* to control model composition, inspired by the Design by Contract methodology from the Eiffel language [123]. The idea of CbC is that models can be associated with composition contracts that govern the eligibility for compositions to change the model. The con-



(a) Crash detection notification aspect (b) Modified crash detection

Figure 1.5: *The crash detection scenario described earlier in Figure 1.2 is modified by the aspect in Figure 1.5(a). It inserts event logging and an emergency call, resulting in undesired behaviour in Figure 1.5(b), because it may delay the unlocking of doors in an emergency situation.*

tracts define the modifications that are allowed by compositions and hence insulate models from unintended and potentially harmful changes.

1.1.4 Model-based Aspect Representation

Allen and Garlan [4] define *software architecture* as a collection of computational *components* together with a collection of *connectors* that describe the interaction of the components. They identify three properties an expressive notation for connectors should have: it should allow specification of common types of communication, such as procedure calls, pipes, and event broadcast, it should allow describing complex interactions between components, and it should allow to make distinctions on connector variations. Connectors in UML have been argued to be too simple to express these kinds of architectural connectors [9].

We device an aspect representation for architectural models in UML that provide *architectural variability* with respect to connectors. It allows complex connector structures to be specified separately and be inserted into an architectural specification in many places.

We also introduce aspect representation for sequence diagram aspects in UML as support for the works on trace-based Java aspects and composition contracts.

1.2 Thesis Structure

This thesis is delivered as a collection of papers with an accompanying overview, and is divided in two main parts. Part I contains the overview, which gives the motivation, background, and overview to the work done. Part II is the main contribution in the form of a set of papers.

Part I contains the following chapters in addition to this introductory chapter.

- In Chapter 2, we give an overview of the background for the work done.
- In Chapter 3, we elaborate the problem area, motivate the work, and define the research topics investigated in this thesis.
- In Chapter 4, we describe the research methods applied in the course of this PhD work.

- In Chapter 5, we give a review of the literature and state-of-the-art.
- In Chapter 6, we give an overview of our contribution and our research papers.
- In Chapter 7, we discuss and evaluate the accomplished work.
- In Chapter 8, we conclude and give some directions for future work.

Part II contains six papers in Appendices A through F, which define the main contribution of this thesis.

Chapter 2

Background

2.1 Introduction

Software has become a part of everyday life, and advanced computer programs are found within cars, cell phones, toys, etc; services that were previously handled by people are being replaced by software services, such as airline booking systems or on-line banking systems. More and more, society depends on software in its operation. The systems are becoming more complex to accommodate concerns such as distribution, security, availability, and performance. New styles of computing, such as *cloud computing* with focus on offering software, infrastructure, and platforms as services, have emerged to accommodate users with high demand on data throughput, response time, and no down time. In service-oriented architectures, such as provided by *Enterprise Service Bus (ESB)* infrastructures, flexible web service technologies are combined with horisontal enterprise services.

As the technologies for implementing systems evolve, the basic principles for software engineering prevail, such as the ability to see abstractions, separating problems into concerns, and applying appropriate methodology. In this work, the focus is on how separate concerns in software programs can be composed, and how such compositions may preserve semantics of its parts within the whole.

In this Chapter, we give a brief background to the problem domain. A more detailed view is given in Chapter 5.

2.2 Separation of Concerns and Modularisation

Separation of concerns (SoC) and *modularisation* are well established strategies for managing complex specifications [48, 141]. It is commonly accepted that the term SoC was coined by Edsger W.Dijkstra in his paper 'On the role of scientific thought' [48]. In that essay, Dijkstra focuses on the importance for software engineers to be able to put focus on individual concerns, such as correctness, without having to think about all other concerns at the same time. Parnas [141] addresses the importance of obtaining a modularisation of the problem that is appropriate for that particular problem.

Object-oriented language mechanisms such as classes, interfaces, and packages provide one way of modularising software according to object-oriented principles. However, it may be difficult to separate the different concerns from each other. This leads to scattering of concerns across many modules and tangling of multiple concerns within

one module (e.g. a single class). The language and the methodology force the developer to focus on one particular dimension when decomposing the system (such as the class dimension) – this problem has been coined *the tyranny of the dominant decomposition* [164].

Aspect-orientation is currently one of the dominating technology trends addressing SoC and composition of concerns both in the modelling and implementation space.

Aspect-Oriented Development. Although software is designed with SoC in mind, the language mechanisms at hand often lead to *scattering* and *tangling* of concerns. Scattering occurs when a concern is defined across multiple artifacts (e.g. classes). Tangling occurs when a concern is not isolated, but intertwined with other concerns. This is illustrated in Figure 2.1.

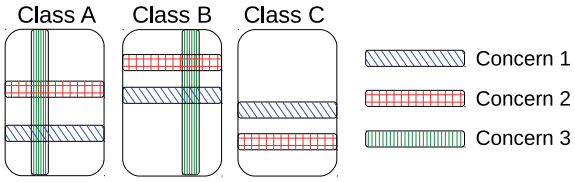


Figure 2.1: *Concerns 1,2, and 3 are scattered across classes A,B, and C. Concern 3 is tangled with concerns 1 and 2 in classes A and B.*

This has motivated a range of language extensions to support concern specification and composition, such as aspects [103] and subjects [33] in programming and modelling. These approaches modularise crosscutting concerns into units, e.g. aspects, that can later be composed by a transformation process commonly referred to as weaving.

Aspect-Oriented Development (AOD) came out of several research efforts, notably the law of Demeter and adaptive programming by Lieberherr et.al. at Northeastern University [120], composition filters by Aksit et.al. at University of Twente [1], subject-oriented programming by Harrison and Ossher at IBM Watson Research Center [76], and aspect-oriented programming and AspectJ by Kiczales et.al. at Xerox PARC [104].

The law of Demeter [120], also known as the *principle of least knowledge*, is a programming style that increases modularity by restricting who an object should talk to; a method M in an object O is only allowed to invoke methods on O itself, objects passed as parameters to M, or objects owned by (as state) O. Method invocations to objects returned by other methods should be avoided. The *object-composition filter* approach [1] was defined as an object-oriented abstraction for database management. It defines filters as first class parts of class definitions. Filters are defined by filter handlers and accept set functions, which determine if a given invocation is accepted, and how to dispatch the invocation. The approach supports multiple views on objects and various database-specific functionality. *Subject-oriented programming* [76] defined subjects as a new programming model where views are specified as subjects with their own state and behaviour, which are composed when activated in the context of a particular object. That work later led to the work on multi-dimensional separation of concerns and hyperslices [164]. The composition semantics, e.g. that behaviour should be *merged*, is specified as part of the composition. *AspectJ* [104] defined *aspects* as a modularising unit where pieces of crosscutting code are defined by advice, which is inserted into locations in the base code that are specified by so-called pointcuts.

Aspect-oriented development, or aspect-oriented software engineering, has emerged into many sub-domains of software engineering. Within analysis and design, so-called *early aspects* and *aspect modelling* have been research topics for several years, especially within targeted workshops on early aspects[1] and aspect modelling[2], resulting in notations, semantics, and tools for aspect-oriented requirements, architecture, and design modelling. Various aspect-oriented development methods have appeared, such as aspect-oriented use case development by Jacobson and Ng [94], the *Theme* approach by Clarke and Baniassad [33], aspect-oriented requirements engineering by Rashid et.al. [144], and aspect-oriented design by France et.al. [62].

Aspect-oriented – or concern-oriented – approaches are often categorised by how their concerns reference other artifacts, e.g. other concerns or a base program or model. In *symmetric* approaches, concerns are symmetrically associated with each other by some type of composition operator, such as merging. In *asymmetric* approaches, concerns are composed with, or inserted into a base – or primary – model. The Hyperslice concept (by Tarr et.al. [164]) is an example of the former, while AspectJ is an example of the latter.

Aspect-orientation in the asymmetric sense has brought with it some characterising terms:

- a *join point* generally refers to elements in the semantic model of the base language, which can be addressed and modified by an aspect. The set of join points is often referred to as a *join point model*. In an aspect-oriented programming language, such as AspectJ, valid join points are method calls or executions, exception handlers, or field access. In a modelling language, these may be any element defined by the language meta-model.

- a *pointcut* is a selection mechanism – or query – that references a set of join points, i.e. it identifies points – or locations – in the base program, or model, which can be modified by aspect advice.

- an *advice* defines the elements that are composed with, or inserted into the base model or program at identified pointcuts, e.g. a piece of code that executes before, after, or instead of existing base code.

- *quantification* refers to the ability to establish one-to-many relationships between aspects and join points in the base model, such that the same advice can be applied at several join points.

- *obliviousness* refers to a base model or program's unawareness of any aspect that may modify that base model or program. Filman and Friedman [57] argued that quantification and obliviousness are the defining characteristics of AOP.

The aspect-oriented mechanisms are powerful and flexible modularisation tools if used correctly. These flexible composition mechanisms do, however, not come without a cost. The detailed knowledge that aspects have of the internals of the base system makes them fragile to base system changes. This problem is known as the *fragile pointcut problem* [112], and has been addressed by works on more robust and semantic

[1] Early Aspects Workshop - http://www.early-aspects.net/

[2] Aspect-Oriented Modelling Workshop - http://www.aspect-modeling.org/

pointcut models, for example by Ostermann et.al. [140]. Aspect interaction, or interference, is another challenge facing software engineers when several aspects interfere with each other such that the composition result depends on the order by which the aspects are composed [51].

2.3 Model-Driven Engineering

Model-Driven Engineering (MDE) – aka Model-Driven Development (MDD) or Model-Driven Architecture (MDA) – is a discipline in which abstract, often graphical models are used as a basis in the software engineering process. Models can be refined toward more detailed, platform-near representations in manual or automatic transformations until implementation code and deployment information is produced.

Model-driven engineering has been around for a long time. Graphical models for database design were proposed in the mid 1970's e.g. in Chens work on the entity relationship (ER) model [30]. In 1976, the first standard for graphical specification of telecommunication systems, ITU-T Specification and Description Language (SDL) [90], was defined. Methods for object-oriented design started appearing in the mid 1980's and early 1990s, e.g. by Booch [25], Wirfs-Brock et.al. [179], Coad and Yourdon [35], Rumbaugh et.al. [148], and Jacobson [93].

In the mid 1990's, the Unified Modeling Language (UML) [134] was defined in a melting-pot process that attempted to incorporate the best-of-breed approaches at the time. It was standardised in its first version in 1997 within the Object Management Group (OMG). Since then, UML has continuously evolved to improve its design and cater for new requirements, and is currently in version 2.2, with version 2.3 on its way. UML is a general-purpose language that supports different facets of software engineering processes through different, interrelated sub languages: classes, components, composite structures, deployments, use cases, interactions, state machines, and activities. (When referring to UML in the following, we mean UML 2.x)

UML has become the de-facto modelling language for software analysis and design, and a natural choice for software development projects. The generality and size of UML, however, is not always found suitable for specific domains. Therefore, specific domains define their own, domain-specific language (DSL). An example is a language defined for architectural specification within the real-time embedded systems domain, the *Architecture Analysis and Design Language (AADL)* [136].

Both UML and DSLs have their strengths and weaknesses [60, 86]. UML provides a standard, general purpose notation, comes with a standard exchange format, and is supported by different open source and commercial tools. With DSLs, the domain knowledge is built into the language. The development and use of DSLs have become more widespread with open source tools like Eclipse/Graphical Modeling Framework (EMF/GMF) [119]. In practise, UML is often used as a DSL, by providing domain-specific extensions through the stereotype extension mechanisms provided by the language. The *Modeling and Analysis of Real-time and Embedded systems (MARTE)* [135] standard is an example of a profile meeting the requirements of the real-time and embedded systems domain.

To support the refinement steps in a model driven process, generative techniques and model transformation tools can be used. This is part of the vision of the Model Driven Architecture (MDA) [128] framework defined by OMG, where central elements

are the meta-model standard – Meta Object Facility (MOF) [129], UML, and standards for model to model transformation (MOF Query, View, Transformation – QVT) and model to text transformation (MOF Model to Text).

Figure 2.2 illustrates (some) central activities and artifacts in a model-driven development process.

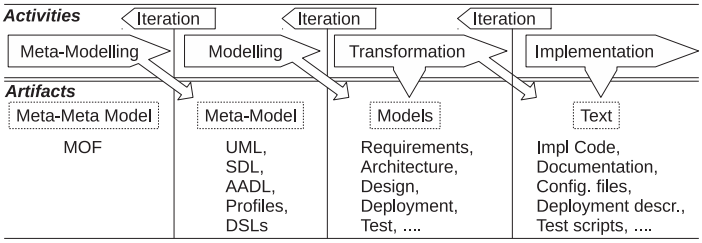


Figure 2.2: *Model-driven engineering activities and artifacts.*

In this thesis, UML and DSLs have been used side by side. UML has been used to specify examples and case studies, e.g. in terms of sequence diagram behaviours and for specifying architectural and behavioural aspects. DSLs have been defined and used e.g. to represent new language concepts, such as composition contracts. Model transformation and code generation have been used to support e.g. product variability resolutions and composition contract generation.

2.4 Languages and Composition

Composition of software parts is a basic characteristic of software construction, e.g. by aggregation of features inside a class, inheriting classes, and calling functions. Support for composition is also a premise for reuse of modularised (or de-composed) pieces of software.

Composition in Modelling Languages. UML [134] provides several mechanism to support modelling of the structures and behaviours of systems. Composition and de-composition of systems can be supported in terms of composite structures and parts. Similarly, UML behaviours can be nested within behaviours, allowing behavioural elements to be composed from other behavioural elements. Another modelling mechanism in UML that might be used to support system configurations is the template mechanism. It is similar to templates in C++ or generics in Java or C#, and can be used to parametrise packages, classes and operations. UML also provides a mechanism called *Package merge*, which allows the contents of UML packages to be merged, primarily based on syntactic matches. UML defines the semantics of merging for structural types (packages, classes, and features), but do not address merging of behaviour.

The Catalysis [54] method defined some powerful extensions to standard UML (1.x) semantics, which provides more advanced (than UML 1.x) mechanisms for refinement and compositions of specifications: one aspect is support for specialisation/extension of packages; another is refinement of different modelling views, such as collaboration refinement; a third is package joining, a predecessor to UML package merge.

Role modelling, as described in OOram [146], defined a foundation for modelling patterns of collaborating objects as role models, with focus on separation of concerns and role responsibilities. OOram defined a process called synthesis, supporting composition of new systems (role models) from (multiple) existing role models and role model refinement. Some ideas from OOram role modelling, such as role collaborations, were proposed as part of the UML standardisation process.

The last 10 years, a wide range of techniques for aspect-oriented, or concern-oriented analysis and design have emerged, such as the Theme approach [33]. There is, however, no agreed standard in this area, and technologies are still too immature. Chapter 5 goes into detail on existing approaches in this area.

Composition in Programming Languages. Object-oriented programming languages, such as Java or C#, have a variety of mechanisms for composition and reuse. *Inheritance* provides reuse of code from super classes. Composition by delegation of responsibility – using the delegation pattern – is a common way of reusing existing code. *Generics*, or templates, allow code frameworks to be reused by type instantiation. Some languages, such as C++ and Eiffel, allow classes to have many direct ancestors – multiple inheritance. This may be very useful in order to aggregate functionality, but may lead to semantic problems when symbols from several classes overlap. *Mixin* classes [27] provide a way of reusing functionality through so-called mixin inheritance or inclusion.

Traits [151] is an approach for composition of collections of behaviours. A trait *provides* and *requires* a set of methods. In the original proposal, a trait did not define any state, not did they access state variables directly. This helped avoiding some of the problems in multiple inheritance and mixins. In more recent proposals, however, traits have been extended with state information [22]. Traits are implemented in several languages, such as Squeak Smalltalk[3], Perl[4], and Scala[5].

Aspects, as described in Section 2.2, have been implemented as extensions to a range of languages, such as AspectJ [103] and AspectC++. Aspect-oriented programming is also integrated into popular enterprise server solutions, such as JBoss and Spring. Aspects are expressed in terms of pointcuts that reference a set of join points in the implementation code and sets of advice that specify the modifications at those join points.

Service Composition. Web services have become a popular development paradigm for development and integration of distributed services. A range of web service standards have evolved for the specification of web services, their interfaces, their quality characteristics, etc.[6]

Composition of web services is a research and development field in its own right. Web service composition technologies such as Business Process Execution Language for Web Services (BPEL4WS), Web Service Choreography Interface (WSCI), and Web Services Conversation Language (WSCL) provide support for specifying business processes, composite services, and service orchestration. The industry collaboration *Open*

[3]http://www.squeak.org/
[4]http://www.perl.org/
[5]http://www.scala-lang.org/
[6]http://www.w3.org/, http://www.oasis-open.org/specs/

Service Oriented Architecture[7] has specified the *Service Component Architecture*, a language-neutral programming model for services and their composition. Curbera et.al. [39] how the web service standards move toward being able to provide robust service composition using process execution, coordination, and transaction services. Adding to that, support for web service policy standards[8], services and composition of services can be controlled by policies and negotiations based on those policies.

2.5 Product Line Engineering

Product line engineering (PLE) [142] evolved from domain engineering as a method for managing reusable artifacts common to several products. Products are characterised by their *features*, which may be *common* for all products or *vary* between products. The terms *commonality* and *variability* are often used to denote the common and variable features within a product line. The feature variability is resolved in a *product resolution – or derivation – process*, wherein feature variability is resolved to produce a *product configuration*. A product line normally also has a common *product line architecture*, on top of which all products are built.

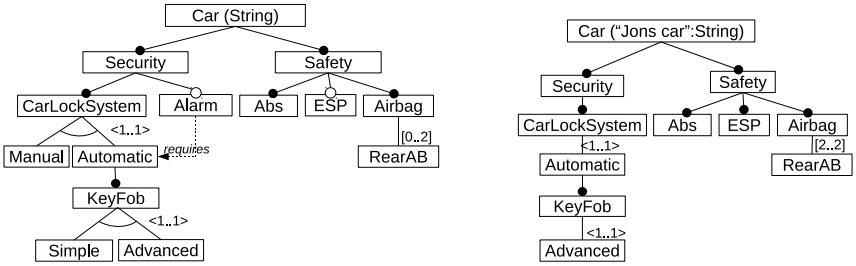
The specification of product lines has traditionally been oriented toward feature modelling and feature management. Feature modelling was introduced by Kang et.al. in the Feature-Oriented Domain Analysis (FODA) [97] approach. FODA defined a method and notation for describing features. More recent approaches are mostly modernised and generalised flavours of this. Examples are the Orthogonal Variability Model (OVM) defined by Pohl et.al. [142], cardinality-based feature modelling by Czarnecki et.al. [42], and the variability model defined by Haugen et.al. [80]. Clements and Northrop [34] present an organisational and process-oriented view on product line management with a focus on technical, management, and organisational practises and how to put these into operation.

A feature model describes the common and variable *features* of a product line. Variations are commonly represented as *optional* or *alternative* features. Feature constraints are used to specify dependencies between features, for instance that one feature *requires* or *conflicts with* another. Figure 2.3(a) shows an example feature model describing (a small part of) the features of a car product line, using the cardinality-based feature modelling notation [42]. One configurable aspect of this feature model is the *feature group CarLockSystem*, which specifies *Manual* and *Automatic* as possible choices for that group. The *optional Alarm* feature (i.e. having cardinality [0..1], denoted by an open circle) requires the *Automatic* feature to be selected in a configuration. A feature model configuration is shown in Figure 2.3(b), where all variabilities have been resolved.

Product line engineering is often coupled with *Generative Programming* (Czarnecki and Eisendecker [41]) as a means of generating products from higher-level product line specifications. Generative programming focuses on using models as assets driving the development of products, similar to MDD. The main difference from general MDD and generative programming is the explicit focus on and support for variability management and feature representation in generative programming.

Feature-oriented programming (FOP) (Batory et.al. [18]) is a style of programming

[7] http://www.osoa.org
[8] http://www.w3.org/TR/2007/REC-ws-policy-20070904



(a) Car feature model using cardinality-based feature modelling notation

(b) A car feature model configuration

Figure 2.3: *Feature modelling example using cardinality-based notation*

focusing on feature configuration for incrementally constructing a software program. Features are considered program transformations, and the complete program is constructed by an expression of such transformations.

2.6 Design by Contract

Design by Contract (DbC) is a methodological approach that was coined by B.Meyer during the development of the Eiffel programming language [123, 124]. The focus in DbC is to increase the quality and robustness of interfaces between system components, and hence increase the robustness and quality of the overall system. This was done by introducing constraints – or assertions – on interfaces in terms of pre-conditions and post-conditions, and invariants for the state of components. Pre-conditions define assumptions on the part of the client of an interface operation, which the client is required to comply with. In Eiffel, the behaviour of routines are specified using *requires* and *ensures* clauses, specifying the obligations and benefits – or assumptions and guarantees – for a client. Post-conditions define guarantees on the part of the provider, i.e. a guarantee with respect to the effect of the operation. Under the DbC paradigm, any contract violation constitute a software *bug*. A violation of the pre-condition manifests a bug in the client, and a violation of the post-condition manifests a bug in the provider.

In the Object Constraint Language (OCL) [130], contracts are provided by pre- and post-conditions and invariants that are specified in the context of model elements, such as illustrated in Figure 2.4.

```
context Car::openDoorsOnCrash ()
pre:  self.peopleInCar > 0;
post: self.doors->forAll(d|d.unlocked = true);
```

Figure 2.4: *Pre- and post-condition specified for a model operation.*

Some of the inspired sources for DbC were the works of Floyd [58] and Hoare [88], who give an axiomatic foundation for reasoning about program correctness, in which assertions are used for reasoning about the values of variables before and after a program execution, just like pre- and post-conditions.

The Ariane 5 disaster was caused by a data conversion error, one of the most expensive software bugs in history. In [95], Jezequel and Meyer argue that this bug – and hence the disaster – could have been avoided if DbC principles had been used.

Helm et.al. [85] give a different perspective on contract-based development, where contracts specify behavioural compositions of object-oriented systems. In their work, a contract defines expected behaviours of sets interacting participants; it specifies pre- and post-conditions of participant operations, and also the required interactions – or *causal obligations*.

2.7 Sequence Diagrams and STAIRS Semantics.

A UML sequence diagram [134], or interaction, is a visual representation of interaction between roles, components, or actors in a software system. Sequence diagrams represent example system behaviours and are excellent tools for specifying and communicating typical or illegal behaviour of a system. Sequence diagrams are often used as a detailing of use cases to describe requirements [94], or for describing detailed behavioural design of component or service interaction. Message Sequence Charts (MSC) [91] is the standard for scenario descriptions within the telecommunication domain, which strongly influenced the standardisation of UML 2.x sequence diagrams.

Since sequence diagrams play a central role in this thesis, an overview of their characteristics is given here. An example sequence diagram annotated with explanations is shown in Figure 2.5. It describes a scenario for the unlocking of a car in the case of an accident; if the car is *not empty*, the doors are unlocked; however, if the car is *stationary* (parked) and *empty*, the doors are kept locked to prevent burglars from provoking door opening.

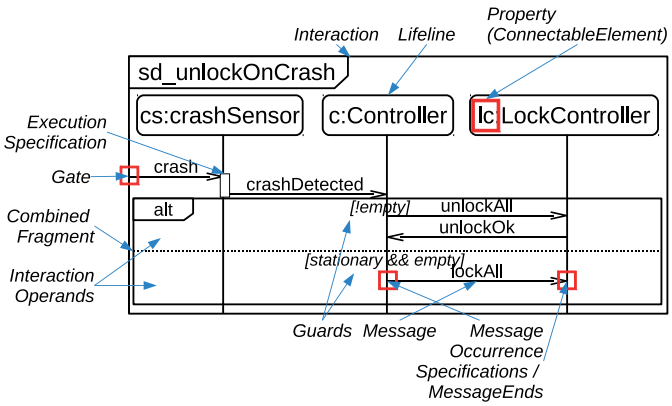


Figure 2.5: *UML sequence diagram notation*

A sequence diagram consists of a set of lifelines representing interacting parts of a system. At the meta-model level, a lifeline represents a *connectable element*, e.g. a property within a classifier. The sequence diagram defines a set of ordered interaction fragments that involves the different lifelines. The basic kind of interaction fragments are occurrence specifications that represent events on lifelines. Messages represent communication between two lifelines or between the environment (represented by a

gate) and a lifeline. A message involves one send and one receive event – or message occurrence specification. Combined fragments allow different kinds of composition of interaction fragments, e.g. sequential (seq), alternative (alt), optional (opt), parallel (par), and loop. Execution specifications represent units of behaviour on a single lifeline. The events in a sequence diagram are constrained by the order of events on each lifeline (weak sequencing) and the causality of send and receive events.

STAIRS. STAIRS [79, 82, 149] is a denotational semantics for UML sequence diagrams based on event traces. A trace is a sequence of events, each defined by its kind – send or receive, a message defining the signal, the sending lifeline, and the receiving lifeline. In STAIRS, The semantics of a sequence diagram *d*, ⟦d⟧, is defined by two sets of traces: the set of *positive* and the set of *negative* traces. Positive traces represent allowed behaviour. Negative traces represent behaviour that is *not* allowed. In addition, any trace that is not covered by the sequence diagram is defined as *inconclusive*. The set of traces in ⟦d⟧ is determined by all possible executions of *d*. The resulting event sequences are constrained by the causality and weak sequencing properties of sequence diagrams: a send event must occur before its corresponding receive event (causality) and the events on a lifeline have the same (relative) ordering in a trace as on the lifeline (weak sequencing). STAIRS defines three kinds of refinement: *supplementing*, which adds positive or negative behaviour by making inconclusive traces either positive or negative, *narrowing*, which changes positive behaviour to negative, and *detailing*, which details existing behaviour in decompositions.

{ t1:**<**!crash, ?crash, !crashDetected, ?crashDetected, !unlockAll, ?unlockAll, !unlockOk, ?unlockOk **>**,
t2:**<**!crash, ?crash, !crashDetected, ?crashDetected, !lockAll, ?lockAll **> }**

Figure 2.6: *Two traces defined by the sequence diagram in Figure 2.5*

Figure 2.6 shows the STAIRS traces from the sequence diagram in Figure 2.5. Send events are prefixed with a '!' symbol and receive events with a '?' symbol. The example sequence diagram produces two traces.

The refinement relationship (denoted by the binary operator '$\rightsquigarrow$') in STAIRS is transitive: if A $\rightsquigarrow$ B and B $\rightsquigarrow$ C, then A $\rightsquigarrow$ C. It is also monotonic with respect to the sequence diagram composition operators *alt, opt, neg, seq*, and *par*: if A $\oplus$ B is the composition with one of those operators of interactions A and B, and A' and B' are refinements of A and B, respectively, then A' $\oplus$ B' is a refinement of A $\oplus$ B. (Proof can be found in [78, 149].) This characteristic is valuable to ensure system consistency during system evolution, so that a system can be specified and refined in parts and later composed.

The full detail of STAIRS semantics can be found in [79] and [149].

2.8 The SWAT Project

This thesis work has been done within the SWAT – Semantics-preserving Weaving - Advancing the Technology – project[9]. SWAT is a project financed by the Norwegian Research Council through their research program STORFORSK. Its goal is to

[9]SWAT Homepage: http://www.ifi.uio.no/swat/index.html

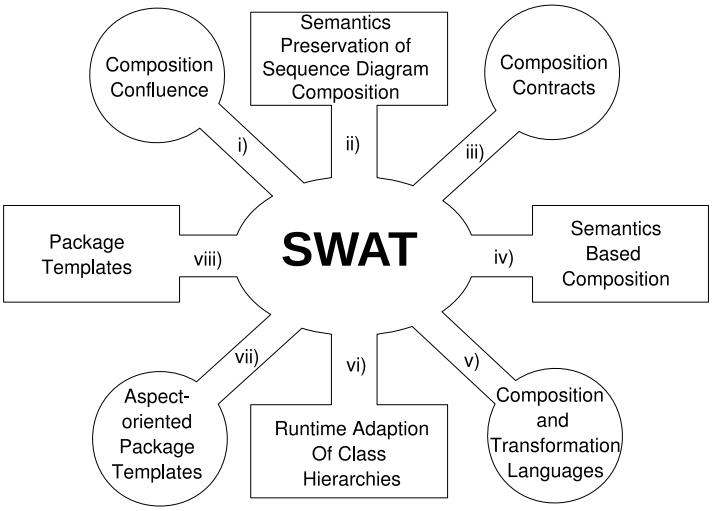


Figure 2.7: *SWAT project context overview*

address and improve programming and modelling language mechanisms for reuse and separation of concerns. A specific concern of the project is that semantics often is compromised when concerns are composed together. An example of this is aspect-oriented programming languages, such as AspectJ [103], where aspects are allowed to access and modify virtually any part of a system, and do so in many places using the quantification mechanism of the aspect pointcut language. This may lead to system modifications that are out of control.

To this end, the project seeks to advance the state-of-the-art on language mechanisms for reuse and SoC toward semantics preservation. The project currently has four PhD scholars that are addressing different topics to support this goal. Figure 2.7 illustrates the topics on a high level.

The four PhD scholars in SWAT are Roy Grønmo, Fredrik Sørensen, Eyvind W. Axelsen, and Jon Oldevik. The focus of Jon Oldevik is defined by this thesis. The foci of the other scholars within the SWAT context are described below. The work in thesis this is most closely related to the work done by Grønmo, as he focuses on model-based aspect composition. Sørensen's and Axelsen's foci are on programming level mechanisms.

The Focus of Roy Grønmo. Grønmo has in his PhD work focused on three areas in the SWAT context: one is composition of sequence diagram aspects that is semantics based [67, 68] and a confluence theory for sequence diagram aspects [72], the other is representation of aspects and transformations in concrete syntax [70, 71], and finally extensions to graph transformation theory to improve the capability of composing sub graphs [69].

The work on semantics-based composition of sequence diagrams is based on the semantic model of STAIRS, which is also the basis for the work on semantics preservation in this thesis (papers III and IV, chapters C and D). Grønmo et.al. establish a definition of partial order equivalence to make trace matching feasible, since the num-

ber of traces in a sequence diagram easily makes aspect matching intractable. In [70], Grønmo and Møller-Pedersen define a modelling notation for activity diagram aspects, which is mapped to graph transformation rules that implement the activity diagram weaving. In [71], Grønmo et.al. present the *concrete syntax-based graph transformation (CGT)*, a graph transformation language based on concrete language, such as UML activity diagrams.

The Focus of Fredrik Sørensen. Sørensen has in his PhD work followed two distinct paths: he contributed to the work on semantics-based weaving in [67, 68]. The main focus of his work, however, has been on *template packages* [114, 115]. A template package is a mechanism for reuse of collections of related classes. It is a parametrised package where its elements can be renamed and template parameters given actual values in package instantiations.

A package template must be instantiated within a package before it can be used. A template instantiation creates a local copy of the template classes – called expansion classes – with potential modifications specified. The instantiating package may expand template definitions by *adding* variables and methods, or overriding or defining abstract template methods. An important property of template packages is that all type references within the template are re-typed according to the bound expansion classes.

The Focus of Eyvind W. Axelsen. Axelsen's research targets different utilisations of the template package concept (from Sørensen and Krogdahl). Specifically, he has addressed how package templates can be paired with limited aspect-oriented extensions to provide flexible implementations of patterns [12, 13]. He has also addressed the implementation of package templates in a dynamic language – Groovy[10] – showing how the dynamic language characteristics can be exploited to make package templates more flexible [11].

[10] http://groovy.codehaus.org

Chapter 3

Problem Statement and Research Topics

3.1 Motivation

In the field of software engineering, there is a constant search for improved tools and methods to support the increasing complexity of software solutions to be built. New engineering paradigms emerge, along with new tools and improved infrastructures. Along with the Internet revolution came advanced middleware solutions supporting distributed enterprise applications and services, e.g. using web services technology. The software solutions require more complex architectures to handle not only the core business concerns, but also crosscutting concerns such as security, transactions, and availability. New modularisation techniques, such as aspect-orientation, have emerged for better to be able to separate such concerns in their own modules.

Aspect-oriented technologies provide great flexibility to the software engineer by allowing concerns to be separated in units that can later be composed with the system, either at design-time, compile-time, or even at run-time. The flexibility lays partly in the query – or *pointcut* – mechanisms, which in many cases (e.g. in AspectJ [103]) use name matching patterns to generalise the query and match multiple *join points*. An aspect may then compose a single *advice* with many *join points* in the base system.

This flexibility does not come without cost, and there are several well-known issues with aspect-oriented composition mechanisms: aspects may overlap with each other and create interference that will disrupt the composition of these aspects. This is addressed e.g. by Douence et.al. in [51]. Vulnerability to base model changes, so-called *pointcut fragility* is also a well-known issue for the pointcut mechanisms in aspect-oriented languages. This may become a problem if pointcuts are referencing obsolete or non-existing elements in the base model or code, or if new unintended matches are introduced [101, 102, 112, 150, 160].

In *product lines*, products are constructed by composition of *features*. Features have several similarities to aspects, as features often represent concerns of a product, which may also be crosscutting. Features often stand in relation with other features, i.e. they interact; sometimes, these interactions are conflicting, or interfering, similar to interfering aspects. In product lines, however, feature interactions have been addressed explicitly by feature models that state the relationship between features, such as *requires* or *conflicts*. The corresponding feature resolution process can then take these interactions into account in the product derivation process.

An issue that is still not fully addressed in the literature is the effect compositions have on the semantics of the code or model they apply to. A composition may have effects that were not intended and that may distort the semantics of the original code or model. A composition, e.g. resulting from a feature resolution, may have interactions with other compositions, requiring a specific ordering of compositions, or leading to an invalid composition result, i.e. a non-functioning system.

For example, an AspectJ aspect may change the methods in a Java program by replacing method contents by alternative behaviour using an around advice. The behaviour is changed and the semantics may no longer be as intended. This may break assumptions about protocols and result in an inconsistent system. Similar arguments can be made for model-based aspects, e.g. for sequence diagram aspects that modify system behaviour described by sequence diagrams.

In this thesis, we address *semantics preservation* of model composition from several perspectives: by defining and applying semantics preservation for aspect compositions, by analysing conflicts of feature compositions, and by guarding models from unintended changes.

This is detailed further in the research topics in the following Section.

3.2 Research Topics

The main research topic is defined as follows:

To what extent – and how – can model-based composition mechanisms *guarantee* consistency and semantics preservation of the models subject to modification?

The fundamental parts of the main research topic can be further decomposed and analysed:

- *Model-based composition mechanisms* refer to techniques for specifying system concerns using modelling languages such as UML, and then composing these concerns.

- *Preserving semantics* of the composition reflects back on the definition from Chapter 1: a system specification is *semantics preserving with respect to composition* if selected semantic characteristics of that specification are kept during composition.

- *Consistency* implies that the models subject to composition should be in a consistent state after composition, i.e. that the effects of concern compositions are not inconsistent or erroneous.

The main research topic sets the context of the research agenda and is decomposed into finer-grained and more focused topics that constitute the main concerns of this thesis.

3.2.1 Research Topic 1 – Semantics Preservation of Trace-Based Aspect Composition (RT_1)

Composition mechanisms for models provide flexible means of composing concerns by allowing pointcuts – or queries – to syntactically or semantically decide what part of a software system that can be modified. The result of such compositions can distort the intended semantics of the model, and destroy established internal model constraints, such as refinement relationships. When specifying system behaviour, maintaining semantic relationships through composition processes can be essential in order to obtain a meaningful final system.

Sequence diagrams, or interactions, are commonly used to specify partial system behaviours and are good tools for specifying things like communication protocols between components. If these interactions represent actual system behaviour, it is of essence that their semantics is not changed beyond the expected.

This research topic is detailed by the following research questions.

- $RT_{1.1}$ – How can semantics preservation for sequence diagrams be defined?
- $RT_{1.2}$ – What is currently lacking for composition of behaviour models to be semantics preserving?
- $RT_{1.3}$ – What are the benefits of semantics preservation?

3.2.2 Research Topic 2 – Confluence and Conflict in Feature Composition (RT_2)

A product line is normally specified in terms of features and dependencies between these features. Features are then linked with software artifacts, such as design elements and implementation code. A resolution process defines a product by selecting features; the product is built using the corresponding design and implementation artifacts.

Each feature represents a concern, or part of a concern, in the product line, which is either common across all products (a commonality) or variable in the product line (i.e. only present in some products). In the construction of the product, the product line variabilities are resolved by selecting the features that will be part of the product. Then, these features are composed.

The features, or concerns, may depend directly or indirectly on each other, and the product composition result may depend on the order by which features are composed. This is similar to *aspect interference*, where two or more aspects interfere with each other, such that the composition result depends on the composition order. These models can be said to be non-confluent with respect to the composition. Why is this a problem? If the fact that two concerns are conflicting is not known, this may lead to undesired or even invalid results. Hence, being able to characterise occurrences of non-confluence can improve awareness of the problem and help avoiding erroneous compositions.

This research topic is detailed by the following research questions:

- $RT_{2.1}$ – How can confluence of feature composition be analysed?
- $RT_{2.2}$ – What are the conditions for being able to determine confluence?

3.2.3 Research Topic 3 – Model Composition Contracts (RT_3)

When a system is designed and implemented, it is planned to adhere to certain explicit and implicit constraints and expectations, in accordance with the system requirements. Each modification to the system should be according to those constraints. The requirements may change over time, and the governing constraints change with them. However, the compliance with these constraints may be difficult, especially for changes imposed by automated mechanisms, such as a composition engine.

Changes that breech these constraints should not be allowed; they should at least be possible to detect and control. If they can, it is possible to gain increased control of the system specification and any process (e.g. composition) that modifies it.

This research topic is detailed by the following research questions:

- $RT_{3.1}$ – What does it mean that a composition is constrained by contracts?
- $RT_{3.2}$ – How can such contracts be specified?

3.2.4 Research Topic 4 – Model-based Aspect Representation (RT_4)

Although there has been a variety of research and publications on model-based aspect representation (as described in Sections 2.2 and 5.1), there is still no community-wide consensus or standard for aspect-oriented modelling. Since software modelling is a very broad topic, there are likely to be unfilled gaps with respect to representing aspect models. In this thesis, we address some of these gaps pertaining to aspect representation in UML.

This research topic is detailed by the following research questions:

- $RT_{4.1}$ – How can UML architectural aspects be represented and with what benefit?
- $RT_{4.2}$ – How can UML interaction aspects be represented and with what benefit?

Chapter 4

Research Method

4.1 Research Method Overview

According to [169], there are two general types of theory-building research: analytic and empirical. Analytic research uses primarily deductive methods while empirical research primarily uses inductive methods to arrive at theories. Analytic reasoning was introduced by Aristotle in his work on prior analytics [7]; this work was later superseded by propositional logic and predicate logic. Empirical research, or the empirical method, can also be traced back to the age of Aristotle, but was formalised by Ibn Sina in about 1000AD [147].

Since its origin, computer science has been closely related to mathematical logic and algorithm theory [46]. Through its applications in almost every thinkable domain, computer science also targets the engineering discipline of constructing and using computer programs. If claiming that a particular technology is effective in a scientific sense, we need a method to show, explain, or prove this. In computer science, there are several ways to do this.

Zelkowitz et.al. [181] identify four categories of research methods:

- The *scientific method*, where a theory is developed to explain some phenomenon. The theory is supported by a hypothesis, which is tested based on data collection.
- The *engineering method*, where solutions are developed and tested to support a hypothesis.
- The *empirical method* is similar to the scientific method, except there may not be a developed theory. The empirical method is often considered as being a part of the scientific method.
- The *analytical method*, wherein a theory is developed, which is supported by empirical observation.

Empirical methods gather information about the research object be means of *observation*, *experiment*, or *experience*. One or more hypothesis are proposed, and a *validation method* is used to gather data to accept or reject the hypothesis. A number of different validation methods exists, with their own characteristics. Zelkowitz et.al. [181] group these into three different categories: *observational*, *historical*, and *controlled*. Project monitoring, field study, case study, and assertions are considered *observational* approaches. Literature search, legacy analysis, and static analysis are

considered *historical* approaches. Replicated experiment, synthetic experiment, dynamic analysis, and simulation are considered *controlled* approaches. The list is not exhaustive.

A slightly different view is presented by Denning [46], who describes three major paradigms in the field of computer science: *theory*, *experimentation*, and *design*. The *theory* paradigm corresponds to the analytic method, the *experimentation* paradigm corresponds to the scientific and empirical methods, and the *design* paradigm corresponds to the engineering method.

The goal of computer science is often to analyse technology, such as languages, algorithms, and methodologies, in order to improve it or to gain a better understanding of its principles and application areas. *Technology research* is defined by Solheim and Stølen [161] as *research to bring forth new and improved artifacts*. Technology research is related to – or equivalent to – what above is referred to as the engineering method. Figure 4.1 illustrates the main process steps involved. The overall goal of technology research is to *obtain new and improved artifacts*, which are established through *innovation*. The nature of these improvements is based on collected requirements, established through *problem analysis*. The overall hypothesis is that the artifacts satisfy the requirements. To *validate* the research, the hypothesis must be tested, i.e. the validity of the statements should be evaluated, either using an empirical or an analytic approach.

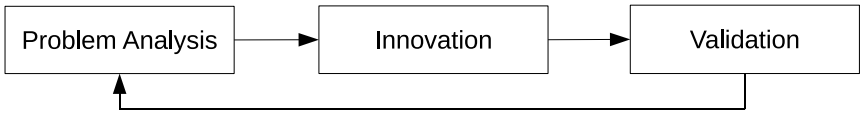


Figure 4.1: *Technology research process elements*

4.2 The Research Methods Applied in this Thesis

Technology research combined with an analytic approach have provided the underlying framework for this thesis work, and a mix of different non-experimental and analytic methods has been applied. Literature search and static analysis of existing techniques and tools have been applied related to all research results. A continuous process of reviewing other research results and technology developments has been undertaken. Case studies and examples were also applied to support the research claims. The analytic approach was also applied for some of the thesis results. Finally, technology analysis was used in some cases to compare technologies. In several cases, prototype tools have been developed to support proposed models or theories. For example, related to paper II (Appendix B), an aspect-oriented extension was developed for the transformation tool MOFScript [137] to improve its support for product line configurations.

A summary of the research process followed is outlined below.

- A problem analysis was conducted, from which requirements for new technology or theory were identified.
- New technology and theory were defined.
- The results were validated with case studies, analysis, prototypes, and to some extent proofs.

4.2.1 Problem Analysis

Literature Review and Analysis of Existing Approaches. In the initial phase of the PhD work, a work through of state-of-the-art within research and practical results within composition and separation of concerns was undertaken. This provided the foundation on which to define the research focus of this thesis work. For each detailed piece of research conducted, as presented in the published papers in Appendices A–F, literature reviews of related research works were performed.

Identifying Problem Areas. From the literature review and analysis of existing approaches, we identified a lack of support for and focus on consistency and semantics preservation in model composition. The technologies provided flexible modularisation and composition mechanisms, but without concern for how compositions affect the semantics of the models they modify. Our analysis resulted in four problem areas being identified, as described by the research topics in Chapter 3: (1) semantics preservation of trace-based aspect composition, (2) confluence and conflict in feature composition, (3) model composition contracts, and (4) model-based aspect representation.

4.2.2 Innovation

The innovation phase addressed the identified problem areas within state-of-the-art model composition. The results were new technology and theory described by the research papers in Appendices A through F.

4.2.3 Validation

Examples and Case Studies. We have established and reused several small and medium-sized case studies in order to test and evaluate our own theories and those of others. For example, in paper V (Appendix E), variability transformations and confluence checking were applied on a train station DSL model defined by the MoSiS[1] project. In paper IV (Appendix D), a drawing application was used for evaluating the approach. Paper VI (Appendix F) uses a case study focusing on distributed collaboration between department stores.

Prototypes. The work in this thesis has been supported by several prototypes to illustrate, test, and validate the concepts and languages defined. In the work in paper II (Appendix B), we developed an aspect-oriented extension to MOFScript [137][2] and implemented transformations for evaluating the research using that extension. In paper IV (Appendix D), we developed transformations for mapping sequence diagrams to an implementation of trace-based aspects. In paper V (Appendix E), we further extended MOFScript to support model transformations, and implemented the transformations to validate the theories in the paper. In paper VI (Appendix F), we developed a prototype for model-based contracts, which supports specification and checking of contracts related to models and model composition.

[1] MoSiS project web page: http://itea-mosis.org/
[2] http://www.eclipse.org/gmt/mofscript/

Technology Analysis. Technology analysis has been used to assess information about static and behavioural features of existing tools and products – typically to gather information on whether particular features are supported or not. When Zelkowitz et.al. [181] refer to dynamic analysis, they refer to analysis of the executing system with respect to some property, such as performance. The analysis done here can more appropriately be referred to as functional analysis.

Analytic Validation. Parts of the work in this PhD thesis are based on analytic validation, where a theory was defined to explain the phenomenon under study. The theory has been validated by empirical observations from applying examples, and also, in some cases, supported by proofs. For example, in paper III (Appendix C), we give a definition of semantics preservation of sequence diagram aspect composition, which is used for evaluating different existing approaches for aspect-oriented modelling of sequence diagrams. In paper IV (Appendix D), these definitions are applied on a defined trace-based semantics for Java. In paper V, we define a set-based theory for reasoning about confluence of feature resolutions.

4.3 Evaluation

We largely base the validation of the thesis research on case studies, examples, and prototypes. In some cases, this may be considered insufficient validation. It does provide, however, example contexts in which the research has been validated; this can be basis for generalisation toward other domains and cases. In research paper III (Appendix C) the defined theory was applied on six different aspect-oriented modelling approaches, which caters for a good coverage of technologies. The case study in paper VI (Appendix F) is a fairly large example case study, which is specified in UML and implemented by 12KLOC Java code.

Potential Weaknesses. The validity of any kind of research can be strengthened by gathering more empirical data, executing additional case studies as input for analysis, performing interviews, etc. This is also the case with the research performed in the course of this thesis. The lack of generalisation in parts of the research represents a weakness of method and potentially of the results. This can be considered a threat to the external validity of the research, i.e. the applicability of the results in other contexts. The same can be said about the size of the case studies/examples used in the research; they are typically small examples. No industrial size case studies have been used. This is due to several factors: the availability of suitable industrial size case studies, the limited time available to perform such cases, and the nature of the research – new language mechanisms – which makes case study candidates difficult to establish. Another, related potential weakness is that the examples used are too narrow to cover all potential shortcomings, which may lead to biased conclusions. The validity may be further strengthened by applying quantitative or qualitative methods to gather statistical data that can support the research claims.

Although it is possible to strengthen the validity, the main thesis results – the research papers – have been subject to validation as described above (Section 4.2.3). Additionally, the papers have been subject to review processes that further strengthen

their validity. In summary, the validation provided shows a reasonable relevance of the results.

Chapter 5

Literature Review and State-of-the-Art

Here, we address in detail related work from literature that is influential and related to what has been achieved in this thesis. In particular, we focus on SoC in modelling, model transformation, product line engineering, conflicts and restrictions in composition, and semantics preservation.

5.1 Separation of Concerns in Modelling

SoC in modelling spans a wide variety of techniques and language mechanisms, and SoC is directly or indirectly incorporated into the modelling process. Some of these have defined mechanisms specifically to address the composition of concerns. Responsibility Driven Design (RDD) [177, 178] is a design process focusing on roles, their responsibilities, and how they interact with other roles. The main abstraction – for representing concerns – is role collaborations; Class (Candidates), Responsibility, and Collaboration (CRC) cards [21] are used to in the process of finding and elaborating the role collaborations. In the Catalysis [54] method, concerns can be separated in packages which can later be joined; this is similar to the *package merge* mechanism in UML [134]. Package merge is a syntactic-based mechanism, and the standard only specifies how static models (classes, packages) are merged. In OOram [146], role models are used to describe patterns of collaborating objects – as concerns – that are composed into systems in a synthesis process. The synthesis process in OOram, however, considered only the synthesis of structural models, not the behaviour. In product line processes, such as KobrA [8], concerns are separated as features, which are composed through a resolution, or product derivation process. KobrA is a component-based product line development process, which represents variability partly in the graphical design model by optional elements. The complete variability management, however, is handled by textual decision models, which map variability resolutions to effects in the design models.

A range of different architecture frameworks have also been widely used for separation of concerns for architectural specification; they address the high-level elements in a specification rather than the specific language mechanisms used to describe them. The Zachman framework [180] defined a framework for representing *information systems architecture* by descriptions capturing different stakeholders needs, or views, of differ-

ent system perspectives. The system perspectives were *data, process, and network*, while the views defined were *scope, business, information, technology, implementation, and running system*. The 4+1 view model of architecture [116] defined Rational Software's view on software intensive architecture. It specifies five views that address concerns of different stakeholders: fours views are used to organise architectural decisions, the *logical, process, physical*, and *development* views, and the *scenario* view is used to illustrate the other views. *Architectural blueprints* are used to describe each view. Several architectural frameworks are also defined by international standards: the ISO Reference Model for Open Distributed Processing (RM-ODP) [92] prescribes five viewpoints by which architecture of distributed systems should be specified – *enterprise, information, computational, engineering*, and *technology* viewpoints; ISO/IEEE 42010 is a standard currently under development, which is a revision and update of the IEEE 1471 standard entitled *Recommended practise for architectural description of software-intensive systems* – it defines guidelines and recommendations for structuring architectural specifications.

In the end of 1990's, aspect-oriented development emerged in the modelling area.

Aspect-Oriented Modelling. Aspect-oriented modelling was spawned by the same needs and ideas as aspect-oriented programming, i.e. that the complexity of models led to scattering and tangling of model elements, and that this could be solved by separating concerns as aspects. An early work in the aspect-oriented modelling arena is documented in Grundy's paper on aspect-oriented requirements engineering [73]. In this work, aspects represent required and provided characteristics of components, which act as a specification for refinement to design and implementation. There is, however, no notion of aspect composition in this work. Clarke et.al. [32] present subject-oriented design, inspired by the concepts of subject-oriented programming [76]. Subject-oriented design supports decomposition of software design into modules, called subjects, that encapsulate parts of a system design belonging to a particular concern. Subjects may overlap each other and are integrated, or composed, by merging, selection, or overriding. Subject-oriented design further evolved into the *Theme* approach [16, 33]. The *Theme* approach is an aspect-oriented analysis and design method, which addresses requirements analysis, design, and composition of themes. A theme is a representation of a concern, or a feature, which may have crosscutting properties. Requirements are specified using Theme/Doc views, in terms of relationships and crosscutting elements of requirements. Design is specified in Theme/UML by base and aspect themes. Crosscutting behaviour is provided by sequence diagrams that are parametrised by template parameters defined on the theme package definition. Themes are composed symmetrically by *merging* or *overriding*. Potential conflicts between matching elements are manually specified, and template bindings are explicitly specified.

Aspect-oriented modelling has been dominated by a series of workshops from early 2001, the aspect-oriented modelling[1] and early aspects[2] workshops. Within these communities a variety of aspect modelling research has been produced. Early aspects refer to aspects at the requirements and architecture level. Rashid et.al. [144, 145] define a model for aspect-oriented requirements engineering (AORE) that separates aspectual requirements, non-aspectual requirements, and composition rules. They establish

[1]Aspect-Oriented Modelling Workshop - http://www.aspect-modeling.org/
[2]Early Aspects Workshop - http://www.early-aspects.net/

positive and *negative* contributions among requirements and concerns, and use these for reasoning about early trade-offs between requirements. In Moreira et.al. [126], the AORE approach is generalised in a multi-dimensional concern model, wherein all concerns are treated uniformly. In [94], Jacobson and Ng describe an approach for aspect-oriented development with use cases with *use case slices*, inspired by *hyperslices* by Tarr et.al. [164]. A use case slice modularises use case realisations and their related, possibly crosscutting, design elements. It contains a collaboration that defines the realisation of the related use case, and may contain additional required classes and class extensions that are defined by aspects.

Aspect-orientation has been defined for several architectural modelling approaches: France et.al. [62] use role templates – applying the Role-Based Metamodeling Language (RBML) [61] – in UML models to describe architecture aspects. Aspects define patterns with template slots that are bound when aspects are instantiated. A template element is denoted using a vertical bar (|) notation in the element name, e.g. $|Server$ as the name of a class defines that class as a template that needs to be bound.

Garcia et.al. [63] present *AspectualACME*, which define aspect extensions to the general purpose architecture language ACME [64]. In ACME, architectures are described with the concepts *system, component, connector, port, role, representation*, and *representation maps*. AspectualACME extends the connector concept with aspectual connectors. While a standard connector connects two roles, an aspectual connector connects a base role and a *cross cutting role*. Aspectual connectors may be attached to multiple base model roles using quantification mechanisms.

In [15], Baniassad et.al. describe how to identify and capture early aspects, and how they are carried from one phase to another.

The AOSD Europe project produced a survey of analysis and design approaches in 2005 [31], which gives a comprehensive overview of aspect-oriented and non-aspect-oriented approaches to analysis and design.

A lot of aspect-oriented analysis and design work has been based on UML extensions, where some of the early works address design aspects related to structural concerns by means of classes, e.g. Clarke et.al. [32], Suzuki and Yamamoto [162], and Herrero et.al. [87]. In [157], Stein et.al. describe a UML-based notation for AspectJ called the Aspect-Oriented Design Model (AODM). That work was succeeded by more focused work on expressing pointcuts in UML in the work on Join Point Designation Diagrams (JPDD) [75,158,159]. JPDD provides a notation, semantics, and a tool implementation for specifying queries graphically using extensions to UML. These queries are specified using the concrete syntax of UML, i.e. classes and interactions, using name patterns and various types of constraints, such as indirect generalisation constraints for classes or control flow constraints for interactions.

The JPDD work is relevant for several parts of this thesis, specifically in the work on mapping sequence diagram aspects to Java trace-based aspects (paper IV, Appendix D).

The topic of behavioural design aspects is addressed in a range of works. Solberg et.al. [155,156] present the Aspect-Oriented Model Driven Framework (AOMDF), which includes an approach for describing sequence diagrams aspects based on stereotype tags. Aspects are composed with base models using explicit binding to the aspects. As such, there is no querying, or pointcuts, involved. The approach uses patterns as described by France et.al. in [61] to parametrise the aspects.

Klein et.al. [109, 110] address weaving of scenarios. They look at the problems of syntax-based composition and provide techniques for semantic-based composition. This is also the focus of Grønmo et.al. [67, 68], who use a trace-based semantics for sequence diagrams as basis for the semantic-based weaving. *Semantic-based weaving* addresses how pointcut matching is according to the semantics of the sequence diagrams (i.e. their behavioural meaning) rather than their syntactic representation.

Other approaches for sequence diagram aspect specification and composition, such as that defined by Solberg et.al. [155], Whittle et.al. [176], and the Theme method [33], are syntactic-based. These approaches are discussed in detail in paper III (Appendix C). In [6, 175], Whittle and Araújo present an approach for capturing requirements with aspect-oriented scenarios. Their approach is based on the *interaction pattern specifications* (IPS) approach from [61], and uses binding of pattern roles to bind aspects – or IPS'es – to concrete scenarios. Their approach is also syntactical, and the binding is explicit, rather than query based. However, although the approach is syntactic-based, it will meet the definition of semantics preservation given in paper III; this is because of the restricted, static binding, which binds single pattern element (messages) in the sequence diagram. This is in accordance with the analysis results of the approaches from paper III.

Deubler et.al. [47] present an approach for MSC aspects targeted specifically at service-oriented platforms, where specialised lifelines represent different kinds of pointcut and advice. Compositions are not done at the MSC level, but as behaviour inserted at service run-time. A similar approach is taken by Krüger et.al. in [118], where crosscutting behaviours for embedded system services are described by MSCs that are joined, using the join operator for MSCs defined by Krüger [117]. This join operator allows composition of overlapping interactions by joining matching messages. In [118], the MSC aspects are used for generating crosscutting run-time monitors.

Kienzle et.al. [106] propose a multi-view approach called *Reusable Aspect Models (RAM)*, which integrates existing aspect modelling techniques for classes, sequence diagrams, and state charts in a coherent model, and packages these an *aspect model*. Aspect models may be reused by, or instantiated by, other aspects, or by non-aspect models. The RAM approach supports dependency checks between aspects and allows aspects to be reused in multiple contexts. An aspect is defined by a structural view, a state view, and a message view. Each view defines template parameters that must be bound when an aspect is used. Consistency is checked and enforced between the views and on aspect bindings. The approach is implemented with tools developed in Kermeta, such as *Kompose*[3] [59]. Kompose defines a generic composition framework that can be specialised for any EMOF-compliant meta-model. It defines interfaces that must be implemented for specific meta-model composition, implements a generic merging operator, and defines a composition directive language to control types of changes that may occur.

Whittle et.al. [173, 174] define a flexible aspect-oriented modelling approach called MATA (Modelling Aspects Using a Transformation Approach), wherein they use graph transformations for composing aspect models in UML. Aspects and base models are defined using the concrete syntax of the modelling languages, in this case UML classes, sequence diagrams, and state machines. They provide a pattern language that allows pointcuts to specify sequences of model elements, and composition directives in the

[3] http://www.kermeta.org/mdk/kompose

form of stereotyped elements to specify the searching for, creation of, or deletion of elements. Base models and composition directives are mapped to graph transformation rules. *Critical pair analysis* is used to check and determine interactions between the rules, and hence interactions between the specified aspects. Two types of interactions are analysed: *conflicts*, which prevent two aspects to be used together, or *dependencies*, which require aspects to be applied in a specific order. After analysis, the transformations are executed and the result is mapped back to a UML representation. Their approach, which is based on syntactic matching, is not semantics preserving according to our definition for sequence diagrams, but it provides a useful and complementary functionality with the conflict analysis, which will help ensuring consistent results of model-based aspect composition.

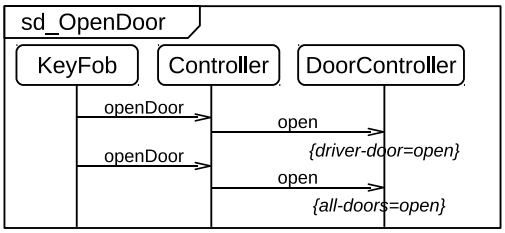
Trace-based Aspects. Scenario specifications describe event traces of system behaviours, which normally represent example runs of system behaviour leading to a specific state. The usage of historic execution information when describing aspects is addressed by many different research and development efforts. These kinds of aspects are referred to as trace-based aspects in the case of using execution traces, or more generally state-based or history-based when using state information in the aspects. Trace-based aspects are the topic of papers III and IV (Appendices C and D), which define semantics preservation for sequence diagram aspect composition and map sequence diagram aspects to a trace-based aspect implementation.

In [50, 52], Douence et.al. define a formal approach for generic composition of stateful and trace-based aspects and introduce applicability conditions for aspects, which are used for analysing aspect interactions and resolving conflicts. Its basis is defined by a framework for detection and resolution of aspect interactions described in [51], which specifies a formal execution model based on observable traces, and an accompanying aspect language. Aspects are defined by rules on the form – $C \rhd I$ – where C is a crosscut and I a program that is executed whenever C matches a join point. The crosscuts are defined by conjunction, disjunction, and negation of terms, but without quantifiers, which makes the analysis of conflict detection feasible. They define independence of aspects and describe an algorithm for checking if two aspects are independent.

Trace-based, or state-based aspects can be supported by several aspect-oriented programming language implementations. In AspectJ, for example, trace-based aspect support can be implemented by storing execution history, which can be used as guards in pointcuts specifications. The specification language in AspectJ, however, does not provide specification mechanisms suitable for specifying pointcuts representing execution traces without explicitly storing and using state variables, except for traces that are part a single control flow, which can be captured by *cflow* pointcut designators.

This is illustrated in Figure 5.1, where the AspectJ code in 5.1(b) specifies pointcuts and advice to capture the trace behaviour specified in the sequence diagram in 5.1(a). The sequence diagram captures that two subsequent calls to the open method of door controller should result in all car doors being opened (represented by state invariants). The AspectJ code needs to keep track of the state of previous *open* calls, and stores this in a local variable.

This shortcoming was addressed in the work on Join Point Designation Diagrams (JPDD) [75], which generated complex pointcut expressions based on JPDD sequence diagrams. This simplifies the specification of complex pointcut expressions that cap-



(a) Open door sequence diagram

```
public aspect OpenDoors {
    private static int door_open_count = 0;
    pointcut ctrlOpen():execution (void Controller.openDoor(..));
    pointcut dcOpen():execution (void DoorController.open());
    pointcut openDoor () : dcOpen() && cflow(ctrlOpen());

    pointcut openDoorTwice(DoorController dc) :
        target(dc) && openDoor() && if(door_open_count == 2);

    after ():openDoor() {
        door_open_count++;
    }
    after (DoorController dc) : openDoorTwice (dc) {
        // door open event has been sent for the second time
        door_open_count = 0;
    }
}
```

(b) Open door AspectJ aspect

Figure 5.1: *Trace-based aspects and their representation in AspectJ*

tures histories, i.e. traces, of the system execution, since they can be expressed graphically in terms of sequence diagrams, similar to Figure 5.1(a). From JPDD, implementation code can be generated in a designated aspect-oriented programming language.

Vanderperren et.al. [167] describe how the aspect-oriented programming language *JAsCo* is extended with support for stateful aspects based on the formal model defined in [50, 52]. Stateful aspects in JAsCo are described declaratively using protocol-based pointcut expressions, where labelled transitions are associated with standard JAsCo pointcuts – such as method calls – and related with destination transitions that should be matched subsequently. The stateful aspect is compiled to a deterministic final state automaton, which is interpreted at run-time.

A similar approach called *Tracematches* is defined by Allan et.al. [3]. It provides an AspectJ-based extension for specifying trace-based aspects. Traces are defined as regular expressions over pointcuts, or symbols referring to pointcuts. The language allows filtering of events that are outside of a chosen context, which was fitting for what was needed for the mapping of sequence diagram aspects in paper IV. In Tracematches, symbols that are not explicitly declared, are ignored when matches are sought for. This was found very useful for representing pointcuts reflecting behaviour specified by sequence diagrams, since it allowed filtering events that were not relevant in the context of a specific interaction. Figure 5.2 illustrates the specification of a *tracematch* for the open door scenario.

Walker and Viggers [171] present an approach called *Declarative event patterns (DEP)* in a language extension to AspectJ. This approach extends AspectJ pointcuts

```
public aspect OpenDoors {
    pointcut ctrlOpen():execution (void Controller.openDoor(..));
    pointcut dcOpen():execution (void DoorController.open());

    tracematch (){
        sym ctrlOpen before: ctrlOpen();
        sym dcOpen after: dcOpen();
        (ctrlOpen dcOpen)[2] {
          // door open event has been sent for the second time
        }
    }
}
```

Figure 5.2: *Open doors aspect in Tracematches*

with specification of *tracecuts*, which can be used to specify histories of execution events, similar to Tracematches. A tracecut is either a regular pointcut, or a sequence or repetition of other tracecuts, representing histories of execution events. DEP allows semantic action blocks to specify detailed acceptance or rejections of events in the context of a tracecut.

In the work by Cottenier et.al. [37, 38], stateful aspects are described in terms of state machines. The approach is implemented in the Motorola WEAVR, and provides a means of describing pointcuts and advice as state machines. The weaver produces a composed model that can be used for code generation.

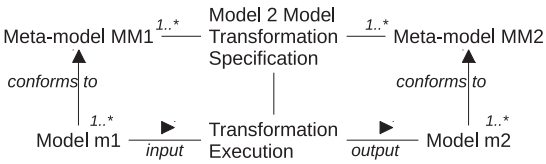
Krüger et.al. [118] propose a solution for run-time monitoring of trace-based system behaviour, where the monitors are defined with MSCs. From the MSCs, distributed run-time monitors are injected into component implementations using aspect-oriented programming techniques. The model-based monitors are used to verify system adherence with the behavioural models.

5.2 Model Transformation

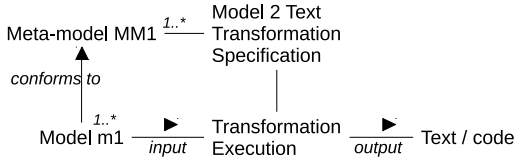
Model Transformation technologies enable declarative or imperative specifications of model manipulations that can be used to implement any kind of transformation of a model – including model compositions. In this thesis work, model transformation has been used primarily as a vehicle for model composition or transforming between model domains.

MDA-style Model Transformation. The Object Management Group (OMG) has defined a range of standards related to model-driven engineering – Model Driven Architecture (MDA) in OMG terms. This includes a standard for model transformation called MOF Query/View/Transformation (QVT) [131] and for model to text transformation [132] (Mof2Text). With QVT, transformations can be authored as relations and mapping rules between meta-models and their properties, or as operational rules. QVT defines two main transformation languages: the relations language and the operational mapping. The relations language allows relations between meta-model domains to be specified in a textual or graphical syntax. Transformations can be either unidirectional or bi-directional. The operational mapping language allows the specification of complete imperative transformations or complementing relational transformation with operational behaviour. The OMG Mof2Text standard defines a language and

meta-model for specifying code – or text – generation in an imperative style using templates. Open source implementations of these standards are being implemented within the Eclipse M2M[4] and M2T[5] projects. Figure 5.3 illustrates the general architecture of these types of model transformations: a model to model transformation (Figure 5.3(a)) specifies transformation rules mapping between two (or more) meta-models. The execution of such a transformation takes model instances *conforming to* the meta-models and performs the specified mappings. A model to text transformation (Figure 5.3(b)) specifies rules, or templates, based on one or more input meta-models, which in turn produce output text, for example implementation code, when executed.



(a) Model to model transformation



(b) Model to text transformation

Figure 5.3: *Model transformation architecture overview*

A wide range of tools provide alternatives to the OMG standards, such as Atlas Transformation Language (ATL)[6], Kermeta[7], Epsilon Transformation Language (ETL)[8] for model transformation, or MOFScript[9], Velocity[10], Xpand[11], or Jet[12] for text transformations.

In this thesis, model and text transformations have been implemented using the MOFScript tool [137], which is an imperative – or operational – style transformation language. MOFScript was originally developed based on one of the proposals for the OMG Mof2Text standard. It has been extended during this thesis work with support for model to model transformations and aspects. MOFScript was used in paper II, paper IV, paper V, and paper VI for code generation of product line code, and model transformation for transformation aspects, feature composition, and contract model generation.

[4]http://www.eclipse.org/m2m/
[5]http://www.eclipse.org/modeling/m2t/
[6]http://www.eclipse.org/m2m/atl/
[7]http://www.kermeta.org/
[8]http://www.eclipse.org/gmt/epsilon/
[9]http://www.eclipse.org/gmt/mofscript/
[10]http://velocity.apache.org/
[11]http://wiki.eclipse.org/Xpand
[12]http://www.eclipse.org/modeling/m2t/

Graph Transformation. Graph transformation, graph grammars, or graph rewriting, evolved as a result of shortcomings of term rewriting systems to express non-linear structures. They were developed as an algebraic theory where graphs are considered algebras that are glued with an operation called *pushout* [36]. Graph transformations – in the algebraic sense – are defined by *productions* (graph transformation rules), *matches*, which are occurrences of the left-hand sides of graph productions, and *rule applications* (direct derivations) of graph occurrences.

A graph production p can be specified as $p : L \rightsquigarrow R$, where L specifies the graph that will be matched, and R specifies the modifications that will be done to matches of graph occurrences. If a match m is found for an occurrence of L in a given graph G, a *derived graph* H is obtained by replacing all occurrences of L with R in G. Graph transformations are convenient for formal analysis of the transformation and compositions, since their underlying mathematical theories allow reasoning about confluence, termination, etc. In Heckel et.al. [84], confluence properties for typed attributed graph transformation systems are defined. This is based on well established theories of parallel independence and critical pair analysis of graph transformations, which have been used to show commutativity and confluence of transformations.

The work in this thesis has not applied graph transformation theory, but the results from paper V (Appendix E), which focus on confluence of product line transformations, are related to graph transformations. In that paper, the transformations were realised using imperative – and dynamic – model transformations on EMF models. Although there exist graph transformation tools that can operate on EMF models, such as the *Tiger EMF Transformation Project*[13] [24], these are not easily applied dynamically and independent of the meta-model.

5.3 Product Line Engineering

As described in Section 2.5, product line engineering is an established approach for management of common and variable features of sets of products that are part of a product line.

In this thesis, product lines are addressed from several perspectives. In paper II, we address the usage of higher-order transformations – or aspectual transformations – for providing variability in product lines. The product line is specified as a UML model, which includes variability information by using stereotypes. We use a generative approach, where products are generated using model to text transformations. The transformations themselves are modified by other transformations – specified by aspects – to accommodate product line variability. *Generative programming* – or *Software Factories* – is an established technique for product lines [41, 65], or product/system families, wherein members of the system family can be automatically generated from a specification in some domain-specific language. In paper V, we provide a domain-independent solution for configuring product line models based on the variability model from Haugen et.al. [80], where the variability is defined by model element substitutions.

The work by Czarnecki and Antkiewicz [40] use model templates with *feature annotations* for generating model configurations. Feature annotations are matched and evaluated against a feature configuration. Variability is modelled as so-called *presence conditions* that are superimposed on the base language (e.g. UML), which determine

[13] http://user.cs.tu-berlin.de/~emftrans/

the presence or absence of a model element in a configuration. The approach is generally applicable to any model domain based on Meta Object Facility (MOF); this is the same for the approach presented in paper V. The underlying implementation, however, is dependent on the domain – or meta-model (e.g. UML), which is avoided in the domain-independent transformation approach in paper V. Another difference is that the superimposed variants are embedded in the base language, which is not the case for the work in paper V.

Batory et.al. define an approach for feature-oriented programming (FOP) in their work on step-wise refinement with GenVoca [19] and AHEAD (Algebraic Hierarchical Equations for Application Design) [17, 18, 20]. GenVoca is a method for compositional refinement of classes, where refinements are specified by equations and implemented by *mixins* [27]. It is generalised in AHEAD, which allow composition using hierarchical equations that represent any kind of artifacts, code and non-code, by associating composition operators with artifact types. Features can be specified as separate units and composed to obtain specific configurations.

The example in Figure 5.4 illustrates feature specification using AHEAD containment hierarchies and composition operator (•). The car *JonsCar* is composed from the two features *Security* and *Safety*, which again consist of features.

```
JonsCar    = Security[a] • Safety
Security[a] = {Automatic-CarLockSystem {AdvancedKeyFob}}
Safety     = {Abs, Esp, Airbag}
Security[b] = Security[a] • Alarm
```

Figure 5.4: *Feature composition and refinement in AHEAD*

Refinements in AHEAD can be provided by feature composition. For example, a different variant of the security feature can be provided by composing $Security^a$ with the Alarm feature, as illustrated in Figure 5.4. The composition operator • is defined to be polymorphic and depends on the underlying artifacts of a feature, e.g. code, design models, or requirements documentation. The AHEAD approach could be applied in the context of both paper II and paper V, in which *hitransforms* and variability substitutions would represent composition operator semantics in AHEAD feature compositions.

Prehofer [143] introduced a programming model for Feature-Oriented Programming (FOP), which allowed objects to be defined by composition of features. A feature is defined similar to a traditional class, by implementing interfaces. In addition, features can *lift* other features, a way of adapting functionality of one feature to the context to another, similar to method overriding. Features may also declare assumptions on the presence of other features.

Haugen and Møller-Pedersen [81] show how to use UML structural classes to specify architectural configurations. This is done by specialising property types and defining subset constraints on parts. They generalise the role concept in UML by letting named subsets of instances play different roles. This is complementary to our work on architectural aspects in paper I, which also use structural classes, but focus on architectural configuration by connector refinement.

Supporting product lines with technologies for crosscutting concerns, as addressed in paper II, has been addressed from several perspectives by others: Anastasopoulos and Muthig [5] evaluate aspect-oriented programming as an implementation strategy

for product lines, using AspectJ as the evaluation language. They analyse its applicability with respect to reuse, positive and negative variability, granularity, testability, integration impact, binding time, and automation. They found AOP suitable for handling variability across several components. They identify shortcomings with respect to testability of aspects, capturing variability related to control structures, and for handling potential conflicts between aspects. Voelter and Groher [168] describe a model-based development framework for product lines, which integrates feature modelling and aspect composition with different development stages. They combine the usage of feature models with analysis and design models, and apply model weaving for generating product configurations. They also apply code generation techniques and weaving of code generation templates for generating application code.

The relationship between features and concerns has been the topic of several early aspect workshops[14] and has also been addressed by the thesis author in [138], which looked at feature representation with model-based aspects.

Kalleberg and Visser [96] present an aspect-oriented extension to the Stratego language, which is a program transformation language based on term rewriting. The rewrite rules operate on terms defined by a language grammar, such as the Java language grammar. The aspect-oriented extension allows crosscutting rewriting rules to be modularised and composed. By integrating the MOFScript transformation language grammar in Strategy, the higher-order transformations described in paper II could be implemented by rewrite rules and aspects.

Morin et.al. [127] present an approach for managing dynamic variability using aspect-oriented and model-driven techniques. Re-configuration of the running system is managed by having a causal connection to a configuration model, which is constructed using aspect composition of variant models with a base model.

Feature Interactions and Confluence. In both papers II and V, we address confluence of variability transformations. In paper II, this is addressed by showing that the feature transformations are confluent for certain types of variability. In paper V, we specify a domain-independent variability transformation, and discuss the criteria for confluence or conflict in the application of this transformation.

Feature, or service interactions have been a well-known problem in the telecommunication sector for some time, and have been addressed by methods and formalisms since the early 1990'ies: Bowen et.al. [26] identify it as a problem in several development phases, i.e. in specification, design, testing, and execution. Wakahara et.al. [170] describe a method for feature interaction detection, and classify feature interactions in six categories: duplication, redundancy, incorrect execution order, inconsistency, non-determinism, and looping.

Similar to telecommunication systems, interactions between features represent a challenge both in product line engineering and aspect-oriented development. In AOD, the problem occurs as interactions – or conflicts – between aspects.

Metzger et.al. [122] describe an approach for semi-automated detection of feature interactions in a product line context. Using goal modelling in the Goal Oriented Requirements Language (GRL), where goals reflect features, they describe an algorithm for detecting feature interactions. This is mapped to a feature modelling context, wherein variation points and variants are explicitly specified with a feature modelling

[14] e.g. Early Aspects at AOSD 2008 and at SPLC 2008

notation [74]. They specifically address vague feature dependencies, categorised as *hints-dependency* and *hinders-dependency*, referring to positive and negative influence between features, respectively.

Czarnecki and Pietroszek [43] present an approach for verification of well-formedness of the feature-based model template approach previously described [40]. They define a semantic extension to OCL, *template interpretation*, which allows constraints to be interpreted for model templates (MOF-based models annotated with presence conditions). Model templates with OCL are verified using a SAT (Boolean Satisfiability Problem) solver based on Binary Decision Diagrams (BDDs), which provide a way of detecting illegal template configurations. This is complementary to our work on conflict detection in paper V; we use set theory to detect model element overlap between pairs of features while they analyse the legality of complete configurations.

Thaker et.al. [166] address type safety of product line composition by analysing global consistency of feature modules and the their combinations. In their work, feature modules are represented by AHEAD expressions [18]. Global consistency, which addresses type references and ambiguities, is checked by code compilation. Type safety of module combinations is analysed based on a set of defined constraints that must be satisfied to ensure type safety. Code analysis is used to extract constraint instances, and a SAT solver is used to check if constraints are violated.

5.4 Interactions, Conflicts and Controlled Compositions in AOD

S. Katz [99] presents a survey of approaches for verification and static analysis of aspects, which gives an overview of verification and analysis techniques for aspect-oriented technologies. He describes some essential issues related to this topic, three of which are strongly related to the work in this thesis.

- Ensuring that desired properties of the base system are maintained (or preserved) in the composed system.
- Establishing whether aspects interfere with each other.
- Ensuring that the desired properties of the aspect are added to the base system.

Aspect Interactions and Conflict Detection. Havinga et.al. [83] use graphs to model and detect composition conflicts that may arise when introducing elements with aspects. Several examples of potential conflicts are given in [83]; one example is the introduction of a method in a class that unintentionally overrides a method in a superclass. They map Java programs to graphs and AspectJ introductions to graph transformation rules. Their approach is implemented in the graph transformation tool GROOVE[15] to execute the matching and application of transformation rules on the program graphs. Conflicts are modelled as graphs representing language violations, such as naming conflicts, which can be checked by the graph transformation tool during rule application. Their approach is complementary to our work on composition

[15] http://groove.cs.utwente.nl/groove-home/

contracts, as they explicitly specify conflicts, while we implicitly specify conflicts by invariants.

Mehner et.al. [121] address detection of interactions and potential inconsistencies in model-based aspects at the requirements-level. They map activity diagrams and aspects to *attributed typed graphs* implemented in AGG [56], and specify pre- and post-conditions for activities as graph productions. They define graph transformation rules for *before, after* and *replace* types of composition, and use critical pair analysis to detect conflicts among the activities. The pre- and post-conditions are assertions on the domain model that define the criteria for detecting conflicts.

Conflicts in aspectual requirements composition is also addressed by Brito et.al. [28], who use the *Analytic Hierarchy Process (AHP)*, a structured technique for dealing with complex decisions. Based on pairwise prioritisation of different concerns (alternatives) in the context of different criteria (in the their example, those were concern contributions and stakeholder importance), priority vectors are calculated, resulting in a prioritised ranking of concerns. This can be used to resolve conflicts arising from concerns that contribute negatively to each other.

Bernardi and Di Lucca [23] propose a taxonomy for interactions introduced by aspects, which considers three interaction categories: interactions introduced by *altering static structure*, *the control flow*, or the *object states*. Their work addresses the interaction between aspects and the base system, not on interactions among aspects. However, a similar taxonomy could be useful also for analysing and understanding aspect interactions.

Kienzle et.al. [107] analyse interactions an aspect-oriented framework for transaction handling and address how AspectJ provides support for managing such interactions. They define a set of language features desirable for handling such interactions: separate aspect binding, inter-aspect configurability, inter-aspect ordering, per-object aspects, dynamic aspects, and per-thread aspects. The established case study can also be useful for similar analysis of other aspect-oriented technologies, such as modelling notations.

Kniesel [111] defines an approach for detection and resolution of interactions in aspect weaving. Weaving interaction and interference are identified as basic problems in aspect-oriented development and *interaction* and *interference* are both defined formally. The work is based on formalisation of language-independent aspect-oriented terms, such as aspect effects and predicates. On this basis, Kniesel defines a fully automated algorithm for detection of weaving interactions, and presents strategies for static and dynamic resolution of the interactions.

Aksit et.al. [2] use graph transformations for analysing aspect interference at shared join points in aspect-oriented programs. They represent join points as graphs, and the run-time semantics as graph production rules, which are derived from aspect oriented programs defined using composition filters. From these graph transformations, a state space represented as a labelled transition system (LTS) is generated, which represents all orders of execution of different aspect advice at shared joint points. The resulting LTS can be used to determine if there are interactions between the original aspects.

In [72], Grønmo et.al. establish a theory for reasoning about confluence of sequence diagram aspects. They show that confluence is undecidable for sequence diagram aspects where pointcuts can contain arbitrary event symbols along lifelines or negative pointcut diagrams. By restricting the expressiveness of pointcuts, they show that confluence can be decided by critical pair analysis.

Constraining Compositions. Flexible composition mechanisms, such as in aspect-orientation, provide power to the concern engineer, but leaves little control over what may happen to the base model or code. The result may be that assumptions the base model engineer made about the base model or program are broken. Design by Contract (DbC) was introduced to assert behavioural compliance between consumers and providers. Similar to DbC, which governs interactions between components, compositions may also be controlled or restricted by constraints. In this thesis, we address the usage of DbC-like principles to help controlling the behaviour of aspect composition (paper VI, Appendix F).

Katz and Sihman [100, 154] describe an approach for validation of aspects using model checking with *superimpositions*. A superimposition is a collection of parametrised aspects and classes, which is specified independently of any base program. A superimposition specifies applicability conditions and desired result assertions globally and for each aspect. A superimposition is defined to be *correct* if, for any base program that satisfies all assumptions and requirements made by the superimposition, the composed program – called augmented program in [100] – obtained by a legal binding of the superimposition with the base program, is correct. Correctness of the base program is based on a specification of its desired properties, which should hold for the base program and the composed program. They implement superimpositions with a preprocessor called SuperJ [154], which translates to AspectJ and Java. The validation of aspects is supported by the *Bandera temporal specification language* [77] for specifying the desired properties of the base program. Katz and Sihman's work relates to our notion of composition contracts; their specification of desired properties for the base program is close to our composition contract post-conditions.

Griswold et.al. [66] present *crosscutting interfaces* (XPIs), which insulate aspects from implementation details and expose only desired behaviours, using defined interfaces (XPIs). The approach supports specification of invariants, checking them, and reporting any violation. Although XPIs are defined using AspectJ, the approach can be applied in other technology contexts as well. An XPI is defined by four elements: a name, a scope, a set of abstract join points, and a partial implementation. The abstract set of join points is expressed in terms of a pointcut descriptor (PCD) signature and a *semantic specification*. The latter states *pre-conditions* that must be satisfied at each point an advice can run, and *post-conditions* that must be satisfied after an advice has run. The XPI approach also prescribe constraints – or design rules – that define how code shall be written in order to ensure consistent application of aspects. Our work on composition contracts is inspired by their work on XPIs, but differ in that we utilise meta-models to define contracts, which are independent of a specific modelling or programming language. This allows contracts to define constraints on elements from different models, or even different modelling languages. The XPI work relies on object-oriented interfaces for specifying the design rules, which are not directly applicable for our modelling context.

Klaeren et.al. [108] define an aspect composition and validation mechanism based on assertions. Their composition approach uses a knowledge base of valid aspect configurations to specify, for each class, which aspects are valid. In particular, it supports specifying valid configurations of several aspects. A similar knowledge base could complement our finer-grained approach for composition contracts, in which we currently do not consider interactions of multiple aspects.

Kizcales and Mezini [105] establish the concept of *aspect-aware interface*, as a way

of representing the interface between an aspect and the java code it crosscuts. The interface is a specification of all references between classes and aspects, which they use to show that modular reasoning is feasible also for aspect-oriented programs. They do not address constraints or restrictions on the composition of aspects. The aspect-aware interface, however, can be used in in conjunction with composition contracts, similar to how *assumption contracts* are used in paper VI.

Ossher [139] describes a hiding mechanism that requires a base program to explicitly *confirm* or *deny* pointcuts that aspects can use in advising it. In addition, the decision can be based on organisational roles and responsibilities, e.g., any aspect defined by a particular organisation is confirmed. The concept of confirmed roles and organisations is not addressed by the *composition contracts* in paper VI, and could be a useful extension to the contract approach. Contracts may be extended with human or system roles, and access policies governing their right to modify the base system.

Dantas and Walker [44] present a language with *harmless advice*, which is designed to prevent interference with the computation of the main program. Hence, harmless advice can be added to a system without fear of breaking any system invariants. The approach allows advice to modify termination behaviour or to use input/output, but may not otherwise change the final result of the main code. They introduce a language based on typed lambda calculus, which is used to specify aspect-oriented systems. The main benefit of the approach is that local reasoning of the main program is preserved regardless of any aspects applied.

Djoko et.al. [49] identify categories of aspects that preserves some classes of properties: observers, aborters, confiners, and weak intruders. It is then sufficient to prove that an aspect belongs to a category to know which properties that are preserved by composed programs. They formally prove that, for any program, the weaving of any aspect in a category preservers the properties of that category class.

5.5 Semantics Preservation

Engels et.al. [55] address preservation of model consistency of UML Real Time (UML-RT) models [152] during evolution. They use UML-RT to describe architecture and behaviour using *capsules, connectors, protocol roles*, and *protocols*. The protocols are defined by state machines, which specify the protocol behaviour between capsules (protocol behaviour) and the internal capsule behaviour. They describe a mapping to Communicating Sequential Processes (CSP) [89], which provide a formal language for describing communicating, concurrent processes. Its semantics is defined in terms of traces, failures, and divergences, and define several notions of *process refinement*: a process Q is a refinement of a process P if the $traces(Q) \subseteq traces(P)$. This semantics is used to analyse protocol consistency and deadlock freedom in UML-RT specifications. The notion of consistency defined in their work is similar to our notion of semantics preservation for sequence diagram aspect composition, which is based on refinements of sequence diagrams represented by trace semantics.

There is a body of work related to program transformation that particularly address semantics preservation. Among the early works on program transformation, Baltzer et.al. [14] introduced *transformation implementation (TI)*. In the TI approach, an abstract program specification is constructed, which is subject to optimisations by following automated transformations. The transformations are maintained in a catalogue

along with the conditions under which they are *equivalence preserving*. In compiler verification, or certification, mechanical or automated proof systems are used to formally prove that compiled code preserve the semantics given in a specification [125]. Program or model refactoring can be considered a special class of transformation, in which semantics preservation is a requirement.

Verification is also addressed in the context of aspect-oriented implementation and specification:

Katz and Katz [98] describe an approach for verifying the conformance of a system to scenario-based aspect specifications. They define a system to be *conform* with a scenario-based specification if all its computations are equivalent to a *convenient* computation. A convenient computation is one in which only specified scenarios occur in some order. Aspects are categorised as *spectative*, *regulative*, or *invasive*. A spectative aspect do not affect state or conditions that modifies exiting behaviour. Regulative aspects may modify the control flow, but may not impact existing behaviour. Finally, invasive aspects may modify state and behaviour. Using fair transition system theory, they verify the conformance of a system to aspect-oriented scenario specifications. Their work addresses a complimentary problems to ours on semantics preservation for sequence diagram aspects; they address the adherence of aspect scenario semantics to a refinement (the implementation), while we address the preservation of refinement relationships when applying sequence diagram aspects.

Another approach for verification of aspect-oriented systems is given by Krishnamurthi et.al. [113]. They modularise verification of aspect-orientation by separating verification of a base program and aspect advice. They employ model checking based on Computation Tree Logic (CTL) and express aspect-oriented programs as state machines. The desired system behaviours, which are subject to verification, are specified as temporal behaviours in CTL. By modularising the verification, the cost of re-verification is reduced when changes are imposed on aspect advice.

Chapter 6

Contribution Overview

The contributions of this thesis are manifested by the research papers in Appendices A through F. They are all concerned with notions of concern composition, primarily at the modelling level, and target our four research topics. In this chapter, we give a brief summary of these contributions.

6.1 Semantics Preservation of Trace-Based Aspect Composition

Sequence diagrams are used to describe interactions between system parts, and represent execution traces of a system. They are commonly used to specify requirements in terms of obligated or prohibited system behaviours, or the detailed specification of such behaviour. In order to address behaviours that crosscut the system, we look at how trace-based aspects – in terms sequence diagram or Java-based aspects – affect the system. Specifically, in papers III and IV (Appendices C and D), we address how semantics can be preserved when sequence diagrams aspects are composed with existing base model or code. In the former, a definition of semantics preservation for sequence diagram aspects is given, which is used for evaluating several existing sequence diagram modelling approaches. In the latter, these ideas are mapped to trace-based aspects in Java.

Semantics Preservation of Sequence Diagram Aspects. In paper III, we use the trace-based semantics defined by STAIRS [82] to establish a definition of semantics preservation of sequence diagram aspects. Two properties are considered as contributing factors to semantics preservation: monotonicity of aspect composition with respect to refinement relationships, and preservation of events in base models. Monotonicity of aspect composition with respect to refinement means that refinement relationships between sequence diagrams are preserved when aspects are applied to them. For two sequence diagrams SD and SD' and an aspect A, where $SD' \rightsquigarrow SD$ ($\rightsquigarrow$ is the refinement operator), the composition of A with SD' ($SD' \oplus A$) should also be a refinement of $SD \oplus A$, i.e. $SD \rightsquigarrow SD' \implies SD \oplus A \rightsquigarrow SD' \oplus A$.

We address the monotonicity both with respect to base model and aspect advice refinement, and define *semantics preservation* of sequence diagram aspect composition based on the two properties monotonicity and event preservation.

This is illustrated in Figure 6.1: Figures 6.1(a) and 6.1(b) illustrate the refinement of the sequence diagram SD and aspect A, respectively. The refinement relation is maintained in the composition in both cases.

(a) Refinement of the sequence diagram (SD)

(b) Refinement of the aspect (A)

Figure 6.1: *Monotonicity of sequence diagram aspect composition with respect to refinement*

We restrict refinement of aspects to *refinement of the advice* and show that refinement of aspect pointcuts generally leads to compositions that are *not* semantics preserving. This is because pointcut refinement potentially adds or removes matches, resulting in compositions that are not refinements. This is further detailed in paper III.

We use our definition to analyse six different sequence diagram approaches with regards to the semantics preservation property, three syntactic and three semantic-based approaches. The results of that analysis show that both syntactic and semantic approaches can be semantics preserving. The one thing that led the syntactic approaches to be semantics preserving was tight restrictions on binding and querying.

Semantics Preservation of Trace-Based Aspects in Java. In paper IV, we define a trace-based semantics for Java inspired by STAIRS sequence diagram semantics, called Java-STAIRS. This semantics is used for reasoning about semantics preservation for Java systems and their refinements, based on the definitions of semantics preservation for sequence diagram aspects given in paper III. In Java-STAIRS, a trace is modelled as a sequence of events, where each event contains information about the sender object, the receiver object, the message called, and the execution thread.

A mapping from sequence diagram aspects to a Java trace-based aspect technology called *Tracematches* [3, 10] is defined. This allows pointcuts and advice initially described by a sequence diagram to be handled by the Java run-time. An example of a tracematch specification was given in Figure 5.2, Section 5.1.

We define *trace filters* for filtering Java execution traces that are not relevant in the context of a specific sequence diagram. The filters are mapped to appropriate mechanisms in the trace-based aspect system. Using a similar refinement semantics for Java-STAIRS as that defined in STAIRS, we show that Java-STAIRS aspects are semantics preserving.

The benefit of only allowing modifications that are valid refinements under the given definitions is, related to the trace-based aspects, that a concern will work consistently for a system also after it has been refined, or, in other words, *semantics is preserved*.

Validation of the Contribution. Our definitions of semantics preservation applies to sequence diagram aspect composition and trace-based aspect composition in Java. It gives a means of assessing if a specification or implementation change will impact

the effects of existing aspects. If the change *is* according to our definitions, we know that the effects imposed by existing aspects will be preserved. It *not*, we know that the effects of existing aspects might be altered. *With this, we can help ensuring a consistent evolution of our system in the presence of trace-based aspects.*

Our definitions are based on the semantics of the languages they address, i.e. traces in sequence diagrams and in Java executions. For evaluation of applicability for sequence diagrams, we analysed six different sequence diagram aspect approaches. For evaluation of applicability for trace-based aspects in Java, we analysed the impact of changes in a graphical Java application with respect to the effect of existing trace-based aspects.

6.2 Confluence and Conflict in Feature Composition

In the works represented by papers II and V (Appendices B and E), we address feature composition using transformations, with a particular focus on how ordering of feature selection influences the resulting product. When a set of features is selected for a product, there may be several paths toward the fully composed product. If these paths yield the same result, the transformations are *confluent*, and the product developer does not have to spend resources checking this. If the different paths yield different results, i.e. they are not confluent, the product developer must decide the ordering of the feature resolution, and hence the transformations.

We address confluence in paper II by using aspectual transformations that modify a generative product line transformation. This is referred to as higher-order transformations (*Hi-Transforms*). The product lines are defined by models, a transformation that generates product code, and higher-order transformations defined by aspects that represent features of the product line. The effect of these are variants of the base transformation, which in turn can produce variants of generated code. The paper showed that the higher-order transformations were especially suited for variability related to the implementation platform (such as representation of properties in the code), and that, given a clear separation between platform and domain information, the variability transformations will be confluent.

We address confluence in paper V by implementing a domain-independent product line transformation, based on a general variability model introduced by Haugen et.al. [80]. Based on the properties of this transformation, we show that certain kinds of features, represented by model fragments, are confluent. We also show under what conditions the features in this model are in conflict. Finally, the paper describes an algorithm for checking confluence. The results of this paper apply theoretical analysis based on an implementation of product line transformations represented by the variability model. The results can be used in practise when building product lines with the variability model, for reasoning about confluence, and for making decisions about the ordering of feature resolutions or detecting conflicts.

Validation of the Contribution. We have described two cases of generative feature composition. In both cases, we implemented transformations for generating the product configurations based on variability resolutions. In the first case, we inserted

transformation directives into product line transformations to accommodate variability. The transformations were applied on an example model and a specific product line code generator. The approach, however, is general and can be applied for other models and transformations. We showed that, if the base transformations separate platform-specific from domain-specific logic, the features can be composed (i.e. the transformation can be applied) order-independently. In the second case, the transformation is general, and can be applied to any domain model; it implements variability resolutions specified by the general variability model from [80]. We defined a theory for reasoning about confluence and conflicts of defined variability, and implemented a confluence checking algorithm. We used an example train station DSL [163] to validate the approach.

6.3 Model Composition Contracts

Contracts are well acknowledged tools for regulating business-level transactions, where several parties agree on mutual obligations and expectations. In software engineering, the contract between components is often defined by interfaces with operation signatures. To strengthen these, interaction protocols may define how components should collaborate, or state machines may specify in detail how an event changes the state of a component. An approach that extends interface definitions, which was called *design by contract* (DbC), was defined as part of the Eiffel programming language [123]. In DbC, the notions of obligations and expectations from business-level contracts are taken into class and method definitions as pre- and post-conditions, and invariants.

This strengthening of the interface – the contract – was used as a quality assurance mechanisms, to build more robust systems, and for providing better documentation for systems.

In Paper VI (Appendix F), we apply the ideas from DbC in the context of model composition, and define *Composition by Contract (CbC)*. (The term Composition by Contract was, however, established within this thesis, not in the paper.) Given a system design, in which concerns are specified by different teams than those responsible for the base application, it is hard to specify a clear interface with obligations and expectations related to concern compositions. This may result in concerns, or aspects, that perform unintended changes to the application, and that may be difficult to discover and expensive to fix. The resulting specifications may have inconsistencies or broken assumptions, which may in turn lead to system faults, integration errors, etc. Model-based composition contracts are intended to remedy this.

The model composition contracts associate constraints with models, which govern the eligibility of compositions to modify the models. Figure 6.2 illustrates the approach. A composition contract is related to a model, and defines constraints governing the access and modification of that model. The constraints specify which *elements* that are allowed to be accessed by a composition, and *accessors* that restrict further what kind of access can be granted specific types of elements (e.g. only classes within the 'CarSystem' namespace). *Modifications* specify, for various accessible elements, what features that can be modified. *Post-conditions* define constraints on the model that should be valid after a composition has been executed. Finally, *helper queries* can be used to define helper operations used within constraints.

The approach uses OCL, with two extensions, for specifying the constraints in

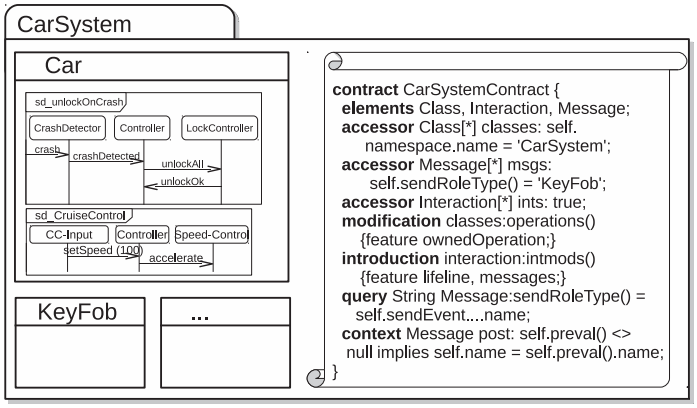


Figure 6.2: *A model with an associated composition contract*

a contract: one extension is the *matches* operation, which provides string matching, which can be useful for quantification within queries. The other is the *preval* operation, which provides access to model element values prior to composition, which is necessary for expressing post-conditions relative to pre-composition states.

To check a composition contract in the course of a composition, the contract is compared with an *assumption contract* that represents the assumptions of the aspect being composed. The export of the assumption contract is the responsibility of the composition engine. The checking process is implemented by a prototype tool, which analyse the contracts; a violating assumption contract yields a negative verdict, which should result in abortion or rollback of the composition.

Validation of the Contribution. We defined and implemented a language and tool support for specifying and checking model composition contracts. The approach helps guarding system specifications against unintended or harmful modifications imposed by aspect compositions. This can help increasing system robustness and ensuring that system specifications stay according to expectations also in the presence of aspects. We evaluated the approach using an established example system for store collaboration [45], which displays many of the complexities of enterprise systems, such as patterns for product exchange between stores.

6.4 Model-based Aspect Representation

In the course of our work, we have addressed language representation of aspect models. In paper I (Appendix A), we address architectural aspects in UML. Papers IV and VI (Appendices D and F) introduce a UML notation for specifying sequence diagram aspects. This notation is used for representing behavioural concerns in the case studies.

Architectural Aspects. The architectural aspects are defined by structured classes containing parts, ports, and connectors. Formal ports defined on the aspect class represent elements that have to be bound during composition. This part of the aspect corresponds to an *advice* in aspect-oriented terms. A textual or graphical binding

language specifies how an aspect is composed with a base model by binding port pairs in the aspect to sets of port pairs, or connectors, in the base model. The binding language corresponds to the *pointcut language* in aspect-oriented terms. This approach provides a way of refining connectors to complex structures, thereby providing a mechanism for architectural variability, which is not supported by the standard UML language mechanisms.

Sequence Diagram Aspects. Sequence diagram aspects are defined using a UML sequence diagram notation in papers IV and VI. The notation builds on existing works by others, specifically that of Whittle et.al. [176] and Hanenberg et.al. [75]. In this thesis work, we used the sequence diagram aspects for generating other artifacts. In the work described by paper IV, the aspects were used to generate code for trace-based aspects in Java, called Tracematches [3], which in turn could be compiled and executed to match traces in Java programs. In the work represented by paper VI, the aspects were used to generate the assumption part of a contract specification, which in turn was used for checking against a specified composition contract.

Validation of the Contribution. Our architectural aspect approach provides architectural variability of UML architecture specifications defined by structured classes. Each aspect act as a refinement of architectural connectors, and allows complex connectors to be modularised and reused many places in an architecture. To validate the approach, we compared it with the built-in mechanisms of UML and showed that it provides useful extensions. We applied the approach on a mobile positioning system example with crosscutting concerns such as access control and transaction. The sequence diagram aspect representation provided a way of specifying trace-based aspects *and* generating code-based aspects and contracts.

Chapter 7

Discussion

In this thesis we addressed a set of research topics – specified in Section 3.2 – through contributions reflected by our research papers. They all contribute toward the main research topic, which was defined as follows:

- To what extent – and how – can model-based composition mechanisms *guarantee* consistency and semantics preservation of the models subject to modification?

Here, we revisit and discuss the research topics.

7.1 Research Topic 1 – Semantics Preservation of Trace-Based Aspect Composition (RT_1)

RT_1 addressed how behavioural composition can be semantics preserving. This was decomposed in three sub topics that address how semantics preservation can be defined ($RT_{1.1}$), what currently is lacking for composition of behaviour models to be semantics preserving ($RT_{1.2}$), and what the benefits of semantics preservation are ($RT_{1.3}$).

We target these topics in paper III and IV (Appendices C and D), wherein we give a definition of semantics preservation ($RT_{1.1}$) for sequence diagram aspect composition based on trace-semantics. The definition uses refinement as a key element for being semantics preserving, such that aspects that are applied on a system S will still work on any system S', if S' is a refinement of S. This is based on the definition of refinement given in STAIRS [79], in which any legal refinement must preserve traces – either as positive or negative. The other key element for semantics preservation in our definition was *preservation of events*.

Our definition of semantics preservation is tied to the refinement semantics defined by STAIRS. It may not work equally for other sequence diagram semantics and may require reworking. For example, in the stream-based MSC semantics defined by Krüger [117], property refinement can restrict behaviour by e.g. removing alternatives or interleaving. In these cases of refinement, the semantics preservation property will not be satisfied. Herein lies interesting areas for future research; how to cope with different notions of refinement in the context of semantics preservation.

In paper IV, we extended this work toward Java implementations by mapping sequence diagram aspects to trace-based aspects in Java. By mapping STAIRS trace semantics to Java (Java-STAIRS), the paper showed how the semantics preservation

definition applied in the implementation context. This result depends on the same refinement notion, only mapped to traces of Java events.

Our analysis of existing approaches revealed that semantics-based sequence diagram aspect approaches will be semantics preserving as long as they do not remove events, i.e. delete messages. However, if we relax our definition to this only consider refinement, all the semantics-based approaches analysed will be semantics preserving. We also saw that syntactic-based approaches can be semantics preserving, if their method for querying and binding is restricted to static binding of elements. For the trace-based aspects in Java, restrictions on allowed program refinements need to be enforced in order to be semantics preserving ($RT_{1.2}$). We argued that our definition of semantics preservation caters for increased consistency of our system during evolution ($RT_{1.3}$).

Our definition of semantics preservation allows systems to be modified by refinement in the presence of trace-based aspects without compromising the effect of the aspects. If arbitrary changes were allowed, no guarantees could be made with regard to aspect effects. However, this can potentially be extended in a controlled manner beyond STAIRS refinement. In the context of a set of existing trace-based aspects, we can define regions in the base models or code, which encapsulate the join points selected by the aspects. Changes that result in non-refinement can then be made outside the join point regions, while changes inside the regions must conform to a definition of local refinement. Such extensions are potential subjects for future work.

7.2 Research Topic 2 – Confluence and Conflict in Feature Composition (RT_2)

This research topic was decomposed in the following sub topics: how to analyse confluence of such compositions ($RT_{2.1}$), and how to establish the conditions for determining confluence ($RT_{2.2}$).

The research papers addressed these topics in two different product line approaches. In paper II (Appendix B), feature variability representation and composition are handled by *hitransforms*, which modify a product line code generator (transformation) to a specific configuration. Platform-specific variability – such as *the implementation choice for associations* – is separated from domain-specific variability – such as *my online bank includes feature electronic invoice*. The paper argues that there is confluence between platform and domain variability transformations, given that the base transformations separate these concerns in the first place. In this case, confluence can be analysed and determined ($RT_{2.1}$, $RT_{2.2}$) based on how well the transformations separate platform-specific concerns from domain-specific concern, e.g. by avoiding mixture of both within a single transformation rule.

In paper V (Appendix E), confluence is addressed more generally, based on domain-independent transformations for product line feature composition. The transformation resolves variability by manipulating the object structure of the product line. Variability is modelled as substitutions, and a feature resolution selects one particular object structure – *replacement fragment* – that should replace another – *placement fragment*. The paper shows how interacting features – defined by overlapping fragments – result in conflicts, and specifies how to check if two features are confluent and without conflict such that they can be resolved without considering their ordering. It shows how confluence can be analysed ($RT_{2.1}$) by navigating the object structures associated with

features, and that confluence can be determined ($RT_{2.2}$) by analysing overlap between their respective object structures.

Knowledge about the confluence characteristics of features does not guarantee error-free feature composition. There may still be conflicts between features that are not captured by structural analysis, e.g. architectural conflicts that need to be explicitly represented by feature conflict relationships. The proposed approach can be augmented with additional conflict analysis techniques to detect and handle these kinds of conflicts. Finding such conflicts is important in order to prevent failures in the deployed products; finding them early in the development process can be crucial for reasons such as safety, cost, and reputation. In the extension of feature conflict analysis, product line configurations may be diagnosed post composition, to detect potential errors in the composition, for instance such as described by White et.al. [172], where feature configurations are translated to constraint satisfaction problems, which in turn are used to diagnose potential problems. Another, complimentary approach, is to increase the tolerance for conflicts in the running system such as in the approach by D'Souza and Gopinathan [53], where they define a priority-based composition scheme to ensure utilisation of features even if conflicts occur.

7.3 Research Topic 3 – Model Composition Contracts (RT_3)

This research topic was decomposed in the following sub topics: what does it mean to constrain compositions by contracts ($RT_{3.1}$), and how can such contracts be specified ($RT_{3.2}$).

These topics were addressed by paper VI (Appendix F), which establishes the *composition contract* concept. Composition contracts allow model compositions to be constrained by specifying policies that identify explicitly allowed changes, hence hindering undesired changes ($RT_{3.1}$). An unwanted modification may be simple structural or behaviour properties that the base model engineer wants to preserve, or details of behaviour, such as interaction protocols, that should be protected from modifications to ensure that the protocols are not broken. The enforcement of the composition contract makes this a controlled composition with respect to the wishes of the policy stakeholder, which might be the base model or program developer. The contracts are specified in a contract language, and reflect on the properties of the base model and how they can be accessed and modified ($RT_{3.2}$).

Our composition contract approach is based on general MOF-based models, and can be applied to any kind of model artifact. As such, the mechanism is very general, as it allows contracts to be specified for different kinds of models based on any kind of meta-model, possibly combining several models or model views in a single contract. One can argue that this generality results in a complicated contract authoring process as well as a complicated contract specification. To a certain extent, this can be remedied by meta-model-specific libraries – or helper queries – that provide simplified *views* on the meta-model elements of interest. This kind of complexity can also be supported by simplifications on the meta-model and model, along the lines of the *meta-model pruning* algorithm described by Sen et.al. [153].

The composition contracts define constraints on behalf of base models, and the checking process requires an *assumption contract* to be exported, or generated, based

on the aspect. In the current contract checker implementation, each aspect is associated with a separate assumption contract. This is, however, not a limitation in the approach, which could allow the assumption of several aspects to be specified in a single contract. Grouping assumptions based on the aspects is a reasonable modularisation, but the approach would benefit from being able to share elements common to many aspects.

7.4 Research Topic 4 – Model-based Aspect Representation (RT_4)

This research topic was decomposed in two sub topics: how UML architectural aspects can be represented and with what benefit ($RT_{4.1}$), and how UML sequence diagram aspects can be represented and with what benefit ($RT_{4.2}$).

We address UML architectural aspects ($RT_{4.1}$) in paper I, where we propose *ArchSpects*, which is an aspect notation and semantics based on UML composite structures. One of the main benefits of the approach is the provision of architectural variability by connector refinement through the aspects. This allows complex connectors to be modularised and reused in many architecture configurations. This kind of connector refinement extends the built-in abilities of UML to refine architectures. With the proposed mechanisms come challenges related to *semantics preservation*: modifications of the structural architecture have implications for the underlying semantics. In the paper, we address the consistency of the types defined on architectural elements. However, behavioural consistency is not addressed. If the underlying architectural elements have associated behaviours, e.g. described by sequence diagrams or state machines, these behaviours ought to be consistently modified along with the architecture.

One path in this direction can be found in alignment with the work of Kienzle et.al. [106], who describe a multi-view approach for aspect-oriented modelling, which maintains consistency between the different views (see Section 5.1). Another path can be found in the work by Engels et.al. [55], who address model consistency preservation of UML-RT models with architecture and state machine behaviour (see Section 5.5).

UML sequence diagram aspect representations ($RT_{4.2}$) were addressed in papers IV and VI. The introduced notation built on other, related notations, but was adapted to fit standard UML notation and tools. The main benefits gained were usage of the sequence diagram aspects for generative purposes: generating trace-based aspects in Java and generating assumption contracts in a contract checking process.

Chapter 8

Conclusion

8.1 Summary of Contributions

In this thesis, we have addressed *semantics preservation in model-based composition* through a set of complimentary focus areas:

- we gave a definition of semantics preservation for sequence diagram aspects, which is based on sequence diagram refinement and event preservation. When complying with this definition, a sequence diagram – base model or aspect advice – can be modified by refinement, and aspects that had an effect on the original base model, will also have the expected effect after the modification. We gave a definition of Java-based trace semantics to which we mapped our definition of semantics preservation. We established a mapping from sequence diagram aspects to trace-based aspects in Java and showed the usefulness of semantics preservation for Java executions. *Semantics preservation helps ensuring that the intended effect of aspects is not lost due to modifications.*

- we established techniques for product line feature composition and theories for analysing confluence and potential conflicts among features. We showed how potential conflicts among features can be detected, and hence analysed and avoided, and how confluence between features can be analysed. Knowledge about confluence helps a product developer to determine if the order of feature composition *is important or not.*

- we defined a technique for model composition contracts – Composition by Contract (CbC), which included a language to specify and a prototype tool to check contracts. The CbC approach allows access constraints to be associated with models in terms of pre-conditions and post-conditions; these govern the eligibility for compositions to access and modify a model, and help the base model engineer to protect her/his assumptions regarding the base model. *CbC helps guarding models from inconsistencies and errors introduced by compositions.*

- we defined specification techniques for model-based aspects, specifically for architectural and behavioural aspects using UML composite structures and sequence diagrams. The architectural aspects – called ArchSpects – provide a way of modularising complex connector structures, which can be used to make crosscutting refinements of architectural connectors and providing architectural variability.

Sequence diagram aspects in UML were utilised for mapping to trace-based aspects in Java and for representing assumption contracts in the work on model composition contracts.

8.2 Directions for Future Work

From the work we have accomplished in this thesis, we see several threads worth investigating as part of future work. We relate this to the domain of complex and adaptive systems, which is – and will increasingly become – an important part of everyday life. Complex, adaptive systems may involve mobile devices, stationary services, embedded software, and sensors, that interact and collaborate to provide a set of services. They may be context dependent, and require adaption and re-configuration to accommodate changes in the environment, users, or architectural elements. One challenge in such re-configuration is to ensure that adaptions do not break functionality or quality properties of the system.

In complex, adaptive systems, *semantics preservation* can be an essential property. In particular for systems that need to ensure that adaptions do not break fundamental functional system properties, mechanisms that can help analyse and confirm semantics preservation will be useful. We want to pursue our work on semantics preservation in the context of complex, adaptive systems. We want to assess if our definition of semantics preservation for trace-based specification is adequate for practical applications of adaptive systems. As previously discussed, we may find that the model requires extensions in order to cope with different notions of refinement. It can also be interesting to address more flexible modes of modifications, e.g. by defining interface boundaries between aspects and base models and allow more extensive changes (than refinement) on the outside of such boundaries.

In addition, a more detailed view of semantics preservation can be useful, e.g. by complementing with state machine behaviour and architecture descriptions. To this end, existing work such as the consistency preservation of UML-RT specifications by Engels et.al. [55] can complement our work. This may also be complemented with feature-oriented views of adaptions, in which analysis of feature configurations can be used to detect inconsistencies.

There is ongoing work on standardisation of a variability meta-model within the OMG in what is called the *Common Variability Language (CVL)* [133]. A natural extension of the results on conflict and confluence analysis is development for supporting the outcome of the CVL standardisation process.

Composition contracts can be developed in several directions. One direction is improved usability and tool support. Tighter integration of contract specification with modelling tools will improve usability, e.g. by allowing in-place constraint annotation of graphical model elements. Integration with a traceability visualisation tool may be used to visualise the dependencies implied by a contract. Another direction is the mapping of composition contracts to implementation platforms in order to check and enforce run-time composition. We have initiated work on mapping composition contracts based on sequence diagrams to corresponding run-time contracts in an Enterprise Service Bus platform. Composition contracts may be mapped to run-time monitors using aspect-oriented technologies, which are used to check compositions – or adaptions – at run-time.

There are several different perspectives to which our techniques, and their potential extensions, can be applied: one is at the level of design models, wherein analysis of semantics preservation, confluence, and contract adherence can be applied design-time. Another is at run-time, on model representations of the executing system (models at run-time). Yet another is on the executing system itself.

Semantics preservation related to refinement and composition of models or code is likely to become increasingly more important in increasingly more complex system domains with demands for adaptivity and re-configuration. Our contributions, and extensions of these, can provide some of this support.

– What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.
T.S.Eliot

Bibliography

[1] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 372–395. Springer-Verlag, 1992.

[2] Mehmet Aksit, Arend Rensink, and Tom Staijen. A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. *8th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 39–50, 2009.

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *20th Annual Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM, 2005.

[4] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transaction on Software Engineering Methodology*, 6(3):213–249, 1997.

[5] Michalis Anastasopoulos and Dirk Muthig. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In *Software Reuse: Methods, Techniques and Tools*, pages 141–156. Springer, 2004.

[6] João Araújo, Jon Whittle, and Dae-Kyoo Kim. Modeling and Composing Scenario-based Requirements with Aspects. In *Proceedings of the 12th IEEE International Requirements Engineering Conference*, pages 58–67, 2004.

[7] Aristotéles. *Prior Analytics, Translated by A.J. Jenkinson.* eBooks@Adelaide, http://ebooks.adelaide.edu.au/a/aristotle/a8pra/, 2007.

[8] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] P. Avgeriou, N. Guelfi, and N. Medvidovic. Software Architecture Description and UML. In *UML Modeling Languages and Applications*, volume 3297/2005 of *LNCS*, pages 23–32. Springer, 2004.

[10] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making Trace Monitors Feasible. Technical Report abc-2007-1, University of Oxford, UK, 2007.

[11] Eyvind W. Axelsen and S. Krogdahl. Groovy Package Templates: Supporting Reuse and Runtime Adaption of Class Hierarchies. *Proceedings of the 5th Symposium on Dynamic Languages (DSL), OOPSLA, ACM*, pages 15–26, 2009.

[12] Eyvind W. Axelsen and Stein Krogdahl. Pluggable Design Patterns Utilizing Package Templates. *Norsk informatikkonferanse (NIK) (Norwegian Conference of Informatics)*, 2009.

[13] Eyvind W. Axelsen, Fredrik Srensen, and Stein Krogdahl. A Reusable Observer Pattern Implementation Using Package Templates. *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 37–42, 2009.

[14] Robert Balzer, Neil Goldman, and David Wile. On the Transformational Implementation Approach to Programming. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 337–344, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[15] E. Baniassad, P.C. Clements, J. Araújo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering Early Aspects. *IEEE Software*, 2006.

[16] Elisa Baniassad and Siobhán Clarke. Theme: an Approach for Aspect-Oriented Analysis and Design. *26th International Conference on Software Engineering (ICSE)*, pages 158–167, 23-28 May 2004.

[17] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 702–703. IEEE Computer Society, 2004.

[18] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating Product-Lines of Product-Families. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE)*, pages 81–92. IEEE Computer Society, 2002.

[19] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[20] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197. IEEE Computer Society, 2003.

[21] Kent Beck and Ward Cunningham. A Laboratory for Teaching Object Oriented Thinking. *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 1–6, 1989.

[22] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful Traits and Their Formalization. *Computer Languages, Systems and Structures, Elsevier*, 34:83–108, 2008.

[23] Mario Luca Bernardi and Giuseppe Antonio Di Lucca. A Taxonomy of Interactions Introduced by Aspects. *International Computer Software and Applications Conference*, pages 726–731, 2008.

[24] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 5301:53–67, 2008.

[25] Grady Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[26] T. F. Bowen, F. S. Jhorack, C. H. Cbow, N. GniReth, G. E. Herman, and Y-J U. The Feature Interaction Problem in Telecommunication Systems. *Proceedings of the 7th IEEE Int. Conf. Soft Eng. Telecom Systems*, 1989.

[27] Gilad Bracha and William Cook. Mixin-based Inheritance. *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) and the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311, 1990.

[28] I. S. Brito, F. Vieira, A. Moreira, and R A. Ribeiro. Handling Conflicts in Aspectual Requirements Compositions. *Transactions on Aspect-Oriented Software Development*, 4620:144–166, 2007.

[29] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20:10–19, 1987.

[30] Peter Pin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transaction of Database Systems*, 1(1):9–36, 1976.

[31] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of Analysis and Design Approaches. Technical report, AOSD Europe, 2005.

[32] S. Clarke, W. Harrison, H Ossher, and P. Tarr. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code. *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 325–339, 1999.

[33] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design, The Theme Approach*. Addison-Wesley, ISBN 0-231-24674-8, 2005.

[34] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, ISBN 0-201-70332-7, August 2001.

[35] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, ISBN 0-13-630070-7, 1991.

[36] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Loewe. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. Technical Report TR-96-17, Department of Informatics, University of Pisa, Italy, 1996.

[37] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola WEAVR: Model Weaving in a Large Industrial Context. *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), Industry track*, 2006.

[38] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Stateful Aspects: the Case for Aspect-Oriented Modeling. In *10th international workshop on Aspect-oriented modeling (AOM)*, pages 7–14. ACM, 2007.

[39] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The Next Step in Web Services. *Communications of the ACM*, 46(10):29–34, 2003.

[40] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. *Generative Programming and Component Engineering*, pages 422–437, 2005.

[41] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co, ISBN 0201309777, 2000.

[42] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the Third Software Product Line Conference (SPLC)*, pages 266–283. Springer, 2004.

[43] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 211–220, 2006.

[44] Daniel S. Dantas and David Walker. Harmless Advice. *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 383–396, 2006.

[45] B. Demchak, V. Ermagan, E. Farcas, T. Huang, I. Krüger, and M. Menarini. *The Common Component Modeling Example: Comparing Software Component Models*, chapter A Rich Services Approach to CoCoME, pages 85–115. LNCS. Springer-Verlag, 2008.

[46] Peter J. Denning. *Encyclopedia of Computer Science*, chapter Computer Science: the Discipline. Nature Publishing Group, 156159248X, 2000.

[47] M. Deubler, M. Meisinger, S. Rittmann, and I. Krüger. Modeling Crosscutting Services with UML Sequence Diagrams. In *Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2005.

[48] Edsger Dijkstra. On the Role of Scientific Thought. *Republished in Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982. ISBN 0387906525*, 1974.

[49] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects Preserving Properties. *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 135–145, 2008.

[50] R. Douence, P. Fradet, and M. Südholt. *Aspect Oriented Software Development*, chapter Trace-based Aspects, pages 201–217. Addison-Wesley, ISBN 0321219767, 2004.

[51] Rémi Douence, Pascal Fradet, and Mario Südholt. A Framework for the Detection and Resolution of Aspect Interactions. *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE)*, pages 173–188, 2002.

[52] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *3rd International Conference on Aspect-oriented Software Development (AOSD)*, pages 141–150. ACM, 2004.

[53] Deepak D'Souza and Madhu Gopinathan. Conflict-Tolerant Features. In *Proceedings of the 20th international conference on Computer Aided Verification (CAV)*, pages 227–239. Springer-Verlag, 2008.

[54] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: the Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[55] G. Engels, R. Heckel, J. Malte Küster, and L. Groenewegen. Consistency-Preserving Model Evolution through Transformations. *5th International Conference on The Unified Modeling Language*, pages 212–226, 2002.

[56] C. Ermel, M. Rudolf, and G. Taentzer. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, chapter The AGG Approach: Language and Environment, pages 551–603. World Scientific Publishing Co., Inc., 1999.

[57] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, 2000.

[58] Robert W. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[59] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing Support for Model Composition in Metamodels. *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 253–266, 2007.

[60] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *IEEE Computer*, 39(2):59, 2006.

[61] R. France, D-K. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, pages 193–206, 2004.

[62] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented Approach to Early Design Modelling. *IEE Proceedings Software*, 151(4):173–185, August 2004.

[63] A. Garcia, C. Chavez, T. Batista, C. Sant'anna, U. Kulesza, A. Rashid, and C. Lucena. *Software Architecture*, chapter On the Modular Representation of Architectural Aspects, pages 82–97. Springer, 2006.

[64] David Garlan, Robert Monroe, and David Wile. Acme: an Architecture Description Interchange Language. *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7, 1997.

[65] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Frameworks*. Wiley Technology Publishing. Wiley, ISBN 0471202843, 2004.

[66] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[67] R. Grønmo, F. Sørensen, B. Møller-Pedersen, and S. Krogdahl. A Semantics-Based Aspect Language for Interactions with the Arbitrary Events Symbol. *European Conference of Model Driven Architecture Foundations and Applications (ECMDA), Springer*, 5095:262–277, Springer–Verlag 2008.

[68] R. Grønmo, F. Sørensen, B. Møller-Pedersen, and S. Krogdahl. Semantics-Based Weaving of UML Sequence Diagrams. *International Conference on Model Transformation (ICMT), Springer*, 5063:122–136, Springer–Verlag 2008.

[69] Roy Grønmo, Stein Krogdahl, and Birger Møller-Pedersen. A Collection Operator for Graph Transformation. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT)*, pages 67–82. Springer-Verlag, 2009.

[70] Roy Grønmo and Birger Møller-Pedersen. Aspect Diagrams for UML Activity Models. In *nternational Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, volume 5088 of *LNCS*, pages 329–344. Springer, 2007.

[71] Roy Grønmo, Birger Møller-Pedersen, and Gøran K. Olsen. Comparison of Three Model Transformation Languages. In *European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA)*, volume 5562 of *LNCS*, pages 2–17. Springer, 2009.

[72] Roy Grønmo, Ragnhild K. Runde, and Birger Møller-Pedersen. Confluence of Aspects for Sequence Diagrams. Research Report 390 ISBN 82-7368-351-6, University of Oslo, 2009.

[73] John Grundy. Aspect-Oriented Requirements Engineering for Component-Based Software Systems. *Requirements Engineering, IEEE International Conference on*, pages 84–91, 1999.

[74] Günter Halmans and Klaus Pohl. Communicating the Variability of a Software-Product Family to Customers. *Software and System Modeling*, 2(1):15–36, 2003.

[75] Stefan Hanenberg, Dominik Stein, and Rainer Unland. From Aspect-oriented Design to Aspect-oriented Programs: Tool-supported Translation of JPDDs Into Code. In *6th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 49–62, New York, NY, USA, 2007. ACM Press.

[76] William Harrison and Harold Ossher. Subject-oriented Programming: a Critique of Pure Objects. *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 411–428., 1993.

[77] John Hatcliff and Matthew B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. *CONCUR 2001 – Concurrency Theory (12st CONCUR'01)*, 2154:39–58, August 2001.

[78] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Why Timed Sequence Diagrams Require Three-Event Semantics. Research Report 309, ISBN 82-7368-261-7, University of Oslo, 2004.

[79] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS Towards Formal Design with Sequence Diagrams. *Software and Systems Modeling*, pages 355–367, 2005.

[80] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. *Software Product Line Conference (SPLC)*, pages 139–148, 2008.

[81] Øystein Haugen and Birger Møller-Pedersen. Configurations by UML. *3rd European Workshop on Software Architecture (EWSA)*, 2006.

[82] Øystein Haugen and Ketil Stølen. STAIRS - Steps To Analyze Interactions with Refinement Semantics. *UML 2003 - The Unified Modeling Language*, 2863:388–402, Springer Verlag 2003.

[83] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A Graph-based Approach to Modeling and Detecting Composition Conflicts Related to Introductions. *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development (AOSD)*, 208:85–95, 2007.

[84] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. *Proceedings of the First International Conference on Graph Transformation*, pages 161–176, 2002.

[85] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-oriented Systems. *ACM SIGPLAN Notices*, 25(10):169–180, 1990.

[86] Brian Henderson-Sellers. UML - the good, the bad or the ugly? perspectives from a panel of experts. *Software and System Modeling*, 4(1):4–13, 2005.

[87] José Luis Herrero, Fernando Sánchez, Fabiola Lucio, and Miguel Toro. Introducing Separation of Aspects at Design Time. *in Proc. of AOP Workshop at ECOOP '00*, 2000.

[88] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[89] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[90] ITU-T. Formal description techniques (FDT) Specification and Description Language (SDL), Z-Series Recommendations. Standard Z.100, International Telecommunication Union (ITU-T), 2003.

[91] ITU-T. Formal description techniques (FDT) Message Sequence Chart (MSC), Z-Series Recommendations. Standard Z.120, International Telecommunication Union (ITU-T), 2004.

[92] ITU-T and ISO/IEC. Reference Model for Open Distributed Processing (RM-ODP), Overview. Standard ITU-T Rec. X.901 — ISO/IEC 10746-1, International Telecommunication Union (ITU-T), International Standards Organisation (ISO), 1998.

[93] Ivar Jacobson. *Object-oriented Software Engineering.* Addison-Wesley, ISBN 0201544350, New York, NY, USA, 1992.

[94] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, ISBN 0321268881, 2004.

[95] Jean-Marc Jézéquel and Bertand Meyer. Design by Contract: the Lessons of Ariane. *IEEE Computer*, 30(1):129–130, Jan 1997.

[96] Karl Trygve Kalleberg and Eelco Visser. Combining aspect-oriented and strategic programming. *Electronic Notes in Theoretical Computer Science*, 147(1):5 – 30, 2006. Proceedings of the 6th International Workshop on Rule-Based Programming (RULE 2005).

[97] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Petersen. Feature-Oriented Domain Analysis (FODA) Feasibility Study, CMU/SEI-90-TR-21. Technical report, Software Engineering Institute (SEI), 1990.

[98] Emilia Katz and Shmuel Katz. Verifying Scenario-Based Aspect Specifications. *International Symposium of Formal Methods Europe (FM)*, pages 432–447, 2005.

[99] Shmuel Katz. A Survey of Verification and Static Analysis for Aspects. Technical report, AOSD Europe Network of Excellence, 2005.

[100] Shmuel Katz and Marcelo Sihman. Aspect Validation Using Model Checking. *Verification: Theory and Practice*, 2772:373–394, 2003.

[101] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. *European Conference on Object-Oriented Programming (ECOOP)*, pages 501–525, 2006.

[102] Raffi T. Khatchadourian and Awais Rashid. Rejuvenate Pointcut: A Tool for Pointcut Expression Recovery in Evolving Aspect-Oriented Software. *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 261–262, Sept. 2008.

[103] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–353, January 2001.

[104] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[105] Gregor Kiczales and Mira Mezini. Aspect-oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 49–58, 15-21 May 2005.

[106] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented Multi-view Modeling. *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98, 2009.

[107] Jörg Kienzle, Ekwa Duala-Ekoko, and Samuel Gélineau. AspectOptima: A Case Study on Aspect Dependencies and Interactions. *Transactions on Aspect-Oriented Software Development V*, pages 187–234, 2009.

[108] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect Composition Applying the Design by Contract Principle. *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering (GCSE)*, pages 57–69, 2001.

[109] Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Multiple Aspects in Sequence Diagrams. *Transactions on Aspect Oriented Software Development (TAOSD)*, pages 167–199, 2007.

[110] Jacques Klein, Luïc Helouet, and Jean-Marc Jézéquel. Semantic-based Weaving of Scenarios. In *The 5th International Conference on Aspect-oriented Software Development (AOSD)*, pages 27–38, New York, NY, USA, 2006. ACM Press.

[111] Günter Kniesel. Detection and Resolution of Weaving Interactions. *Transactions on Aspect-Oriented Software Development V*, pages 135–186, 2009.

[112] Christian Koppen and Maximilian Storzer. PCDiff: Attacking the Fragile Pointcut Problem. *European Interactive Workshop on Aspects in Software (EIWAS), Berlin, Germany*, 2004.

[113] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying Aspect Advice Modularly. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, pages 137–146, 2004.

[114] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of Package Templates for Flexible re-use of Collections of Related Classes. *Journal of Object Technology*, 2009.

[115] Stein Krogdahl and Frederik Sørensen. Generic Packages with Expandable Classes Compared with Similar Approaches. *Norsk informatikkonferanse (NIK) (Norwegian Conference of Informatics)*, 2007.

[116] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[117] Ingolf H. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technischen Universität München, 2000.

[118] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Interaction-based Runtime Verification for Systems of Systems Integration. *Journal of Logic Computation*, 2008.

[119] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL Frameworks. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 602–616, 2006.

[120] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-oriented Programming: An Objective Sense of Style. *International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 323–334, September 1988.

[121] Katharina Mehner, Mattia Monga, and Gabriele Taentzer. Analysis of Aspect-Oriented Model Weaving. *Transactions on Aspect-Oriented Software Development V*, pages 235–263, 2009.

[122] A. Metzger, S. Bühne, K. Lauenroth, and K. Pohl. Considering Feature Interactions in Product Lines: Towards the Automatic Derivation of Dependencies between Product Variants. *8th International Conference on Feature Interactions in Telecommunications and Software Systems*, pages 198–216, 2005.

[123] Bertand Meyer. Applying 'design by contract'. *IEEE Computer*, 25:40–51, 1992.

[124] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.

[125] J. Strother Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[126] Ana Moreira, Awais Rashid, and João Araújo. Multi-Dimensional Separation of Concerns in Requirements Engineering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE)*, pages 285–296, Washington, DC, USA, 2005. IEEE Computer Society.

[127] B. Morin, F. Fleurey, N. Bencomo, J-M. Jézéquel, A. Solberg, V. Dehlen, and G.S. Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. *11th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 5301:782–796, 2008.

[128] Object Management Group (OMG). MDA Guide V1.0.1. OMG document, omg/2003-06-01, OMG, 2003.

[129] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification. Standard, formal/06-01-01, Object Management Group (OMG), 2006.

[130] Object Management Group (OMG). Object Constraint Language, Version 2.0. Standard, formal/06-05-01, OMG, 2006.

[131] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation, Version 1.0. Standard, ptc/05-11-01, Object Management Group, 2008.

[132] Object Management Group (OMG). MOF Models to Text Transformation Language, Version 1.0. Standard, formal/2008-01-16, Object Management Group, 2008.

[133] Object Management Group (OMG). Common Variability Language. Draft RFP ad/2009-11-02, OMG, 2009.

[134] Object Management Group (OMG). The Unified Modeling Language: Superstructure, Version 2.2. Standard, formal/2009-02-02, OMG, 2009.

[135] Object Management Group (OMG). UML Profile for Modeling and Analysis of Real-time and Embedded systems (MARTE). Standard, ptc/08-06-09, OMG, 2009.

[136] Society of Automotive Engineers (SAE). Architecture Analysis and Design Language (AADL). Standard AS5506, SAE, 2009.

[137] J. Oldevik, T. Neple, R. Grønmo, J. Aagedal, and A. Berre. Toward Standardised Model to Text Transformations. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA)*, pages 239–253, Nuremberg, 2005. Springer.

[138] Jon Oldevik. Can Aspects Model Product Lines? *Proceedings of the 2008 AOSD Workshop on Early aspects*, pages 1–8, 2008.

[139] Harold Ossher. Confirmed Joinpoints. *AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2006.

[140] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. *European Conference on Object Oriented Programming (ECOOP)*, 3586/2005:214–240, 2005.

[141] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 1972.

[142] Klaus Pohl, Gunter Bockle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, ISBN 3540243720, 2005.

[143] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object Oriented Programming (ECOOP)*, pages 419–443, 1997.

[144] A. Rashid, P. Sawyer, A. Moreira, and J. Araújo. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. *IEEE Joint International Conference on Requirements Engineering*, pages 199–202, 2002.

[145] Awais Rashid, Ana Moreira, and João Araújo. Modularisation and Composition of Aspectual Requirements. *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD)*, pages 11–20, 2003.

[146] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects: The OORAM Software Engineering Method*. Englewood Cliffs: Prentice Hall, ISBN 0-13-452930-8, 1995.

[147] Sajjad H. Rizvi. Avicenna (Ibn Sina). *Internet Encyclopedia of Philosophy*, 2006.

[148] J. Rumbaugh, M. Blaha, W. Lorenson, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design*. Prentice-Hall, 1990.

[149] R.K. Runde. *STAIRS - Understanding and Developing Specifications Expressed as UML Interaction Diagrams*. PhD thesis, Department of Informatics, University of Oslo, 2007.

[150] Kouhei Sakurai and Hidehiko Masuhara. Test-based Pointcuts for Robust and Fine-grained Join Point Specification. *Proceedings of the 7th international conference on Aspect-oriented software development (AOSD)*, pages 96–107, 2008.

[151] N. Schärli, S. Ducasse, O. Nierstrasz, and A.P. Black. Traits: Composable units of behaviour. *Lecture Notes in Computer Science*, pages 248 – 274, 2003.

[152] Bran Selic. Using UML for Modeling Complex Real-Time Systems. *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, 1998.

[153] S. Sen, N. Moha, B. Baudry, and J-M.Jézéquel. Meta-model pruning. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2009.

[154] Marcelo Sihman and Shmuel Katz. Superimpositions and Aspect-Oriented Programming. *The Computer Journal*, 46(5), 2003.

[155] A. Solberg, D. Simmonds, R. Reddy, R. France, S. Ghosh, and J. Aagedal. Developing Distributed Services Using an Aspect Oriented Model Driven Framework. *International Journal of Cooperative Information Systems*, 15:535–564, 2006.

[156] Arnor Solberg. *An Aspect-Oriented Model-Driven Approach for QoS-Aware Sofwtare Engineering*. PhD thesis, University of Oslo, 2007.

[157] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A UML-based Aspect-Oriented Design Notation for AspectJ. In *Proceedings of the 1st international conference on Aspect-oriented software development (AOSD'02)*, pages 106–112, New York, NY, USA, 2002. ACM Press.

[158] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query Models. In *7th International Conference of Modelling Languages and Applications*, volume 3273/2004, pages 98–112, Lisbon, Portugal, 2004. Springer.

[159] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Expressing Different Conceptual Models of Join Point Selections in Aspect-oriented Design. In *5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 15–26, New York, NY, USA, 2006. ACM Press.

[160] Maximilian Stoerzer and Juergen Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-oriented Software. *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 653–656, Sept. 2005.

[161] Ketil Stølen and Ida Solheim. Technology Research Explained. SINTEF Report A313, ISBN 82-14-04039-6, SINTEF, 2007.

[162] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 299–300, 1999.

[163] A. Svendsen, G. K. Olsen, J. Endresen, T. Moen, E. Carlson, K. J. Alme, and Ø. Haugen. The Future of Train Signaling. In *11th international conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 128–142, Berlin, Heidelberg, 2008. Springer-Verlag.

[164] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *International Conference on Software Engineering*, pages 107–119, 1999.

[165] Paul Taylor. *Practical Foundations of Mathematics*. Number ISBN 0-521-63107-6 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1999.

[166] A. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104, 2007.

[167] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. *4th Intl. Workshop of Software Composition*, 2005.

[168] Markus. Voelter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 233–242, 10-14 Sept. 2007.

[169] John G. Wacker. A Definition of Theory: Research Guidelines for Different Theory-Building Research Methods in Operations Management. *Journal of Operations Management*, 16, 1998.

[170] Y. Wakahara, M. Fujioka, H. Kukuta, H. Yagi, and S.-I. Sakai. A Method for Detecting Service Interactions. *Communications Magazine, IEEE*, 31(8):32–37, Aug 1993.

[171] Robert J. Walker and Kevin Viggers. Implementing Protocols via Declarative Event Patterns. In *12th ACM International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, pages 159–169, New York, NY, USA, 2004. ACM.

[172] J. White, D.C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortès. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. *Software Product Line Conference, International*, pages 225–234, 2008.

[173] J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira, and J. Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *Transactions on Aspect-Oriented Software Development VI*, 5560:191–237, 2009.

[174] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi. An Expressive Aspect Composition Language for UML State Diagrams. *MODELS. International Conference on Model Driven Engineering Languages and Systems, Nashville, TN*, 2007.

[175] Jon Whittle and João Araújo. Scenario Modelling with Aspects. *Software, IEE Proceedings*, 151(4):157–171, Aug. 2004.

[176] Jon Whittle and Praveen K. Jayaraman. MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. *11th International Workshop on Aspect-Oriented Modeling (AOM)*, 2007.

[177] Rebecca Wirfs-Brock and Alan McKean. *Object Design – Roles, Responsibilities and Collaborations*. Addison-Wesley, ISBN 0-201-37943-0, 2003.

[178] Rebecca Wirfs-Brock and Brian Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 71–75, 1989.

[179] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, ISBN 0-13-629825-7, New Jersey, 1990.

[180] John A. Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[181] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental Models for Validating Technology. *IEEE Computer*, 31(5):23–31, 1998.

Part II

Research Papers

Appendix A

Paper I: Architectural Aspects in UML

Authors. Jon Oldevik and Øystein Haugen.

Paper Summary. In this paper, we present an approach for modelling and composing architectural aspects based on UML structured classes, which provides a flexible way of modularising architectural concerns. The approach is shown especially useful for providing refinements or variants of connectors in the architecture.

Author Contribution. Jon Oldevik was the main author and responsible for the main part of the research and writing of this paper, accounting to about 90% of the work. Specifically, Jon Oldevik was the main contributor in development of the concepts, notations, and techniques for architectural aspects and the evaluation of these.

Publication Arena. Published in the proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS) 2007. Acceptance rate 28% (45/158).

This article is removed.

Appendix B

Paper II: Higher-Order Transformations for Product Lines

Authors. Jon Oldevik and Øystein Haugen.

Paper Summary. In this paper, we present higher-order transformations in the form of transformational aspects, which provide product line variability by modifying the transformations that generate products. We distinguish platform from domain variability and show that if these concerns are well separated in base transformations, the feature transformations are confluent, i.e. they can be applied order-independent.

Author Contribution. Jon Oldevik was the main author and responsible for the main part of the research and writing of this paper, accounting to about 90% of the work. Specifically, Jon Oldevik was the main contributor in the techniques for higher-order transformation, their implementation, and their evaluation.

Publication Arena. Published in the proceedings of the 11th Software Product Line Conference (SPLC) 2007. Acceptance rate 35% (28/80).

Higher-Order Transformations for Product Lines

Jon Oldevik
University of Oslo
Department of Informatics
Oslo, Norway
jonold at ifi.uio.no

Øystein Haugen
University of Oslo
Department of Informatics
Oslo, Norway
oysteinh at ifi.uio.no

Abstract

An aspect-based extension to a text transformation language provides higher-order transformations that can be used to represent variability in generative product line engineering. We show by example how these higher-order transformations compare with first order transformations. We also detail how the approach has been implemented as an extension of MOFScript, an existing model-to-text transformation language.

1. Introduction

Researchers, tool vendors, and end users pay more attention to model-driven development (MDD) than before as the field is maturing. The standardisation and commercialisation of technologies within the MDD domain, especially UML 2, have a great impact in this regard. More recently, standardisation of technologies for automating MDD has appeared within Object Management Group (OMG), such as the MOF Query/View/Transformation language[15], which is a language for defining model-to-model transformations, and the MOF Model to Text Transformation[16], which is a language for transforming models to text.

For product lines (PL), generative programming[8] is an established approach for generating product implementations based on higher-level specifications. There is a close relationship and clear overlap in the philosophies of generative programming and MDD. The most significant difference is the explicit focus on product lines found in generative programming[7], which is not there in MDD. MDD focuses on using models as assets driving the development, and may use generative approaches to refine models or generate code. It does not, however, explicitly address variability management and product line feature representation.

The paper investigates the applicability of higher-order transformation for product line variability, using the MOFScript language[14]. We devise an aspect-oriented extension to MOFScript that we call *Hi-Transform*. The implementation of the aspect weaving is done by a transformation in MOFScript itself. An example product line model is used to analyse the appropriateness of model-to-text transformations with aspects for product line variability. We address two kinds of product line variability: Variability related to the product line platform (cross-platform variability) and variability related to the product line domain model (intra-domain variability). Cross-platform variability is variability pertaining to the commonalities and variabilities between e.g. programming language versions, component infrastructures, etc. Intra-domain variability is variability pertaining to the commonalities and variabilities of the PL domain concepts (i.e. the features of the domain).

Our investigation seeks to answer the following:

- *Can Hi-Transform provide fruitful representation and realisation of variability for platform variability and intra-domain variability?*

Our conjecture is that higher-order transformations such as devised in Hi-Transform can be more independently described than corresponding first-order transformations. In the following, we go in depth on the higher-order text transformation approach for product lines. In Section 2, we describe the basics of model-to-text transformation. In Section 3, we study through an example how variants can be supported with aspects in model-to-text transformations. In Section 4, we give the details of Hi-Transform, the aspect-based extension to MOFScript. In Section 5, we describe related work and in Section 6 we conclude the paper.

2. Model-to-Text Transformation Basics

A model-based text transformation language has a set of basic properties[16]. As illustrated by Figure 2, a model-to-text transformation takes one or more models as input and generates output text. The input models are instances of

0-7695-2888-0/07 $25.00 © 2007 IEEE
DOI 10.1109/SPLINE.2007.11

IEEE computer society

metamodels that define the concepts used in the models. A transformation contains rules that define the logic producing text output by combining literal expressions and input model properties.

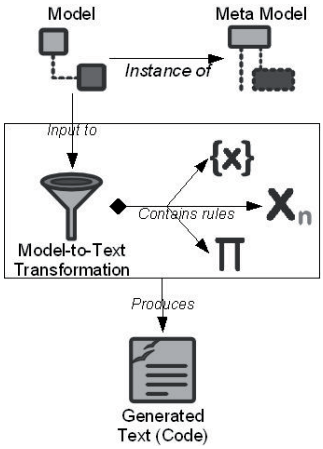


Figure 1: Model-to-Text Transformation Basics

In MOFScript, model-to-text transformation rules are similar to Java methods. However, their primary objective is to operate on model types from the input model(s) of the transformation. A transformation rule in MOFScript is defined with a model type as its context, e.g. *Class* from UML2's metamodel. Logically, this associates the rule with the *Class* type and allows it to operate on Class properties. A transformation rule invokes other transformation rules imperatively to perform the desired behaviour and output. A simple example that produces Java code from UML models is given in Program 1. It creates an output file for each class and generates private Java variables for each UML property.

Program 1: Example Transformation

```
texttransformation SimpleJava (in uml:"UML2") {
   public uml.Class::genClass() {
    file (self.name + ".java")
     'public class ' self.name ' { '
      self.ownedAttribute->forEach(a)
        a.genAttribute()
     '}'
   }

   public uml.Property::genAttribute() {
     'private ' self.type.name ' ' self.name ';'
   }
}
```

In order to provide support for string manipulation, MOFScript implements a range of string operations, such as *toUpper(), toLower, trim(), replace*, etc. To facilitate modularisation and reuse of transformations, MOFScript has additional mechanisms: *Import* of transformations facilitates reuse of transformation libraries. *Inheritance* of transformations provides the means of specialising and thereby reusing and overriding defined transformation rules.

Alternative Technologies. The main benefit of a specific model-based text generation language is the provision of tailored mechanisms for the specific purpose of text generation, i.e. it is domain-specific and it has explicit support for using models and model properties.

A programming language (e.g. Java) can provide the same functionality. A Java representation of the transformation in Program 1 will become less compact, but the difference is not dramatic (maybe 35 per cent). If a Java-based approach provides a tailored framework with abstractions, e.g. through generic classes, it would further simplify the Java approach. This is similar to the idea presented by Akehurst[1], where a simple Java-based framework for model transformation is presented. This approach suggests that using standard programming languages in many cases is just as suitable as tailored transformation languages, due to the well-known language mechanisms and good tool support.

In addition to Java, or Java frameworks, a wide range of alternative technologies are available for performing text generation. One example is Java Emitter Templates (JET), which provides an XML-based scripting language tightly integrated with Java, Eclipse end EMF. Another example, closer to the approach in MOFScript, is the *openArchitectureWare* tool, which provides a template-based language based on EMF metamodels. Interestingly, it implements aspect functionality in the language[18]. Their argument in favour of aspects is to support generation of code for several platform variants within the same transformation. This suggests that aspects certainly can be useful in model-to-text transformations for product line engineering.

Introducing Hi-Transform. An aspect in the aspect-oriented programming sense is a specification of a transformation that will alter the structure and behaviour of a specification when applied (weaved). In the context of text transformation, an aspect is a specification of a transformation that will alter the structure and behaviour of a text transformation; it thus represents a higher-order transformation where the source and target are both text transformations.

To provide such higher-order transformations in MOFScript, we have extended MOFScript with aspect-like functionality and called it *Hi-Transform*. It defines the following concepts:

- *Hi-Transform* is a representation of text transforma-

tional aspects. It is itself a text transformation representing a specific concern, and it contains *Hi-Queries*, *Hi-Rules*, as well as standard text transformation rules.

- *Hi-Query* is a description of places in a transformation that may be modified by the hi-transform. This concept is analogous to the aspect-oriented programming concept pointcut.
- *Hi-Rule* is a representation of text transformation code which is used to modify a text transformation identified by Hi-Queries. This concept is analogous to the aspect-oriented programming concept advice. It specifies text transformation code to be inserted *before*, *after*, or to *replace* existing code.

A detailed description of Hi-Transform is given in Section 4.

Higher-Order Transformations. The implementation of Hi-Transform is a higher-order transformation that applies to two text transformations, the base transformation and the hi-transform, and produces a modified text transformation.

The input for the transformations in Figure 2 is a *product line model*. The *base transformation* may produce base product code, which may be abstract (i.e. not executable), or it may not be produced at all. When combined with *hi-transforms*, a *higher-order transformation* produces a *modified transformation*, which in turn produces a variant of the product code.

Section 3 investigates how hi-transforms can be used to provide variability in product line development.

3. Providing Variability

Aspects in text transformations are instruments for providing variability with respect to generating code for different platforms, different QoS requirements, language variations, or target frameworks. All these can be viewed as platform variations for the product line.

Another, not so obvious application of higher-order text transformations is to provide variability with respect to the product line domain model, thus being able to provide intra-domain as well as platform variability for the product line. We examine how higher-order transformations can express both these kinds of variability.

Expressing choices in generative languages will often result in lots of variant checking and conditional code generation scattered throughout the transformation, which makes it less readable. A more elegant alternative is to write the general transformation, and specialise it for different variants. This, however, will result in an explosion of specialisation and overriding if the variants are many, and different

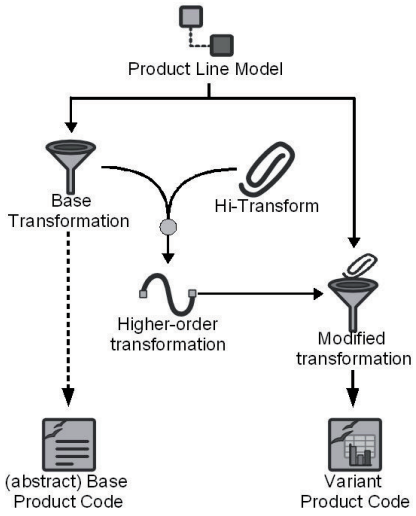


Figure 2: Higher-order transformation.

variant code occurs in many places. Our Hi-Transform approach should remedy these undesired effects.

The next section describes an example and shows how transformational aspects can be applied to support product variability.

3.1. Variability Example

In this example, we illustrate the usage of Hi-Transform to generate variants of the same system. The example used for illustration is a simple *BookStore* system described in UML(Figure 3). The BookStore system defines a *BookStore class*, which contains a set of *books*, a set of *authors*, and a set of *categories*. Furthermore, a book has a *bookCover*. The Category class represents an optional feature, and the BookCover class represents an alternative feature with two alternatives, its two subclasses.

We will use the example in Figure 3 to illustrate two kinds of product line variability that can be provided by text transformation aspects: product line platform and product line intra-domain variability.

3.2. Platform Variability

The variability we want to illustrate is in how relations are mapped to implementation in the product line platform. In the base text transformation, which generates Java classes from a UML model, associations are mapped to

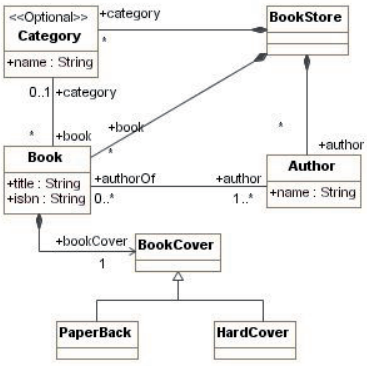


Figure 3: Bookstore Product Line UML model

HashMap variables in the class and a set of operations (get / getAll / remove / add / create) for each association.

To illustrate, the transformation code for association variable declaration and a retrieval operation is shown in Program 2.

Program 2: Base Transformation Rules for Associations

```
uml.Property::classPrivateAssociations () {
 '    private HashMap ' + self.propertyPrivateName() ';
 '
}

uml.Property::associationGetAllMethod () {
 '
 public Collection get' self.propertyPluralName() '(){
    return ' self.propertyPrivateName() '.values();
 }'
}
```

When the transformation is executed, these rules are invoked once for each UML association property, which results in the generation of a private variable of type HashMap and a getter operation that returns a general Collection.

We want our system to be flexible with respect to how associations are implemented to cope with different Java platforms, and possibly the performance and flexibility of association operations. A set of transformational aspects are written to provide different variants of these implementation properties.

Figure 4 illustrates the principle: The base *UML2Java* transformation produces a base Java implementation with HashMap (variant 1). Three different hi-transforms provide variant implementations: one using Java Generics for the collection classes, one using Java List instead of HashMap, and one using a custom implementation of a Collection class that is also provided by the hi-transform.

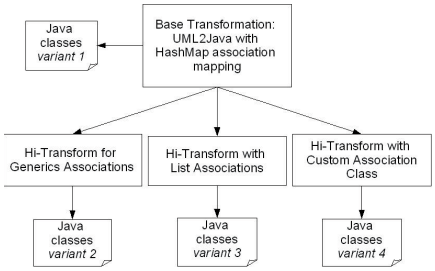


Figure 4: Variability Example with Hi-Transforms

The code in Program 3 shows (part of) the hi-transform. It specifies some *hi-queries* that target the relevant rules of the UML2Java transformation and *hi-rules* that contain the new code to be generated. In this example, the original *HashMap* representation is replaced with a typed HashMap using Java Generics syntax available in Java 5.

Program 3: Hi-Transform with Generic Associations

```
hi-transform GenericAssoc {
  hi-query privateAssociation(Property) def
    ("classPrivateAssociations")
  hi-query associationInit (Property) def
    ("associationInit")
  hi-query associationGetAllMethod (Property) def
    ("associationGetAllMethod")

  hi-rule replace privateAssociation {
    '
    private HashMap<String, ' self.type.name '>' +
       self.propertyPrivateName() ';
    '
  }
  hi-rule replace associationInit {
    result = self.propertyPrivateName() + ' = new
       HashMap<String, ' self.type.name + '> ();'
  }
  hi-rule replace associationGetAllMethod {
    '
    public Collection <'self.type.uml2JavaType()'> get'
        self.propertyPluralName()'(){
      return ' self.propertyPrivateName() '.values();
    }'
  }
}
```

The *replace* keyword signifies a Hi-Rule that replaces the contents at the locations given by the referenced Hi-Query. (For further details on Hi-Transform, please refer to Section 4.) When the hi-transform is applied on the base text transformation, a modified text transformation with different implementations of the rules matched by the hi-transform is obtained. Executing the modified transformation then yields a modified Java implementation. The code extracts in Programs 4 and 5 show parts of the code generated by the original UML2Java transformer and the one generated by the new one.

Program 4: Bookstore Base Product

```
class BookStore {
    private HashMap _book;
    private HashMap _author;
    public BookStore () {
      _book = new HashMap ();
      _author = new HashMap ();
    }
    public Collection getBooks ( ) {
       return _book.values();
    }
}
```

Program 5: Bookstore Modified Product

```
class BookStore {
    private HashMap<String, Book> _book;
    private HashMap<String, Author> _author;
    public BookStore () {
      _book = new HashMap<String, Book> ();
      _author = new HashMap<String, Author> ();
    }
    public Collection <Book> getBooks ( ) {
            return _book.values();
    }
}
```

The second hi-transform, which uses a *List* implementation of associations instead of HashMap, requires more modifications, since all association methods need to be modified to use the List type. This hi-transform defines hi-queries for all association rules and hi-rules with an appropriate List implementation mapping.

The final association implementation hi-transform provides yet another mapping, which replaces the original association representation with a custom collection class.

Platform Features. The variability illustrated in the example fits well with the notion of features in product lines. In this case, the features are alternative implementations of associations. A product line architecture description could specify these along other product line features in a feature diagram. This can then be used by the product designer to select the variants desired, which can be mapped to the appropriate hi-transform.

Figure 5 shows a FODA-style[10] feature diagram, which has the *Association Implementation Feature* as one of several features in the *Implementation Architecture*. It defines four alternative features corresponding to the standard UML2Java transformation plus the three hi-transforms.

3.3. Intra-Domain Variability

A product line may define variability at many different levels. In Section 3.2, we illustrated platform variability with respect to specific implementation variants. Another application of hi-transforms is for handling intra-domain model variability of the product line. This is somewhat more complex as it requires that domain knowledge is built into the transformations.

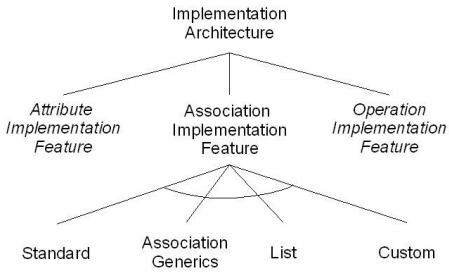


Figure 5: A Feature Diagram for the Platform Variants

To illustrate, we show a feature-style representation of the domain model from Figure 3, where the *Category* class is optional and the *BookCover* represents alternative variants: *HardCover* and *PaperBack* (Figure 6).

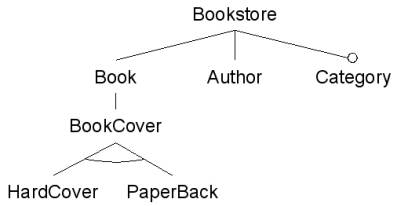


Figure 6: Bookstore Features

When generating platform variability in Hi-Transform, general (i.e. domain-independent) Java transformations and hi-transforms can be used, since the transformations can be oblivious to domain-specific concepts. The platform variability previously illustrated can therefore apply to a wide range of domain models. This is not true for variability of the product line domain model itself. To support product generation for the domain model that can handle the variability, a domain-specific (in this case, a Bookstore-specific) transformation is required. We want a transformation that can generate all variants, where the variants are provided by hi-transforms.

In order to handle the commonalities, we define a single *BookStore* transformation. It consists of some domain-specific parts and some general, reusable parts. Program 6 shows some domain-specific parts of the transformation, covering the BookStore class. The code for the other classes follows the same pattern. All the variable parts of the

Program 6: Domain-Specific Base Transformation

```
uml.Class::classGeneration(packageName:String){
  if (className.equals("BookStore"))
     self.generateBookStore (packageName)
  ...
}
uml.Class::generateBookStore(packageName:String) {
  self.genClassStart(packageName)
  self.generateBookstoreAttributes ()
  self.generateBookstoreAssociations()
  self.genClassEnd()
}

uml.Class::generateBookstoreAssociations() {
  var books:uml.Property = self.ownedAttribute->select(
    p:uml.Property|p.name="book").first()
  books.genAssociation()
  var authors:uml.Property = self.ownedAttribute->select(
    p:uml.Property|p.name="author").first()
  authors.genAssociation()
}
```

Program 7: Hi-Transform for Category Feature

```
hi-transform CategoryHit(in uml:"UML2")
{
  hi-query classGeneration(Class) def
    ("classGeneration")
  hi-query generateBookstoreAssociations (Class) def
    ("generateBookstoreAssociations")

  hi-rule after classGeneration {
    if (className.equals("Category")) {
      self.generateCategory(packageName)
    }
  }

  hi-rule after generateBookstoreAssociations {
    var category:uml.Property = self.ownedAttribute->
      select(p:uml.Property | p.name="category").first()
    category.genAssociation()
  }
}
```

product line is kept out of this transformation and provided by separate hi-transforms. Some of the rules called within the domain-specific code uses general rules, such as *self.genClassStart()* and *books.genAssociation()*. The transformation is logically divided into two parts: one with general transformation rules, and one with domain-specific rules that use the general ones. The general rules could be defined separately and imported by the domain-specific transformation.

To provide the variability, we define hi-transforms for each variant in the product line model, which is the optional *Category* feature and the two alternatives for BookCover. If we focus on the *Category feature hi-transform*, it needs to do the following:

- Insert code that invokes the category class rule.
- Insert code that adds a category association to the BookStore.
- Insert code that adds a category association to the Book.
- Provide the rules that generate the Category class.

Program 7 shows an excerpt of the hi-transform for the Category feature doing exactly this. It defines two *Hi-Queries* that reference the *classGeneration* and the *generateBookstoreAssociations* rules. Two *Hi-Rules* provide the code to be inserted into the base transformation *after* the existing code.

We can now represent each variability of the product line in terms of hi-transforms. To produce a transformation for a specific product, the hi-transforms corresponding to the desired features should be applied to the base transformation using the higher-order transformation. Selecting and applying the hi-transforms become the variability resolution process for the product line. To resolve several variations in the product line domain model, several hi-transforms must be applied in sequence. The same might also be the case for the platform variability, if there are several categories of platform features having variants (e.g. attribute implementation in addition to association implementation). If there are further constraints on the features, such as conflict or requirement dependencies, these should be catered for in the selection and application of hi-transforms.

The base transformations and hi-transforms pertaining to the domain variability are naturally flavoured by the concepts of the product line domain. As illustrated by programs 6 and 7, the base transformation contains *Bookstore-specific* rules representing the commonalities, while the *Category hi-transform* explicitly references features related to *Category*. These parts of the base transformation may however be derived from the product line feature model.

3.4. Discussing the Variability Approach.

The previous examples show usage of hi-transforms providing both platform and domain model variability for product lines. By combining these, complete product code can be generated from a product line model. Figure 7 shows the pieces put together and illustrates the difference between higher-order transformations with hi-transforms (a) and first order transformations (b). With hi-transforms, a series of higher-order transformations can be applied to obtain new text transformations. All steps take text transformations (MOFScript) as input and produce MOFScript output, This finally results in a *Product Transformation*, which is used to generate the *Product*. With a first-order transformation, a model-to-model transformation can create a *Modified Product Line Model*, which in turn is input to a *Platform Transformation*. The figure shows the correspondence between the two approaches; a variable feature will result in either a hi-transform or alternatively a first-order transformation. The notable difference here is that the *type* of transforma-

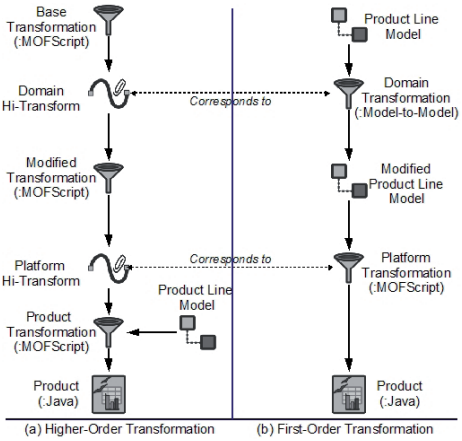


Figure 7: Hi-Transforms vs. First Order Transformations

tion changes for the first order case, not for the higher order case.

Higher-order model-to-text transformations in Hi-Transform seems flexible with respect to representing and implementing product line variability. Two kinds of variability have been shown, platform variability and intra-domain variability. If transformations are designed by carefully separating platform-specific from domain-specific logic, these can be combined using hi-transforms without considering ordering. I.e, a platform variability hi-transform can be applied before or after an intra-domain variability hi-transform; the resulting transformation will be the same, so the variability transformations are confluent.

An alternative approach is to manage variability with model-to-model transformations and code generation. Assuming that the product line is represented by a model, model-to-model transformations could handle the intra-domain variability to create product-specific models. An approach using higher-order transformations to provide variability could also be applied in this scenario. The platform variants could then be supported either by transformations to platform-specific models followed by code generation, or by a Hi-Transform-like approach.

Using an aspect-oriented programming language such as AspectJ[12] proves to be difficult. The intra-domain variability can be provided by inter-type member declarations, which insert variant elements into existing classes. This would, however, require introduction of new classes representing variable features, which is not possible in AspectJ; the classes would have to be pre-defined. Platform variability is impossible, since it requires modifications to declared variables and method signatures, which is illegal in AspectJ.

Assuming a language without such limitations, it could be used to provide the intra-domain and platform variability. It would however be more restrictive in how transformations are applied. The intra-domain and platform variability transformations are no longer confluent. Platform variability aspects must always succeed intra-domain variability aspects, as these also contain platform details.

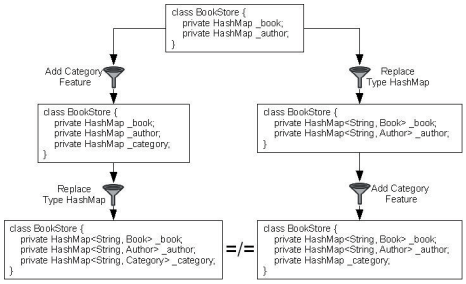


Figure 8: Non-Confluent Transformations

This is illustrated by Figure 8, where the two branches of transformation ends with different result. On the left hand side, the domain variability is handled first by adding the category feature. This is followed by applying the platform transformation (replacing the HashMap type), yielding a correct result. On the right hand side, the order is reversed, and we see that the two results are different.

Our conjecture that higher-order transformations can be more independently described than corresponding first order transformations seems plausible. In practice this means that hi-transforms are more often confluent than a set of first order transformations (e.g. a set of MOFScript transformations). However, there must be constraints on the base transformation for this to be true, e.g. that domain-specific rules should be clearly separated from general rules.

4 Detailing Hi-Transform

As described in Section 2, Hi-Transform is an aspect-based extension of the model-to-text transformation language MOFScript. It contains concepts similar to those in aspect-oriented programming[12], adapted to the text transformation domain.

Hi-Transform is defined in terms of three assets:

- *A MetaModel:* The Hi-Transform metamodel defines each hi-transform concept. It is used for populating hi-transform instances that are used at runtime.
- *A Language Grammar:* The grammar defines the concrete syntax of Hi-Transform and is used in the parser

implementation.

- *A Higher-Order Transformation:* The execution of a hi-transform is handled by a MOFScript transformation that takes a hi-transform and a base transformation as input, and produces a modification of the base transformation.

Hi-Transform MetaModel. The metamodel defines Hi-Transform as a special kind of MOFScript transformation (Figure 9).

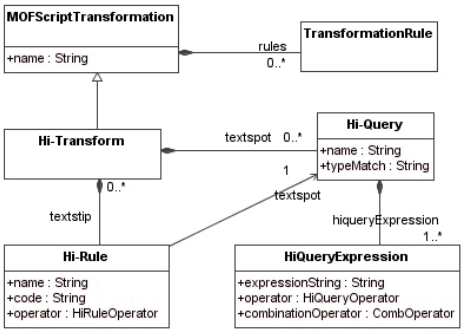


Figure 9: Hi-Transform Metamodel

A hi-query is a analogous to the concept pointcut in AspectJ addressing either the invocation or definition of transformation rules. The following hi-query types are defined in this model by the HiQueryOperator enumeration.

- *Definition hi-query:* This type identifies a set of transformation rule definitions.
- *Call hi-query:* This type identifies a set of calls to transformation rules.

In addition, a hi-query may define a *typeMatch* (similar to a *target pointcut* in AspectJ, which further constrains the hi-query to transformation rules with a specific context type. The notation for the hi-query expression is based on Java regular expressions within quotation marks (".*" matches any sequence of characters). The example `hi-query propChange(Class) def("property.*")` identifies all transformation rules with a context type *Class*, which starts with the name *property*.

A hi-rule is analogous to advice types in AspectJ and specifies text transformation code to be inserted at positions identified by a hi-query.

- *Before hi-rule:* Targets the transformation part immediately before the referenced hi-query. It represents an insertion of the hi-rule code before all places identified by the hi-query.
- *After hi-rule:* Targets the transformation part immediately after the referenced hi-query. It represents an insertion of the hi-rule code after all places identified by the hi-query.
- *Replace hi-rule:* Targets the transformation part identified by the reference hi-query and replaces this transformation code by the body of the Hi-Rule.

The example `hi-rule before propChange {log ("Property rule executed.")}` applies to the *propChange* hi-query and inserts a logging statement.

Hi-Transform Concrete Language. The hi-transform language is defined as an extension to the MOFScript language. Program 8 describes the elements of the language extensions as an EBNF grammar. Several of the productions are reused from mofscript and are not detailed here.

Program 8: Hi-Transform grammar

```
hitransform      =  "hi-transform" simpleName "("
                     modelParameters ")" "{"
                     ( hiquery | hirule | trRule )*
                     "}" ;
hiquery          =  "hi-query" simpleName ["("
                     <typeName> ")"] ("def" | "call")
                     "(" <matchCriterion> ")" ;
hirule           =  "hi-rule" ("before" | "after" |
                    "replace") hiqueryRef ruleBody;
typeName         =  simpleName ;
matchCriterion   =  "\"" regular expression "\""
hiqueryRef       =  simpleName
modelParameters  =  mofscript model parameters
simpleName       =  mofscript simple name
trRule           =  mofscript transformation rule
ruleBody         =  mofscript transformation rule body
```

Implementation of Hi-Transform. The transformation represented by a hi-transform is implemented by a higher-order transformation in MOFScript. The higher-order transformation modifies the base transformation by adding and removing features as instructed by the hi-transform. The result is a modified text transformation with the changes from the hi-transform. The hi-transform parser checks the overall structure of the hi-transform and that hi-rules contain valid transformation code before execution.

We implemented Hi-Transform as a language extension to MOFScript, where the hi-transforms are specialised text transformations, which in addition to hi-queries and hi-rules may define ordinary transformation rules. The hi-transform metamodel could have been defined as a separate, independent metamodel, which would make it more reusable for other tools. The higher-order transformation could also

have been realised by a model to model transformation tool with access to the hi-transform metamodel, such as ATL[6].

In this implementation, all hi-transform code is applied at design time by generating modified transformations from a hi-transform and a base transformation. The benefit of the static weaving is that complete transformations for specific products can be inspected or stored for reuse, without having to perform the weaving transformation each time.

Hi-transforms might also be applied dynamically, i.e. during the execution of the transformation. Dynamic weaving would save the user from viewing the composed resulting transformation, which is probably a good thing. The user is more likely to be interested in the finally generated code rather than the composed transformation. Dynamic weaving support for hi-transforms would require an extension to the transformation runtime, to match hi-queries dynamically and execute the relevant hi-rules. The result should be the same, except that runtime support would allow for more advanced hi-queries taking advantage of dynamic information such as the control flow. (E.g. inserting code in a rule only if that rule is called from a specific context). At this time we have not seen the need for this in Hi-Transform.

5 Related Work

To our knowledge, there is little work done on using higher-order text transformations to support product line variability. A similar approach is provided in the xPand language[18], which provides support for aspect-oriented concepts and argues its usefulness for supporting platform variability in product lines. There is no analysis or examples relating this to domain model variability.

This work is strongly related to ideas in generative programming, coined by Czarnecki and Eisendecker in [8]. It is specifically contrasting in the usage of higher-order transformations as a variability mechanism.

Aspect-oriented programming[11] can solve the same issues as targeted by this paper using for instance AspectJ[12]. This would however require a generative approach based on Java rather than a dedicated transformation language, i.e. a text transformation written in Java (as discussed in Section 3.4.

The work by Batory et al on step-wise refinement[4] provides an approach for composing refinements based on hierachical equations and shows how it can be applied for code and non-code artifacts, given that composition operators are defined for the artifact type. Layers/features can be specified as separate units and composed to obtain specific configurations. We could integrate Hi-Transform with this approach using the higher-order transformation as our composition operator and provide the benefits of the formalisms and tool support for step-wise refinement.

A lot of work has been done in the area of supporting product lines with technologies for cross-cutting of concerns, such as aspects.

Griss[9] outlines an approach for feature-driven, aspect-oriented product line engineering. He advocates the usage of feature-driven analysis and design combined with aspect-oriented implementation techniques, where code fragments (aspects) represent features at the code level. Application design is done by selecting features and by selecting and weaving aspects.

Anastasopoulos and Gacek[2] evaluate implementation approaches with respect to providing product line variability. They address among other things AOP, frames, dynamic libraries, and parameterisation, and define a framework for comparing the approaches. They conclude that no approach can sufficiently capture product line variability requirements and that approaches need to be improved or used together. Higher-order transformations are not considered.

Loughran and Awais[13] describe *Framed Aspects*, which combine the cross cutting features of aspects with the configuration capacities of frames, which cater for better support for variability.

Anastasopoulos and Muthig[3] analyse the suitability of AOP for implementing product lines, concluding aspects suitable for cross-component variability. An observation made is the limited adequacy for supporting so-called negative variability (e.g. removing a method). Although probably an often undesired property, this is not a limitation with higher-order text transformations.

Saleh and Gomaa[17] describe an approach for separation of concerns for product lines. It uses a model-based representation of feature models that represents commonalities and variabilities, which is mapped to a domain-specific programming language tailored for handling variability.

Bayer et al[5] describe a consolidated metamodel for product lines, which provides a reference model for product line variability. It defines the concept Transformer, which is one way to specify variability in a separate variation model. A hi-transform can be interpreted as a special kind of transformer.

6 Conclusions

This paper has described a model-based approach for product line variability with higher-order text transformations in Hi-Transform. Hi-Transform implements aspect-based functionality as an extension to the text transformation language MOFScript. We have used a product line example model as basis for analysing the applicability of Hi-Transform for variability.

The ideas presented combine generative programming with aspects to provide variability in product line engineering. The variability is described in terms of a model of the

product line, capturing the common and variable features of the domain and platform. This variability can be realised by higher-order transformations.

In conclusion, we recap the question raised in the introduction (Section 1) based on the results shown in the paper:

- *Can Hi-Tranform provide fruitful representation and realisation of variability for platform variability and intra-domain variability?*

Our analysis shows that product line variability related to differences in implementation platform is well supported by hi-transforms. The platform variability can often be seen independently of the variability of the domain model. As such, it can be treated at a more general level with transformations that are independent of specific product line domain concepts. Our analysis also shows that hi-transforms indeed can provide variability also for a product line domain model. This requires more complexity on the part of the base transformation and hi-transforms, as they must reflect domain concepts from the product line domain model.

The usage of Hi-Transform for product line variability seems to be a powerful approach, especially when considering the combination of platform and domain variability support, and how these can be independently applied.

Future work is required in order to analyse how well this approach scales for larger product lines, especially with respect to supporting many variants, and feature variants containing or depending on other feature variants. Work is also needed to investigate how feature resolution can be better integrated with the approach, e.g. with tool support. Further experience and empirical evidence are needed to conclude additionally on the characteristics of higher-order transformations for product lines.

Acknowledgement. This work has been carried out in the context of the SWAT project (Semantics-preserving Weaving - Advancing the Technology), funded by the Norwegian Research Council (project number 167172/V30).

References

[1] D. Akehurst, B. Bordbar, M. Evans, W. Gareth, J. Howells, and K. McDonald-Maier. SiTra: Simple Transformations in Java. *Model Driven Engineering Languages and Systems (MODELS)*, 2006.

[2] M. Anastasopoules and C. Gacek. Implementing Product Line Variabilities. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[3] M. Anastasopoulos and D. Muthig. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In *Software Reuse: Methods, Techniques and Tools*, pages 141–156. Springer, 2004.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Stepwise Refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[5] J. Bayer, S. Gerard, Ø. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J. Thibault, and T. Widen. Consolidated Product Line Variability Modeling. In *Software Product Lines, Research Issues in Engineering and Management*. Springer, 2006.

[6] J. Bezivin, G. Dupe, F. Jouault, G. Pitette, and J. Rougui. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture,*, 2003.

[7] K. Czarnecki. Overview of Generative Software Development. *Unconventional Programming Paradigms*, 3566/2005, 2005.

[8] K. Czarnecki and U. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[9] M. L. Griss. Implementing Product-Line Features by Composing Aspects. In *Proceedings of the first conference on Software product lines : experience and research directions*, pages 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Petersen. Feature-Oriented Domain Analysis (FODA) Feasibility Study, CMU/SEI-90-TR-21. Technical report, Software Engineering Institute (SEI), 1990.

[11] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–, Jan. 2001.

[13] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *Software Reuse: Methods, Techniques and Tools*, pages 127–140. Springer, 2004.

[14] J. Oldevik, T.Neple, R. Grønmo, J. Aagedal, and A.Berre. Toward Standardised Model to Text Transformations. *European Conference on Model Driven Architecture - Foundations and Applications, Nuremberg*, 2005.

[15] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation. Technical Report OMG document id ptc/05-11-01, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf, 2005.

[16] OMG. MOF Models to Text Transformation Language Final Adopted Specification, OMG document number ptc/06-11-01. Technical report, Object Management Group, 2006.

[17] M. Saleh and H. Gomaa. Separation of Concerns in Software Product Line Engineering. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[18] M. Volter. Best Practices for Model-to-Text Transformations. In *Eclipse Summit Europe, Modeling Symposium*, 2006.

Appendix C

Paper III: Semantics Preservation of Sequence Diagram Aspects

Authors Jon Oldevik and Øystein Haugen.

Paper Summary. In this paper, we give a definition of semantics preservation with respect to sequence diagram aspects. The definition is based on sequence diagram refinement and event preservation. With basis in that definition, we evaluate existing techniques for sequence diagram aspects with respect to the semantics preserving property. This notion of semantics preservation helps ensuring that aspects will have the expected effect when specifications evolve.

Author Contribution. Jon Oldevik was the main author and responsible for the main part of the research and writing of this paper, accounting to about 90% of the work. Specifically, Jon Oldevik was the main contributor of the formal definitions for semantics preservation, analysing its application in the context of syntactic and semantic aspect approaches, and for evaluating existing approaches.

Publication Arena. Published in the proceedings of the 4th European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA) 2008. Acceptance rate 33% (31/87).

This article is removed.

Appendix D

Paper IV: From Sequence Diagrams to Java-STAIRS Aspects

Authors. Jon Oldevik and Øystein Haugen.

Paper Summary. In this paper, we give a definition of trace-based semantics in Java, which is used to reason about semantics preservation in Java systems. We define a trace semantics for Java programs, which is based trace semantics used in Paper III. We show how sequence diagrams can be mapped to the defined Java trace semantics, and how it can be used to reason about semantics preservation when the Java system is refined.

Author Contribution. Jon Oldevik was the main author and responsible for the main part of the research and writing of this paper, accounting to about 90% of the work. Specifically, Jon Oldevik was the main contributor of the definitions of trace-based semantics and semantics preservation for Java and for applying and evaluating the approach.

Publication Arena. Published in the proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD) 2009. Acceptance rate 22% (19/86).

From Sequence Diagrams to Java-STAIRS Aspects

Jon Oldevik
University of Oslo/SINTEF ICT
Oslo, Norway
Univ. of California, San Diego, La Jolla, USA
jonold at ifi.uio.no

Øystein Haugen
SINTEF ICT / University. of Oslo
Oslo, Norway
oysteinh at sintef.no

ABSTRACT

Execution traces are naturally represented at the design level with UML sequence diagrams. During a system execution, trace-based aspects can be used to monitor behavioral patterns and protocols and may e.g. provide mitigation strategies for unwanted behavior. Trace-based and stateful aspects have evolved to handle such reoccurring interaction patterns at the implementation level. We define a STAIRS-inspired semantics for trace-based Java aspects, and a sequence diagram aspect notation with a mapping to a trace-based Java implementation. We use this to show that aspect composition is semantics preserving with respect to refinement under the given semantics.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.10 [**Software Engineering**]: Design—*Representation*

General Terms

Design, Languages

Keywords

Sequence Diagram Aspects, Trace-Based Aspects, Refinement

1. INTRODUCTION

In the aspect-oriented community, concerns have traditionally centered around capturing single system events (e.g. method calls), such as provided by the joinpoint model in AspectJ [14]. More recent works, however, focus on the need for more advanced models of aspects, to be able to capture traces or histories of events in a program execution [29, 30, 9, 24, 16, 1]. These kinds of aspects are commonly called *stateful* or *trace-based* aspects, and have been shown useful for detecting behavioral patterns and protocols and then possibly changing the course of a program. Aspects describing traces of system executions have a natural representation in graphical specification languages such as UML sequence diagrams (SDs) or message sequence charts (MSCs), which describe interactions between objects in the system. For many years, techniques for describing object interactions have been used in domains where systems inherently are reactive and asynchronous, such as in the telecommunication domain. In service-oriented systems engineering, interactions provide an intuitive way of specifying services [4]. A mapping of these abstractions to the programming and program execution level can help minimizing the gap between design and implementation. Several research efforts have addressed modularization of design-level cross cutting concerns for sequence diagrams or message sequence charts. [15, 31, 10]. These approaches focus only on the modeling level. In [18], programming level aspects are generated from MSCs to provide runtime monitors for MSC behaviors. Join Point Designation Diagrams (JPDDs) [25] were developed as a means of raising the abstraction when describing aspect pointcuts, and mappings from JPDDs to AspectJ pointcuts [11] show that the sequence diagram abstraction has definite advantages. We are missing in existing sequence diagram aspect approaches, however, provision of trace-based pointcuts *and* advice in a standard sequence diagram notation with a clear conceptual and semantic mapping to Java. To that end, we propose Java-STAIRS Aspects, a sequence diagram notation for aspects with an associated STAIRS-inspired [13], trace-based semantics for Java that allows sequence diagram semantics and Java semantics to be correlated. We use the semantics to show that aspect composition is semantics preserving with respect to Java system refinement. This is tied to earlier work on STAIRS and sequence diagram aspects [21], and allows us to check if a Java program *p2* is a refinement of another program *p1*, and by this gaining knowledge that a Java-STAIRS aspect that matches *p1* will match similarly for *p2*. We map sequence diagram aspects to an existing trace-based aspect implementation, *Tracematches* [1], and use a figure drawing application to test and illustrate how the approach works.

Outline. In Section 2, we elaborate the background and motivation for this work. In Section 3, we define a trace-based semantics for Java that is based on STAIRS semantics for sequence diagrams. In Section 4, we propose a notation for sequence diagram aspects and define how these are mapped to Java and Tracematches. In Section 5, we discuss our results and compare with other work, and in Section 6, we summarize.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.
Copyright 2009 ACM 978-1-60558-442-3/09/03 ...$5.00.

2. MOTIVATION AND BACKGROUND

2.1 Trace-Based Aspects

The motivations for trace-based aspects are manifold and are adequately described in other works, where some motivators are: to improve the robustness (vulnerability to changes) of aspects [29, 24]; to provide better abstractions for specifying and implementing protocols [30, 28]; and being able to capture histories or traces in the execution [9, 1, 2]. At the modeling level, trace based aspects have found their natural place in the design space in sequence diagram or MSC aspects.

Several approaches for aspects at the sequence diagram level have appeared for handling cross cutting concerns at design-level [15, 31, 10, 6, 3]. Join Point Designation Diagrams (JPDDs) [25] bridge the gap between design abstractions and implementation-level, trace-based aspects. However, as we will see later, although JPDDs provide powerful mechanisms for describing implementation pointcuts, its matching semantics does not provide what we consider a natural conceptual mapping of a matching pattern. For example, it is not possible to specify a pointcut that matches the sending/receiving of two consecutive messages between two roles only if there are no other messages in between. Also it is not possible to ignore messages being sent to other roles that are not part of the *concern* captured by the sequence diagram. Furthermore, JPDD was designed to express pointcuts only and does not provide a way of expressing advice. Finally, it is based on a non-standard notation, and it lacks notational support for useful things such as combined fragments to express conditionals, loops etc.

We believe that trace-based aspects have a more intuitive representation in a sequence diagram than in an aspect-oriented implementation. Therefore, we propose to express trace-based aspect using UML sequence diagrams, which can be mapped to a Java implementation. However, there is no standard interpretation of a sequence diagram in the context of a Java system. An execution trace in Java will often be much more detailed, involving more events and more objects than those represented by a sequence diagram. In such cases, we would like to reason about the Java traces in the context of the sequence diagram by only seeing events between the specific objects or roles specified by the sequence diagram.

When using sequence diagrams as matching patterns for Java systems, there are several possible ways this can be mapped to matching semantics in Java. Here we motivate our way of doing it. Let us assume we have a set of interacting components (a,b,c,d,e), each of which participates in different collaborations to fulfill different tasks. Lets say we want to capture one particular interaction pattern P involving a,b, and c, which we describe as a sequence diagram aspect. This can be considered one specific concern of these components. The question is *what* the criteria are for P to be detected. In this scenario, we assume that we do not know if or how a,b, or c interacts with d or e. These may again be interacting with other, perhaps external components. The actual component interaction may be richer and span other concerns than the P that we are looking for. A component should therefore be allowed to participate in interactions with parties outside the concern P without interfering with the detection of P. For interactions between the components in P, however, this is not always the intended semantics. If the pattern we are interested in is the exact trace(s) specified by the sequence diagram, we cannot allow arbitrary other message passings to occur between our components. Instead, the matching semantics should be strict with respect to the interactions between the components in P, while the pattern language should allow arbitrary messages to be specified, hence supporting both types of semantics. Since different contexts might require different semantics, however, there should be ways to configure matching semantics to different needs.

To support this, we propose a semantics for representing traces in Java, which is based on sequence diagram semantics, and we give an interpretation of sequence diagram aspects in terms of this semantics. Flexible matching semantics is supported by an event filtering mechanism. We define a mapping of sequence diagram aspects to a trace-based aspect implementation solution and support this with model-driven techniques that generate the trace-based implementation from sequence diagram models.

Aspect vulnerability with respect to changes in the base program, known as the fragile pointcut problem [17], is also a motivator for this work. We propose a notion of refinement in our semantics, based on refinement in STAIRS, which ensures that potential execution traces are not removed. This will in turn make a system and its trace-based aspects less vulnerable to changes.

2.2 Sequence Diagram Overview

A UML [23] Sequence Diagram (SD) is an *Interaction* that consists of a set of lifelines representing interacting roles or objects in a system. At the metamodel level, a lifeline represents a part within a classifier. The SD defines a set of ordered interaction fragments that involve the different lifelines. The basic kind of interaction fragments are occurrence specifications that represent events on lifelines. Messages between lifelines represent a communication between two lifelines involving one *send* and one *receive* event. Combined fragments allow different kinds of composition of interaction fragments, e.g. sequential (seq), alternative (alt), parallel (par), and loop composition. The events in a sequence diagram are constrained by the order of events on each lifeline (weak sequencing) and the causality of send and receive events.

2.3 STAIRS and refinement

STAIRS [13] is a denotational semantics for UML sequence diagrams based on event traces. A trace is a sequence of events, each defined by its kind (send or receive), a message defining the signal, the sending lifeline, and the receiving lifeline. In STAIRS, The semantics of a sequence diagram d, $[\![\mathrm{d}]\!]$, is defined by two sets of traces: the set of *positive* and the set of *negative* traces. Positive traces represent allowed behavior. Negative traces represent behavior that is *not* allowed. In addition, any trace that is not covered by the sequence diagram is defined as *inconclusive*. The set of traces in $[\![\mathrm{d}]\!]$ is determined by all possible executions of d. The resulting event sequences are constrained by the causality and weak sequencing properties of sequence diagrams: a send event must occur before its corresponding receive event (causality) and the events on a lifeline have the same (relative) ordering in a trace as on the lifeline (weak sequencing). STAIRS defines three kinds of refinement: *supplementing*, which adds positive or negative behavior by making incon-

clusive traces either positive or negative, *narrowing*, which changes positive behavior to negative, and *detailing*, which details existing behavior in decompositions. In this context, we will only address supplementing refinement, and only positive traces of Java programs. For more details on STAIRS semantics, the interested reader may refer to [13]. STAIRS provides the formal foundation for Java-STAIRS semantics and Java-STAIRS Aspects.

2.4 Semantics Preservation of Sequence Diagram Aspects

In [21], Oldevik et al. give a definition of semantics preservation for sequence diagram aspects. It is based on monotonicity of aspect composition with respect to (STAIRS) refinement and the preservation of events. The monotonicity of aspect composition is essentially preservation of the refinement relationship before and after aspects are applied. We will show that the same ideas of semantics preservation can be applied to Java based on our definitions.

3. JAVA-STAIRS TRACE SEMANTICS AND REFINEMENT

In this Section, we define a trace-based semantics for Java that we call Java-STAIRS, which is inspired by the STAIRS semantics for sequence diagrams [13]. The purpose of Java-STAIRS is to define a semantics for Java executions that can be used to represent traces described by sequence diagrams. The Java-STAIRS semantics is defined based on the execution traces of methods of individual objects, only considering messages between different objects, and not assignments or other expressions. We restrict our focus in this paper to synchronous, possibly multi-threaded Java systems. This is a simplification of the asynchronous semantics of sequence diagrams. However, we will show that our Java semantics can be mapped to a general STAIRS sequence diagram semantics.

3.1 Trace-Based Semantics in Java

We define a Java system $\mathcal{S}$ by a configuration of objects typed by classes and a set of execution threads operating on those objects: $\mathcal{S} = \{\mathrm{O}_S, \mathrm{Th}_S\}$, where $\mathrm{O}_S = \{o_1 : C_1, o_2 : C_2, o_3 : C_3,...,o_n : C_n\}$, where each $C_i \in S$ is a class and each o_i is a unique instance of C_i. Each class C is defined by a set of methods $M_C = \{m_{c1}, m_{c2}, ..., m_{ck}\}$. The threads of $\mathcal{S}$ are defined by a non-empty set, $\mathrm{Th}_S = \{Th_1, ... Th_k\}$, where each Th_i contains a subset of O_S. We do not consider the variables defined by a class, even though their values may influence the execution trace.

Although we may infer some information about system interactions by static analysis, we cannot generally determine the behavior of a particular system configuration before instantiating and running it, due to inheritance and polymorphism. We can statically see which *send events* (method invocations) that are potential from analyzing a method specification, but we cannot generally know statically the implementation of the methods that are called. Hence, we must define the Java-STAIRS trace semantics relative to the *objects* making up the system. We define the semantics of a method m in the context of object *o:C* by the set of traces of message sending that represent possible executions of m.

One trace of m, $\mathcal{T}_m$, represents one possible execution of m. $\mathcal{T}_m$ is defined by the *transitive closure* of each *method invocation* occurring between the *start* and *end* of the execution of m (i.e. all method calls that are put on the stack between the start and end of the method execution). Since we only consider synchronous communication in Java, we do not have to separate the sending and receiving of a message as different events.

We choose to model a trace as a sequence of events, where each event contains information about the *sender object*, *receiver object*, the *message called*, and the *execution thread*. A trace of message m called on object o can be defined by $\mathcal{T}(m,o) = \{<e_0(o_{s1},o_{r1},m_1,th_1),\ e_1(o_{s2},o_{r2},m_2,th_2),....,e_i(o_{si},\ o_{ri},m_i,\ th_i)>\}$, where each e_i is an event, each o_{si} is a sender object, each o_{ri} is a receiver object, each m_i is the message sent, and each th_i is the execution thread. For a single-threaded Java application, each th_i will be identical.

The *semantics* of m in the context of the object o, $[\![\mathrm{m(o)}]\!]$, is defined by the set of *potential traces* of m in the context of object o. This contrasts with the semantics of sequence diagrams in STAIRS, which is exactly the sets of positive and negative traces defined by the sequence diagram. In our definition of the semantics of m, we want to ignore any message being sent to *self*, as these can mostly be considered irrelevant in the context of interactions (since we want to focus on the *inter*action between objects). This is done by applying a default trace filter ($\mathcal{F}_{exclude-this}$ - see Section 3.3 below), which filters away any such events from the trace.

For a specific object $o \in S$, with methods M_o={m_0, m_1, m_2,...m_k}, the semantics of o, $[\![\mathrm{o}]\!] = [\![m_0]\!] \cup [\![m_1]\!] \cup [\![m_2]\!] \cup$ ... $\cup\ [\![m_k]\!]$, i.e. all potential traces for all methods on that object. Similarly, the semantics of system S is defined as the union of the semantics of all its objects: $[\![\mathrm{S}]\!] = [\![o_1]\!] \cup [\![o_2]\!]$ $\cup$ ... $\cup\ [\![o_n]\!]$, i.e. all potential traces of all objects in the system.

Program 1 Example One

```
class DrawController {
  protected Figure f = new Figure();
  protected Display d = new Display (f);

  public void addPoint (int x, int y) {
    Point p1 = f.makePoint (x, y);
    d.display(p1);
  }

  public void addPoint2(int x, int y) {
    addPoint(x, y);
    Point p3 = f.makePoint (x+100, y+100);
    d.display(p3);
  }

  public void addPoint3 (int x, int y, boolean addOne) {
    addPoint(x, y);
    if (addOne) {
      Point p3 = f.makePoint(x*2, y*2);
      d.display(p3);
    }
  }
}
```

Looking at Program 1, we can consider the semantics of an object O of type *DrawController*. It is defined by the union of the semantics of each of its methods. This gives the following semantics: $[\![O]\!] = [\![addPoint]\!] \cup [\![addPoint2]\!]$ $\cup$ $[\![addPoint3]\!]$, which in turn equals {<*makePoint, display*>} $\cup$ {<*makePoint, display, makePoint, display*>} $\cup$ {<*makePoint,*

display>, <*makePoint, display, makePoint, display*>}. Set duplicates can be removed, resulting in the two traces {<*makePoint, display*>, <*makePoint, display, makePoint, display*>}. For brevity, an event is just denoted by the message name without the sending/receiving objects or thread. The calls to *addPoint* from *addPoint2* and *addPoint3* are not events in these traces, since they are filtered by $\mathcal{F}_{exclude-this}$.

If we were to extend Java-STAIRS to model asynchronous communication, the events can be extended with send and receive flags and an id to match corresponding send and receive events. Time is not considered in this paper, but it could also be an extension to the event by using time ticks or time stamps.

3.2 Java-STAIRS Refinement

Refinement of a sequence diagram in STAIRS is defined by new sequence diagrams that supplement, narrow, or detail another sequence diagram. For Java-STAIRS, we restrict this to supplementing refinement. Refinements are established when methods are redefined. A supplementing refinement of a sequence diagram adds new traces to the sequence diagram e.g. by adding optional fragments, adding operands to an alternative, or creating an alternative to an existing (set of) fragment(s). Basically, any modification that produces a superset of the traces of the original sequence diagram is a supplementing refinement.

For Java-STAIRS, a refinement is defined based on the potential traces that can be produced by methods executions. For methods M and M': a method M' is a refinement of M ($M \rightsquigarrow M'$) iff $[\![M]\!] \supseteq [\![M]\!]$, that is, the potential traces of M are also potential traces of M', i.e. $\mathcal{T}(M') \supseteq \mathcal{T}(M)$. This is consistent with the definition of supplementing refinement in STAIRS [13]; this is also the basis for the semantics preservation definition for sequence diagram aspects in [21].

Two traces $\tau(M)$ and $\tau(M')$ are considered *equal* if their sizes are the same (i.e. they contain the same number of events) and the message in each $\tau(M')$ event refers to the same method, or a redefined method, as the corresponding event in $\tau(M)$. By allowing methods to be redefined, two traces can also be equal if objects of corresponding events are in a sub/supertype relationship.

Refinement of objects is based on refinement of methods. We say that an object o' is a refinement of o ($o \rightsquigarrow o'$) iff $[\![o']\!] \supseteq [\![o]\!]$. For systems S and S', we say that S' is a refinement of S ($S \rightsquigarrow S'$) iff $[\![S']\!] \supseteq [\![S]\!]$.

For the refinement relationship, we can further define the following properties:

1. The refinement identity - a system s is always a refinement of itself: $\mathcal{S} \rightsquigarrow \mathcal{S}$.
2. Refinement is transitive - $\mathcal{S} \rightsquigarrow \mathcal{S}'$ and $\mathcal{S}' \rightsquigarrow \mathcal{S}'' \Rightarrow \mathcal{S} \rightsquigarrow \mathcal{S}''$.

The identity property is trivial. The transitivity property has been proven for sequence diagrams in [12]. Here, refinement is defined by the superset operator on traces, which gives the transitivity property. Our definition of refinement only supports supplementing. This, however, is not a limitation on the effect any aspect may have on the base program, only on the changes that can be made to the base program that will constitute a refinement. As we will see later (in Section 4.3), refinements will allow us to know that an aspect will have effect when the base program evolves.

In STAIRS, refinement semantics is defined for various sequence diagram composition operators, such as sequential, alternative, and parallel composition. This allows for compositional reasoning when refining a system specification. Similar composition rules could be defined for Java by using conditional statements and sequential method calls. We will, however, not focus on composition operators in this paper. The Java-STAIRS semantics is a simplification of STAIRS in that it does not describe asynchronous communication. It also only covers the actual potential positive behavior, while STAIRS categorize behavior (traces) as being either positive, negative, or inconclusive. We can, however, map traces in Java-STAIRS to STAIRS by, for each trace, create a positive STAIRS trace that contains one send and one receive event for each Java-STAIRS event. The resulting STAIRS traces describe asynchronous events, but in a synchronous order. With such a mapping to STAIRS we could rely on STAIRS analysis techniques for Java-STAIRS traces.

3.3 Trace Filters for Interaction Contexts.

In order to focus on the traces essential for a set of interacting objects, we want to be able to abstract away events that are not considered part of that particular object interaction. For example, if we want to analyze the interaction between objects o_1, o_2, and o_3, we want to hide traces involving other objects. To this end, we introduce filter constraints, which allow to specify filters for the kinds of objects who's interactions are (or are not) interesting in this context. These filters are constraints on trace sets that restrict the sets of traces visible, and create projected views on a complete set of execution traces. This gives focus to the essential communication between the objects involved in a specific interaction. We define a set of default filters that helps projecting Java-STAIRS traces to match a particular sequence diagram context.

The default exclude filter for excluding message events from an object to itself, $\mathcal{F}_{exclude-this}$, is defined by filtering (removing) from $\mathcal{T}$(m,o) all events e(o_s, o_r, m, th) where $o_s = o_r$. Formally, this can be defined by the following constraint on the system $\mathcal{S}$:

- $\mathcal{F}_{exclude-this}(\mathcal{S}) = \forall$ t $\in [\![\mathcal{S}]\!]$, $\forall$ e $\in$ t $\mid e.o_s \neq e.o_r$.

Two other filters are considered important for capturing sequence diagram semantics: restricting the type of objects involved in interaction and restricting the instances involved. Related to a specific interaction (defined by a sequence diagram), we want to restrict the traces on the Java level to only involve objects of the types from that particular interaction. We call that the $\mathcal{F}_{restrict-type}$ function. Furthermore, we want to restrict our traces to the particular objects interacting. We don't want other objects of the same type to interfere with the traces. We call that the *one-object-per-type (oopt)* filter. The former function is defined by:

- $\mathcal{F}_{restrict-type}(\mathcal{S}$, Set<Class> typeSet$) = \forall$ t $\in [\![\mathcal{S}]\!]$, $\forall$ e $\in$ t $\mid e.o_s.type \in$ typeSet $\wedge\ e.o_r.type \in$ typeSet.

The latter function is defined by:

- $\mathcal{F}_{oopt}(\mathcal{S}) = \forall$ t *in* $[\![\mathcal{S}]\!]$, $\forall$ e$_1$, e$_2$ $\in$ t $\mid e_1.o_s.type = e_2.o_r.type \Rightarrow o_s = o_r$.

The need for *oopt* filter can be seen e.g. if capturing traces in a graphical editor with several (concurrent) active editor windows, and we want to avoid matching traces involving

more than one of the editors. If the system is multi-threaded and these objects live in different threads, a *per-thread* filter may be used instead. Other filters and filter functions can be defined as required by the application context.

The filters provide a means for semantic variability in the trace matching. Other filters can be defined to filter out events outside the scope of interest for an interaction. The semantics of Java-STAIRS Aspects is controlled by the sequence diagram(s) in question. If a sequence diagram specifies multiple roles of the same type, the *oopt* constraint must be relaxed to allow this.

3.4 Examples of Refinement.

We will now address how this applies to the example Java code in Program 2, which shows some classes extending the *DrawController* class from Program 1. The original class defines the *addPoint* method, which is redefined in these subclasses. We will look at the refinement relationships between the different *addPoint* in the original class and the ones introduced in Program 2.

Program 2 Refinement Examples

```
class DrawCtrl2 extends DrawController {
  public void addPoint (int x, int y) {
    super.addPoint(x,y);
  }
}

class DrawCtrl3 extends DrawController {
  public void addPoint (int x, int y) {
    super.addPoint(x, y);
    Point p3 = f.makePoint(x,200);
    d.display(p3);
  }
}
class DrawCtrl4 extends DrawController {
  public void addPoint (int x, int y) {
    Point p1 = f.makePoint (x, y);
    d.display(p1);
    logger.log(Level.INFO, "Point created");
  }
}
```

By our definition of refinement, addPoint2 in DrawController (Program 1) *(mp2)* is *not* a refinement of addPoint *(mp1)*. It will never produce the same trace, since it introduces a new call to *makePoint* that will always be invoked. The possible traces of mp2 are, as we saw earlier (Section 3.1) {<*makePoint, display, makePoint, display*>}, which is not a superset of the *mp1* traces - {<*makePoint, display*>}. In DrawController::addPoint3 (mp3), a call to *mp1* is supplemented with a conditional that may or may not add additional events to the trace. As earlier seen, the potential traces of *mp3* ($\mathcal{T}$(mp3)) are a superset of *mp1* traces ($\mathcal{T}$(mp1)). We can therefore say that *mp3* is a refinement of *mp1*.

The subclass *DrawCtrl2* redefines the *addPoints* method, but it only invokes the *addPoint* method of the superclass. The traceset for the two will be the same; for *DrawCtrl3*, the situation is similar to that of *addPoint2*. It introduces new method invocations and will never be able to produce the same trace as *addPoint* in the superclass. Thus, it is not a refinement. Finally, lets look at class *DrawCtrl4*, which also redefines *addPoint*. It makes the same two calls to makePoint and display, but in addition it has a call to a logger. The trace of that method (in any object context) will be {<*makePoint, display, log*>}, which would not be a refinement of the superclass addPoint. However, if we apply a trace filter that focuses on the collaboration between DrawController and Figure, it would be a refinement, as equal traces constitute valid refinements.

3.5 Summary of Java-STAIRS

We have defined a semantics for execution traces in Java, which is based on STAIRS semantics for UML sequence diagrams. This provides a foundation for reasoning about the execution traces in a collaboration of interacting Java objects. Furthermore, we have defined a mechanism for specifying the relevant interaction patterns by using trace filters. This makes it possible to define the scope of the interaction within which we want to reason about trace semantics.

We then defined refinement for Java-STAIRS based on supplementing refinement in STAIRS. This provides a way of reasoning about refinement relations with respect to system execution traces. At a design level with sequence diagrams, this is useful for being able to determine system consistency during refinement. At the Java level, we will see that this is very useful in order to provide a consistent application of trace-based aspects.

4. Java-STAIRS Aspects

Java-STAIRS Aspects is an application of trace-based aspects for Java programs using the trace semantics defined in Section 3. We extend the standard aspect-oriented paradigm to support aspects defined by traces such as those expressible by sequence diagrams. In contrast to other trace-based approaches, we define a trace-based semantics that can be filtered according to specific sequence diagram contexts, and describe aspects using standard UML sequence diagrams that are later mapped to trace-based aspects in Java.

There is an obvious, fundamental difference between sequence diagram aspects and trace-based Java aspects. A sequence diagram describes a static model of a system execution scenario. As such, any part of that scenario can be modified by an aspect. For example, a sequence diagram aspect can look for an interaction pattern and then insert new interaction fragments before or between fragments of any match found in the base sequence diagram. Since the traces of a Java program execution can only be determined during the actual execution, it is only possible to advice new behavior *after* the occurrence of a trace.

We use UML sequence diagrams as our instrument for describing pointcuts, similar to the work of Stein et al. [26] with Join Point Designation Diagrams (JPDD). We evaluated JPDD semantics and tool support for implementing our approach, and found two main reasons for not choosing JPDD: most importantly, the semantics defined and implemented by JPDD does not provide any means for filtering the communication to support desired communication semantics for the sequence diagram objects. Secondly, the JPDD approach uses a non-standard UML notation that is hard to support in a standard UML tool, and their notation and tool do not support UML control structures like conditionals, loops, or alternatives.

The *Tracematches* approach, proposed by by Allan et al. [1], defines a language and a semantics for trace-based matching that *seems to* meet our specification needs on the implementation level. Avgustinov et al. [2] have imple-

mented the tracematch semantics within the *AspectBench* platform. The Tracematches approach allows filtering of events with event symbols that are defined by ordinary AspectJ pointcuts. A match for a tracematch is defined by a regular expression combining these symbols. We will see later how *Tracematches* can capture interactions specified by sequence diagrams, and we will analyze its strengths and weaknesses in this respect.

4.1 Pointcut/Advice Description in Sequence Diagrams

There are several approaches that define notations for specifying sequence diagram or MSC aspects and define the semantics for composing or weaving these aspects at a modeling level [15, 10, 31]. Join Point Designation Diagrams (JPDDs) [25, 26] defines a UML-based notation for expression advanced pointcuts, without addressing weaving at the model level. A mapping of JPDDs to code-level pointcuts is described in [11]. That work has been supported by a prototype JPDD modeling tool and a pointcut generator to AspectJ [14] and JAScO [27].

For our purposes, we employ standard UML sequence diagrams to represent trace-based aspects. If the pointcut semantics needs to be extended, we rely on the existing work in this area. We illustrate with examples from a *figure drawing application*, which we used for experiment with our approach. Figure 1 shows an aspect with a pointcut and advice, which applies to the example application. The pointcut is defined by fragments (messages and combined fragments) as a standard sequence diagram. The advice is defined *after* the fragments defining the pointcut by applying the <<create>> stereotype (from Whittle et al. [31]) on a *seq* combined fragment. Any interaction fragment within this is treated as part of the advice. To accommodate specification of explicit advice code, the model advice can embed Java code in addition to the advice interaction fragments.

The notation used for return types (<?p1>) is borrowed from the JPDD notations for identifiers. The return value of a message is stored in the identifier and can be referenced later in the sequence diagram, typically in the advice part. Message names and parameters may be specified using wildcard conventions as in AspectJ and JPDD. Parameters for messages may also be defined by the associated method definition in the model, if any.

This simple aspect looks for a sequence of calls where a *DrawController* invokes *makePoint* on a *Figure* object followed by a call to *display* on a *Display* object. The two calls should occur twice. The trace we need to find is thus {<*makePoint, display, makePoint, display*>}. If this occurs, the advice will create a line that links the two points (*makeLine*) and display it on a display object.

In addition to basic sequences of messages, we allow a sequence diagram aspect to specify combined fragments that represent loops (repetitions), alternatives, optionals, or parallel composition. Other kinds of combined fragments from UML are not supported. To specify quantified message fragments or lifelines, the notation from JPDDs, which adheres to the conventions of AspectJ, can be used for message names and type names of lifelines.

Figure 2 illustrates a Java-STAIRS aspect that looks for an interaction pattern with a loop. It also illustrates that the advice introduces a new lifeline that invokes behavior on the involved objects. A sequence diagram with only sequential

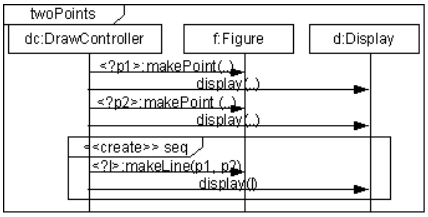


Figure 1: Detecting Creation of Two Points

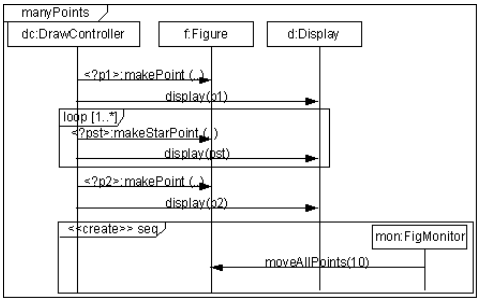


Figure 2: More Complex Example

messages will define a *single* trace (since we consider only synchronous semantics). Introducing alternatives, optionals, loops, or parallel merge in the sequence diagram will result in several traces being defined as part of the Java-STAIRS aspect pointcut. An alternative defines one alternative sub trace for each operand. An optional will define two alternative sub traces (where one is empty). A loop defines a range of possible traces defined by the loop bounds. A parallel merge defines a set of traces based on a interleaving the events of the operands without obstructing the event order within each operand. Events from different parallel merge operands are associated with different thread identifiers.

Figure 3 shows examples of sequence diagrams with different kinds of combined fragments. The names of messages may use wildcard notation similar to that of AspectJ such as '..' or 'make*'. The *altex* defines the traces {<*makePoint, display, make**>, <*makePoint, display, repaint*>}. The *optex* defines the traces {<*makePoint, display*>, <*makePoint, display, make**>}. The *loopex* defines an unbounded set of traces: {<*makePoint, display, make**>, <*makePoint, display, make*, make** >, ...}. The *parex* defines the traces resulting from interleaving the events of the operands: {<*makePoint*, $display_{t1}$, $update_{t1}$, $make*_{t2}$, $repaint_{t2}$>, <*makePoint*, $display_{t1}$, $make*_{t2}$, $update_{t1}$, $repaint_{t2}$> , <*makePoint*, $display_{t1}$, $make*_{t2}$, $repaint_{t2}$, $update_{t1}$>, <*makePoint*, $make*_{t2}$, $display_{t1}$, $repaint_{t2}$, $update_{t1}$>, <*makePoint*, $make*_{t2}$, $display_{t1}$, $update_{t1}$, $repaint_{t2}$>, <*makePoint*, $make*_{t2}$, $repaint_{t2}$, $display_{t1}$, $update_{t1}$>}. All *display* and *update* events in this trace have the same thread id (t1); so will all *make** and *repaint* events (t2). Since a trace defined by a Java-STAIRS aspect may describe quantified events, it may match many *different* traces in a base program, similar to AspectJ pointcuts that quantify message calls.

The knowledge of how to create or access the new lifeline

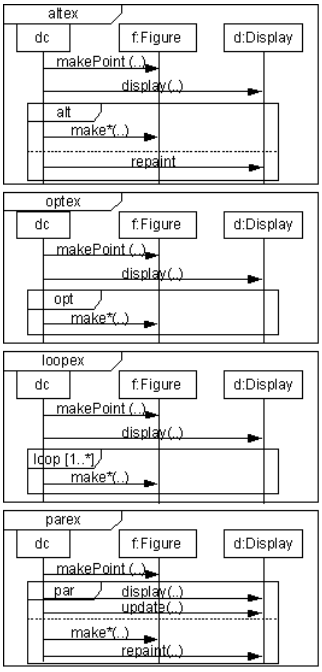


Figure 3: Combined Fragment Examples

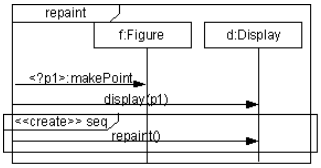


Figure 4: Using the Formal Gates

can be embedded in the advice code if needed. The advice of a Java-STAIRS aspect may also define behavior originating from the environment (the formal gates of the sequence diagram). The pointcut part may also define fragments involving the formal gates of the sequence diagram. These are considered as originating from any kind of object, i.e. the same as specifying a lifeline with wildcards (Figure 4).

The composition of a Java-STAIRS aspect with a Java system is done by mapping the sequence diagram aspect to an implementation level trace-based aspect. Formally, we define the composition of a system $\mathcal{S}$ with a Java-STAIRS aspect $\mathcal{A}$ - $\mathcal{S} \oplus \mathcal{A}$ - by the set of (potential) traces in $[\![\mathcal{S} \oplus \mathcal{A}]\!]$, with the exclusion and type restriction filters applied ($\mathcal{F}_{oopt}$ ($\mathcal{F}_{restrict-type}$ ($\mathcal{F}_{exclude-this}$ ($[\![\mathcal{S} \oplus \mathcal{A}]\!]$)))). The next section describes the mapping in detail.

4.2 Mapping to Trace-Based Java Aspects

To implement Java-STAIRS Aspects, we analyzed several existing technologies, such as JPDD, Tracematches, Declarative Event Patterns [30], and JaSCo [28]. As JPDDs are closely related, it was a natural starting point. The semantics of JPDDs is defined by the generators to different aspect language platforms, and we analyzed the JPDD generator to AspectJ. As earlier mentioned, we found one major obstacle with JPDD. The JPDD interpretation of the aspect (the pointcut part) of Figure 1 is to match a call to *makePoint* followed by another call to *makePoint*, even if there are other calls in between. In Java-STAIRS Aspects, we want to filter any events that are not between the types of the objects involved in the interactions ($\mathcal{F}_{restrict-type}$) and the events that are just calls to self ($\mathcal{F}_{exclude-this}$). This has proven difficult, since the JPDD semantics allows any message to come between messages that are part of the trace pattern we are looking for. We ran several rounds of test cases and discussed with the JPDD developers, but were unable to support the desired semantics. It would, however, be possible to redefine the JPDD semantics and re-implement the tool to support this semantics, which would be like implementing a new tool. Instead of pursuing this, we looked to other alternatives.

Tracematches was defined by Allan et al. in [1] to support trace-based matching of execution traces as an extension to AspectJ. It has the ability to filter events that are out of context, which seemingly is exactly what is needed as a platform for Java-STAIRS Aspects. Tracematches extends AspectJ with the ability to define symbols that refer to standard AspectJ pointcuts. The symbols declare the joinpoints of interest. All other joinpoints in an execution are ignored when looking for a match to a trace. A regular expression pattern of these symbols defines the traces of interest. By defining symbols that are not used in the matching pattern, events of interest that are not wanted in a match can be specified. Occurrences of these events will break an ongoing trace match.

Program 3 shows a tracematch for the example from Figure 1. As this is a simple example, it could easily been handled by standard AspectJ with some state information. However, as argued by [1, 25] and others, it is *not* straightforward to design these aspects when the interactions get more complicated. The tracematch defines one pointcut for each specific message (e.g. makePoint) from the sequence diagram. In addition, it defines one pointcut (*others*) for filtering other calls between the involved types. Corresponding symbols and the trace to match are defined within the *tracematch* definition. We use a type pattern *(+)* in the within clauses to also capture calls made from subclass objects.

The tracematch in Program 3 defines three *symbols*, each related to a specific pointcut. The *mp* symbol is associated with the *makePoint* pointcut, which is triggered by any call to the method *makePoint* on a Figure object from within *DrawController*. The symbol *dis* represents the display event occurring when a DrawController invokes *display* on a *Display* object, while the *oth* symbol captures any other events that are calls to any of *DrawController, Display, or Figure* from within any of the two others. The regular expression of symbols within the tracematch is defined by: *mp dis mp dis*, which maps to the trace defined by our aspect in Figure 1. The tracematch will ignore any event not captured by a symbol, and so it will only look for events that

Program 3 TraceMatch

```
public aspect PointMatcher {
  pointcut makePoint(Figure f): target(f) &&
    call (* Figure.makePoint(..)) &&
    within (DrawController+);
  pointcut display(Display d): target(d) &&
    call (* Display.display(..)) &&
    within (DrawController+);
  pointcut others(): (target(DrawController) &&
    (within(Figure+ || Display+))) ||
    (target(Figure) && (within(DrawController+ ||
    Display+))) || (target(Display) &&
    (within(Figure+ || DrawController+)));

  tracematch (Figure f, Display d) {
    sym mp  after: makePoint(f);
    sym dis after: display(d);
    sym oth after: others();
    mp dis mp dis {
      System.out.println ("Two points were created.
        Now add a line..");
    }
  }
}
```

match the *mp, dis*, and *oth* symbols.

Figure 5 illustrates what we would like to happen: when the user adds a point, which is followed by a another one, the tracematch should add a line between the two points.

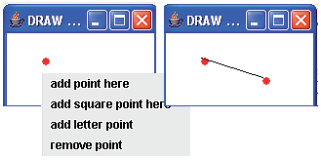


Figure 5: Drawing Two Points

Now, the actions defined by the *tracematch* does not yet implement the desired behavior of the advice. The availability of information for individual pointcuts is limited within a tracematch, and in order to implement the desired behavior, we need to access the return value of the *makePoint* calls. This must be handled by additional standard AspectJ advice that can store the relevant information (e.g. storing the points in a list). Similar additions are needed to access joinpoint information for single call events, such as accessing the caller and callee object of each event in a trace.

Program 4 shows the tracematch for the sequence diagram with loop from Figure 2. Other symbols and pointcuts are as illustrated before. The loop in the trace is modeled as a one-or-more repetition (+) of a group of symbols *(mStar dis)*.

Expectation Mismatch.

The matches given by the defined tracematch meet our criteria of matching the traces <*makePoint, display, makePoint, display*>. However, the implementation tested (the AspectBench compiler v1.3.0) provided matches that we did not desire: the trace {<*makePoint, display, makePoint, display, makePoint, display*>} results in two matches rather than one, and any subsequent call to makePoint+display will result

Program 4 Example with Loop

```
pointcut makeStarPoint(Figure f): target(f) && call (
   Point+ Figure.makeStarPoint(..)) &&
   within (DrawController+);
tracematch (Figure f, Display d) {
  sym mStar after: makeStarPoint(f);
  mp dis (mStar dis)+ mp dis {
    f.moveAllPoints(10);
  }
}
```

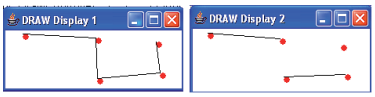


Figure 6: Wrongly (left) and Correctly Matched (right)

in a new match, if there is no event breaking the sequence. This is according to the semantics of *Tracematches*, which looks for matches against all suffixes of the program trace. However, this semantics goes against the intuition of how trace-based aspects work on the sequence diagram level; since an aspect on a sequence diagram typically will modify the matched traces by inserting or removing fragments before, after, or in between the matched trace(s); the result will be a modified trace set where already matched events may have been removed, so it is in general not feasible to include the same events in multiple matches. We would like the events included in an aspect match to be unique for each aspect.

Figure 6 illustrates graphically the implications of the unintended trace match given by the *tracematch* implementation. The leftmost viewer shows that every point created gets connected to the previously created point, while the rightmost viewer only has connected each pair of points, as intended. We cannot avoid the *tracematch* from detecting a match, but we can restrict the specification to connect two and two points pairwise by defining a *superfluous symbol* in the tracematch that is not part of the tracematch expression. This is similar to how the same problem was solved in [1] to provide automatic save after five command executions: for our example, we define a symbol *breaker* that is triggered by the pointcut *traceStop*. It matches any call to the method *breakTrace*, which is defined within the aspect and invoked each time a match is found (Progam 5).

Program 5 Extra Symbol

```
public aspect PointMatcher {
  pointcut traceStop ():
     call (void PointMatcher.breakTrace());
  public void breakTrace () {}
  tracematch (Figure f, Display d) {
    sym breaker after: traceStop();
    // other symbols as before
  mp dis mp dis {
    breakTrace();
    // code as before
  }
}
```

Instance-Level Reasoning.

Tracematch semantics allows symbols to be associated with instances through variables that are referenced by symbols through their pointcut variables (if any). If a variable is bound more than once in a trace, the value of the instance is compared with the old value. If it is not equal, the symbol (the event) is ignored for this particular trace. This allows a tracematch to match only if the same objects are involved in an interaction, which in many cases can be important. Consider for example the graphical application executing with two different viewers, or displays. Each viewer is an instance of *Display* and draws graphical elements on a *Figure*.

Figure 7 shows what may happen when a trace-based aspect does not distinguish different instances when finding trace matches. Here, two points are created consecutively, but in two different displays. The *tracematch* assumes that this is a match for the trace and creates a line between the two points, even though only one of them is visible on that display.

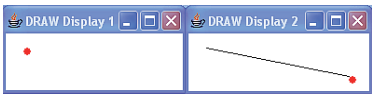


Figure 7: Points Created in Different Displays

For the sake of Tracematches, however, a problem arises when combining the superfluous symbol that breaks the *all suffixes matching* and catching traces consistently for several objects of the same type. Since we trigger the same superfluous symbol to reset the matching process when a match is found, regardless of which object is involved, finding a match on one object will reset the trace for the other object(s) as well.

Handling interactions in which there are multiple instances of the same type and being able to distinguish them are also important features of handling system traces. Just consider a scenario that describes behavior involving two or more client objects talking to one service instance in a specific way; we would like to capture this too. Support for specifying distinct object involvement in interactions has recently been implemented as part of the Tracematches in the Aspectbench framework, but it has not been tested here.

Mapping Rules.

The following rules are followed in the mapping from Java-STAIRS Aspects to *Tracematches*:

1. The mapping follows the weak sequencing and causality rules of sequence diagrams. For one sequence diagram, one *aspect* containing one *tracematch* is defined.

2. Each message with one send and one receive event is mapped to an aspect pointcut, using the name of the message and the sender and receiver roles as identity keys. If the same method (with the same identity) appears more than once, only one pointcut is defined. For each pointcut, a named symbol is defined in the tracematch.

3. For each combined fragment, a mapping is done recursively for its operands until leaf nodes (messages) are mapped to their symbol representation within the combined fragment mapping.

4. An optional fragment is mapped to two alternative branches, each representing the trace with or without the optional: *(a opt(b) c) => (a b c | a c)*. As there is no support for specifying an optional pattern in a tracematch, the specification can become awkward if there are many optionals in the Java-STAIRS aspect.

5. An alternative fragment is mapped to group alternatives in the tracematch. Java-STAIRS Aspect operand fragments are mapped to symbols in each tracematch group: *alt(ab|bc|cd) => ((ab)| (bc)| (cd))*.

6. A loop fragment is mapped to group repeats in a tracematch, either using the '*' or '+' repetition for the group if the loop is unbounded: *loop[*](abc)=> (abc)**. If the loop has a lower bound, but no upper bound, it can be mapped to a numbered repetition followed by an unbounded repetition: *loop[2..*](abc)=> ((abc[2] (abc)*))*. If the loop has an upper bound as well, it can be mapped to group alternatives, with a numbered repetition for each iteration: *loop[1..3](abc)=> ((abc) (abc)[2] (abc)[3])*.

7. A parallel fragment is mapped to group alternatives for each possible alternative resulting from interleaving the events of the different fragment operands: *par(ab| cd) => (abcd|acbd|acdb|cabd|cadb|cdab)*.

8. For all pairs of objects involved in the Java-STAIRS Aspect, a single pointcut is defined to represent actions that are not described by the sequence diagram. A corresponding symbol is defined in the tracematch, but not included in the trace expression. This symbol is defined to avoid matches if there is other communication between the involved objects in a specific trace.

The mapping rules can easily be implemented by a model transformation to automatically generate tracematches from Java-STAIRS Aspects. We have implemented such a mapping using a model-to-text transformation from UML sequence diagrams to *Tracematches* using MOFScript [22]. The transformation is straightforward by following the rules above.

4.3 Refinement and Semantics Preservation

Let us recap the motivation to maintain refinement relationships during system evolution. In this context, we may consider the interaction between a group of objects in our system as a behavioral contract, or a protocol agreement between system roles.

Aspects defined on the trace-level rely on that contract in order to be functional. Modifying a system by using the above defined Java-STAIRS refinement ensures that traces that were potential system traces before refinement still are potential after refinement. In the following we show that applying Java-STAIRS Aspects maintains refinement. If S' is a refinement of S, then applying an aspect A to S' will result in something that is a refinement of the result from applying A to S.

In [21], the authors address semantics preservation of sequence diagram aspects. Semantics preservation was defined by two characteristics: monotonicity of aspect composition with respect to refinement *and* event preservation (i.e. that events are not removed).

For Java systems $\mathcal{S}$ and $\mathcal{S}'$, where $\mathcal{S} \rightsquigarrow \mathcal{S}'$ ($\mathcal{S}'$ is a refinement of $\mathcal{S}$), and a Java-STAIRS aspect $\mathcal{A}$, we can observe the following properties:

- $[\![\mathcal{S}' \oplus \mathcal{A}]\!] \supseteq [\![\mathcal{S} \oplus \mathcal{A}]\!]$.
- All traces $\tau \in [\![\mathrm{S}]\!]$ have a corresponding trace τ' $\in [\![\mathcal{S} \oplus \mathcal{A}]\!]$ such that τ is a subtrace of τ' meaning that τ can be formed from τ' by just inserting events.

In other words, the composition of $\mathcal{S}'$ with $\mathcal{A}$ is a refinement of $\mathcal{S}$ composed with $\mathcal{A}$, and there is no loss of events. This is consistent with the definition of *semantics preservation* for sequence diagram aspects given in [21]; and we can declare Java-STAIRS Aspects to be *semantics preserving* with respect to refinement. **Rationale:** Since $\mathcal{S} \rightsquigarrow \mathcal{S}'$, we know that the traces in $\mathcal{S}$ must also be in $\mathcal{S}'$. If $\mathcal{A}$ matches a trace in $\mathcal{S}$, the same match will be found in $\mathcal{S}'$. Each trace in $\mathcal{S}$ that is modified by $\mathcal{A}$ will have a corresponding trace in $\mathcal{S}'$ that also will be modified by $\mathcal{A}$. Hence, each modified trace in $\mathcal{S}$ will have an equal modified trace in $\mathcal{S}'$. It follows that $\mathcal{S} \oplus \mathcal{A} \rightsquigarrow \mathcal{S}' \oplus \mathcal{A}$.

Consider again the example where parts of our system are redefined (Program 6). In this case, the class *DrawController* is specialized and the method *addPoint* is redefined. The redefinition calls the addPoint of the super class and invokes in addition a *repaint* on the *display* (Program 6). Now consider that the user draws points on the display canvas and triggers the *addPoint* twice, and that we have defined the Java-STAIRS aspect from Figure 1. Using *DrawController2*, the aspect will never match, because there is always a call to *display.repaint* that breaks the trace being searched for.

Program 6 Refinement Examples

```
---- original DrawController::addPoint------
protected void addPoint (int x, int y) {
  Displayable d = figure.makePoint(x, y);
  if (displayable != null) {
    myDisplay.display(d);
  }
}

---- redefined DrawController2::addPoint ------
protected void addPoint (int x, iny y) {
  super.addPoint(x, y);
  myDisplay.repaint();
}
```

We could have anticipated such a modification, and built our aspects so that they could allow for a larger space of traces. However, this is not an ideal situation, as it is not feasible to foresee all future changes. The aspect could also be modified to cope with the new situation, by modifying the trace(s) that should be matched. By allowing only modifications that are Java-STAIRS refinements, we ensure that behaviors (traces) are not removed from the system, so that trace-based aspects can work consistently also in refined systems, i.e. that semantics is preserved.

Refinement Analysis.

In software engineering, it is invaluable to accurately verify that a system behaves as expected. In a software design with sequence diagrams, system refinement can be checked by analyzing and comparing trace sets of behaviors. Sequence diagrams specification with asynchronous communication may represent a lot of traces, so computing and comparing traces become intractable [10]. In his PhD thesis, Lund defined a method for refinement testing based on test generation and execution of sequence diagrams [19]. A similar approach may be taken at the implementation level; testing can be applied for analyzing the refinement relationships between code modifications. Static analysis can also to a certain extent be used to determine refinement validity. Although potential traces cannot be determined in the general case, the examples in Program 6 illustrated that local contributions to traces can be analyzed and used to determine refinement relationships.

4.4 Multiple Aspects and Interference

In the case that multiple sequence diagram aspects are involved, interference or conflict between them may occur. We cannot prevent interference or conflict from happening, but we have reasonable control over when it may occur. If two sequence diagrams involve the same set of lifelines, they are at risk of interfering with each other. If one (or both) introduces new message interactions between the common lifelines, this may result in removal of matches or addition of new matches, but only if it triggers relevant communication initiated outside of the aspects; communication from the aspect is discarded by filters. Detecting of such conflicts is out of scope for this paper. The topic of aspect conflict resolution is thoroughly addressed by others, e.g. by Douence et al. [7].

5. DISCUSSION AND RELATED WORK.

The work presented is strongly related to works on stateful and trace-based aspects, as we provide an integration between design-level and implementation-level trace-based aspects. The work on semantics preservation of sequence diagrams aspects by Oldevik and Haugen [21] was a motivator for addressing this topic in the space between design and implementation. The semantic definitions for Java-STAIRS Aspects and the notion of refinement are based on that of STAIRS [13]. Our approach is complementary to trace-based aspects at the implementation and design level.

There is a range of programming language work on trace-based aspect languages that relates to our choice of technology for mapping Java-STAIRS Aspects to an implementation level, namely *Tracematches* [1]. One of our criteria was that it should be feasible to use in practice and open for a natural mapping from sequence diagrams. Other, related approaches such as Declarative Event Patterns [30] and JaSCo with stateful aspects [28] were also considered. Walker and Viggers [30] introduce *Declarative Event Patterns (DEP)* as a means of implementing protocols. They define a language extension to AspectJ called *tracecuts* and provide a proof-of-concept implementation. This language is more expressive as it based on context-free grammars rather than regular expressions. However, it does not seem to support the filtering mechanisms that we want for our Java-STAIRS Aspects semantics. Vanderperren et al. defined and implemented stateful aspects in JaSCo [28, 27], modeled by the theoretical work of Douence et al. [8, 9]. They can distinguish interactions on a per thread, class, object, method, cflow, or everything basis, which is not possible in Tracematches. However, the approach does not seem to be able to describe interactions that need to distinguish different objects of the same type. Both of the latter approaches will provide matches for traces with occurring events in between,

which is not what we want for events between our objects of interest. Klose and Ostermann [16] define the *Gamma* language in which pointcuts can be considered as predicates over the complete execution trace, including past and future events. The execution trace is stored in a database as Prolog facts, and pointcuts are in effect Prolog queries. A major drawback with the approach is that the system must be run at least once before pointcuts can be determined. The concepts, however, match well that of aspects on the sequence diagram level, where the complete execution trace is available.

In the design space, relevant work on trace-based aspects has been done with state machine modeling [5] and with sequence diagrams [25, 11]. Cottenier et al. [5] argue that stateful aspects at the programming level is a symptom of unsuitable abstractions of the problem, and that a model-based state machine oriented approach is a better model and abstraction for stateful aspects within *reactive* systems. Even if a reactive system is implemented by a state-machine abstraction, there might be a need to capture the interaction between the reactive components or services of the system; these may well be captured by interactions (e.g. sequence diagrams) and be subject to trace-based aspects too. This is shown by Krüger et al. [18], who model trace-based aspects pertinent to reactive components (services) and implement the behavior by a (reactive) trace monitor. Maoz and Harel [20] generate aspect-based systems from Live Sequence Charts (LSC) in what they call scenario aspects, where the aspects are used as implementation technology for the LSCs. Pre-charts of an LSC act as guards for main-charts, which in that sense resemble trace-based aspects.

The Join Point Designation Diagram (JPDD) approach [25, 11] defines a notation for describing pointcuts based on UML sequence diagrams, acknowledging that some type of pointcuts are really hard to describe at the code level. They provide mappings from the JPDD specifications to implementations in both AspectJ and JaSCo. We evaluated JPDDs and the supporting tools for generating Java-STAIRS Aspects, but found that they could not express the semantics desired for our sequence diagram mappings. Furthermore, the JPDD tools provide only limited UML support without notation for expressing things like alternatives, loops, conditionals, etc. We compared JPDD with our approach by testing aspects on the *figure drawing example* with both approaches.

In this work, we only addressed synchronous communication of possibly multi-threaded Java programs. This could be generalized to explicitly explicitly support asynchronous communication by extending the Java-STAIRS semantics. A limitation in the approach is its inherent single-process (multi-threaded) nature, constrained by the mapping to Java and Tracematches. Providing trace-based aspects in distributed systems is a different ballgame, which requires some kind of common event monitor service, similar to that described in [18].

6. CONCLUSION AND FUTURE WORK.

We have proposed a modeling notation for trace-based aspects using UML sequence diagrams, which is used for representing traces that are pointcuts of Java executions, and advice that inserts behavior when aspects are matched. We defined Java-STAIRS, a trace semantics for Java based on STAIRS sequence diagram semantics, which gives the theory for reasoning about the sequence diagram aspects at Java level. Filtering mechanisms on Java-STAIRS traces were defined to allow a Java system and its execution traces to be analyzed in the context of the roles and interactions defined by a sequence diagram. We defined refinement of Java in terms of the Java-STAIRS semantics, and showed that Java-STAIRS Aspects are semantics preserving with respect to this refinement notion. We defined a mapping of Java-STAIRS Aspects to *Tracematches* in Java and implemented this using a model-to-text transformation. Finally, we tested and illustrated the approach on a figure drawing application.

For future work, we will investigate further how composition operators in sequence diagrams relate to Java mechanisms, and how refinement and aspect application is influenced by such composition operators. We also need to analyze further the scalability of the implementation approach with respect to complexity of the trace expression, e.g. when handling many alternatives or larger parallel fragments. We will also look further into models for supporting trace-based aspects in distributed and asynchronous systems.

7. ACKNOWLEDGMENTS

This work has been done in the context of the SWAT project (Semantics-preserving Weaving - Advancing the Technology), funded by the Norwegian Research Council (project number 167172/V30).

8. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *20th Annual Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364, New York, USA, 2005. ACM.

[2] P. Avgustinov, J. Tibble, and O. de Moor. Making Trace Monitors Feasible. Technical Report abc-2007-1, University of Oxford, UK, 2007.

[3] E. Baniassad and S. Clarke. Theme: an Approach for Aspect-Oriented Analysis and Design. *26th International Conference on Software Engineering (ICSE)*, pages 158–167, 23-28 May 2004.

[4] M. Broy, I. H. Krüger, and M. Meisinger. A Formal Model of Services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):5, 2007.

[5] T. Cottenier, A. van den Berg, and T. Elrad. Stateful Aspects: the Case for Aspect-Oriented Modeling. In *10th international workshop on Aspect-oriented modeling (AOM)*, pages 7–14, New York, USA, 2007. ACM.

[6] M. Deubler, M. Meisinger, S. Rittmann, and I. Krüger. Modeling Crosscutting Services with UML Sequence Diagrams. In *Model Driven Engineering Languages and Systems, MODELS*, Jamaica, 2005. Springer.

[7] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative*

Programming and Component Engineering, pages 173–188, 2002.

[8] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *3rd International Conference on Aspect-oriented Software Development (AOSD)*, pages 141–150, New York, NY, USA, 2004. ACM.

[9] R. Douence, P. Fradet, and M. Südholt. *in Trace-based Aspects*, chapter Trace-based Aspects., pages 201–217. Addison-Wesley, ISBN 0-32-121976, 2004.

[10] R. Grønmo, F. Sørensen, B. Møller-Pedersen, and S. Krogdahl. A Semantics-Based Aspect Language for Interactions with the Arbitrary Events Symbol. *European Conference of Model Driven Architecture, Foundations and Applications (ECMDA), Springer*, 5095/2008, 2008.

[11] S. Hanenberg, D. Stein, and R. Unland. From Aspect-oriented Design to Aspect-oriented Programs: Tool-supported Translation of JPDDs Into Code. In *6th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 49–62, New York, NY, USA, 2007. ACM Press.

[12] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Why Timed Sequence Diagrams Require Three-Event Semantics. Research Report 309, ISBN 82-7368-261-7, University of Oslo, 2004.

[13] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS Towards Formal Design with Sequence Diagrams. *Software and Systems Modeling*, pages 355–367, 2005.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–, Jan. 2001.

[15] J. Klein, L. Helouet, and J. Jezequel. Semantic-based Weaving of Scenarios. In *AOSD '06: The 5th International Conference on Aspect-oriented Software Development*, pages 27–38, New York, NY, USA, 2006. ACM Press.

[16] K. Klose and K. Ostermann. Back to the Future: Pointcuts as Predicates over Traces. *Foundations of Aspect-Oriented Languages workshop (FOAL)*, 2005.

[17] C. Koppen and M. Storzer. PCDiff: Attacking the Fragile Pointcut Problem. *European Interactive Workshop on Aspects in Software (EIWAS), Berlin, Germany*, 2004.

[18] I. H. Krüger, M. Meisinger, and M. Menarini. Runtime Verification of Interactions: From MSCs to Aspects. *Workshop on Runtime Verification*, 2007.

[19] M. S. Lund. *Operational Analysis of Sequence Diagram Specifications*. PhD thesis, Department of Informatics, University of Oslo, 2008.

[20] S. Maoz and D. Harel. From multi-modal scenarios to code: Compiling lscs into aspectj. In *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE*, pages 219–230, New York, NY, USA, 2006. ACM.

[21] J. Oldevik and Ø. Haugen. Semantics Preservation of Sequence Diagram Aspects. *Fourth European Conference on Model Driven Architecture Foundations and Applications (ECMDA)*, pages 215–230, 2008.

[22] J. Oldevik, T.Neple, R. Grønmo, J. Aagedal, and A.Berre. Toward Standardised Model to Text Transformations. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA)*, pages 239–253, Nuremberg, 2005. Springer.

[23] OMG. The Unified Modeling Language: Superstructure, Version 2.1. Standard ptc/06-01-02, OMG, 2006.

[24] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. *European Conference on Object Oriented Programming (ECOOP)*, 3586/2005:214–240, 2005.

[25] D. Stein, S. Hanenberg, and R. Unland. Query Models. In *7th International Conference of Modelling Languages and Applications*, volume 3273/2004, pages 98–112, Lisbon, Portugal, 2004. Springer.

[26] D. Stein, S. Hanenberg, and R. Unland. Expressing Different Conceptual Models of Join Point Selections in Aspect-oriented Design. In *5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 15–26, New York, NY, USA, 2006. ACM Press.

[27] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-oriented Approach Tailored for Component Based Software Development. *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 21–29, 2003.

[28] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. D. Fraine. Stateful Aspects in JAsCo. *4th Intl. Workshop of Software Composition*, 2005.

[29] R. J. Walker and G. C. Murphy. Joinpoints as Ordered Events: Towards Applying Implicit Context to Aspect-Orientation. *Workshop on Advanced Separation of Concerns at ICSE*, 2001.

[30] R. J. Walker and K. Viggers. Implementing Protocols via Declarative Event Patterns. In *12th ACM International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, pages 159–169, New York, NY, USA, 2004. ACM.

[31] J. Whittle and P. Jayaraman. MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. *11th International Workshop on Aspect-Oriented Modeling (AOM)*, 2007.

Appendix E

Paper V: Confluence in Domain-Independent Product Line Transformations

Authors. Jon Oldevik, Øystein Haugen, and Birger Møller-Pedersen.

Paper Summary. In this paper, we describe an implementation of a transformation for a domain-independent variability language. This implementation is then used for reasoning about the confluence properties of variability resolutions. We give the criteria for when these variability transformations are confluent and when they are conflicting. We also describe how to analyse confluence in the context if given variability resolutions.

Author Contribution. Jon Oldevik was the main author and responsible for the main part of the research and writing of this paper, accounting to about 90% of the work. Specifically, Jon Oldevik was the main contributor in the implementation of the variability transformation and for the formal definitions and approaches supporting analysis of confluence and conflicts.

Publication Arena. Published in the proceedings of Fundamental Approaches to Software Engineering (FASE) 2009. Acceptance rate 24% (30/124).

This article is removed.

Appendix F

Paper VI: Model Composition Contracts

Authors. Jon Oldevik, Massimiliano Menarini, Ingolf Krüger

Paper Summary. In this paper, we describe a technique for specifying and checking composition contracts for models. The composition contracts provide a level of control for what kind of modifications a composition can be allowed, or should be denied, to perform on a base model. This is done in terms of constraints defined in contracts, which restrict the eligibility for composition.

Author Contribution. Jon Oldevik was the main author and responsible for a major part of the research and writing of this paper. Specifically, Jon Oldevik was a major contributor to the composition contract concept, the language, and the tool prototype development.

Publication Arena. Published in the proceedings from ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS) 2009. Acceptance rate 23% (46/211).

This article is removed.