

Abstract Delta Modeling

Software Product Lines and Beyond

Proefschrift

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden,

op gezag van Rector Magnificus Prof. Mr. C.J.J.M. Stolker,

volgens besluit van het College voor Promoties

te verdedigen op woensdag 12 november 2014

klokke 11:15 uur

door

Michiel Helvensteijn

geboren te Voorburg

in 1986

Promotor

Prof. Dr. F.S. de Boer (Universiteit Leiden)

Copromotor

Dr. D. Clarke (Uppsala Universitet)

Promotiecommissie

Prof. Dr. F. Arbab (Universiteit Leiden)

Dr. M.M. Bonsangue (Universiteit Leiden)

Prof. Dr. R. Hähnle (Technische Universität Darmstadt)

Prof. Dr. J.N. Kok (Universiteit Leiden)



Universiteit
Leiden



The work in this thesis has been carried out at Centrum Wiskunde & Informatica and Leiden University, and under the auspices of the research school IPA: Institute for Programming research and Algorithmics. The research was partially funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models

Copyright © 2014 by Michiel Helvensteijn (www.mhelvens.net)

Cover design and artwork by Tim Hengeveld (www.timhengeveld.com)

Typeset with \LaTeX

Printed by Ipskamp Drukkers BV

Published by Michiel Helvensteijn

ISBN: 978-90-822936-0-9

IPA Dissertation Series 2014-14

Contents

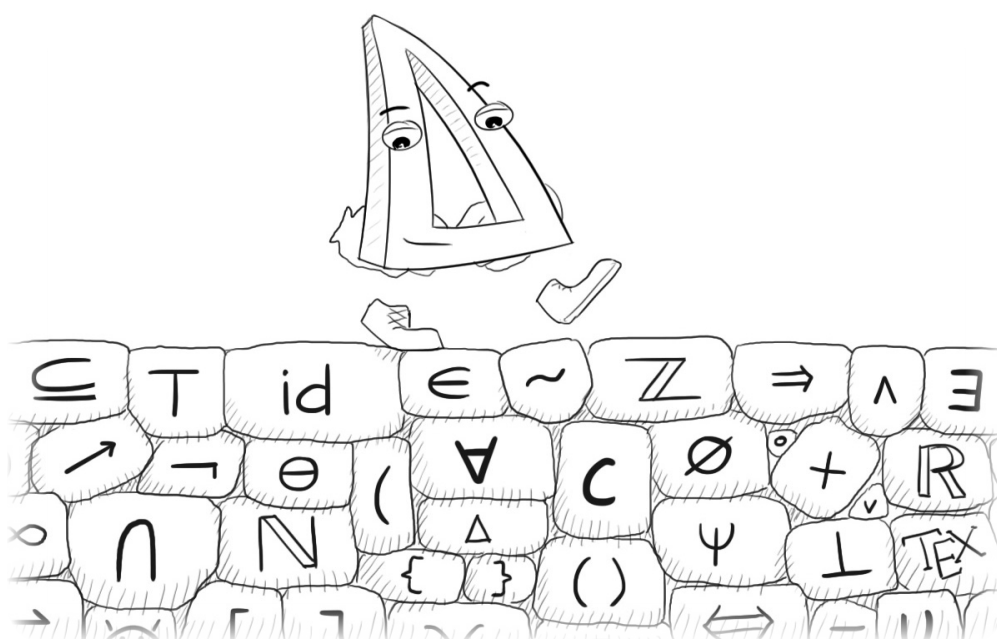
Contents	i
1 Introduction	2
1.1 Problem Statement	4
1.2 Existing Approaches	5
1.3 The Abstract Delta Modeling Approach	6
1.4 A Running Example: The Editor Product Line	7
1.5 Papers & Chapters	10
1.6 Typographic Conventions	16
1.7 Mathematical Preliminaries	18
2 Algebraic Delta Modeling	30
2.1 Introduction	31
2.2 Deltas & Products	36
2.3 The Software Deltoid	39
2.4 The Semantics of Deltas	44
2.5 Delta Refinement and Equivalence	47
2.6 Delta Algebras	48
2.7 Classification of Deltoids	56
2.8 Encoding Related Approaches	60
2.9 Conclusion	64
2.10 Related Work	64
3 Delta Models	68
3.1 Introduction	69
3.2 The Delta Model	73
3.3 A Conflict Resolution Model	75
3.4 A Fine Grained Software Deltoid	82
3.5 Ambiguous Delta Models	88
3.6 Nested Delta Models	90
3.7 Conclusion	93
3.8 Related Work	93
4 Product Lines	96
4.1 Introduction	97
4.2 Feature Modeling	99

4.3	Product Line Implementation	102
4.4	Product Line Specification	106
4.5	Parametric Deltas	109
4.6	Nested Product Lines	114
4.7	Conclusion	116
4.8	Related Work	116
5	L^AT_EX Meets Delta Modeling	118
5.1	Introduction	119
5.2	delta-modules: Deltas for Document Generation	120
5.3	pkgloader: An ADM-based Package Manager	126
5.4	Conclusion	131
5.5	Related Work	132
6	Delta Logic	134
6.1	Introduction	135
6.2	A Multimodal Language	136
6.3	Kripke Frames	136
6.4	Kripke Models	142
6.5	Conclusion	147
6.6	Related Work	147
7	Delta Modeling Workflow	148
7.1	Introduction	149
7.2	The Subfeature Relation	150
7.3	Locality	151
7.4	Workflow Description	152
7.5	The Abstract Behavioral Specification Language	155
7.6	The Fredhopper Access Server	161
7.7	Conclusion	163
7.8	Related Work	163
8	Dynamic Product Lines	166
8.1	Introduction	167
8.2	Automated Profile Management	168
8.3	An Operational Semantics	175
8.4	Cost and Optimization	188
8.5	Conclusion	192
8.6	Related Work	194
9	Conclusion	196
9.1	A Look Back	197
9.2	A Look Forward	205
A	DMW Operational Semantics	210
A.1	The Subfeature Relation	210
A.2	Non-interference	211
A.3	An Operational Semantics	211
A.4	Analysis	216

<i>CONTENTS</i>	iii
Summary	220
Samenvatting	222
Index	226
Bibliography of My Publications	234
Main Bibliography	236

Introduction

Motivation and Mathematical Foundation



If you are reading this thesis, you probably won't need much convincing of the fact that *software* is now an essential ingredient in many different aspects of our society. The improvement of software and its development is therefore a broad area of academic pursuit.

A lot of important research is about writing software that is *correct*, *efficient* and *secure*. The research presented in this thesis, however, is primarily about writing software that is *modular* and *easy to maintain*. Now that software is updated over the internet—even hosted entirely online—, release cycles become ever shorter and it becomes ever more important that software be easy to adapt and extend without making it too complex.

Specifically, this thesis is about *Abstract Delta Modeling (ADM)*, a formal framework developed to achieve modularity and separation of concerns in software, as well as provide the opportunity for variability management and automated product generation in *Software Product Line Engineering (SPLE)*.

The thesis follows a predominantly formal approach. This is important, as it avoids vagueness and ambiguity. It allows the use of mathematical proof techniques, which gives the academic community a high level of confidence in the results. While software engineering in general has come a long way when it comes to formal analysis, SPLE has been mostly an empirical field of study. But this has changed in recent years. This thesis is a product of the European HATS project [80]:

HATS: Highly Adaptable and Trustworthy Software using Formal Models

This thesis presents a formal foundation for the techniques of *delta modeling*, which was the main approach to variability used by the HATS project. To do this, it employs (among other things) abstract algebra, modal logic, operational semantics and Mealy machines, and lays the bridges between the different disciplines as we go. The chapters to come provide a broad overview of the ADM framework and its possibilities, as well as a number of existing practical applications, laying a foundation for further research and development.

This **Introduction** chapter is organized as follows. Section 1.1 introduces the main problems we are trying to solve. Section 1.2 introduces a number of existing approaches to solving those problems and points out shortcomings that we will try to overcome. Section 1.3 then outlines the general delta modeling approach proposed in this thesis. To illustrate and motivate this approach we study an example in Section 1.4, which we'll be referring back to throughout the rest of the thesis. Section 1.5 outlines the structure of the thesis and relates the chapters to my academic publications. This thesis is primarily a work of *formal methods*, introducing and building upon mathematical and logical notions. To help the reader, it adheres to a number of typographic conventions and the theory is based on well-established concepts of discrete mathematics. These are introduced, respectively, in Sections 1.6 and 1.7.

1.1 Problem Statement

Programming is an activity very prone to human error. As more and more features are implemented in a software system by different programmers, progress will often slow to a crawl. It is all too easy for programmers to lose overview of what their code is doing when it is spread across the code base surrounded by the code of others. This can result in bugs and inevitably much time will need to be spent on maintenance. This, in turn, results in more expensive software that takes longer to reach the user.

To prevent a large software system from collapsing under its own complexity, its code needs to be well-structured. Manny Lehman (remembered as the Father of Software Evolution) stated the following as his second law of software evolution [48, 119]:

“As a program is evolved its complexity increases unless work is done to maintain or reduce it.”

Ideally we want all code related to a certain *feature* (sometimes called *concern*) to be grouped together in one module —which is called *feature modularity* or *feature locality* [89, 109, 156]— and code belonging to different features *not* to be mixed together — which is called *separation of concerns* [96, 112, 114, 147]. But many concerns cannot be easily captured by existing abstractions. They are known as *cross-cutting concerns*. By their very nature their implementation needs to be spread around the code base, so modularization and separation of concerns are still elusive.

The software engineering discipline that has the most to gain from those properties is *Software Product Line Engineering (SPLE)*, a relatively new development. To quote van der Linden, Schmid and Rommes [122]:

“Software product lines represent perhaps the most exciting paradigm shift in software development since the advent of high-level programming languages.”

SPLE is concerned with the development and maintenance of *multiple* software systems at the same time, each possessing a different (but often overlapping) set of features — a form of *mass customization* [117, 155].¹ This gives rise to an additional need. It is no longer enough that the code for a given feature is separated and modular; it also need to be *composable* and able to deal gracefully with the presence or absence of other features. We need to be able to make a selection from a set of available features and have the corresponding software mechanically generated for us — a process known as *automated product derivation* [1, 2, 65, 163] (Figure 1.1). That is no small order.

¹The term ‘product line’ is really a misnomer. It comes from 1834, when a typical retailer had only a small number of products, *lined up* in front of him [55, 157]. In a (software) product line, neither the products nor the features need to be linearly ordered in any way. The term ‘product family’, used primarily in Europe [155], is therefore technically more accurate. Nonetheless, ‘product line’ is used much more widely, so we’ll stick with it.

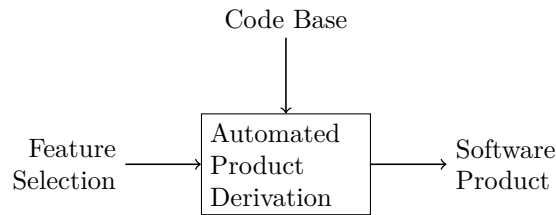


Figure 1.1: The process of Automated Product Derivation.

1.2 Existing Approaches

A number of programming paradigms have recently emerged out of a need to solve the problem of cross-cutting concerns. Among the more recent are *Aspect Oriented Programming (AOP)* and *Feature Oriented Software Development (FOSD)* [13, 104, 110, 125, 156]. We now summarize these existing approaches.

1.2.1 Aspect Oriented Programming

Around the turn of the millenium, *Aspect Oriented Programming (AOP)* [107, 112, 114, 126, 133, 144] was proposed to tackle cross-cutting concerns. An *aspect* can specify code (called *advice*) to be added at specific locations (called *pointcuts*) in an existing code base. All code belonging to a concern can be grouped together.

AOP is a step in the right direction, as many related code fragments that once had to be spread around can now be grouped into one aspect. Generally, however, AOP only supports the insertion of statements around identified join-points inside methods and the addition of members to an existing class (using *inter-type declarations* [12]). Moreover, there has been a general lack of support from AOP to help us reason about —and coordinate— the interaction between different concerns [51], though there has been recent work attempting to improve this situation [129].

1.2.2 Feature Oriented Software Development

In literature, *Feature Oriented Software Development (FOSD)* is often equated with software product line engineering, so approaches that claim to follow FOSD are often capable of automated product derivation to some degree. A *Software Product Line (SPL)* is a collection of similar software systems (or *software products*) that differ only by which features they support and can, therefore, share a lot of the same source code. FOSD and SPLE share basically the same techniques. Compared to AOP, these techniques have the added benefit of supporting automated product derivation.

To accomplish this, methods were developed to express the *variability* between products: where and how can the code of one product differ from that of another, and which features account for those differences?

Techniques for expressing SPL variability can be divided into two main categories [108]: *annotative* techniques and *compositional* techniques.

1.2.3 Annotative Software Product Line Techniques

An *annotative* code-base consists of the totality of available code — all features included. Specific annotated parts of that code — parts belonging to features we don't want — can be removed to create a final program [106, 108, 181]. Of all approaches to automate product derivation, the annotative approach is probably most popular one practice. A prominent example of this is the Linux kernel, which is an immense collection of C code annotated by `#ifdef` preprocessor directives, which allow conditional compilation [171].

But in the annotative approach, code is not gathered in modules; just annotated wherever it appears. Consequently, it does not enjoy the modularity or separation of concerns we want. Furthermore, it forces feature related code into an improper structure: a linear textual order. This is an *overspecification*, as it can never be clear whether the order between two different lines of code was by design, or forced upon the developers by the annotative paradigm. As the order between two statements can be semantically significant in an imperative programming model, this can cause unanticipated bugs and complexity in the long run.

1.2.4 Compositional Software Product Line Techniques

Of particular interest to us are the so-called *compositional techniques*, such as GenVoca [30], AHEAD [31] and Delta Modeling [160, 162–164]. In contrast to the annotative techniques, the idea here is to gather all code belonging to a feature — or a closely related set of features — into a single module: a *feature module*. To obtain any specific product from the product line, one only has to choose the appropriate set of modules and *apply* them to the *core product* — the code that contains only the bare basics — in the proper order. When applied, they can then *modify* the core in order to integrate their features. This is generally done by a process known as *invasive composition* [23], called so because feature modules often need break object oriented encapsulation and class boundaries in order to apply the proper modifications to the code.

1.3 The Abstract Delta Modeling Approach

This thesis is about *Delta Modeling*, a compositional technique for implementing variability of software product lines. Its feature modules are called *deltas*, as they describe the *difference* between two systems.

Rather than focus merely on software, we define an abstract approach to delta modeling called *Abstract Delta Modeling (ADM)*, which comprises a number of related formalisms. These allow us to reason about delta modeling without having to consider any specific programming paradigm. A great number of relevant concepts can be discussed without ever mentioning software. In fact, focussing on software too early — or worse, a specific programming language — could narrow our vision and cause us to miss good ideas.

Dave Clarke, Ina Schaefer and myself [1, 2] first introduced ADM as a formal approach for modeling software product lines. These articles give an algebraic description of deltas and how they can be combined and linked to the higher level notion of feature. One of the main contributions of that work was a way of organizing deltas in a partially ordered structure. This allows us

to express relations and dependencies between deltas that closely mirror the original design intentions. Specifically, we could now reason about *conflicting deltas*—independent deltas with incompatible implementations—and *conflict resolving deltas*—specialized modules to mediate conflicts and streamline feature interaction.

Already in that work, delta modeling was not restricted to software, but rather product lines of *any* domain. So even though the main inspiration for ADM was to structure software systems, it can just as readily be used to formulate sets of mathematical equations, model possible hardware configurations or design a line of office furniture.

Since the original article, delta modeling has been extended and refined in several directions [3–7], all of which we will discuss in this thesis. In particular, a *modal logic* was created to make it easier to reason about the semantics of products and deltas; a *development workflow* for product lines was introduced; and last but not least, an abstract semantics for *dynamic delta modeling* was developed, allowing deltas to be applied at runtime, on demand, based on environmental conditions. Much of this theory has already been put into practice. Delta modeling was implemented in the ABS modeling language [8, 52] (developed by the HATS project), and I have implemented it for Javascript and L^AT_EX. The development workflow was applied to the Fredhopper Access Server [7]—an industrial scale case study—, and I applied dynamic delta modeling techniques to the development of a profile management application for Android.

1.4 A Running Example: The Editor Product Line

In this section we introduce the plans for a fictional software product line of source code editors such as you might find in IntelliJ IDEA [99] or Eclipse [139]. We will use this product line as an illustrative example throughout most of the thesis.

Section 1.4.1 describes the features we want to implement and Section 1.4.2 gives an idea of how our delta modeling based implementation will work.

1.4.1 The Specification

The specification of the *Editor product line* includes a set of *feature configurations* corresponding to the different kinds of editors we want to be able to generate. Each represents a different selection of features to include in the final product. Such a set of feature configurations is called a *feature model*. Figure 1.2 shows a *feature diagram* [66] representing the feature model of the Editor product line. We’ll consider the following features:

- *Editor* (*Ed*) is the only mandatory feature of the product line. It represents basic text editing functionality.
- *Printing* (*Pr*) allows the user to print the code in the editor on paper.
- *Syntax Highlighting* (*SH*) displays code in color for easier recognition of different programming language constructs.
- *Error Checking* (*EC*) performs simple grammatical analysis on code and underlines certain errors. Hovering over an error with the mouse-pointer triggers a tooltip with extra information.

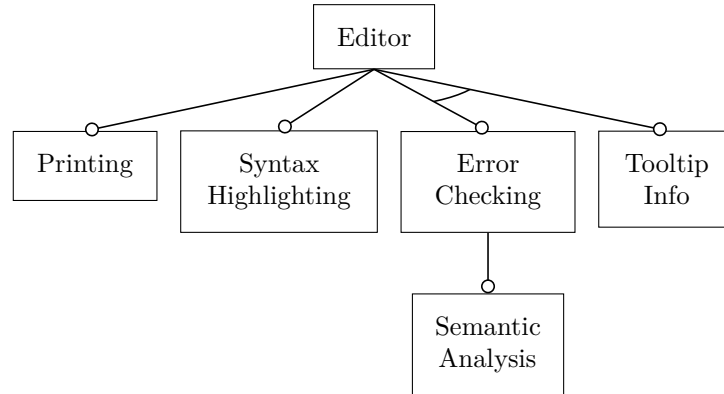


Figure 1.2: The feature diagram of the Editor product line. Each box represents a feature. The base feature (Editor) is mandatory. A connector with an open circle at the end $\text{---}\circ$ indicates an optional subfeature. A subfeature can only be selected if its parent feature is also selected. A curve between two connectors \wedge indicates a mutually exclusive choice between subfeatures.

- Error Checking has an optional subfeature: *Semantic Analysis* (SA). It performs more sophisticated error analysis of program code.
- *Tooltip Information* (TI) shows contextual information in a tooltip when the mouse-pointer hovers over some code. Since Error Checking can also trigger tooltips, we decide to make the two features mutually exclusive, i.e., a final product should not include both.

This product line consists of 16 different editors, as there are 16 possible feature configurations.

1.4.2 An Implementation using Deltas

We now look at a delta-based implementation of the Editor product line in some object oriented programming language. It will be complete enough to generate all 16 possible end products, yet modular enough not to require any code duplication.

Figure 1.3 shows an overview of the entire code-base. Each delta is represented by a dashed box, displaying the modifications it can perform to the program. The internal boxes mimic UML class-notation [159]. The keywords **add**, **mod** and **rep** respectively indicate addition, modification and replacement of code artefacts. A modification descends one level to apply more fine-grained transformations. Each delta is also annotated with its *application condition*, indicating the feature configurations for which it should be applied.

The Editor code-base consists entirely of deltas. They are designed to incrementally modify the *empty program* (which is not shown). Each feature f is implemented by a single delta — which we'll call d_f . For example, d_{Ed} implements the basic editing functionality of *Ed* by adding the `Editor` class to the empty program. It is always applied, because *Ed* is a mandatory feature.

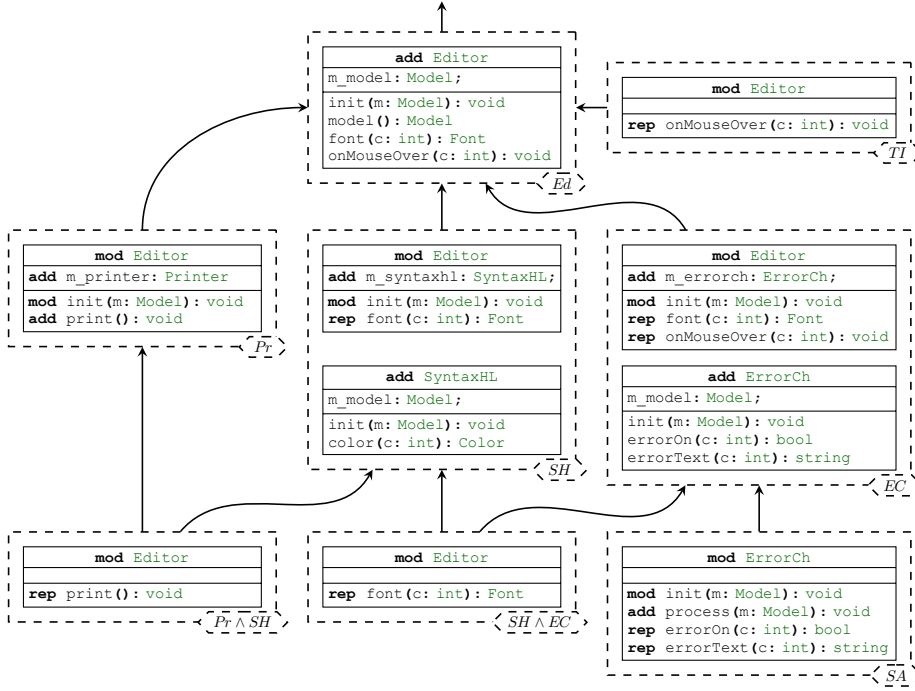


Figure 1.3: A delta diagram of the Editor product line implementation. Each dashed box represents a delta with an overview of the modifications it can apply to the core product in a UML-like notation. Method bodies are omitted for brevity. The arrows \longrightarrow indicate which deltas are allowed to overwrite or alter the modifications of others. The propositional logic $\langle \text{formula} \rangle$ attached to the bottom right corner of each delta represents the feature configurations for which that delta should be applied.

The other deltas add some functionality on top of the basic editor, just as a programmer would in traditional software engineering. For instance, d_{Pr} implements the Pr feature by making some modifications to the `Editor` class. It adds a field `m_printer`, a method `print()` and it modifies the pre-existing `init()` method.

Since d_{Pr} continues where d_{Ed} left off, it is important that they are applied in the right order. That's where the arrows in the diagram come in. They represent the partial *application order*, a part of the code-base design. Because of this order, d_{Pr} can be written with the certainty that d_{Ed} will be applied first. This is always necessary for deltas that implement subfeatures. Besides the subfeatures on the first level, it is also the case for d_{SA} , which implements the second-level subfeature SA . By *not* placing an order between two deltas, a developer indicates that the order in which the deltas are applied should not matter — an important design intention.

This makes the code-base very robust to change, as code dependencies are made explicit in the design. If the code of d_{Ed} is ever changed, the developers of Pr may receive an automated warning, so they can determine whether they should make corresponding changes to d_{Pr} .

In certain situations the application order can even ensure an automated error message when two independent deltas make incompatible changes to the program. For example, because of the limited interface of `Editor`, d_{SH} and d_{EC} both need to replace the `font(int)` method; the former to change the color of the content, the latter to underline it in case of errors. (Incidentally, note that besides modifying `Editor`, both deltas also add a new class of their own.) Since neither delta has priority over the other —and rightfully so— there is a conflict in the code-base that needs to be resolved, and it can be automatically detected.

We resolve the conflict with what we call a *conflict resolving delta*. The delta annotated with $SH \wedge EC$ —which we'll call $d_{SH \wedge EC}$ — is applied only in situations where this conflict would occur and replaces the `font(int)` method with a final version that properly combines the implementations of d_{SH} and d_{EC} .

At first glance it appears as though d_{EC} and d_{TI} are also in conflict, as they both replace the `onMouseOver(int)` method. However, this will never be a problem. By the feature model (Figure 1.2) the features EC and TI can never be selected together, so there can never be a conflict in the first place.

So what is $d_{Pr \wedge SH}$ doing? d_{Pr} and d_{SH} are not in conflict. However, we'd like these two features to be more than the sum of their parts. When we can both highlight the code and print it, it makes sense that we should also be able to print the code *in color*. We call $d_{Pr \wedge SH}$ an *interaction implementation delta*. It replaces `print()` with a version that makes use of the `font(int)` method from d_{SH} . It is similar in many ways to a conflict resolving delta, but the reason we need it cannot be detected automatically.

And so we have an implementation that explicitly links features to their corresponding code. It is modular: all code belonging to the same feature (combination) is grouped together in a delta. It has separation of concerns: code belonging to different features is separated. We also have automated product derivation: given any desired feature configuration, we can select the relevant deltas from the model, then apply them in the proper order. This description did leave out a lot of details. We will spend the rest of this thesis exploring those details.

1.5 Papers & Chapters

This section outlines the chapter structure of this thesis and lists the publications on which it is based.

1.5.1 My Publications

In October 2009 I presented my idea for conflict resolving deltas at a HATS working meeting in Leuven, Belgium. This began my collaboration with Dave Clarke and Ina Schaefer on Abstract Delta Modeling. I was fortunate to stumble upon a viable research idea so early on. It provided me with a sense of focus

for more than three years, improving and extending ADM. All of my publications since the abovementioned collaboration, therefore, are on the same topic, allowing for a thesis with a coherent narrative. A bibliography-style list of those publications can be found on Page 234. They are split off from the main bibliography, and are numbered sequentially throughout the thesis: [1–10]

[1, 9] Abstract Delta Modeling

This paper was written by Dave Clarke, Ina Schaefer and myself, and forms the theoretical core of this entire thesis. A technical report [9] accompanied the main paper [1], containing full proofs that did not fit within the page limit. I presented the paper at GPCE 2010 in Eindhoven, the Netherlands. It introduces an abstract notion of products, of deltas that can transform products, partially ordered delta structures called delta models and delta-based product lines. This theory is treated in Chapters 2 to 4. This paper is also the source of the Editor Product Line example of Section 1.4. I’ve chosen to extend it for this thesis, since it seems to have done a great job in helping fellow researchers understand the practical application of the theory. It is simple and familiar, yet flexible enough to demonstrate most of ADM’s benefits.

[5] Delta Modeling Workflow

This paper was written by me in 2011 and presented at VaMoS 2012 in Leipzig, Germany, together with its companion paper [7]. ADM is meant to model real-world software product lines, but the main work only presented a formal framework; one that encompasses a vast expressive space, but without any guidelines to its recommended use. So this paper proposed a development workflow based on ADM, which allows concurrent and isolated development of features while preserving beneficial global properties. The workflow is treated in Chapter 7, and a more thorough formalization can be found in Appendix A.

[7] Delta Modeling in Practice, a Fredhopper Case Study

This paper was written by Radu Muschevici, Peter Wong and myself in 2011, and put the Delta Modeling Workflow through its paces on the industrial-scale case study of the Fredhopper Access Server. I presented it at VaMoS 2012 together with the theoretical paper described above. It includes an analysis on the effectiveness of the workflow in a practical setting, which is included in Chapter 7.

[3] A Modal Logic for Abstract Delta Modeling

This paper was written by Frank de Boer, Joost Winter and myself, one of three publications presented by me at SPLC 2012 in Salvador, Brazil. It presents a multimodal logic meant for reasoning about the effects of deltas and the semantics of products. Its main innovation is a modality for delta models. This theory is treated fully in Chapter 6.

This was the first paper to interpret deltas as mathematical relations between products, rather than functions. This idea has been fully integrated into the main ADM theory of this thesis — something which has not yet been published. It’s had a particularly profound effect on Chapters 2 and 3.

[6] Dynamic Delta Modeling

This paper was written by me and presented at SPLC 2012 in Salvador, Brazil. It put the flexibility of ADM to the test, as it applies the formalism in a dynamic setting: the selected feature configuration can change *at runtime*, and the system has to adapt in real time. The main case-study is an Android application which allows automated profile management, reconfiguring a mobile device's operating profile based on environmental factors. This paper is covered fully in Chapter 8, together with its submitted extension [x1].

[4] Abstract Delta Modeling: My Research Plan

This is a PhD research plan submitted to the doctoral symposium colocated with SPLC 2012. It describes my plans for this thesis and was published in the digital proceedings of the conference. The thesis does not fulfill all of my earlier predictions, but I believe that in most of those instances, the result was a better reading experience.

[2] Abstract Delta Modeling (Journal Version)

This paper was written by Dave Clarke, Ina Schaefer and myself as an extended version of the ADM conference paper [1, 9], accepted to a special issue of MSCS. (We received notification of acceptance a long time ago, but the article has not yet been published as of this writing, because of a backlog.) This article subsumes the original work. Aside from a more detailed treatment of the formalism, its main addition was that of nested delta models, which are discussed in Sections 3.6 and 4.5.

[8] HATS Abstract Behavioral Specification: The Architectural View

This paper was written by Reiner Hähnle, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, Peter Y. H. Wong and myself and submitted as a HATS publication, published by Springer in 2013. It describes the Abstract Behavioral Specification (ABS) language from an architectural perspective, a perspective that includes delta modeling. My contribution to this article was an accounting of the Delta Modeling Workflow tailored to ABS, which is summarized in Chapter 7.

[10] The pkgloader and lt3graph Packages: Toward simple and powerful package management for LaTeX

I was invited to write this article by the editor of TUGboat, the Communications of the TeX Users Group, based on my work on the pkgloader LaTeX package. This package oversees the package loading process and uses delta modeling principles to address one of the major frustrations of LaTeX: package conflicts. The article introduces the pkgloader package, as well as lt3graph, a LaTeX3 library used by pkgloader to do most of the heavy lifting. This practical application of delta modeling is treated in Chapter 5.

1.5.2 Unpublished Work

The following have been written, but not yet accepted for publication.

[I1] An Operational Semantics for Dynamic Product Lines

This paper was written by me in 2013, and submitted to a SoSyM Special Issue on Integrated Formal Methods. It is loosely based on the first paper on dynamic delta modeling [6]. It takes a more formal, more general approach and subsumes the earlier work. The main new contribution is an operational semantics, which now formalizes the previously vague notion of ‘strategy’. The new approach incorporates unrestricted feature models as well as relational deltas, whereas the original work required that the feature model considers all features to be independent and all deltas to have a functional behavior. Finally, the description of the case-study has been greatly extended. It is on this article that Chapter 8 is based.

[I2] A Formal Software Product Line Development Workflow

This paper was written by me in 2013, but has not yet been submitted. It extends the first abstract paper on the Delta Modeling Workflow [5], and subsumes most of the theory from that paper. The main contribution absent from the earlier work is a focus on concurrent development and a proof that such concurrent development is possible without sacrificing correctness. To that end, an operational semantics is used to model the various implementation steps of the workflow. This theory is presented in Appendix A.

1.5.3 The Chapters

The distribution of the work over the chapters of the thesis is based on narrative merit, rather than a one-to-one correspondance. Pretty much every publication has influenced every chapter in some way, but there are some clear connections, which are mentioned in the summaries below.

Chapter 1: Introduction

What remains of the current chapter is a set of typographic conventions in Section 1.6 —recommended, if you plan on reading a significant portion of the other chapters— and some preliminary theory on discrete mathematics in Section 1.7, which is meant primarily as a reference.

Chapter 2: Algebraic Delta Modeling

This chapter introduces the basic building blocks of delta modeling: *products and deltas*, as well as their characteristics and interactions. Most of the theory comes from the original papers [1, 2], but it adopts refinements introduced in other papers. Notably, it introduces the semantics of deltas as *relational*, rather than functional, which was first done in the papers on delta logic [3] and dynamic delta modeling [6]. It also contains some unpublished material. In particular, a number of new *algebraic interpretations* are discussed in Section 2.6.

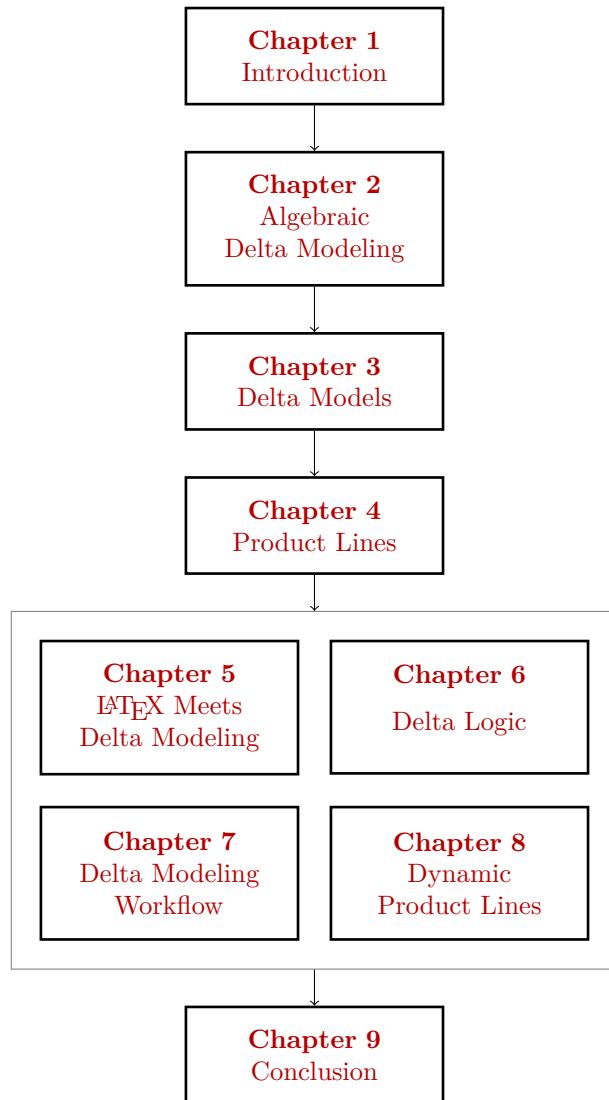


Figure 1.4: The suggested reading order of the thesis. Chapters 5 to 8 can basically be read in any order. Before that, however, each chapter builds upon the theory of its predecessor.

Chapter 3: Delta Models

This chapter introduces the *partially ordered structure* and *conflict resolution models* that first inspired the work on ADM. Again, most of this comes from the original ADM papers [1, 2], but includes adaptation to the relational semantic view. The chapter also introduces a distinction between several different types of delta model semantics, one of which —conjunctive semantics— is as of yet unpublished.

Chapter 4: Product Lines

This chapter introduces *features* and *ADM-based product lines* and is the final chapter based on the original work [1, 2]. The theory in this chapter has been influenced most by the work on the development workflow [5], which split up the description of a delta-based product line into a specification and an implementation, and introduced the concept of *parametric deltas*: deltas that behave differently for different feature configurations.

Chapter 5: L^AT_EX Meets Delta Modeling

This chapter demonstrates the L^AT_EX implementation of delta modeling by documenting two new L^AT_EX-packages. The `delta-modules` package defines deltas that can be used for the preparation of documents, and families of related documents. For example, this thesis was prepared using the `delta-modules` package, and different versions of the thesis are available that skip certain topics for a different reading experience.

The `pkgloader` package solves a long-existing problem with the L^AT_EX ecosystem: that of package management. Many document authors suffer from the fact that various L^AT_EX packages are mutually incompatible, and the fact that this is very poorly documented. The `pkgloader` package uses delta modeling principles to load third party packages in the proper order, apply code to resolve certain package conflicts or, as a last resort, provide the user with a clear error message.

Chapter 6: Delta Logic

This chapter introduces a modal logic for reasoning about products and deltas. It is almost fully based on [3], but some theory was already introduced in earlier chapters. It presents a multi-modal language and a number of proof techniques with accompanying soundness and completeness proofs. A possible new adaptation to *hybrid logics* is briefly discussed as well.

Chapter 7: Delta Modeling Workflow

This chapter introduces a recommended *workflow* for building delta-based product lines. It is almost fully based on the corresponding publications [5, 7, 8]. The theory is strengthened by a new formulation in terms of operational semantics based on an as of yet unpublished paper [2]. To improve readability, the operational semantics is not exposed in the chapter itself, but is instead presented in Appendix A.

Chapter 8: Dynamic Product Lines

This chapter introduces *dynamic delta modeling*: a basis for modeling product lines that can adapt at runtime to a dynamically changing feature configuration. It is based on the corresponding paper [6], though thoroughly reworked and extended since 2012. The new theory, based on an interplay between operational semantics and Mealy machines, has been submitted to a SoSyM special issue on integrated formal methods [1].

Chapter 9: Conclusion

This chapter summarizes the main contributions, goals and lessons of each chapter, and then presents a number of possible directions for future work.

1.6 Typographic Conventions

This section contains the various typographic conventions used throughout this thesis. Knowing them is not necessary for understanding the theory, but it can help the reader to spot certain constructs at a glance and to read and parse the text more efficiently.

1.6.1 Fonts

The following variations in font serve a special purpose:

- A Serif Roman typeface, apart from being used for the prose, is also used for the names of mathematical functions.
- *Italic type* is used to put emphasis on certain words or phrases, either in a linguistic sense (“*we* can represent product lines of *any* domain”) or to mark certain notions as important to the theory (“units called *deltas*”). It is also used for the names of mathematical variables and constants.
- A Sans Serif typeface is used for the names of mathematical predicates, relations and classes.
- *Caligraphic* or *Fraktur* typefaces are used in mathematical formulas for specific kinds of variables related to the theory of discourse, usually to maintain consistency with previous work.
- A Monospaced typeface is used for (fragments of) source-code. Inside source code, keywords and core language constructs are **bold**, values are *italic*, comments are */*gray*/* and types are *green*. The color contrast should still be high enough for the last two to be readable (though not distinguishable) when printed in grayscale.
- SMALL CAPS is used in Chapter 8 and Appendix A for the names of specific inference rules in the operational semantics. The definition of new inference rules sets the pace of the respective chapters and their names form convenient reference points.

1.6.2 Formal Concepts

The main chapters of this thesis present various theories. The definitions, assumptions and results that constitute the *formalisms* for those theories are placed in clearly delimited blocks that break up the running text. These concepts are numbered sequentially per chapter, and they come in the following flavors:

0.1. Definition: a formal definition ┘

0.2. Notation: a formal notation ┘

0.3. Action: a formal action that may be taken by a developer ┘

0.4. Example: an example of a formal concept ┘

0.5. Axiom: a formal restriction on an existing definition ┘

0.6. Theorem: an important formal result □

▷ **0.7. Lemma:** an intermediate or smaller formal result □

▶ **0.8. Corollary:** a formal result that readily follows from a previous result □ ♣

In the left margin of each of these types of blocks you may find one of two symbols. A white triangle (▷) indicates that the concept belongs to one of the concrete examples or case studies of the thesis, rather than the overall abstract formalism. A black triangle (▶) indicates concepts of a greater scope. Blocks without either symbol are local in nature and could be skipped without missing too much in the long run. But a black triangle indicates that later sections or chapters will refer back to the concept in question.

Not all formal results will be accompanied by a proof. A lot of the required proofs are conceptually quite simple, but long, tedious and generally uninteresting to read. A proof is included only if reading it would provide valuable insight into the theory. For some results about the running example, a mechanically checked proof has been written with the Coq proof assistant [36]. Those results show the Coq logo (♣) in the right margin. This gives the reader some confidence in the result without requiring them to plow through quadruply nested case distinctions.

1.6.3 PDF Features

When viewing the PDF file of this thesis on a computer monitor, you have access to several extra features. First, there are clickable hyperlinks between sections, formal environments, bibliography references and more. Second, the logo next to the results checked with the Coq proof assistant can be clicked to download an embedded copy of the Coq proofs. This will only work for certain PDF viewers, such as Adobe Reader.

1.7 Mathematical Preliminaries

This section establishes the basic mathematical notations and definitions that are used in this thesis. We assume that the reader already has an intuitive grasp of the concepts in this section, as the treatment will be dense. A basic grounding in the topics of Sections 1.7.1 to 1.7.7 can be gained from any introductory textbook on discrete mathematics [123]. As for the more specialized material of Sections 1.7.8, 1.7.9, 1.7.10 and 1.7.11, there are some great introductions on abstract algebra [179], modal logic [42] and operational semantics [153] to be found. I found these cited sources well written and accessible.

1.7.1 Sets

A *set* is an unordered collection of *elements*—finite or infinite—which does not contain the same element more than once. For this thesis, an understanding of *naive set theory* [83] is sufficient.

- **1.1. Definition (Basic Set Concepts):** The basic set notations are as follows, for any set S , elements e, e_1, \dots, e_n , predicate P , relation R and function f :

$\{e_1, \dots, e_n\}$	the finite set containing only the elements e_1, \dots, e_n
$\{f(e) \mid P(e)\}$	the set of all elements $f(e)$ such that e satisfies P
$\{e R g \mid P(e)\}$	an abbreviation for $\{e \mid e R g \wedge P(e)\}$
$e \in S$	e is a member of S
$ S $	the cardinality of S

The following relations and operations are defined for all sets S, T :

$S \subseteq T$	$\stackrel{\text{def}}{\iff} \forall e \in S: e \in T$	$S \setminus T$	$\stackrel{\text{def}}{=} \{e \in S \mid e \notin T\}$
$S \subset T$	$\stackrel{\text{def}}{\iff} S \subseteq T \wedge S \neq T$	$S \ominus T$	$\stackrel{\text{def}}{=} (S \cup T) \setminus (S \cap T)$
$S \cup T$	$\stackrel{\text{def}}{=} \{e \mid e \in S \vee e \in T\}$	$\text{Pow}(S)$	$\stackrel{\text{def}}{=} \{S' \mid S' \subseteq S\}$
$S \cap T$	$\stackrel{\text{def}}{=} \{e \mid e \in S \wedge e \in T\}$	S^c	$\stackrel{\text{def}}{=} U \setminus S$

A universal set U is assumed to be clear from context when the S^c notation is used. ┘

- **1.2. Definition (n -ary Set Operations):** The n -ary set operations \bigcup and \bigcap are defined as follows, for all sets S , properties P and functions f :

$\bigcup S$	$\stackrel{\text{def}}{=} \{e \mid \exists s \in S: e \in s\}$	$\bigcup_{P(g)} f(g)$	$\stackrel{\text{def}}{=} \bigcup \{f(g) \mid P(g)\}$
$\bigcap S$	$\stackrel{\text{def}}{=} \{e \mid \forall s \in S: e \in s\}$	$\bigcap_{P(g)} f(g)$	$\stackrel{\text{def}}{=} \bigcap \{f(g) \mid P(g)\}$

where g is a fresh variable name. ┘

- **1.3. Notation (Important Sets):** The following specific sets are used often:

\emptyset	the empty set
\mathbb{N}^+	the positive natural numbers $1, 2, 3, \dots$
\mathbb{N}	the natural numbers $0, 1, 2, \dots$
\mathbb{Z}	the integers $\dots, -2, -1, 0, 1, 2, \dots$
\mathbb{R}	the real numbers

Each is a strict subset of the ones that follow. ┘

1.7.2 Tuples

To represent ordered collections of elements, we use *tuples*.

- **1.4. Notation (Tuples):** A *tuple* can be given directly as a comma-separated list of elements between parentheses (e, g, h) , or sometimes angle brackets $\langle e, g, h \rangle$. The order between the elements is significant. ┘

Tuples with 2, 3, 4, 5 and n elements are respectively called *pairs*, *triples*, *quadruples*, *quintuples* and *n-tuples*.

- **1.5. Definition (Cartesian Product):** The *Cartesian product* operation \times on two sets S, T is a set of pairs defined as follows:

$$S \times T \stackrel{\text{def}}{=} \{ (e, g) \mid e \in S \wedge g \in T \} \quad \text{┘}$$

- **1.6. Definition (n -ary Cartesian Product):** For some number $n \in \mathbb{N}^+$, the *n-ary Cartesian product* on sets S_1, \dots, S_n is a set of n -tuples defined as follows:

$$S_1 \times \dots \times S_n \stackrel{\text{def}}{=} \{ (e_1, \dots, e_n) \mid e_1 \in S_1 \wedge \dots \wedge e_n \in S_n \} \quad \text{┘}$$

- **1.7. Definition (Cartesian Power):** For any number $n \in \mathbb{N}^+$, the *n-th Cartesian power* of a set S is defined as follows:

$$S^n \stackrel{\text{def}}{=} \underbrace{S \times \dots \times S}_n \quad \text{┘}$$

1.7.3 Sequences

Sequences are basically tuples, but always contain elements of the same type and are usually of unspecified (but finite) length.

- **1.8. Definition (Sequences):** Given a set S , the set of all finite *sequences* of elements from S is defined as follows:

$$S^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} S^n \quad \text{┘}$$

- **1.9. Definition (Sequence Concatenation):** The *concatenation* of two sequences $\bar{e} = (e_1, \dots, e_n)$ and $\bar{g} = (g_1, \dots, g_m)$ with $n, m \in \mathbb{N}$ is defined as follows:

$$\bar{e} \frown \bar{g} \stackrel{\text{def}}{=} (e_1, \dots, e_n, g_1, \dots, g_m)$$

A sequence on either side with only one element may be abbreviated by omitting the parentheses, e.g., $e \frown (g_1, g_2)$ instead of $(e) \frown (g_1, g_2)$. ┘

1.7.4 Relations

- **1.10. Definition (Relation):** Given $n \in \mathbb{N}^+$, an n -ary relation R over the sets S_1, \dots, S_n is a subset of their Cartesian product: $R \subseteq S_1 \times \dots \times S_n$. A unary relation is also called a *predicate*. \lrcorner

- **1.11. Definition (Relation Operations):** Given two binary relations $R \subseteq S \times T$ and $Q \subseteq T \times U$ we define the following operations:

$$\begin{aligned} Q \circ R &\stackrel{\text{def}}{=} \{ (e, h) \mid \exists g \in T: (e, g) \in R \wedge (g, h) \in Q \} \\ R^{-1} &\stackrel{\text{def}}{=} \{ (g, e) \mid (e, g) \in R \} \\ \text{id}_S &\stackrel{\text{def}}{=} \{ (e, e) \mid e \in S \} \end{aligned}$$

with $Q \circ R \subseteq S \times U$ and $R^{-1} \subseteq T \times S$ and $\text{id}_S \subseteq S^2$. Note that relation composition \circ should be read from right to left. \lrcorner

- **1.12. Notation:** For all sets S, S' , elements $e, g, h \in S$ and predicates $P \subseteq S$:

$$P(e) \stackrel{\text{def}}{\iff} e \in P \qquad P(S') \stackrel{\text{def}}{\iff} S' \subseteq P$$

Additionally, for all binary relations $R, Q \subseteq S \times T$:

$$\begin{aligned} e R g &\stackrel{\text{def}}{\iff} (e, g) \in R & R(e) &\stackrel{\text{def}}{=} \{ g \mid e R g \} \\ e \not R g &\stackrel{\text{def}}{\iff} \neg(e R g) & R(S') &\stackrel{\text{def}}{=} \bigcup_{e \in S'} R(e) \\ e \mathcal{R} g &\stackrel{\text{def}}{\iff} g R e & \text{img}(R) &\stackrel{\text{def}}{=} R(S) \\ e R g \ Q h &\stackrel{\text{def}}{\iff} e R g \wedge g Q h & \text{pre}(R) &\stackrel{\text{def}}{=} R^{-1}(S) \\ e, g R h &\stackrel{\text{def}}{\iff} e R h \wedge g R h & e \not R &\stackrel{\text{def}}{\iff} R(e) = \emptyset \end{aligned}$$

$R(e)$ is called the *image* of e in R and $R^{-1}(g)$ is called the *preimage* of g in R . $\text{img}(R)$ and $\text{pre}(R)$ are called the image and preimage of the relation R itself.

The notational conventions regarding $\not R$ and \mathcal{R} often hold in literature, but are not often formalized. In this thesis they always hold; as does the implied convention that binary relation symbols with a horizontal symmetry, such as $=$, \equiv and \iff , are also symmetric in a relational sense (Definition 1.13). \lrcorner

- **1.13. Definition (Binary Relation Properties):** A binary relation $R \subseteq S \times T$ may have the following named properties:

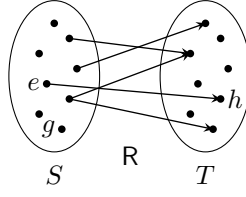


Figure 1.5: The visualization of a specific relation $R \subseteq S \times T$. The two groups of nodes represent the domain S and codomain T of the relation with $e, g \in S$ and $h \in T$. The arrows represent the relation R itself. They show, for example, that $e R h$ and that $g \not R h$.

uniquely defined:	$\forall e, g, h \in S:$	$e R g \wedge e R h \Rightarrow g = h$
fully defined:		$\text{pre}(R) = S$
well defined:		uniquely and fully defined
injective:	$\forall e, g, h \in S:$	$e R h \wedge g R h \Rightarrow e = g$
surjective:		$\text{img}(R) = T$
bijective:		surjective and injective
one-to-one:		uniquely defined and injective

Uniquely defined and fully defined are often called ‘functional’ and ‘total’. In the case that $S = T$ we also distinguish the following:

reflexive:	$\forall e \in S:$	$e R e$
symmetric:	$\forall e, g \in S:$	$e R g \Rightarrow e \mathcal{R} g$
transitive:	$\forall e, g, h \in S:$	$e R g \wedge g R h \Rightarrow e R h$
irreflexive:	$\forall e \in S:$	$e \not R e$
antisymmetric:	$\forall e, g \in S:$	$e R g \wedge e \mathcal{R} g \Rightarrow e = g$
asymmetric:	$\forall e, g \in S:$	$e R g \Rightarrow e \not \mathcal{R} g$
total:	$\forall e, g \in S:$	$e R g \vee e \mathcal{R} g$
discrete:	$\forall e, g \in S:$	$e \not R g \wedge e \not \mathcal{R} g$

- **1.14. Definition (Transitive Closure):** Given a binary relation $R \subseteq S \times S$, define its *transitive closure* $R^+ \subseteq S \times S$ and *reflexive transitive closure* $R^* \subseteq S \times S$ as follows for all elements $e, g \in S$:

$$\begin{aligned}
 e R^+ g &\stackrel{\text{def}}{\iff} e R g \vee \exists h \in S: e R h R^+ g \\
 e R^* g &\stackrel{\text{def}}{\iff} e = g \vee e R^+ g
 \end{aligned}$$

- **1.15. Notation (Inference Rule):** A specific relation or set of relations is sometimes defined as the smallest relation(s) satisfying a particular set of *inference rules*. An inference rule consists of a *conclusion* and a set of *premises*, separated by a horizontal line:

$$\frac{\langle \text{premise 1} \rangle \quad \langle \text{premise 2} \rangle \quad \dots}{\langle \text{conclusion} \rangle}$$

Free variables present in an inference rule can be consistently replaced by any concrete value, i.e., they are under an implicit universal quantification. \lrcorner

Relations are important in this thesis, particularly because the semantics of deltas is expressed by relations. Relation diagrams such as the one in Figure 1.5 are occasionally used to illustrate relational concepts.

1.7.5 Functions

- **1.16. Definition (Function):** A *function* from S into T is a well defined relation over the sets S, T , which are respectively called its *domain* and *codomain*.

$S \rightarrow T$ denotes the set of all functions from S into T . To declare a function f of that type, write $f: S \rightarrow T$. When a function is declared this way, and we have $(e, g) \in f$, we say that $f(e) = g$. This overrides Notation 1.12 (as we have $f(e) \in T$ rather than $f(e) \subseteq T$). \lrcorner

- **1.17. Definition (Partial Function):** A uniquely defined relation over S, T that is not necessarily fully defined is called a *partial function* from S into T .

The set of all partial functions from S into T is denoted $S \rightharpoonup T$. To declare a partial function f of that type we write $f: S \rightharpoonup T$. When $e \notin \text{pre}(f)$ we say that $f(e)$ is *undefined*, or that $f(e) = \perp$ (assuming that $\perp \notin T$). Otherwise, the notation is the same as for functions. \lrcorner

- **1.18. Notation:** A pair $(e, g) \in f$ in some partial function f , indicating that e maps to g , can also be denoted $e \mapsto g$. \lrcorner

- **1.19. Definition (Function Update):** Given a function $f: S \rightarrow T$ and elements $e \in S$ and $g \in T$, define the *updated function* $f[e \mapsto g]$ as follows, for all elements $e' \in S$:

$$f[e \mapsto g](e') \stackrel{\text{def}}{=} \begin{cases} g & \text{if } e = e' \\ f(e') & \text{otherwise} \end{cases}$$

The same notation is defined for partial functions, in which case the update $f[e \mapsto \perp]$ is also available. \lrcorner

1.7.6 Transitive Relations

- **1.20. Definition (Preorder):** A *preorder* is a transitive and reflexive relation. \lrcorner

- **1.21. Definition (Equivalence Relation):** An *equivalence relation* is a transitive and symmetric relation (i.e., a symmetric preorder) denoted $=, \approx, \equiv, \Leftrightarrow, \dots$ \lrcorner

- **1.22. Definition (Orders):** An *order* is a transitive and antisymmetric relation that is either:

- reflexive, in which case it is called *inclusive* and denoted $\leq, \preceq, \subseteq, \sqsubseteq, \dots$
- or irreflexive, in which case it is called *strict* and denoted $<, \prec, \subset, \sqsubset, \dots$

Orders are generally called *partial orders*, to emphasize the fact that they are not necessarily total (Definition 1.13). \lrcorner

- **1.23. Definition (Minimal and Maximal Elements):** Given an order \prec on a set S , an element $e \in S$ is a *minimal element* in \prec iff there exists no $g \in S$ such that $g \prec e$ and a *maximal element* in \prec iff there exists no g such that $e \prec g$. \lrcorner

- **1.24. Definition (Extension):** Given two orders \sqsubseteq and \preceq on the same set S , call \sqsubseteq an *extension* of \preceq iff $\preceq \subseteq \sqsubseteq$. If \sqsubseteq is total, call it a *linear extension*. \lrcorner

1.7.7 Quotient Sets

- **1.25. Definition (Equivalence Class):** Given a set S , an element $e \in S$ and an equivalence relation $\sim \subseteq S \times S$, the \sim -*equivalence class* of e is defined as follows:

$$[e]_{\sim} \stackrel{\text{def}}{=} \{g \in S \mid e \sim g\} \quad \lrcorner$$

- **1.26. Definition (Quotient Set):** The *quotient set* of a set S by an equivalence relation $\sim \subseteq S \times S$ is defined as follows:

$$S/\sim \stackrel{\text{def}}{=} \{[e]_{\sim} \mid e \in S\} \quad \lrcorner$$

- **1.27. Notation (Implicit Canonical Projection):** When we work with a quotient set S/\sim and the equivalence relation \sim is clear from context, we can choose to omit it. We then treat S as an abbreviation for S/\sim and e as an abbreviation for $[e]_{\sim}$ — in effect, turning equivalence into equality. \lrcorner

1.7.8 Operations

- **1.28. Definition (Operation):** Given some set S , an n -*ary operation* (or *operator*) on S is a function from S^n into S . $n \in \mathbb{N}$ is called the *arity* of the operation. As a special exception, a 0-ary operation is an element of S , also called a *constant*. \lrcorner

- **1.29. Definition (Binary Operation Properties):** There are a number of important properties a binary operation $\star: S \times S \rightarrow S$ may have. The following are the most common. For all elements $e, g, h \in S$:

$$\begin{aligned} \star \text{ is associative:} \quad & e \star (g \star h) = (e \star g) \star h \\ \star \text{ is commutative:} \quad & (e \star g) = (g \star e) \\ \star \text{ is idempotent:} \quad & e \star e = e \end{aligned}$$

With regard to another binary operation $\curlywedge: S \times S \rightarrow S$ we may have:

$$\curlywedge \text{ distributes over } \star: \quad e \curlywedge (g \star h) = (e \curlywedge g) \star (e \curlywedge h)$$

With regard to a specific element $\epsilon \in S$ we may have:

$$\begin{aligned} \epsilon \text{ is an identity for } \star: \quad & \epsilon \star e = e = e \star \epsilon \\ \epsilon \text{ absorbs } \star: \quad & \epsilon \star e = \epsilon = e \star \epsilon \end{aligned} \quad \lrcorner$$

1.7.9 Algebraic Structures

The following are some concepts regarding abstract algebra required for Chapters 2 and 3. The definitions that follow are quite limited compared to those in existing literature [98], but they are sufficient for this thesis.

- **1.30. Definition (Algebraic Structure):** An *algebraic structure* or *algebra* consists of a tuple $(S, \star_1, \dots, \star_n)$ —where S is a *carrier set* and $\{\star_1, \dots, \star_n\}$ is a set of nullary, unary and binary operators on S . The tuple is also called an *algebraic signature*. \lrcorner
- **1.31. Notation:** We can write S to refer to an algebraic structure $(S, \star_1, \dots, \star_n)$ and vice versa, if this does not introduce ambiguity. \lrcorner

When the carrier of an algebraic structure is a quotient set, but its operations are defined in terms of the original set, some conditions are imposed:

- **1.32. Definition (Quotient Algebra):** Given any algebraic structure with carrier set S , nullary operators $x \in S$, unary operators $^\circ: S \rightarrow S$ and binary operators $\star: S \times S \rightarrow S$, the corresponding *quotient algebra* by equivalence relation $\sim \subseteq S \times S$ would be the algebra with carrier set S/\sim and corresponding operators $[x]_\sim \in (S/\sim)$, $^\circ_\sim: (S/\sim) \rightarrow (S/\sim)$ and $\star_\sim: (S/\sim) \times (S/\sim) \rightarrow (S/\sim)$, which must satisfy the following for all $e, g \in S$:

$$\begin{aligned} [e]_\sim^{\circ_\sim} &= [e^\circ]_\sim \\ [e]_\sim \star_\sim [g]_\sim &= [e \star g]_\sim \end{aligned} \quad \lrcorner$$

This allows us to extend implicit canonical projection (Notation 1.27) to the operators of the algebra and use x , $^\circ$ and \star as abbreviations for $[x]_\sim$, $^\circ_\sim$ and \star_\sim .

Here follow three common algebraic structures:

- **1.33. Definition (Monoid):** A *monoid* is a tuple (S, \cdot, ε) where $\cdot: S \times S \rightarrow S$ is a composition operator and $\varepsilon \in S$ is a neutral element, satisfying the following axioms for all elements $e, g, h \in S$:

- a. associativity: $(e \cdot g) \cdot h = e \cdot (g \cdot h)$
- b. identity element ε : $\varepsilon \cdot e = e = e \cdot \varepsilon$ \lrcorner

- **1.34. Definition (Boolean Algebra):** A *Boolean algebra* is a tuple $(S, \sqcup, \sqcap, \neg, \perp, \top)$ where $\sqcup: S \times S \rightarrow S$ is a disjunction operator, $\sqcap: S \times S \rightarrow S$ is a conjunction operator, $\neg: S \rightarrow S$ is a negation operator, $\perp \in S$ is an empty element and $\top \in S$ is a complete element. For all elements $e, g, h \in S$ it satisfies the following:

- a. associativity: $(e \sqcup g) \sqcup h = e \sqcup (g \sqcup h)$
- b. commutativity: $e \sqcup g = g \sqcup e$
- c. identity: $e \sqcup \perp = e$
- d. distributivity: $e \sqcup (g \sqcap h) = (e \sqcup g) \sqcap (e \sqcup h)$
- e. complement: $e \sqcup e^- = \top$

It also satisfies the *dual* axioms **a'**, **b'**, **c'**, **d'** and **e'** formed by taking an original axiom and exchanging \sqcup with \sqcap and \perp with \top . \lrcorner

- **1.35. Definition (Relation Algebra):** A *relation algebra* $(S, \sqcup, \sqcap, \neg, \perp, \top, \cdot, \cdot^-, \cdot^\circ)$ is a tuple with a monoid (S, \cdot, ε) , a Boolean algebra $(S, \sqcup, \sqcap, \neg, \perp, \top)$ and a converse operator $\cdot^-: S \rightarrow S$. In addition to the axioms from Definitions 1.33 and 1.34, it also satisfies the following for all $e, g, h \in S$:

- a. \sim is its own inverse $e^{\sim\sim} = e$
- b. \sim is an anti-involution for \cdot $(e \cdot g)^{\sim} = g^{\sim} \cdot e^{\sim}$
- c. \cdot distributes over \sqcup $e \cdot (g \sqcup h) = (e \cdot g) \sqcup (e \cdot h)$
- d. \sim distributes over \sqcup $(e \sqcup g)^{\sim} = e^{\sim} \sqcup g^{\sim}$
- e. Tarski's axiom $(e^{\sim} \cdot (e \cdot g)^{-}) \sqcup g^{-} = g^{-}$ \lrcorner

1.7.10 Modal Logic

This section provides a terse introduction to the theory of modal logic [40]. It is required knowledge for Chapter 6, where a new modal logic is presented.

- **1.36. Definition (Multimodal Language):** Given a set of *modal labels* M and a set of *propositional variables* $PROP$, a *multimodal language* Ψ is defined with the following grammar:

$$\Psi \ni \varphi ::= \top \mid k \mid \neg\varphi \mid \varphi \vee \psi \mid \langle m \rangle \varphi$$

where $k \in PROP$ is a propositional variable and $\langle m \rangle$ is a *modality* based on a modal label $m \in M$. The following formulas are *abbreviations*, so formally we need only be concerned with the grammar above. For all formulas $\varphi, \psi \in \Psi$:

$$\begin{aligned} \perp &\stackrel{\text{def}}{=} \neg\top \\ [m]\varphi &\stackrel{\text{def}}{=} \neg\langle m \rangle \neg\varphi \\ \varphi \wedge \psi &\stackrel{\text{def}}{=} \neg(\neg\varphi \vee \neg\psi) \\ \varphi \rightarrow \psi &\stackrel{\text{def}}{=} \neg\varphi \vee \psi \\ \varphi \leftrightarrow \psi &\stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \end{aligned}$$

To resolve ambiguity we assume the traditional set of precedence rules (e.g. \wedge binds stronger than \vee) and allow parentheses to override those rules. \lrcorner

- **1.37. Definition (Kripke Frame):** A *Kripke frame* is a tuple (W, M, R) , often denoted by \mathfrak{F} , with a set of *worlds* W , a set of modal labels M and a function $R: M \rightarrow \text{Pow}(W \times W)$ which maps each modal label $m \in M$ to a corresponding *accessability relation* $R_m \subseteq W \times W$. \lrcorner
- **1.38. Definition (Valuation Function):** Given a set of worlds W and a set of propositional variables $PROP$, a *valuation function* is a function $V: PROP \rightarrow \text{Pow}(W)$ which maps each propositional variable to the set of worlds in which it is true. \lrcorner
- **1.39. Definition (Kripke Model):** Given a set of propositional variables $PROP$, a *Kripke model* is a tuple (W, M, R, V) , often denoted by \mathfrak{M} , where (W, M, R) is a Kripke frame and $V: PROP \rightarrow \text{Pow}(W)$ is a valuation function. \lrcorner

The truth or falsehood of a formula is determined by the forcing relation:

- **1.40. Definition (Forcing Relation):** Given a modal language Ψ , a Kripke model $\mathfrak{M} = (W, M, R, V)$ and a world $w \in W$, the *forcing relation* $\Vdash \subseteq W \times \Psi$ is defined by induction on the shape of the formula:

$$\begin{aligned}
w \Vdash \top & \stackrel{\text{def}}{\iff} \text{always} \\
w \Vdash k & \stackrel{\text{def}}{\iff} w \in V(k) \\
w \Vdash \varphi \vee \psi & \stackrel{\text{def}}{\iff} (w \Vdash \varphi) \vee (w \Vdash \psi) \\
w \Vdash \neg \varphi & \stackrel{\text{def}}{\iff} \neg(w \Vdash \varphi) \\
w \Vdash \langle m \rangle \varphi & \stackrel{\text{def}}{\iff} \exists w' \in W: (w, w') \in R(m) \wedge (w' \Vdash \varphi)
\end{aligned}$$

When $w \Vdash \varphi$, we say that φ is *true for w*. If the Kripke model is not clear from context, we attach a subscript as in $\Vdash_{\mathfrak{M}}$. \lrcorner

We also define the following extensions for this notation:

- **1.41. Notation:** Given a modal language Ψ , for every Kripke model \mathfrak{M} , Kripke frame \mathfrak{F} , world $w \in W$ and formula $\varphi \in \Psi$ define:

$$\begin{aligned}
\Vdash_{\mathfrak{M}} \varphi & \stackrel{\text{def}}{\iff} \forall w' \in W: w' \Vdash_{\mathfrak{M}} \varphi \\
w \Vdash_{\mathfrak{F}} \varphi & \stackrel{\text{def}}{\iff} \forall V: PROP \rightarrow \text{Pow}(W): w \Vdash_{(\mathfrak{F}, V)} \varphi \\
\Vdash_{\mathfrak{F}} \varphi & \stackrel{\text{def}}{\iff} \forall w' \in W: w' \Vdash_{\mathfrak{F}} \varphi
\end{aligned}$$

denoting respectively that φ is *globally true in \mathfrak{M}* , *valid for p* and *valid on \mathfrak{F}* . \lrcorner

The forcing relation \Vdash is overloaded for when a formula is semantically entailed by a set of premises (also formulas):

- **1.42. Definition (Local/Global Consequence):** Given a set of formulas Γ and a class of Kripke models \mathcal{S} , we say that $\varphi \in \Psi$ is a *local consequence* or *global consequence* of Γ iff:

$$\begin{aligned}
\Gamma \Vdash_{\mathcal{S}} \varphi & \stackrel{\text{def}}{\iff} \forall \mathfrak{M} \in \mathcal{S}: \forall w \in W: (\mathfrak{M}, w \Vdash \Gamma) \implies (\mathfrak{M}, w \Vdash \varphi) \\
\Gamma \Vdash_{\mathcal{S}}^g \varphi & \stackrel{\text{def}}{\iff} \forall \mathfrak{M} \in \mathcal{S}: (\forall w \in W: \mathfrak{M}, w \Vdash \Gamma) \implies (\forall w \in W: \mathfrak{M}, w \Vdash \varphi) \lrcorner
\end{aligned}$$

A *normal modal logic* is defined by a set of axioms and closure rules:

- **1.43. Definition (Normal Modal Logic):** Given a modal language Ψ based on a set of modal labels M and a set of propositional variables $PROP$, a *normal modal logic* is a set of formulas $\Gamma \subseteq \Psi$ which, for all formulas $\varphi, \psi \in \Psi$ and modal labels $m \in M$, contains at least the following:

- *all propositional tautologies* χ : $\chi \in \Gamma$
- *the formula K*: $[m](\varphi \rightarrow \psi) \rightarrow ([m]\varphi \rightarrow [m]\psi) \in \Gamma$
- *the formula Dual*: $\langle m \rangle \varphi \leftrightarrow \neg[m]\neg\varphi \in \Gamma$

and is closed by the following properties:

- *modus ponens*: $\varphi, \varphi \rightarrow \psi \in \Gamma \implies \psi \in \Gamma$
- *generalization*: $\varphi \in \Gamma \implies [m]\varphi \in \Gamma$
- *uniform substitution*: $\forall k \in PROP: \varphi \in \Gamma \implies \varphi[\psi/k] \in \Gamma$

Given any set of formulas Γ , a smallest normal modal logic containing all formulas in Γ always exists. It is called the modal logic *generated by Γ* . \lrcorner

Normal modal logics can be used as proof-systems. Their axioms consist of all propositional tautologies, the formulas **K**, **Dual**, and the other axioms of the logic in question. Their proof rules consist of the three closure properties: modus ponens, generalization and uniform substitution.

- **1.44. Definition (Provability):** Given a modal language Ψ and normal modal logic $\Lambda \subseteq \Psi$, the *provability relation* $\vdash_\Lambda \subseteq \text{Pow}(\Psi \times \Psi)$ is defined as follows, for all sets of formulas $\Gamma \subseteq \Psi$:

$$\Gamma \vdash_\Lambda \varphi \quad \stackrel{\text{def}}{\iff} \quad \exists \psi_1, \dots, \psi_n \in \Gamma: \left(\bigwedge_{1 \leq i \leq n} \psi_i \right) \rightarrow \varphi \in \Lambda$$

A shorthand notation is defined for provability without premises:

$$\vdash_\Lambda \varphi \quad \stackrel{\text{def}}{\iff} \quad \emptyset \vdash_\Lambda \varphi \quad \lrcorner$$

- **1.45. Definition (Soundness):** Given a modal language Ψ , a normal modal logic $\Lambda \subseteq \Psi$ is called *sound* with respect to a class of frames \mathcal{S} , if for any set of formulas $\Gamma \subseteq \Psi$ and any formula $\varphi \in \Psi$, we have:

$$\Gamma \vdash_\Lambda \psi \quad \implies \quad \Gamma \Vdash_{\mathcal{S}} \varphi \quad \lrcorner$$

- **1.46. Definition (Strong Completeness):** Given a modal language Ψ , a normal modal logic $\Lambda \subseteq \Psi$ is called *strongly complete* with respect to a class of frames \mathcal{S} , if for any set of formulas $\Gamma \subseteq \Psi$ and any formula $\varphi \in \Psi$, we have:

$$\Gamma \Vdash_{\mathcal{S}} \varphi \quad \implies \quad \Gamma \vdash_\Lambda \psi \quad \lrcorner$$

- **1.47. Theorem:** The normal modal logic **K**, generated by the empty set, is sound and strongly complete with respect to the class of all frames \mathcal{F} .

Proof: See [42]. □

1.7.11 Operational Semantics

An operational semantics [88, 153, 154] describes the progress of a dynamic system by defining a *transition relation* \longrightarrow over a set of *configurations* which model possible states of the system:

$$\langle cn_1 \rangle \longrightarrow \langle cn_2 \rangle \longrightarrow \langle cn_3 \rangle \longrightarrow \langle cn_4 \rangle \longrightarrow \dots$$

This allows us to both visualize a system moving from state to state and to reason about whether it could reach certain desirable or undesirable states. Operational semantics are used in Chapter 8 and Appendix A .

- **1.48. Notation (Configuration Space):** A *configuration space* CS is a set of *configurations* cn which represent the states of a dynamic system. Specific configurations are often typeset inside angle brackets; $\langle cn \rangle$. \lrcorner

- **1.49. Notation (Transition Relation):** Given a configuration space CS , a *transition relation* is a binary relation $\rightarrow \subseteq CS \times CS$ where

$$\langle cn \rangle \rightarrow \langle cn' \rangle$$

means that cn' is a possible *next configuration* of cn . We use \rightarrow^+ and \rightarrow^* for, respectively, its transitive closure and *reflexive transitive closure* (Definition 1.14). In a nondeterministic system there may be more than one next configuration. If a configuration cn has no next configurations, we say that it is *stuck*, denoted $\langle cn \rangle \nrightarrow$ (Notation 1.12). \lrcorner

- **1.50. Definition (Infinite Transition Path):** Given a configuration space CS , a transition relation $\rightarrow \subseteq CS \times CS$ and a configuration $cn \in CS$, say there is *no infinite transition path from cn* iff the following holds:

$$\langle cn \rangle \nrightarrow^\infty \quad \stackrel{\text{def}}{\iff} \quad \forall cn' \in CS: \quad \langle cn \rangle \rightarrow \langle cn' \rangle \implies \langle cn' \rangle \nrightarrow^\infty \quad \lrcorner$$

The nature of the configuration space determines what kinds of properties we can express about a system. For example, the original purpose of operational semantics [153] was to describe the execution of imperative source-code, using configurations $\langle st, \sigma \rangle$ containing a next statement st and a state σ , mapping variables to their current value.

Transition relations are defined using inference rules (Notation 1.15):

- 1.51. Example (Inference Rule):** The following two classical inference rules describe the semantics of a while loop in an imperative programming language:

$$\frac{\text{eval}(B, \sigma) = \text{true}}{\langle \text{while } B \text{ do } st \text{ od}, \sigma \rangle \rightarrow \langle st; \text{while } B \text{ do } st \text{ od}, \sigma \rangle}$$

$$\frac{\text{eval}(B, \sigma) = \text{false}}{\langle \text{while } B \text{ do } st \text{ od}, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle} \quad \lrcorner$$

The specifics of this particular example are not important. The operational semantics presented in Chapter 8 and Appendix A use fundamentally different configuration spaces, as they describe very different kinds of systems.

1.7.12 Mealy Machines

A Mealy machine is a finite-state machine with an input symbol and an output symbol on each transition [131]. In other words, given a current state and some input, the system receive some output and can go to a next state. Mealy machines are used in Chapter 8.

- **1.52. Definition (Mealy Machine):** A *Mealy machine* is a 5-tuple $(S, \Sigma, \Lambda, T, O)$.

S is a set of states, Σ is an input alphabet and Λ is an output alphabet. Given a state and input symbol, the transition function $T: S \times \Sigma \rightarrow S$ returns the next state and the output function $O: S \times \Sigma \rightarrow \Lambda$ returns the corresponding output symbol. The functions O and T are defined for the same inputs, i.e., $\text{pre}(O) = \text{pre}(T)$. A ‘current’ state $s \in S$ and input symbol $i \in \Sigma$ together constitute a transition in the machine iff $(s, i) \in \text{pre}(T)$. \lrcorner

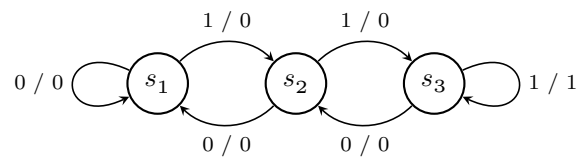
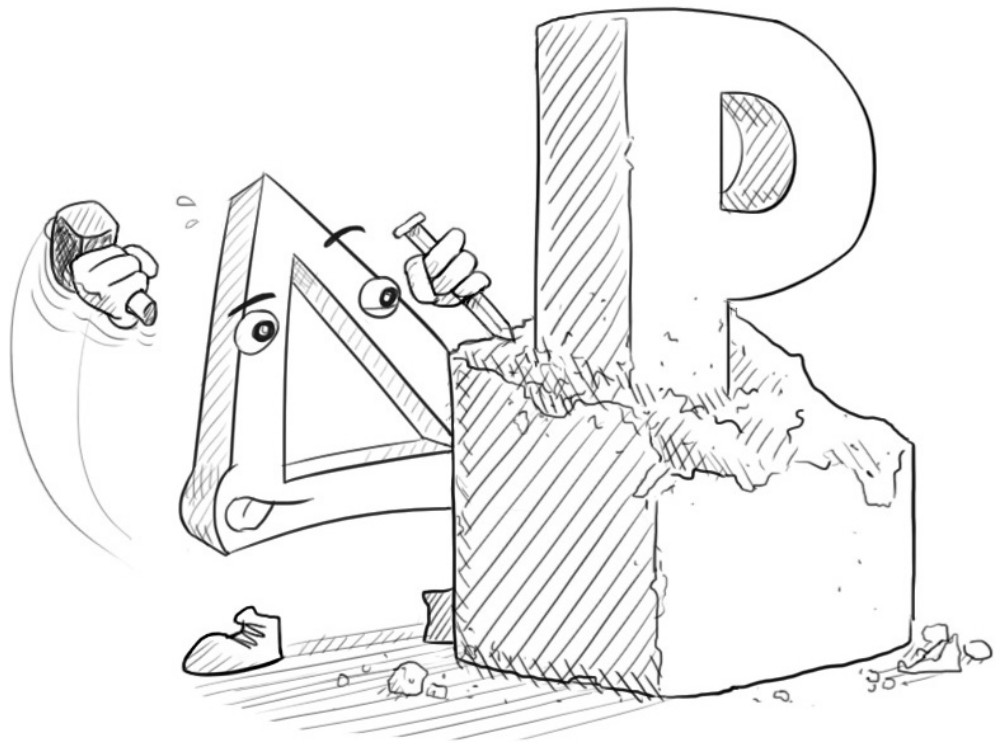


Figure 1.6: An example of a simple Mealy machine with $\Sigma = \Lambda = \{0, 1\}$. Each node represents a state. Each arrow is annotated with i/o : an input and an output symbol.

This definition differs from most formulations in two ways. First, the transition and output functions are *partial*. That means that we can have states that do not accept all input symbols. Secondly, the machines do not have an initial state. For our purposes in Chapter 8 an initial state will not have much meaning, so we choose to omit it. See Figure 1.6 for a small visual example.

Algebraic Delta Modeling

On the Theory of Incremental Product Modification



2.1 Introduction

This chapter introduces the basic building blocks of delta modeling —products and deltas— and lays much of the formal foundation for this thesis.

A *product* represents the kind of artefact we want to manufacture. In practice, it will be built up out of many smaller artefacts. For example: packages, classes, methods and fields, in an object oriented programming language, together forming a program. The problem is that the artefacts in such a product almost never map directly to the higher level concept of *feature*. Indeed, a feature can relate to many classes, and a class can relate to many features.

To understand how *deltas* can help in this regard, let us examine existing software engineering practices from a formal perspective. First, we introduce a rudimentary object oriented programming language.

2.1.1 A Simple Programming Language

The definitions that follow describe a set of abstract syntax trees (ASTs). First we introduce the concept of *identifiers*, to be used as names for product artefacts such as packages, classes, methods and fields:

- **2.1. Notation (Identifiers):** *Identifiers* are denoted by *id*. Sets of identifiers are denoted by *ID* or \mathcal{ID} . ┘

We do not define semantics for this language (Example 1.51, page 28). This thesis is about manipulating program *structure*. So we see *statements* and *types* in the same abstract way as identifiers, i.e., as arbitrary strings:

- **2.2. Notation (Statements and Types):** *Statements* are denoted by *st*. Sets of statements are denoted by *ST* or \mathcal{ST} .
Types are denoted by *tp*. Sets of types are denoted by *TP* or \mathcal{TP} . ┘

From this point on, we assume a global set of identifiers \mathcal{ID} , a global set of statements \mathcal{ST} and a global set of types \mathcal{TP} .

This leads to the definition of classes, which can contain methods and fields:

- **2.3. Definition (Classes, Methods and Fields):** A *method* is represented by a type and a sequence of statements. A *field* is represented by only a type. The set of all *classes* is a finite partial map:

$$\mathcal{Mtd} \stackrel{\text{def}}{=} \mathcal{TP} \times \mathcal{ST}^* \quad \mathcal{Fld} \stackrel{\text{def}}{=} \mathcal{TP} \quad \mathcal{CL} \stackrel{\text{def}}{=} \mathcal{ID} \rightharpoonup (\mathcal{Mtd} \cup \mathcal{Fld})$$

A class $cl \in \mathcal{CL}$ maps each identifier $id \in \text{pre}(cl)$ to either a method $cl(id) \in \mathcal{Mtd}$ or a field $cl(id) \in \mathcal{Fld}$. ┘

- **2.4. Example:** An example of a class is:

$$cl = \left\{ \begin{array}{ll} \text{"m_name"} & \mapsto \text{"String"}, \\ \text{"run"} & \mapsto \text{"String[] -> void"}, \\ & \left(\begin{array}{l} \text{"m_name = args[1]"}, \\ \text{"output(\"Hello " + m_name)} \end{array} \right) \end{array} \right\}$$

But from now on we'll often use pseudo-code instead, and assume the abstract mathematical structure to be understood:

```

1  class {
2      m_name: String;
3      run(args: String[]) : void {
4          m_name = args[1];
5          output("Hello " + m_name);
6      }
7  }

```

And finally, we define packages. A package can contain any number of classes mapped by name:

- **2.5. Definition (Packages):** A *package* is a finite partial function $pkg: \mathcal{ID} \rightarrow \mathcal{CL}$, mapping identifiers to classes. The set of all packages is denoted \mathcal{PKG} . ┘

2.1.2 The `DeltaEditor` Package

We now use this language to write a small package implementing a bare-bone version of the source code editor introduced in Section 1.4:

- **2.6. Example:** The software product “DeltaEditor core”:

```

1  package DeltaEditor {
2      class Editor {
3          m_model: Model;
4
5          init(m : Model) : void {
6              m_model = m;
7          };
8
9          model() : Model { return m_model; };
10
11         font(c : int) : Font {
12             Font result = new Font();
13             result.setColor(Color.BLACK);
14             result.setBold(false);
15             result.setUnderlined(false);
16             return result;
17         };
18
19         onMouseOver(c : int) : void { };
20     };
21 }

```

Assume that some other class (imported from a widget library perhaps) does most of the work, drawing and managing the visual interface. It is our job to implement the `model()`, `font(int)` and `onMouseOver(int)` methods so that the widget library has the necessary information to manage the editor.

2.1.3 Implementing Syntax Highlighting

Now, we implement some additional features in the traditional manner. This will demonstrate some of the disadvantages of the traditional approach—a lack of modularity, separation of concerns and variability control—and thereby motivate the work on delta modeling.

The first feature is Syntax Highlighting, which changes the font of the content to provide a visual distinction between different language constructs. To accomplish this we develop a new class inside the `DeltaEditor` package to handle the business logic of parsing the model and determining the correct font for each individual character. We then add an instance of it to the `Editor` class, initialize it and, finally, replace the `font(int)` method with one that consults the new class. The resulting program looks as follows (the modified lines have been highlighted):

▷ **2.7. Example:** The software product “DeltaEditor with *SH*”:

```

1  package DeltaEditor {
2      class Editor {
3          m_model : Model;
4          m_syntaxhl : SyntaxHL;
5
6          init(m : Model) : void {
7              m_model = m;
8              m_syntaxhl = new SyntaxHL(m);
9          };
10
11         model() : Model { return m_model; };
12
13         font(c : int) : Font {
14             return m_syntaxhl.font(c);
15         };
16
17         onMouseOver(c : int) : void { };
18     };
19
20     class SyntaxHL {
21         m_model : Model;
22
23         init(m : Model) : void { m_model = m; };
24
25         font(c : int) : Font { /* something complicated */; };
26     };
27 };

```

Note that to implement this one feature, we were forced to make changes in four different places. When, in the future, another developer needs to change one of the highlighted code-fragments, they may well neglect to make corresponding changes to the other fragments, which is how bugs are introduced. Also, keep in mind that this is an oversimplified example. In a full application,

the implementation of a feature like this would involve designing toolbar buttons and configuration screens, developing code for user interaction and code to link models, views and controllers — not to mention the code necessary for proper interaction with other features.

The point is, practically all software features are *cross cutting concerns*: their code *needs* to be spread around the code base to do its job properly, at least if we're using programming models like OOP. This is a well-known problem in the world of software engineering. When software approaches certain levels of complexity, it becomes harder and harder to properly maintain it. We therefore strive towards the following goal:

Goal: Find a way to ‘group together’ code related to the same feature.

This is called *feature modularity* or *feature locality* [89, 109, 156].

2.1.4 Implementing Error Checking

We now add another feature: Error Checking. We'd like certain syntactic errors to be underlined, and to show a tooltip when the mouse cursor hovers over them. Similar to before, a new class is responsible for the business logic, and several lines in the base class are added or modified to accomodate the new functionality. After implementing this feature, the resulting package might look as follows (again with the modified lines highlighted):

▷ **2.8. Example:** The software product “DeltaEditor with *SH* and *EC*”:

```

1 package DeltaEditor {
2     class Editor {
3         m_model      : Model;
4         m_syntaxhl    : SyntaxHL;
5         m_errorch     : ErrorChecker;
6
7         init(m : Model) : void {
8             m_model      = m;
9             m_syntaxhl   = new SyntaxHL(m);
10            m_errorch    = new ErrorChecker(m);
11        };
12
13        model() : Model { return m_model; };
14
15        font(c : int) : Font {
16            Font result = m_syntaxhl.font(c);
17            result.setUnderlined(m_errorch.errorOn(c));
18            return result;
19        };
20
21        onMouseOver(c : int) : void {
22            if (m_errorch.errorOn(c)) {
23                super.showTooltip(m_errorch.errorText(c));
24            }
25        };
26    };
27
```



```

28  class SyntaxHL {
29      m_model : Model;
30
31      init(m : Model) { m_model = m; };
32
33      font(c : int) : Font { /* something complicated */; };
34  };
35
36  class ErrorChecker {
37      m_model : Model;
38
39      init(m : Model) { m_model = m; };
40
41      errorOn(c : int) : bool { /* some code */; };
42
43      errorText(c : int) : string { /* more code */; };
44  };
45  };

```

The code for this feature has to be spread around just like before. But the thing to note here is that we had to change the `font(int)` method again. The new version handles both Syntax Highlighting and Error Checking correctly, but it is now hard to say where one feature ends and the other begins. Our original intention is obscured, even in this local context. If we ever want to expand either feature—or fix a bug—we risk accidentally tampering with the other feature too, perhaps breaking it without warning. This kind of problem clearly makes maintenance more difficult. So we also strive for the following goal:

Goal: Find a way to ‘separate’ code belonging to different features.

This is generally referred to as *separation of concerns* [96, 112, 114, 147].

Our answer to both problems consists of implementing each feature as a delta which can mechanically modify the core product (Example 2.6), rather than implementing them in the product directly. This chapter explores the interaction between products and deltas which makes this possible.

The remainder of the chapter is structured as follows. In Section 2.2 we start our abstract treatment of delta modeling by introducing the notions of product, delta, and how the latter can modify the former. It places these main ingredients in a structure called a *deltoid*. It also makes explicit our distinction between *syntax* and *semantics* and discusses the notion of *quotient deltoid*. Section 2.3 then applies these concepts to the software packages introduced in Section 2.1.1.

Section 2.4 further explores the semantic aspects of deltas. It presents notions of delta definedness, (non)determinism and specification. Section 2.5 then uses delta specifications to give a formulation of refinement and equivalence: when can one delta behaviorally take the place of another?

In Section 2.6 we explain how to reason syntactically about deltas, and briefly explore the field of abstract algebras. This is where the *delta monoid* is introduced, a structure always present in previous work on ADM. It gives us the notions of *delta composition* and the *neutral delta*. We also take a

particular look at the relation algebra introduced by Tarski [175], which proves to be quite relevant. Then, in Section 2.7, we classify deltoids by a number of *expressiveness* properties, as well as by means of a notion of deltoid *refinement*, defined in terms of product- and delta-homomorphisms.

Section 2.8 compares ADM with some other prominent algebraic formulations of feature-oriented programming, namely the work of Apel et al [17] and Batory et al [32], both based on the Quark model. We encode these formalisms within our own setting and demonstrate thereby the wide applicability and expressiveness of ADM. Finally, Section 2.9 offers concluding remarks and Section 2.10 discusses related work in a number of different directions.

2.2 Deltas & Products

This section presents the three main ingredients of the ADM formalism: deltas, products, and an operation to apply the former to the latter in order to generate new products.

2.2.1 Products

The object we are ultimately concerned with is the program or, abstractly put, the *product*. That is the object of traditional software engineering and that is what we ship to the end user. This thesis is about modularizing their design using deltas, but for deltas to make any sense, we first need something to apply them to.

On the abstract level, we do not specify the concrete nature of products. They could represent different kinds of development artefacts (e.g., documentation, models or code) on any level of abstraction (e.g., component level or class level). The set \mathcal{P} from Definition 2.5 is a good example of a product set, and we shall be following up on that formulation throughout the thesis. However, products might also model something radically different, like something that comes out of a physical production line.

- **2.9. Notation (Products):** We denote *products* by the symbols p, q . Sets of products are denoted by P or \mathcal{P} .

2.2.2 Deltas

We then introduce the main ingredient: *deltas*, which can transform one product into another. We don't specify their concrete nature either. They could be mathematical functions or relations performing the changes directly. But in practice, those changes will have some finite syntactic representation tailored to the product domain we are working with.

- **2.10. Notation (Deltas):** We denote *deltas* by the symbols d, w, x, y, z . Sets of deltas are denoted by D or \mathcal{D} . ┘

In Section 2.3 we design a set of deltas to operate on the software products from Section 2.1.

2.2.3 Delta Application

Deltas are applied to products in a process called *delta application*. Imagine for the moment that deltas are mathematical functions mapping products to products. Then applying a delta consists of simply calling the function, so $d(p)$ would be the product resulting from applying delta d to product p . More interesting cases, such as those in software product lines, involve the *incremental application* of a number of deltas d_1, \dots, d_n to a minimal core product c , each changing a specific aspect of it:

$$d_n(\dots d_1(c) \dots).$$

As mentioned in Section 2.2.2, in practice deltas are not mathematical functions or relations, but finite (syntactic) representations of such functions or relations. The semantics of deltas are given by the third main ingredient of the formalism, the *delta evaluation* operator. Together, a set of products, a set of deltas and a delta evaluation operator form a *deltoid*:

- **2.11. Definition (Deltoid):** A *deltoid* is a triple $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$ with a set of products \mathcal{P} , a set of deltas \mathcal{D} and a unary *delta evaluation* operator $\llbracket - \rrbracket: \mathcal{D} \rightarrow \text{Pow}(\mathcal{P} \times \mathcal{P})$. If $d \in \mathcal{D}$ is a delta, then $\llbracket d \rrbracket \subseteq \mathcal{P} \times \mathcal{P}$ is a binary relation over the set of products, sometimes called a *semantic delta*.

By Notation 1.12, $p \llbracket d \rrbracket q$ indicates that q may result from applying d to p and $\llbracket d \rrbracket(p)$ represents the set of *all* products that may result from such an application. We use both notations regularly. \lrcorner

A deltoid describes all building blocks necessary to model delta-based systems for a specific domain and abstraction level. The sets \mathcal{P} and \mathcal{D} represent the *potential* products and deltas of the domain of discourse, and are usually infinite in size (e.g., ‘all object oriented programs and deltas’).

The notion of deltoid presented in Definition 2.11 is a *generalization* of the one presented in earlier work [1, 2]. It differs in two important ways:

- Firstly, it does not require that deltas form a *monoid* with an application operator and neutral element. However, when they do, the new definition coincides with the traditional one. Delta composition and other algebraic topics are discussed in some detail in Section 2.6.
- Instead of a delta evaluation operator, the earlier works define a *delta application* operator $-(-): \mathcal{D} \times \mathcal{P} \rightarrow \mathcal{D}$. In essence, all deltas behaved like *functions*, whereas we now allow for them to be specified *relationally*. A delta may not apply to certain products (i.e., it may be partially defined). Conversely, it may apply and have more than one possible output product (i.e., it may be *non-deterministic*). We discuss these notions more thoroughly in Section 2.4.

With regard to that second point: the semantic evaluation operator often serves to *specify* delta application for a concrete domain, without actually *implementing* it. For a deltoid to be usable in practice, an *effective procedure* (i.e., an executable algorithm) for delta application must be written, roughly corresponding to the $-(-)$ operator of the earlier work:

- **2.12. Definition (Delta Application):** Given a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, *delta application* is a partial function $\text{apply}: \mathcal{D} \times \mathcal{P} \rightharpoonup \mathcal{P}$, representing an effective procedure satisfying the following axiom for all deltas $d \in \mathcal{D}$ and products $p \in \mathcal{P}$:

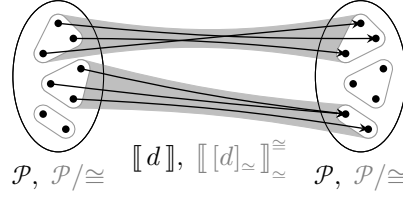


Figure 2.1: A delta d and its equivalence class (in gray) acting on a set of products \mathcal{P} and its quotient set (in gray).

$$\mathbf{a.} \ (d, p) \in \text{pre}(\text{apply}) \implies p \llbracket d \rrbracket \text{apply}(d, p) \quad \lrcorner$$

This thesis maintains a firm distinction between syntax and semantics. *Syntax* is concerned with deltas. *Semantics* is concerned with products. The bridge between these two worlds is the $\llbracket - \rrbracket$ notation, as witnessed in Definition 2.11. In general, we keep to the following convention:

$$\llbracket \langle \text{something syntactic} \rangle \rrbracket = \langle \text{something semantic} \rangle$$

We introduce several extensions of the $\llbracket - \rrbracket$ notation over the course of the thesis. We begin with a straightforward extension to sets of deltas:

► **2.13. Notation:** Given any delta set $D \subseteq \mathcal{D}$, we define $\llbracket D \rrbracket \stackrel{\text{def}}{=} \{ \llbracket d \rrbracket \mid d \in D \}$. \lrcorner

2.2.4 Quotient Deltoids

Recall Section 1.7.7 on quotient sets. If the delta set \mathcal{D} and/or the product set \mathcal{P} happen to be quotient sets (Definition 1.26), we require that the delta evaluation operator $\llbracket - \rrbracket$ behave appropriately with regard to the associated equivalence relations, as we would for algebraic operations (Definition 1.32):

► **2.14. Definition (Quotient Deltoid):** Given any deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, the corresponding *quotient deltoid* by equivalence relations $\cong \subseteq \mathcal{P} \times \mathcal{P}$ and $\simeq \subseteq \mathcal{D} \times \mathcal{D}$, denoted $(\mathcal{P}/\cong, \mathcal{D}/\simeq, \llbracket - \rrbracket_{\simeq}^{\cong})$, exists iff a delta evaluation operator $\llbracket - \rrbracket_{\simeq}^{\cong}: (\mathcal{D}/\simeq) \rightarrow \text{Pow}((\mathcal{P}/\cong) \times (\mathcal{P}/\cong))$ exists such that for all deltas $d \in \mathcal{D}$ and products $p, q \in \mathcal{P}$:

$$p \llbracket d \rrbracket q \iff [p]_{\cong} \llbracket [d]_{\simeq} \rrbracket_{\simeq} [q]_{\simeq} \quad \lrcorner$$

Figure 2.1 illustrates this concept. To prove that such a quotient counterpart of delta evaluation exists, it suffices to prove the following property for a given delta evaluation operator:

► **2.15. Lemma:** The quotient of a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$ may be used iff for all products $p_1, p_2, q_1, q_2 \in \mathcal{P}$ and all deltas $d_1, d_2 \in \mathcal{D}$:

$$p_1 \cong p_2 \wedge q_1 \cong q_2 \wedge d_1 \simeq d_2 \implies (p_1 \llbracket d_1 \rrbracket q_1 \iff p_2 \llbracket d_2 \rrbracket q_2) \quad \square$$

The existence of a quotient deltoid allows us to extend implicit canonical projection (Notation 1.27) to delta evaluation and use $\llbracket - \rrbracket$ as an abbreviation for $\llbracket - \rrbracket_{\simeq}^{\cong}$.

2.3 The Software Deltoid

We now continue what we started at the beginning of the chapter and build a deltoid around the notion of software package from Definition 2.5.

2.3.1 Software Deltas

We need to come up with a language for software deltas that can express modifications to software packages in \mathcal{PKG} . It needs to be intuitive for developers and powerful enough to describe implementations of the kinds of features we are interested in, such as those from the Editor specification (Section 1.4.1). Recently, some work has been done in automatically deriving a delta language from a product language [76], but generally the task requires knowledge of the problem domain and an adequate understanding of the language. We propose the following, expressive enough to set up most of the Editor features in a modular fashion:

- ▷ **2.16. Definition (Software Deltas):** We define software deltas on two levels: packages and classes. We start on the lower level. *Software class deltas* are defined as finite partial maps:

$$\mathcal{D}_{cl} \stackrel{\text{def}}{=} \mathcal{ID} \multimap \mathcal{OP}_{cl}$$

mapping each identifier to a class-level operation:

$$\mathcal{OP}_{cl} \stackrel{\text{def}}{=} \left(\begin{array}{l} \{\mathbf{add}\} \times (\mathcal{Mtd} \cup \mathcal{Fld}) \cup \\ \{\mathbf{rem}\} \cup \\ \{\mathbf{rep}\} \times (\mathcal{Mtd} \cup \mathcal{Fld}) \cup \\ \{\mathbf{frb}\} \cup \\ \{\mathbf{err}\} \end{array} \right)$$

An **add** operation adds a new entity with the given identifier, and therefore requires that the identifier is not yet in use. A **rem** (**remove**) operation removes the entity currently using the identifier, and requires that such an entity exists.

A **rep** (**replace**) is the same as a **remove** followed by an **add**, and replaces the entity using the given identifier with the given product value. Similarly (though perhaps less intuitively), a **frb** (**forbid**) is the same as an **add** followed by a **remove**. This is really more an assertion than an operation. It does not modify anything, but still imposes the condition inherited from **add**: that the given identifier is not currently in use.

Finally, an **err** (**error**) may be present. A delta with this placeholder on any level is *invalid* and will not yield any results when applied to a product. This construct is presumably never used by developers, but is useful for propagating the result of invalid delta operations, which are examined in detail in Section 2.6.

We now move to the package level, and define *software package deltas* (or *software deltas* for short) as finite partial maps as well:

$$\mathcal{D}_{pkg} \stackrel{\text{def}}{=} \mathcal{ID} \multimap \mathcal{OP}_{pkg}$$

mapping each identifier to a package-level operation:

$$\mathcal{OP}_{\text{pkg}} \stackrel{\text{def}}{=} \begin{pmatrix} \{\mathbf{add}\} \times \mathcal{CL} & \cup \\ \{\mathbf{rem}\} & \cup \\ \{\mathbf{rep}\} \times \mathcal{CL} & \cup \\ \{\mathbf{mod}\} \times \mathcal{D}_{\text{cl}} & \cup \\ \{\mathbf{frb}\} & \cup \\ \{\mathbf{err}\} & \end{pmatrix}$$

Deltas on this level are intuitively very similar to deltas on the class level. The operations can add and remove full classes. However, there is one important addition: the **mod** (**modify**) operation descends one level in order to make modifications of a finer granularity. We can only do so on the package level (at least for now). These deltas cannot, for example, tinker with the type or individual statements of a method. \lrcorner

As you can see, these deltas follow the structure of Definition 2.5, providing operations on both the package and class levels. They employ *invasive composition* [23], as they disregard object-oriented encapsulation by referencing—from the outside—artifacts of arbitrary nesting depth and ignoring class boundaries. The depth at which a modification occurs is called its *granularity*. Generally, a modification inside the body of a method is called *fine-grained*. Modifications on a higher level are called *coarse-grained* [108]. By this terminology, the deltas above are only capable of making coarse-grained modifications.

Since we want to keep our examples as simple as possible, there are obvious limits to this set of operations. They do not work on a fine-grained level and cannot alter types or parameters. We have not introduced class inheritance in the programming language, so these deltas cannot alter the inheritance hierarchy. But these are merely artificial limits. Chapter 3 will extend software deltas so they are able to manipulate method bodies. Haber et al. [77, 79] extended software deltas with **connect** and **disconnect** operations for software components. In 2013 they applied delta modeling to Matlab/Simulink [78], a graphical language.

The following is an example of a software delta:

▷ 2.17. **Example:** The software delta “SH implementation”:

```

1  modify package DeltaEditor {
2      modify class Editor {
3          add m_syntaxhl : SyntaxHL;
4
5          replace init(m : Model) : void {
6              m_model = m;
7              m_syntaxhl = new SyntaxHL(m);
8          };
9
10         replace font(c : int) : Font {
11             return m_syntaxhl.font(c);
12         };
13     };
14 
```

```

15     add class SyntaxHL {
16         m_model : Model;
17
18         init(m : Model) : void { m_model = m; };
19
20         font(c : int) : Font { /* something complicated */; };
21     };
22 };

```

2.3.2 Software Delta Application

The meaning of software deltas is, hopefully, already somewhat intuitive. The following definition formalizes their semantics by defining the software delta evaluation operator, completing the basic *software deltoid*:

- ▷ **2.18. Definition (Software Deltoid):** *Software deltoid* $Dt_{\text{pkg}} \stackrel{\text{def}}{=} (\mathcal{PKG}, \mathcal{D}_{\text{pkg}}, \llbracket - \rrbracket)$ comprises \mathcal{PKG} from Definition 2.5 as its product set and \mathcal{D}_{pkg} from Definition 2.16 as its delta set. We define semantic evaluation of software deltas by specifying a set of inference rules (Notation 1.15). We do so on *four levels*: full package deltas, package level operations, full class deltas and class-level operations. As before, we start at the lowest level.

a. Class-level Operations

The semantics of class-level operations is defined by the smallest semantic evaluation operator $\llbracket - \rrbracket: \mathcal{ID} \times \mathcal{OP}_{\text{cl}} \rightarrow \text{Pow}(\mathcal{CL} \times \mathcal{CL})$ satisfying the following inference rules. For all identifiers $id \in \mathcal{ID}$, classes $cl \in \mathcal{CL}$ and methods or fields $mf \in \mathcal{Mtd} \cup \mathcal{Fld}$:

$$\begin{array}{c}
 \frac{id \notin \text{pre}(cl)}{cl \llbracket id \mapsto \mathbf{add} \ mf \rrbracket \ cl[id \mapsto mf]} \quad \text{method/field addition} \\
 \\
 \frac{id \in \text{pre}(cl)}{cl \llbracket id \mapsto \mathbf{rem} \rrbracket \ cl[id \mapsto \perp]} \quad \text{method/field removal} \\
 \\
 \frac{id \in \text{pre}(cl)}{cl \llbracket id \mapsto \mathbf{rep} \ mf \rrbracket \ cl[id \mapsto mf]} \quad \text{method/field replacement} \\
 \\
 \frac{id \notin \text{pre}(cl)}{cl \llbracket id \mapsto \mathbf{frb} \rrbracket \ cl} \quad \text{method/field forbiddance} \\
 \\
 \frac{}{cl \llbracket id \mapsto \perp \rrbracket \ cl} \quad \text{no operation}
 \end{array}$$

Note that the ‘precondition’ of each operation —the presence or absence of a particular identifier— is specified as a premise for each rule. Note in particular that the **error** operation is not mentioned. Indeed, this means that $\llbracket id \mapsto \mathbf{err} \rrbracket = \emptyset$. A software delta with an **error** inside cannot produce a valid result. The error is propagated to the higher levels.

b. Class Deltas

The semantics of class deltas is defined by the smallest semantic evaluation operator $\llbracket - \rrbracket: \mathcal{D}_{cl} \rightarrow \text{Pow}(\mathcal{C} \times \mathcal{C})$ satisfying the following inference rule. For all classes $cl \in \mathcal{C}$ and class deltas $d_{cl} \in \mathcal{D}_{cl}$:

$$\frac{\forall id \in \mathcal{I}: \quad cl(id) \llbracket id \mapsto d_{cl}(id) \rrbracket cl'(id)}{cl \llbracket d_{cl} \rrbracket cl'} \quad \begin{array}{l} \text{software class} \\ \text{delta application} \end{array}$$

This basically lifts class-level operation semantics to the level of full class deltas, applying them for every (relevant) identifier.

c. Package Level Operations

The semantics of package-level operations is defined by the smallest semantic evaluation operator $\llbracket - \rrbracket: \mathcal{I} \times \mathcal{OP}_{pkg} \rightarrow \text{Pow}(\mathcal{PKG} \times \mathcal{PKG})$ satisfying the following inference rules. For all identifiers $id \in \mathcal{I}$, packages $pkg \in \mathcal{PKG}$, classes $cl \in \mathcal{C}$ and class deltas $d_{cl} \in \mathcal{D}_{cl}$:

$$\begin{array}{c} \frac{id \notin \text{pre}(pkg)}{pkg \llbracket id \mapsto \mathbf{add} \ cl \rrbracket \ pkg[id \mapsto cl]} \quad \text{class addition} \\[1em] \frac{id \in \text{pre}(pkg)}{pkg \llbracket id \mapsto \mathbf{rem} \rrbracket \ pkg[id \mapsto \perp]} \quad \text{class removal} \\[1em] \frac{id \in \text{pre}(pkg)}{pkg \llbracket id \mapsto \mathbf{rep} \ cl \rrbracket \ pkg[id \mapsto cl]} \quad \text{class replacement} \\[1em] \frac{id \notin \text{pre}(pkg)}{pkg \llbracket id \mapsto \mathbf{fxb} \rrbracket \ pkg} \quad \text{class forbiddance} \\[1em] \frac{id \in \text{pre}(pkg) \quad pkg(id) \llbracket d_{cl} \rrbracket \ cl}{pkg \llbracket id \mapsto \mathbf{mod} \ d_{cl} \rrbracket \ pkg[id \mapsto cl]} \quad \text{class modification} \\[1em] \frac{}{pkg \llbracket id \mapsto \perp \rrbracket \ pkg} \quad \text{no operation} \end{array}$$

The interesting new rule is the one for class modification. Its second premise states that applying the class level delta d_{cl} to the existing class $pkg(id)$ can result in a new class cl . After ‘delegating’ to the lower level, the rule replaces the existing class with the new class.

d. Package Deltas

The semantics of package deltas is defined by the smallest semantic evaluation operator $\llbracket - \rrbracket: \mathcal{D}_{\text{pkg}} \rightarrow \text{Pow}(\mathcal{PKG} \times \mathcal{PKG})$ satisfying the following inference rule. For all packages $pkg \in \mathcal{PKG}$ and package deltas $d_{\text{pkg}} \in \mathcal{D}_{\text{pkg}}$:

$$\frac{\forall id \in \mathcal{ID}: \quad pkg(id) \llbracket id \mapsto d_{\text{pkg}}(id) \rrbracket pkg'(id)}{pkg \llbracket d_{\text{pkg}} \rrbracket pkg'} \quad \begin{array}{l} \text{software package} \\ \text{delta application} \end{array}$$

This rule lifts operations to the full delta level, as before. \lrcorner

- ▷ **2.19. Definition (Software Delta Application):** *Software delta application* is an effective procedure, apply: $\mathcal{D}_{\text{pkg}} \times \mathcal{PKG} \rightarrow \mathcal{PKG}$, as per Definition 2.12.

Semantic software delta evaluation was defined in a straightforward and constructive way, so this definition would be almost a repeat of Definition 2.18. We assume that this partial function is defined to satisfy Axiom 2.12a. For a definition of this style, the reader is referred to the ADM papers [1, 2]. \lrcorner

- ▷ **2.20. Lemma:** Referring to Examples 2.6, 2.7 and 2.17, we have:

$$\text{“DeltaEditor core”} \llbracket \text{“SH implementation”} \rrbracket \text{“DeltaEditor with SH”} \quad \square$$

The thesis often refers back to this deltoid.

2.3.3 Software Delta Equivalence

When working with concrete syntax as we are now, it soon becomes useful to define equivalence relations in order to treat structurally different products and/or deltas the same way.

For software packages this is not the case. But for software deltas it is. We can equate all deltas that contain an **error** at any level of nesting. Afterwards we can work in the quotient set and use implicit canonical projection (Notation 1.27).

We require one intermediate definition: a predicate for identifying invalid software deltas (those that contain an **error**):

- ▷ **2.21. Definition (Invalid Software Deltas):** The *invalid software delta* predicate $\text{Err} \subseteq \mathcal{D}_{\text{pkg}} \cup \mathcal{OP}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}} \cup \mathcal{OP}_{\text{cl}}$ holds for software deltas and operations that contain an **error** at any nesting level. It is the smallest predicate so that the following hold for all identifiers $id \in \mathcal{ID}$ and deltas $d_{\text{cl}} \in \mathcal{D}_{\text{cl}}$ and $d_{\text{pkg}} \in \mathcal{D}_{\text{pkg}}$:

$$\begin{aligned} \text{Err}(d_{\text{pkg}}) & \quad \text{if } \exists id \in \text{pre}(d_{\text{pkg}}): \text{Err}(d_{\text{pkg}}(id)) \\ \text{Err}(\mathbf{modify} \ d_{\text{cl}}) & \quad \text{if } \text{Err}(d_{\text{cl}}) \\ \text{Err}(d_{\text{cl}}) & \quad \text{if } \exists id \in \text{pre}(d_{\text{cl}}): \text{Err}(d_{\text{cl}}(id)) \\ \text{Err}(\mathbf{error}) & \end{aligned} \quad \lrcorner$$

This makes it simple to define the equivalence relation:

- **2.22. Definition (Software Delta Equivalence):** The *software delta equivalence* relation $\simeq \subseteq \mathcal{D}_{\text{pkg}} \times \mathcal{D}_{\text{pkg}}$ is defined to equate all invalid deltas, as well as each delta with itself:

$$\simeq \stackrel{\text{def}}{=} \text{Err}^2 \cup \text{id}_{\mathcal{D}_{\text{pkg}}} \quad \lrcorner$$

Finally, we prove that Definition 2.18 respects this equivalence relation, as required by Definition 2.14 by using Lemma 2.15. We can then work in the quotient deltoid.

- **2.23. Theorem:** For all packages $p_1, p_2, q_1, q_2 \in \mathcal{PKG}$ and software deltas $d_1, d_2 \in \mathcal{D}_{\text{pkg}}$ we have:

$$p_1 \cong p_2 \wedge q_1 \cong q_2 \wedge d_1 \simeq d_2 \implies (p_1 \llbracket d_1 \rrbracket q_1 \Leftrightarrow p_2 \llbracket d_2 \rrbracket q_2)$$

Proof: As we have no special package equivalence relation, this is simplified to:

$$d_1 \simeq d_2 \implies (p \llbracket d_1 \rrbracket q \Leftrightarrow p \llbracket d_2 \rrbracket q)$$

Then it only remains to prove that two deltas that both have an **error** at any level have the same behavior. This is trivial, as, by Definition 2.18, all semantic software deltas with an error are empty relations. This propagates from the lowest to the highest level, as noted in Definition 2.18a. \square

2.4 The Semantics of Deltas

In earlier work the semantics of deltas were *functions* [1, 2]. But since then the need arose to make them more expressive, hence the current interpretation of semantic deltas as *relations* (Definition 2.11). In this section we explore the implications.

2.4.1 Definedness and Determinism

Deltas may now be *partially defined* and *non-deterministic*.

- **2.24. Definition (Product Acceptance):** Given a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, a delta $d \in \mathcal{D}$ is said to *accept* a product $p \in \mathcal{P}$ iff $p \in \text{pre} \llbracket d \rrbracket$. \lrcorner
- **2.25. Definition (Fully Defined Delta):** Given a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, a delta $d \in \mathcal{D}$ is said to be *fully defined* iff it accepts all products, i.e., iff $\text{pre} \llbracket d \rrbracket = \mathcal{P}$. A delta that is not fully defined is *partially defined*. A delta that accepts no products at all is called *undefined* or *invalid*. \lrcorner
- **2.26. Definition (Deterministic Delta):** Given a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, a delta $d \in \mathcal{D}$ is said to be *deterministic* iff $\llbracket d \rrbracket$ is uniquely defined (Definition 1.13). Otherwise it is called *non-deterministic*. \lrcorner

When compared to the old functional interpretation, Definitions 2.25 and 2.26 give useful generalizations of the intuitive concept of a modification. Partiality (Figure 2.2) allows us to model modifications that only make sense for certain products. This applies, for example, when a software delta **removes** an identifier that is not present in the product, or **add** an identifier that *is* present.

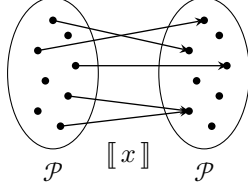


Figure 2.2: A partially defined, deterministic delta x .

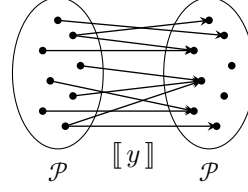


Figure 2.3: A fully defined, non-deterministic delta y .

Non-determinism (Figure 2.3) allows us to model deltas that have more than one possible result when transforming a product, with no guarantee as to which it might be. We can use this in certain situations to prove that the choice ‘does not matter’. We use the term ‘deterministic’ rather than the term ‘uniquely defined’ (Definition 1.13) because it fits better with the idea of a delta performing a transformation.

Software deltas are *deterministic*; at least for now. Each operation can only modify a software product in a single, specific way:

- **2.27. Lemma:** All software deltas $d \in \mathcal{D}_{\text{pkg}}$ (Definition 2.18) are deterministic. All except \emptyset (the empty map) are partially defined. \square

Chapter 3 will introduce software deltas that can insert statements in arbitrary positions inside a method body. This makes them not only fine-grained, but potentially nondeterministic as well.

2.4.2 Delta Specifications

To help us reason about the behavior of deltas we introduce the notion of delta specifications:

- **2.28. Definition (Delta Specification):** Given a deltoid $Dt = (\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, the corresponding set of *delta specifications* $\mathcal{S} \stackrel{\text{def}}{=} \text{Pow}(\mathcal{P} \times \mathcal{P})$ consists of the full set of product relations.

If the deltoid is not clear from context, we attach a subscript as in \mathcal{S}_{Dt} . \lrcorner

Basically, delta specifications can express any behavior a semantic delta may have. They look and feel like semantic deltas, but they don’t require a syntactic counterpart in \mathcal{D} . Now that we have delta specifications, we can establish a notion of delta correctness. We distinguish between *partial* and *total* correctness, based on the same distinction in Hoare Logic [93, 94]:

- **2.29. Definition (Delta Correctness):** Given a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, to indicate that a delta $d \in \mathcal{D}$ is *partially or totally correct* with regard to a specification $s \in \mathcal{S}$ we use the satisfaction relation $\models \subseteq \mathcal{D} \times \mathcal{S}$:

$$\begin{aligned}
 d \models s &\iff \underbrace{\forall p \in \text{pre}(s): p \in \text{pre } \llbracket d \rrbracket}_{\text{a}} \Rightarrow \underbrace{\llbracket d \rrbracket(p) \subseteq s(p)}_{\text{c}} \\
 d \models_{\text{tot}} s &\iff \underbrace{\forall p \in \text{pre}(s): p \in \text{pre } \llbracket d \rrbracket}_{\text{a}} \wedge \underbrace{\llbracket d \rrbracket(p) \subseteq s(p)}_{\text{c}}
 \end{aligned}$$

If the deltoid is not clear from context, we attach a subscript as in \models_{Dt} . \lrcorner

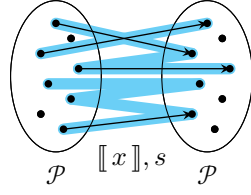


Figure 2.4: A delta x partially correct w.r.t. a specification s .

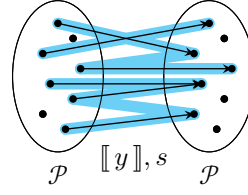


Figure 2.5: A delta y totally correct w.r.t. a specification s .

In other words, (a) for all products that satisfy the *precondition* of s , (b) if delta d accepts the product, (c) then any product resulting from its application will satisfy the *postcondition* of s . Partial correctness guarantees that d won't return an invalid result, but it doesn't actually guarantee that there will be a result at all. Total correctness guarantees both.

An interesting equivalent formulation is the following:

2.30. Lemma: For any delta $d \in \mathcal{D}$ and any delta specification $s \in \mathcal{S}$ we have:

$$\begin{aligned} d \models s &\stackrel{\text{def}}{\iff} \llbracket d \rrbracket \subseteq s \\ d \models_{\text{tot}} s &\stackrel{\text{def}}{\iff} \llbracket d \rrbracket \subseteq s \wedge \text{pre} \llbracket d \rrbracket = \text{pre}(s) \end{aligned} \quad \square$$

This formulation can be visualized using relation diagrams. Figures 2.4 and 2.5 show examples of partially correct deltas. The delta in Figure 2.5 is also totally correct.

2.4.3 Delta Derivation

A related but less expressive concept is that of *derived deltas*, introduced in [6]. They won't be used much until Chapter 8. We introduce them here because they are interesting to compare to delta specifications.

We will use them to express a number of useful notions further on. They also serve to put the power of delta specifications in perspective.

► **2.31. Definition (Delta Derivation):** Given deltoid $Dt = (\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, a delta *derived* from two product sets $P, P' \subseteq \mathcal{P}$ is one that can transform any product from the first set into some product from the second. This kind of specification can also be separated into a *partial* and *total* correctness version, denoted by the operators $\models, \models_{\text{tot}}: \text{Pow}(\mathcal{P}) \times \text{Pow}(\mathcal{P}) \rightarrow \text{Pow}(\mathcal{D})$, defined as follows:

$$\begin{aligned} P \models P' &\stackrel{\text{def}}{=} \{ d \in \mathcal{D} \mid \forall p \in P: \llbracket d \rrbracket(p) \subseteq P' \} \\ P \models_{\text{tot}} P' &\stackrel{\text{def}}{=} \{ d \in \mathcal{D} \mid \forall p \in P: \emptyset \subset \llbracket d \rrbracket(p) \subseteq P' \} \end{aligned}$$

If the deltoid is not clear from context, we attach a subscript as in \models_{Dt} . \dashv

We now show the connection between derivations and specifications:

2.32. Lemma: For all deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$ and all deltas $d \in \mathcal{D}$ and all product sets $P, P' \subseteq \mathcal{P}$ we have:

$$\begin{aligned} d \models P \times P' &\iff d \in (P \models P') \\ d \models_{\text{tot}} P \times P' &\iff d \in (P \models_{\text{tot}} P') \end{aligned}$$

Proof: The proof for \models_{tot} proceeds as follows:

$$\begin{aligned}
d &\models_{\text{tot}} P \times P' && \iff (\text{Def. 2.29}) \\
\forall p \in \text{pre}(P \times P'): p \in \text{pre} \llbracket d \rrbracket \wedge \llbracket d \rrbracket(p) \subseteq (P \times P')(p) &&& \iff (\text{Not. 1.12}) \\
\forall p \in P: p \in \text{pre} \llbracket d \rrbracket \wedge \llbracket d \rrbracket(p) \subseteq P' &&& \iff (\text{Not. 1.12}) \\
\forall p \in P: \emptyset \subset \llbracket d \rrbracket(p) \subseteq P' &&& \iff (\text{Not. 1.1}) \\
d \in \{ x \in \mathcal{D} \mid \forall p \in P: \emptyset \subset \llbracket x \rrbracket(p) \subseteq P' \} &&& \iff (\text{Def. 2.31}) \\
d \in (P \Rightarrow_{\text{tot}} P') &&&
\end{aligned}$$

The proof for \models is somewhat simpler but mostly analogous. \square

Lemma 2.32 shows us exactly how delta derivation is a weaker notion than delta specification: it can only express delta specifications that are full Cartesian products. To make an analogy with the field of type theory: one could say that a delta derivation is to a delta specification what a *product type* is to a *dependent product type* [151, 152]. A delta specification is able to express how the output type depends on the input. In return, however, delta derivation is often decidable — something we make good use of in Chapter 8.

2.5 Delta Refinement and Equivalence

A deltoid may contain any number of distinct deltas that have very similar—or even identical—effects on products. To express such facts, this section defines semantic notions of *delta refinement* and *delta equivalence*.

When a given delta satisfies at least the same specifications as another, and can thus always safely take its place, it is said to *refine* the other delta. There are two kinds of refinement: one that preserves partial correctness and one that preserves total correctness. When two deltas refine each other, and thus act identically in every situation, they are *equivalent*.

- **2.33. Definition (Semantic Delta Refinement):** Given a deltoid $Dt = (\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, *semantic delta refinement* is a preorder $\sqsupseteq \subseteq \mathcal{D} \times \mathcal{D}$. Delta $x \in \mathcal{D}$ is a *semantic refinement* of delta $y \in \mathcal{D}$ iff its corresponding semantic delta is a subset of that of the other¹:

$$\begin{aligned}
x \sqsupseteq y &\stackrel{\text{def}}{\iff} \llbracket x \rrbracket \subseteq \llbracket y \rrbracket \\
x \sqsupseteq_{\text{tot}} y &\stackrel{\text{def}}{\iff} \llbracket x \rrbracket \subseteq \llbracket y \rrbracket \wedge \text{pre} \llbracket x \rrbracket = \text{pre} \llbracket y \rrbracket
\end{aligned}$$

If the deltoid is not clear from context, we attach a subscript as in \sqsupseteq_{Dt} . \lrcorner

The following establishes the expected correspondance between semantic delta refinement and delta correctness:

- 2.34. Lemma:** Delta x refines delta y iff x satisfies all specifications that y does:

$$\begin{aligned}
x \sqsupseteq y &\iff \forall s \in \mathcal{S}: y \models s \implies x \models s \\
x \sqsupseteq_{\text{tot}} y &\iff \forall s \in \mathcal{S}: y \models_{\text{tot}} s \implies x \models_{\text{tot}} s \quad \square
\end{aligned}$$

¹The direction of the refinement symbol in $x \sqsupseteq y$ may feel counterintuitive, but it is the standard direction used in literature [44, 135]. Think of it as x being *more* refined.

And sometimes it is easier to reason about refinement of relations if it is specified in the following way:

- 2.35. Lemma:** Delta x refines delta y iff, for every product p , x will only produce a subset of y 's possible results:

$$\begin{aligned} x \sqsupseteq y &\iff \forall p \in \text{pre } \llbracket y \rrbracket: && \llbracket x \rrbracket(p) \subseteq \llbracket y \rrbracket(p) \\ x \sqsupseteq_{\text{tot}} y &\iff \forall p \in \text{pre } \llbracket y \rrbracket: && \emptyset \subset \llbracket x \rrbracket(p) \subseteq \llbracket y \rrbracket(p) \quad \square \end{aligned}$$

Note that the semantic delta refinement preorder is not always a partial order. There may be multiple distinct syntactic deltas which are mapped to the same relation. Such deltas are *equivalent*. Semantic equivalence of deltas is straightforwardly based on refinement, as is often the case in literature:

- **2.36. Definition (Semantic Delta Equivalence):** Given a deltoid $Dt = (\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$ we define *semantic delta equivalence* $\equiv \subseteq \mathcal{D} \times \mathcal{D}$ as follows for all $x, y \in \mathcal{D}$:

$$x \equiv y \stackrel{\text{def}}{\iff} x \sqsupseteq y \wedge x \sqsubseteq y$$

If the deltoid is not clear from context, we attach a subscript as in \equiv_{Dt} . \lrcorner

The following is an equivalent formulation:

- 2.37. Lemma:** For every two deltas $x, y \in \mathcal{D}$ we have:

$$x \equiv y \iff \llbracket x \rrbracket = \llbracket y \rrbracket$$

$$\begin{aligned} \text{Proof: } x \equiv y &\iff x \sqsupseteq y \wedge x \sqsubseteq y \iff \\ &\llbracket x \rrbracket \subseteq \llbracket y \rrbracket \wedge \llbracket x \rrbracket \supseteq \llbracket y \rrbracket \iff \llbracket x \rrbracket = \llbracket y \rrbracket \quad \square \end{aligned}$$

When no two distinct deltas are semantically equivalent, it makes it easier to reason syntactically about their effects. This is why it can be useful to establish a quotient deltoid (Definition 2.14). If a syntactic equivalence relation \simeq can be defined such that $\text{id}_{\mathcal{D}} \subset \simeq \subseteq \equiv$, syntactic equality of the quotient will approach semantic equivalence.

2.6 Delta Algebras

The previous sections form a picture of the relation between deltas and products. This section explores deltas from an algebraic perspective. That is, it introduces a number of useful delta operations and categorizes them in the style of abstract algebra [98] (Section 1.7.9). This allows us to reason about deltas on a syntactic level, without actually involving products.

For instance, to reason about incremental application, we need to introduce a composition operation \cdot , so that applying $y \cdot x$ is the same as applying first x and then y . It then makes sense to define a delta ε which is neutral in \cdot and thus ‘modifies nothing’. Deltas have been presented with a *monoid* structure $(\mathcal{D}, \cdot, \varepsilon)$ (Definition 1.33) since our first publication about ADM [1].

A later publication [3] also introduces the operator \sqcup for non-deterministic choice between two deltas,² to express the ambiguity of delta models that contain unresolved conflicts (Chapter 3).

This section discusses these operations. But it does beg the questions: How are the operations related, and which others might be useful? The key insight is that the abovementioned operators have obvious interpretations on a semantic level. Namely relation composition \circ , the identity relation $\text{id}_{\mathcal{P}}$ and set union \cup (Definitions 1.1 and 1.11). Delta semantics are given in terms of relations, and any operation that makes sense for relations potentially has a syntactic counterpart that makes sense for deltas.

Relation Algebras

Taking this relational point of view to its logical conclusion leads us to *relation algebras*, pioneered by Tarski [101, 102, 175]. Relation algebras capture the meaning of the standard relational operators (Definitions 1.1 and 1.11) and are thus worth studying as the limit of what abstract deltas could express.

Relation algebras are formally introduced in Definition 1.35. To summarize, their signature is $(S, \sqcup, \sqcap, \neg, \perp, \top, \cdot, \varepsilon, \smile)$, with carrier set S , disjunction operator $\sqcup: S \times S \rightarrow S$, conjunction operator $\sqcap: S \times S \rightarrow S$, negation operator $\neg: S \rightarrow S$, an empty element $\perp \in S$, a full element $\top \in S$, a composition operator $\cdot: S \times S \rightarrow S$, a neutral element $\varepsilon \in S$ and a converse operator $\smile: S \rightarrow S$. If we take $S = \mathcal{D}$ to be the set of deltas from a deltoid, we would assume any of these operators, when implemented, to respect the following semantics:

- **2.38. Definition (Relation Algebra Semantics):** A relation algebra operator implemented for a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$ should respect the following interpretations. For any two deltas $x, y \in \mathcal{D}$:

$$\begin{array}{llll} \llbracket x \sqcup y \rrbracket & = & \llbracket x \rrbracket \cup \llbracket y \rrbracket & \llbracket \perp \rrbracket & = & \emptyset \\ \llbracket x \sqcap y \rrbracket & = & \llbracket x \rrbracket \cap \llbracket y \rrbracket & \llbracket \top \rrbracket & = & \mathcal{P} \times \mathcal{P} \\ \llbracket y \cdot x \rrbracket & = & \llbracket y \rrbracket \circ \llbracket x \rrbracket & \llbracket \varepsilon \rrbracket & = & \text{id}_{\mathcal{P}} \\ \llbracket x^\smile \rrbracket & = & \llbracket x \rrbracket^{-1} & \llbracket x^- \rrbracket & = & \llbracket x \rrbracket^c \end{array} \quad \lrcorner$$

In the case where deltas are semantic deltas (and delta evaluation $\llbracket - \rrbracket = \text{id}_{\mathcal{P} \times \mathcal{P}}$ is simply the identity function), they form what is known as a *proper relation algebra* [101, 127, 128], with the signature $(\mathcal{D}, \cup, \cap, ^c, \emptyset, \mathcal{P} \times \mathcal{P}, \circ, \text{id}_{\mathcal{P}}, ^{-1})$. Delta evaluation $\llbracket - \rrbracket$ is always a homomorphism from the current ‘delta algebra’ to the proper relation algebra. This behavior is guaranteed if the operator implementations satisfy the axioms of Definitions 1.33 to 1.35.

So what intuitive interpretation we can attribute to each of these operators? The monoid operators of composition \cdot and the neutral element ε would respectively represent sequential application of deltas and the delta that modifies nothing — notions that have proved their usefulness in previous work. Disjunction \sqcup represents nondeterministic choice. Its dual, conjunction \sqcap , represent agreement or consensus between two deltas. The empty element \perp represents an invalid delta (Definition 2.25). Those are the ones with the most obviously

²Actually, in [3] the \cup symbol is used for this, but a new notation was chosen to emphasize the difference between syntax and semantics.

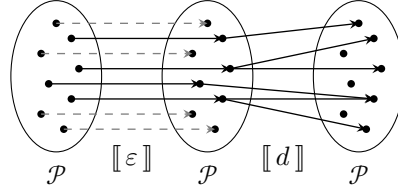


Figure 2.6: A diagrammatic representation of delta composition $d \cdot \varepsilon$. The dashed arrows are part of ε , but not of the full composition.

useful interpretations. The converse operator \smile can, for some deltas d , provide a delta d^\smile that acts as an undo-operation. The full element \top represents the delta that accepts all products, but then guarantees nothing about the output; it discards all information. The negation operator $-$, somewhat less intuitively, will provide a delta d^- that can perform only the modifications that delta d cannot (and vice versa).

- **2.39. Notation:** We can include available algebraic operators in the deltoid tuple, e.g., $(\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, \sqcap, \perp, \llbracket - \rrbracket)$ (dropping the inner parentheses), following the convention of notationally confusing an algebra with its carrier (Notation 1.31). This allows us to directly designate a notation for the available operators. ┘

2.6.1 Monoids

Delta composition, more than anything else, allows us to focus on deltas in this thesis rather than on products. Instead of seeing incremental application as a product undergoing a series of delta applications as in $\llbracket d_n \rrbracket(\dots \llbracket d_1 \rrbracket(c) \dots)$, we can see it as the application of a single composed delta:

$$\llbracket d_n \cdot \dots \cdot d_1 \rrbracket(c)$$

The composed delta $y \cdot x \in \mathcal{D}$ applies first x and then y . *Delta composition*, like relation composition (Definition 1.11), is read from right to left. The fact that \cdot is interpreted as relation composition, and thus

$$\llbracket y \cdot x \rrbracket(p) = \llbracket y \rrbracket(\llbracket x \rrbracket(p)),$$

makes delta application a *monoid action*.

The *neutral delta* ε is the delta that accepts all products but doesn't do anything, returning them unchanged (Figure 2.6). It is fully defined and deterministic (Definitions 2.25 and 2.26).

Delta monoids are not necessarily commutative. The order in which two deltas are applied is often quite significant. The software delta **replace** operation, for example, *overwrites* a previous value with a new one. The delta that does this last determines the result.

- **2.40. Definition (Commuting Deltas):** In any delta algebra (\mathcal{D}, \cdot) , two deltas $x, y \in \mathcal{D}$ are said to *commute* iff $y \cdot x = x \cdot y$. ┘

See Figure 2.7 for an example of this property. A lack of commutativity between deltas helps define the notion of conflict in Chapter 3.

We define the following derived operations:

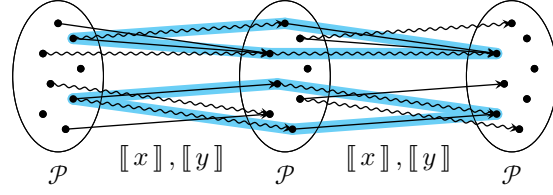


Figure 2.7: A diagrammatic representation of commuting deltas x and y . Highlighting has been added to clarify this commutativity.

► **2.41. Notation:** Given a delta set $D \subseteq \mathcal{D}$, the notations

$$\begin{aligned} D^* &\stackrel{\text{def}}{=} \{x_n \cdot \dots \cdot x_1 \mid x_1, \dots, x_n \in D\} \quad \text{and} \\ D^+ &\stackrel{\text{def}}{=} \{x_n \cdot \dots \cdot x_1 \mid x_1, \dots, x_n \in D \wedge n > 0\} \end{aligned}$$

denote the set of all possible delta compositions from D and the set of all possible non-vacuous delta compositions from D respectively. \lrcorner

By definition we have $\varepsilon \in D^*$. Depending on D , we may also have $\varepsilon \in D^+$.

2.6.2 Boolean Algebras

The semantics of deltas are generally relational and can thus be partially defined or non-deterministic (Section 2.4). The syntactic operators that are able to manipulate that aspect of deltas are those defined in the Boolean algebra (Definition 1.34).

A *delta choice* operator \sqcup represents nondeterministic choice. For example, the term $x \sqcup y$ represents all modifications available when choosing either x or y . When one is not applicable, the other is used instead.

A *delta consensus* operator \sqcap can express the set of modifications that two given deltas *agree* on. The delta $x \sqcap y$ is only applicable if both x and y are, and, when applied, produces a product that might also be produced from applying only x or from applying only y .

An *empty delta* \perp (Figure 2.8) is a delta that does not accept any product. This concept is useful because it can model *invalid* deltas (Definition 2.25).

Error handling

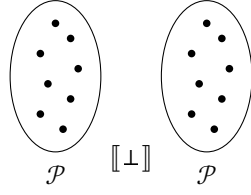
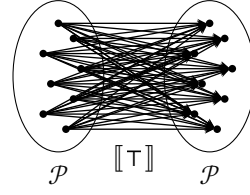
If a delta d cannot be applied to a product p because $p \notin \text{pre} \llbracket d \rrbracket$, it possibly represents an *error* of some sort. Perhaps d was not designed to operate on p in the first place. For example, in the software deltoid we have

$$\llbracket \text{remove } D; \rrbracket (\text{class } C \ \{ \}) = \emptyset.$$

because the product in question does not contain an element D to remove.

If $y \cdot x = \perp$, it could mean that x and y should never be applied one-after-the-other, even if they are individually valid. For example:

$$(\text{add class } C \ \{ \}) \cdot (\text{add class } C \ \{ \}) \equiv \perp$$

Figure 2.8: The empty delta \perp .Figure 2.9: The full delta \top .

because adding two packages with the same name in a row is always invalid. (This will be formalized in Definition 2.48.)

So the empty delta \perp itself represents an *invalid delta*, which is not applicable to any product because of an internal error. In software deltas this is indicated by the **error** placeholder. A composition with an invalid delta is, itself, invalid.

Constructivism

The remaining operators from the Boolean algebra, *negation* \neg and the *full element* \top (Figure 2.9) have an intriguing property. In contrast to the other operators, they are not *constructive* (or *intuitionistic*). Boolean algebras cannot serve as a semantic model for constructive logic, as they can be used to deduce the law of excluded middle $e \sqcup e^-$ [92].

Constructivism is a particularly useful property for modeling deltas, because constructing things is exactly what deltas are all about. Given some product p , a delta should be able to produce another product predictably (within the bounds of its possibly non-deterministic nature), regardless of the full set of potential products \mathcal{P} . However, product sets such as $\llbracket d^- \rrbracket(p)$ and $\llbracket \top \rrbracket$ could be changed just by extending the set of potential products \mathcal{P} , without changing the definitions of d , p or $\llbracket - \rrbracket$.

Constructive alternatives to Boolean algebras exist, for example, in *Heyting algebras* [92] and *co-Heyting algebras* [34, 177]. Studying their interpretation in delta modeling, and integrating them with full relation algebras, is a topic proposed as future work in Chapter 9.

Syntactic Delta Refinement

A boolean algebra is also a *lattice*. Lattices are not only studied from an algebraic, but also from an order-theoretic point of view. For deltas, this point of view yields a reasonable notion of syntactic delta refinement:

- **2.42. Definition (Syntactic Delta Refinement):** *Syntactic delta refinement* is a preorder $\lesssim \subseteq \mathcal{D} \times \mathcal{D}$ satisfying the following equivalences. For all $x, y \in \mathcal{D}$:

$$\begin{aligned} x \lesssim y &\iff y = x \sqcup y \\ x \lesssim_{\text{tot}} y &\iff y = x \sqcup y \wedge \top \cdot x = \top \cdot y \end{aligned} \quad \lrcorner$$

The semantic interpretation of \lesssim is the subset relation \subseteq . For the total correctness case, the \top element had to be used, which allows a syntactic comparison between the ‘preconditions’ of the two deltas. Alas, in an abstract setting, this cannot be done constructively.

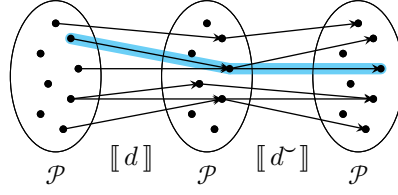


Figure 2.10: A diagrammatic representation of the composition $d^\sim \cdot d$. The highlighted path demonstrates that $d^\sim \cdot d \not\sqsubseteq \varepsilon$.

This leads to a notion of *syntactic delta equivalence*:

- **2.43. Definition (Syntactic Delta Equivalence):** *Syntactic delta equivalence* is an equivalence relation $\simeq \subseteq \mathcal{D} \times \mathcal{D}$ satisfying the following equivalences. For all $x, y \in \mathcal{D}$:

$$x \simeq y \iff x \lesssim y \wedge x \gtrsim y \quad \lrcorner$$

If the rules of Definition 2.38 are followed, these syntactic relations will always be subsets of their semantic counterparts introduced in Section 2.5:

- 2.44. Lemma:** For any given deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$ well-structured under a lattice order \lesssim , we have:

$$\begin{aligned} \lesssim & \subseteq \sqsupseteq \\ \lesssim_{\text{tot}} & \subseteq \sqsupseteq_{\text{tot}} \\ \simeq & \subseteq \equiv \end{aligned} \quad \square$$

2.6.3 Relation Algebra

Last is the full relation algebra, which adds the *converse* operator $^\sim$ (Definition 1.35). An implementation of this operator would, quite simply, reverse the arrows on a delta's relation diagram (Figure 2.10).

This does *not* mean d^\sim will always reverse the effects of d ; it is not a universal undo-operation. At least, not for all deltas. This is simply because some deltas are fundamentally not ‘undoable’. Software deltas that overwrite a value, for example, have no memory to restore that value when converted. A delta d is undoable when it satisfies the following:

$$d^\sim \cdot d \sqsupseteq \varepsilon$$

Figure 2.10 shows that this is not the case for all deltas. For a delta to be ‘undoable’, it needs to be semantically one-to-one (Definition 1.13). In other words, it and its converse need to be deterministic.

In Darcs patch theory [97] it is required that all *patches* have this property, so that any modification can be reversed. This is accomplished by tailoring each patch to the product that it modifies. Their **replace** operation, for example, includes the old value as well as the new. They can only be applied if the old values match, and are therefore undoable.

This thesis won’t consider the converse operator any further. But its possible rôle in ADM is an interesting topic for future work.

2.6.4 The Algebraic Software Deltoid

We now apply the concepts of this section to the software deltoid of Section 2.3.

The Software Delta Algebra

Software deltas support the algebraic signature: $(\mathcal{D}_{\text{pkg}}, \sqcap, \perp, \cdot, \varepsilon)$.

The easiest operators to define are the empty and neutral software deltas:

- ▷ **2.45. Definition (Empty and Neutral Software Deltas):** Define the *empty* and *neutral software deltas* as follows:

$$\begin{aligned} \perp &\stackrel{\text{def}}{=} \{ \text{"id"} \mapsto \mathbf{err} \} \\ \varepsilon &\stackrel{\text{def}}{=} \emptyset \end{aligned} \quad \lrcorner$$

The empty delta needs to have the property that $\llbracket \perp \rrbracket = \emptyset$, so we just choose an arbitrary invalid delta (Definition 2.21). For the neutral delta there was no other candidate than the empty function.

Next, we define syntactic refinement (Definition 2.42), as it will help us define consensus. Software deltas, as defined right now, are deterministic. That means refinement is not all that difficult to define. It would be trivial if not for the **forbid** operation, which can change the precondition of a delta without changing its output. Without **forbid** in the picture, we would have $x \lesssim y$ if and only if $x \simeq y$ (Definition 2.22). As it is, there is still another possibility:

- ▷ **2.46. Definition (Syntactic Software Delta Refinement):** We define *syntactic software delta refinement* $\lesssim \subseteq \mathcal{D}_{\text{pkg}} \times \mathcal{D}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}} \times \mathcal{D}_{\text{cl}}$ as the smallest preorder satisfying the following inference rules. Two deltas $x, y \in \mathcal{D}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}}$ refine each other if they are both invalid:

$$\frac{\text{Err}(x) \wedge \text{Err}(y)}{x \lesssim y} \quad \text{by mutual invalidity}$$

and they refine each other if each identifier is either mapped to the same exact operation, or to operations that refine each other accordingly:

$$\frac{\forall id \in \mathcal{I}: x(id) = y(id) \vee x(id) \lesssim y(id)}{x \lesssim y} \quad \text{by pairwise comparison}$$

with the following refinement rules on the operation level:

$$\frac{}{\mathbf{frb} \lesssim \perp} \quad \text{by forbiddance} \qquad \frac{x \lesssim y}{\mathbf{mod} x \lesssim \mathbf{mod} y} \quad \text{by delegation} \quad \lrcorner$$

So a software delta is only a partial refinement of another if it is equal, or its only difference is that it has additional **forbid** operations.

- ▷ **2.47. Definition (Software Delta Consensus):** *Software delta consensus* is then defined as follows for all $x, y \in \mathcal{D}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}}$:

$$x \sqcap y \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \lesssim y \\ y & \text{if } y \lesssim x \\ \perp & \text{otherwise} \end{cases} \quad \lrcorner$$

Finally we define software delta composition \cdot . This is a bit more involved:

- ▷ **2.48. Definition (Software Delta Composition):** We define *software delta composition* operator $\cdot : \mathcal{D}_{\text{pkg}} \times \mathcal{D}_{\text{pkg}} \rightarrow \mathcal{D}_{\text{pkg}}$ in a few stages: (a) we eliminate the case where one of the operands is invalid, (b) we define composition for valid delta operations on both the package and class levels, and then (c) we lift that definition to the software deltas themselves.

a. Invalid deltas and delta operations

We get invalid deltas and invalid delta operations out of the way as follows. For all invalid deltas $e \in \text{Err}$ and all deltas $d \in \mathcal{D}_{\text{pkg}}$:

$$\begin{aligned} e \cdot d &\stackrel{\text{def}}{=} \perp \\ d \cdot e &\stackrel{\text{def}}{=} \perp \end{aligned}$$

An invalid delta composed with any other delta results in an invalid delta.

b. Valid delta operations — package and class levels

We now define composition for class-level and package-level delta operations at the same time, using the following table. The left column is a case distinction on a delta operation $op_2 \in (\mathcal{OP}_{\text{pkg}} \cup \mathcal{OP}_{\text{cl}})$. The top row is a case distinction on a delta operation $op_1 \in (\mathcal{OP}_{\text{pkg}} \cup \mathcal{OP}_{\text{cl}})$. The inner cells form op_{21} so that:

$$op_2 \cdot op_1 \stackrel{\text{def}}{=} op_{21}$$

	add p_1	rep p_1	mod d_1	rem	frb
add p_2	err	err	err	rep p_2	add p_2
rep p_2	add p_2	rep p_2	rep p_2	err	err
mod d_2	add $a(d_2, p_1)$	rep $a(d_2, p_1)$	mod $d_2 \cdot d_1$	err	err
rem	frb	rem	rem	err	err
frb	err	err	err	rem	frb

The ‘a’ in the table above is an abbreviation of ‘apply’, the software delta application function (Definition 2.19).

c. Valid deltas — package and class levels







Finally, we lift the operator level definition to deltas. For all deltas $x, y \in (\mathcal{D}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}})$ and all identifiers $id \in \mathcal{ID}$:

$$(x \cdot y)(id) \stackrel{\text{def}}{=} x(id) \cdot y(id) \quad \lrcorner$$

Combining the above, we have the software delta algebra:

- ▷ **2.49. Definition (Software Delta Algebra):** The *software delta algebra* is the quotient algebra $(\mathcal{D}_{\text{pkg}}, \sqcap, \perp, \cdot, \varepsilon)$ under \simeq (Definition 2.22) with \mathcal{D}_{pkg} from Definition 2.16 and the operators \sqcap, \perp, \cdot and ε from Definitions 2.45, 2.47 and 2.48. \lrcorner

We now prove that this algebra is well-behaved by establishing a number of required results from the syntactic domain. Each was proved with the Coq proof assistant³:

- ▷ **2.50. Lemma:** Software delta consensus \sqcap respects equivalence \simeq . □ 
- ▷ **2.51. Lemma:** Software delta composition \cdot respects equivalence \simeq . □ 
- ▷ **2.52. Lemma:** Software delta composition \cdot is associative. □ 
- ▷ **2.53. Lemma:** ε is an identity element for software delta composition \cdot . □ 
- ▷ **2.54. Lemma:** \perp is an absorbing element for software delta consensus \sqcap . □ 
- ▷ **2.55. Lemma:** \perp is an absorbing element for software delta composition \cdot . □ 

Because the software delta operations satisfy the above properties, they satisfy the requirements of Definition 2.38.

2.7 Classification of Deltoids

It is sometimes useful to group deltoids into classes, both for a good overview, and to formally compare their properties. One dimension in which to classify a deltoid is the algebraic signature it supports, as discussed in the previous section. Another method is to analyze a deltoid in terms of its expressiveness. Section 2.7.1 introduces some semantic classifications based on expressiveness properties. Finally, Section 2.7.2 introduces the semantic notion of deltoid refinement. It then proceeds to introduce three useful classifications based on refinement.

2.7.1 Classification Based on Expressiveness

We consider a number of expressiveness properties. Expressiveness of a deltoid is measured by what kind of product modifications can be expressed by \mathcal{D} .

For interests sake, here is the strongest possible expressiveness property:

- ▷ **2.56. Definition (Fully Expressive Deltoids):** The class *Full* of *fully expressive* deltoids contains those deltoids for which semantic delta evaluation is *surjective* (Definition 1.13). Formally, for all deltoids $Dt = (\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$:

$$Dt \in \text{Full} \quad \stackrel{\text{def}}{\iff} \quad \text{img } \llbracket - \rrbracket = \mathcal{P} \times \mathcal{P}$$

(See Notation 2.13 and Definition 2.28.) \lrcorner

³Some of these proofs are written out fully in the ADM journal article [2] and the technical report that accompanied the earlier conference paper [9]. Software deltas were simpler then—their operations did not have preconditions—but each proof still took up several pages.

This class would comprise those deltoids with a set of deltas rich enough to obtain any possible semantic delta. But this is a theoretical property, not achievable for deltas with a finite syntactic representation and an infinite set of products [127].

The following weaker property states that any product can be mapped to any other product by applying the proper delta:

- **2.57. Definition (Maximal Expressiveness):** The class **Max** of *maximally expressive* deltoids is defined as follows, for all deltoids $Dt = (\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$:

$$Dt \in \mathbf{Max} \stackrel{\text{def}}{\iff} \forall p, p' \in \mathcal{P}: (\{p\} \Rightarrow_{\text{tot}} \{p'\}) \neq \emptyset$$

(See Definition 2.31.) ┘

This property ensures that we can reach any product from any other product.

The expressiveness of a particular deltoid can also be characterised in terms of the existence of an element in the product set from which all products can be generated:

- **2.58. Definition (Initial Product):** Given a deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, a product $0 \in \mathcal{P}$ is an *initial product*, indicated by the predicate $\text{init} \subseteq \mathcal{P}$, iff:

$$\text{init}(0) \stackrel{\text{def}}{\iff} \forall p \in \mathcal{P}: \exists d \in \mathcal{D}: \llbracket d \rrbracket(0) = \{p\} \quad \text{┘}$$

The existence of an initial product indicates that the delta set is sufficiently expressive to describe any product. They are therefore ideal candidates for the rôle of core product in a product line (more about this in Chapter 4).

- **2.59. Definition:** The class of deltoids that have an initial product is denoted **Init**. ┘

Not every deltoid has an initial product. An effective illustration of this is the following small example deltoid, which inspired the chapter illustration on Page 30.

- **2.60. Definition (Stone Carving Deltoid):** Imagine $Dt_{\text{sc}} = (\mathcal{P}_{\text{sc}}, \mathcal{D}_{\text{sc}}, \cdot, \varepsilon, \llbracket - \rrbracket)$, a deltoid which models the art of stone-carving. A product $p \in \mathcal{P}_{\text{sc}}$ contains an (infinitely dense) set of coordinates in 3-dimensional space, describing a stone sculpture. A delta $d \in \mathcal{D}_{\text{sc}}$ is the set of coordinates from which excess material should be carved away.

The set of deltas is defined simply as follows:

$$\mathcal{D}_{\text{sc}} \stackrel{\text{def}}{=} \text{Pow}(\mathbb{R}^3)$$

But the coordinates comprising a sculpture are finitely bounded in all directions. After all, it is unrealistic to model a slab of marble infinite in size:

$$\mathcal{P}_{\text{sc}} \stackrel{\text{def}}{=} \{ B \subseteq \mathbb{R}^3 \mid \exists l \in \mathbb{R}: \forall (X, Y, Z) \in B: -l < X, Y, Z < l \}$$

Delta application $\llbracket - \rrbracket(-) \stackrel{\text{def}}{=} \setminus$ is set difference, composition $\cdot \stackrel{\text{def}}{=} \cup$ is set union and the neutral delta $\varepsilon \stackrel{\text{def}}{=} \emptyset$ is the empty set. Stone carving deltas can only carve material away, but cannot sincerely add new material back on.⁴ ┘

⁴For readers who do not get the joke: A popular folk etymology proposes that the word “sincere” derives from the Latin “sine sera”, meaning, “without wax”. When unethical Roman stonemasons accidentally chipped a marble sculpture, they would fill it in with wax to cover the flaw, or so the story goes. Modern etymologists have since debunked this theory. [158]

- **2.61. Lemma:** The stone carving deltoid has no initial product. Because stone sculptures are finite in size, whichever one is chosen as a candidate initial product, there will exist one which is larger. And because stone carving deltas cannot make a sculpture larger, there cannot be an initial product. \square

The existence of an initial product would allow us to reason about deltas without having to talk about products at all. An incremental application $\llbracket d_n \cdot \dots \cdot d_1 \rrbracket(c)$ could also be expressed based on an initial product 0 and a delta d_c such that $\llbracket d_c \rrbracket(0) = \{c\}$. This allows us to instead write

$$\llbracket d_n \cdot \dots \cdot d_1 \cdot d_c \rrbracket(0),$$

disregard products completely and focus on compositions like $d_n \cdot \dots \cdot d_1 \cdot d_c$.

Finally, we connect this notion with the notion of maximal expressiveness:

- 2.62. Lemma:** Every product in a maximally expressive deltoid is initial. Conversely, any deltoid in which every product is initial is maximally expressive.

Proof: Assume a maximally expressive deltoid $(\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$, so then:

$$\begin{aligned} \forall p, p' \in \mathcal{P}: (\{p\} \mapsto_{\text{tot}} \{p'\}) &\neq \emptyset && \iff (\text{Def. 2.31}) \\ \forall p, p' \in \mathcal{P}: \exists d \in \mathcal{D}: \emptyset \subset \llbracket d \rrbracket(p) \subseteq \{p'\} &&& \iff (\text{Def. 1.1}) \\ \forall p, p' \in \mathcal{P}: \exists d \in \mathcal{D}: \llbracket d \rrbracket(p) &= \{p'\} && \square \end{aligned}$$

2.7.2 Classification Based on Deltoid Refinement

Another way to classify deltoids, compare them and justify transferring results from one to another, is to use a notion of refinement based on homomorphisms:

- **2.63. Definition (Deltoid Refinement):** A deltoid $Dt_1 = (\mathcal{P}_1, \mathcal{D}_1, \llbracket - \rrbracket)$ is said to *refine* another deltoid $Dt_2 = (\mathcal{P}_2, \mathcal{D}_2, \llbracket - \rrbracket)$, denoted

$$Dt_1 \sqsupseteq Dt_2$$

iff there exists a delta homomorphism $\beta: \mathcal{D}_1 \rightarrow \mathcal{D}_2$ which preserves the algebraic axioms from Dt_1 and a product homomorphism $\alpha: \mathcal{P}_1 \rightarrow \mathcal{P}_2$ such that for all products $p, q \in \mathcal{P}_1$ and all deltas $d \in \mathcal{D}_1$:

$$p \llbracket d \rrbracket q \implies \alpha(p) \llbracket \beta(d) \rrbracket \alpha(q)$$

The pair (α, β) is called a *deltoid homomorphism*. When $\alpha = \text{id}_{\mathcal{P}_1}$ we can also call β by itself a deltoid homomorphism. \lrcorner

We can classify deltoids by their refinement of specific prototypical deltoids. We follow up with three such classifications:

- **2.64. Definition (Relational Deltoids):** The class Rel of *relational deltoids* is defined as follows, for all deltoids Dt :

$$Dt \in \text{Rel} \stackrel{\text{def}}{\iff} Dt \sqsupseteq (\mathcal{P}, \mathcal{P} \times \mathcal{P}, \text{id}_{\mathcal{P} \times \mathcal{P}}) \quad \lrcorner$$

We then move to the class of deltoids with deltas that are always deterministic, but still possibly partially defined — the class in which deltas represent partial functions (Definition 1.17).

Potential Operators									
Rel	\sqcup	\sqcap	$-$	\perp	\top	\cdot	ε	\smile	
PFun		\sqcap		\perp		\cdot	ε	\smile	
Fun						\cdot	ε	\smile	

Table 2.11: This table summarizes potential support for the relation algebra operators by the relational (Rel), partially functional (PFun) and functional (Fun) deltoid classes. It is assumed that any deltoid under consideration has a ‘realistic variety’ of products and deltas: at least three products and at least two deltas that produce different outputs when applied to the same product.

- **2.65. Definition (Partially Functional Deltoids):** The class PFun of *partially functional deltoids* is defined as follows, for all deltoids Dt :

$$Dt \in \text{PFun} \quad \stackrel{\text{def}}{\iff} \quad Dt \sqsupseteq (\mathcal{P}, \mathcal{P} \multimap \mathcal{P}, \text{id}_{\mathcal{P} \multimap \mathcal{P}}) \quad \lrcorner$$

And the next logical step is the class of deltoids with deltas that are always deterministic *and* fully defined — the class in which deltas represent functions (Definition 1.16).

- **2.66. Definition (Functional Deltoids):** The class Fun of *functional deltoids* is defined as follows, for all deltoids Dt :

$$Dt \in \text{Fun} \quad \stackrel{\text{def}}{\iff} \quad Dt \sqsupseteq (\mathcal{P}, \mathcal{P} \rightarrow \mathcal{P}, \text{id}_{\mathcal{P} \rightarrow \mathcal{P}}) \quad \lrcorner$$

We close off with some observations about these classes:

- 2.67. Lemma:** The class Rel encompasses all possible deltoids, because the semantic evaluation operator $\llbracket - \rrbracket$ (Definition 2.11) is, by definition, a deltoid homomorphism from Dt to the ‘proper relation deltoid’ $(\mathcal{P}, \mathcal{P} \times \mathcal{P}, \text{id}_{\mathcal{P} \times \mathcal{P}})$.

And for completeness sake: a functional deltoid is also a partially functional deltoid, and a partially functional deltoid is also a relational deltoid:

$$\text{Fun} \subset \text{PFun} \subset \text{Rel} \quad \square$$

- **2.68. Lemma:** The software deltoid (Definition 2.18) is partially functional, but not functional:

$$Dt \in \text{PFun} \setminus \text{Fun}$$

The stone-carving deltoid (Definition 2.60) *is* functional:

$$Dt_{\text{sc}} \in \text{Fun} \quad \square$$

Finally, we list some practical correlations between these refinement-based classifications above and the algebraic classifications based on Definition 2.38.

A *relational deltoid* can potentially support all operators of the relation algebra (Definition 1.35).

Any practical *partially functional* deltoid, however, cannot support the \top , $-$ or \sqcup operators. If it has at least two products, \top is non-deterministic. If it has at least three products and one delta d , either d or d^- would logically be non-deterministic. And if any two of its deltas x, y produce different outputs when applied to the same product, their union $x \sqcup y$ would be non-deterministic.

Similarly, a practical functional deltoid can, additionally, not support \perp or \sqcap . If it has at least one product, \perp is not fully defined. And if any two of its deltas x, y behave differently, their intersection $x \sqcap y$ would not be fully defined.

This information is summarized in Table 2.11.

2.8 Encoding Related Approaches

A number of other approaches describing the underlying structure of software product lines have been proposed [17, 32]. They formalize the mechanisms underlying AHEAD [31], GenVoca [30] and FeatureHouse [15].

Their approach exhibits some similarities and some differences to ours. Like us, they distinguish between a semantic and a syntactic notion of product transformation, the latter of which is structured algebraically.

However, their algebraic treatment is limited to the realm of monoids. In particular, their transformations cannot be partially defined or nondeterministic (Section 2.4).

Additionally, whereas they equate those product transformations with *features*, we do not make such a claim for deltas. In Chapter 4 we introduce our own notion of feature, which can structure and combine deltas in more flexible ways.

Furthermore, they model products as *Feature Structure Trees (FSTs)* (which are essentially abstract syntax trees with a coarser granularity) and distinguish between two types of what we call deltas: *introductions* and *modifications*, which respectively model FST *superimposition* —which merges two trees into one— and *quantification and weaving* — which targets a specific node using some query language and performs a specific change there. In contrast, the formalism of products and deltas presented in this chapter is more abstract and algebraically simpler, since we assume a single, unified collection of deltas.

In this section we analyze the *quark model* [17] in terms of the concepts of this chapter. We first did this in 2010 [1, 2] (also encoding the algebraically similar system of *Finite Map Spaces* [32]), but this section adapts it to the more recent formulation of ADM.

Section 2.8.1 provides an overview of quarks and Section 2.8.2 encodes quarks within the ADM setting.

2.8.1 The Quark Model

In this subsection we introduce the basic notions of introductions, modifications and quarks as introduced by Apel et al. [17]. They don't formalize FSTs with any detail. They point out the isomorphism between introductions and FSTs and, during their algebraic description, focus on introductions and modifications only, so we will too.

By a stroke of luck, their notation is entirely separate from ours, so this section can faithfully preserve both notations.

Introductions

2.69. Definition (Introductions): *Introductions* form a monoid $(I, \oplus, \mathbf{0})$ with a ‘distant idempotence’ property (see Axiom **c** below), where I is a set of *introductions*, $\oplus: I \times I \rightarrow I$ is the *introduction sum* operator and $\mathbf{0}$ is the *empty introduction*, satisfying the following axioms for all $i_1, i_2, i_3 \in I$:

- a. associativity: $(i_3 \oplus i_2) \oplus i_1 = i_3 \oplus (i_2 \oplus i_1)$
- b. identity element $\mathbf{0}$: $\mathbf{0} \oplus i_1 = i_1 = i_1 \oplus \mathbf{0}$
- c. distant idempotence: $i_1 \oplus i_2 \oplus i_1 = i_2 \oplus i_1$ ┘

2.70. Lemma (Direct Idempotence of Sum): For all $i \in I$, we have $i \oplus i = i$ by taking $i_2 = \mathbf{0}$ in Axiom **2.69c**. □

2.71. Definition (Introduction Equivalence): *Introduction equivalence* $\sim \subseteq I \times I$ is an equivalence relation defined as follows for all introductions $i_1, i_2 \in I$:

$$i_1 \sim i_2 \iff i_1 \oplus i_2 \oplus i_1 = i_1 \quad \text{┘}$$

2.72. Lemma (Quasi-commutativity w.r.t. \sim): We have $i_1 \oplus i_2 \sim i_2 \oplus i_1$ by applying Axiom **2.69c** to Definition **2.71**. □

From here on, we will be working with the quotient algebra I/\sim (Definition **1.32**)—which is, by Lemma **2.72**, a commutative monoid—and relying on implicit canonical projection (Notation **1.27**).

Modifications

2.73. Definition (Modifications): *Modifications* form a monoid $(M, \otimes, \mathbf{1})$, where M is a set of *modifications*, $\otimes: M \times M \rightarrow M$ is the *modification product* operator and $\mathbf{1}$ is the *identity modification*, satisfying the following axioms for all $m_1, m_2, m_3 \in M$:

- a. associativity: $(m_3 \otimes m_2) \otimes m_1 = m_3 \otimes (m_2 \otimes m_1)$
- b. identity element $\mathbf{1}$: $\mathbf{1} \otimes m_1 = m_1 = m_1 \otimes \mathbf{1}$ ┘

2.74. Definition (Modification Application): Given introduction monoid $(I, \oplus, \mathbf{0})$ and modification monoid $(M, \otimes, \mathbf{1})$, *modification application* is a binary operator $\odot: M \times I \rightarrow I$ satisfying the following axioms for all $i_1, i_2 \in I$ and all $m_1, m_2 \in M$:

- a. \odot distributes over \oplus : $m_1 \odot (i_2 \oplus i_1) = (m_1 \odot i_2) \oplus (m_1 \odot i_1)$
- b. identity modification $\mathbf{1}$: $\mathbf{1} \odot i_1 = i_1$
- c. iterative application \otimes : $(m_2 \otimes m_1) \odot i_1 = m_2 \odot (m_1 \odot i_1)$ ┘

Axiom **2.74c** makes \odot a monoid action.

Quarks

Introductions and modifications are combined in the quark model, which defines composition on a set of *quarks* Q , corresponding roughly to our deltas. They define four different kinds of quarks, each with a composition operator $\diamond: Q \times Q \rightarrow Q$ that behaves differently for each kind of quark. They also define an *empty quark*, which does not perform any transformations, and a way to extract the introduction from a quark corresponding to the ‘end product’. They do not introduce a notation for those last two concepts, so we use $\mathbf{1}_Q \in Q$ and $\text{image}: Q \rightarrow I$ respectively.

Three of the kinds of quarks (simple quarks, local quarks and global quarks) are essentially a restriction on the fourth (full quarks) so, to save space, that is how we define them.

- 2.75. Definition (Full Quarks):** Given an introduction monoid $(I, \oplus, \mathbf{0})$, a modification monoid $(M, \otimes, \mathbf{1})$ and a modification application operator \odot , *full quarks* form a magma⁵ $(Q_f, \diamond, \mathbf{1}_Q, \text{image})$ where $Q_f \stackrel{\text{def}}{=} M \times I \times M$ is a set of *full features* defined as triples $\langle g, i, l \rangle$ —containing a *global* modification g , an introduction i , and a *local* modification l —, the empty feature $\mathbf{1}_Q \stackrel{\text{def}}{=} \langle \mathbf{1}, \mathbf{0}, \mathbf{1} \rangle$ is the triple of respective identity elements, the projection function $\text{image}: Q_f \rightarrow I$ is defined as follows for all introductions $i \in I$ and all modifications $g, l \in M$:

$$\text{image}(\langle g, i, l \rangle) \stackrel{\text{def}}{=} i,$$

and quark composition $\diamond: Q_f \times Q_f \rightarrow Q_f$ is defined as follows for all introductions $i_1, i_2 \in I$ and all modifications $g_1, g_2, l_1, l_2 \in M$:

$$\langle g_2, i_2, l_2 \rangle \diamond \langle g_1, i_1, l_1 \rangle \stackrel{\text{def}}{=} \langle g_2 \otimes g_1, (g_2 \otimes g_1) \odot (i_2 \oplus (l_2 \odot i_1)), l_2 \otimes l_1 \rangle$$

┘

- 2.76. Definition (Local Quarks):** *Local quarks* $(Q_l, \diamond, \mathbf{1}_Q)$ are a restriction on full quarks, disallowing global modifications. The set $Q_l \stackrel{\text{def}}{=} \{ \mathbf{1} \} \times I \times M$ contains triples $\langle \mathbf{1}, i, l \rangle$ which are abbreviated to $\langle i, l \rangle$. ┘

- 2.77. Lemma:** Local quark composition —derived from Definitions 2.75 and 2.76— work as follows for all $i_1, i_2 \in I$ and all $l_1, l_2 \in M$:

$$\langle i_2, l_2 \rangle \diamond \langle i_1, l_1 \rangle = \langle i_2 \oplus (l_2 \odot i_1), l_2 \otimes l_1 \rangle \quad \square$$

- 2.78. Definition (Global Quarks):** *Global quarks* $(Q_g, \diamond, \mathbf{1}_Q)$ are a restriction on full quarks, disallowing local modifications. The set $Q_g \stackrel{\text{def}}{=} M \times I \times \{ \mathbf{1} \}$ contains triples $\langle g, i, \mathbf{1} \rangle$ which are abbreviated to $\langle i, g \rangle$. ┘

- 2.79. Lemma:** Global quark composition —derived from Definitions 2.75 and 2.78— work as follows for all $i_1, i_2 \in I$ and all $g_1, g_2 \in M$:

$$\langle i_2, g_2 \rangle \diamond \langle i_1, g_1 \rangle = \langle (g_2 \otimes g_1) \odot (i_2 \oplus i_1), g_2 \otimes g_1 \rangle \quad \square$$

- 2.80. Definition (Simple Quarks):** *Simple quarks* $(Q_s, \diamond, \mathbf{1}_Q)$ are the intersection between local quarks and global quarks (in that they disallow all modifications and are thus, essentially, introductions). The set $Q_s \stackrel{\text{def}}{=} \{ \mathbf{1} \} \times I \times \{ \mathbf{1} \}$ contains triples $\langle \mathbf{1}, i, \mathbf{1} \rangle$ which are abbreviated to i . ┘

⁵A *magma* is an algebraic structure with a binary operator that need not be associative.

2.81. Lemma: Simple quark composition \blacklozenge —derived from Definitions 2.75 and 2.80—is the same as introduction sum \oplus (Definition 2.69). \square

They further observe that while local quarks (and, of course, simple quarks) form a monoid, composition for global and full quarks has no identity element and is not even associative. This is due to the fact that global modifications are applied multiple times — at least once for every composition. For example, global quark composition produces results such as the following (underlining specific segments to call attention to them):

$$\begin{aligned} (\langle i_3, g_3 \rangle \blacklozenge \langle i_2, g_2 \rangle) \blacklozenge \langle i_1, g_1 \rangle &= \\ \langle (g_3 \otimes g_2 \otimes g_1) \odot ((\underline{(g_3 \otimes g_2)} \odot (i_3 \oplus i_2)) \oplus i_1), g_3 \otimes g_2 \otimes g_1 \rangle \\ \langle i_3, g_3 \rangle \blacklozenge (\langle i_2, g_2 \rangle \blacklozenge \langle i_1, g_1 \rangle) &= \\ \langle (g_3 \otimes g_2 \otimes g_1) \odot (\underline{i_3 \oplus ((g_2 \otimes g_1) \odot (i_2 \oplus i_1))}), g_3 \otimes g_2 \otimes g_1 \rangle \end{aligned}$$

To make global quark composition behave they propose to make modification composition \otimes (Definition 2.73) distantly idempotent and commutative, which would grant associativity. This is quite a strong restriction, however, excluding useful modifications such as method wrapping. We will not pursue this proposal.

2.8.2 Quarks as Deltas

We now wrap quarks into our notion of deltoid (Definition 2.11), to make the relation between our two formalisms explicit. We say ‘wrap’ rather than encode because we use a very straightforward interpretation of quarks as deltas. This allows us to analyze quarks using our own measures.

▷ **2.82. Definition (Quark Deltoid):** Given a commutative introduction monoid I , a modification monoid $(M, \mathbf{1})$, a modification application operator and associated quark magma $(Q, \blacklozenge, \text{image})$, we define the *quark deltoid* $Dt_Q = (\mathcal{P}, \mathcal{D}, \llbracket - \rrbracket)$ where the set of products $\mathcal{P} \stackrel{\text{def}}{=} I$ consists of all introductions, the set of deltas $\mathcal{D} \stackrel{\text{def}}{=} Q$ consists of all quarks and delta evaluation $\llbracket - \rrbracket: \mathcal{D} \rightarrow \text{Pow}(\mathcal{P} \times \mathcal{P})$ is defined as follows for all deltas $\langle g, i, l \rangle \in \mathcal{D}$ and all products $p \in \mathcal{P}$:

$$\llbracket \langle g, i, l \rangle \rrbracket(p) \stackrel{\text{def}}{=} \{ \text{image}(\langle g, i, l \rangle \blacklozenge \langle \mathbf{1}, p, \mathbf{1} \rangle) \} \quad \lrcorner$$

Finally, we classify quarks in our own context:

▷ **2.83. Theorem:** Every quark deltoid Dt_Q is functional: $Dt_Q \in \text{Fun}$

Proof: By Definition 2.82, every semantic delta $\llbracket d \rrbracket \in \llbracket \mathcal{D}_{Dt_Q} \rrbracket$ is uniquely and fully defined (Definitions 1.13 and 1.16), so the deltoid homomorphism we need for the proof is simply $\llbracket - \rrbracket: \mathcal{D} \rightarrow (\mathcal{P} \rightarrow \mathcal{P})$, its output interpreted as a function (rather than a relation). \square

This demonstrates that, indeed, all quarks are fully defined and deterministic. According to Table 2.11, functional deltoids can potentially support the monoid operators and converse operator in a well-behaved manner (Definition 2.38). Which are actually supported by the four types of quarks? We confirm what Apel et al. [17] stated about this:

- ▷ **2.84. Theorem:** The simple and local quark deltoids support the monoid operators, but not the converse operator. The global and full quark deltoids have a neutral element (the empty feature 1_Q), but don't support associative composition. \square

2.9 Conclusion

At the beginning of this chapter we stated two goals: *feature modularity* and *separation of concerns*, a duality we aim for with the delta modeling approach. This chapter thoroughly explores the interaction between deltas and the interaction between a delta and a product, thereby introducing the fundamentals of *Abstract Delta Modeling (ADM)*, built upon by chapters to follow. The notion of *deltoid* is introduced, which contains the full sets of products and deltas representing a specific domain, as well as the semantics of deltas: how they modify products. By working abstractly, ADM is ready to encode any domain, not limited to specific programming language, nor even to software.

To jumpstart the running example introduced in Section 1.4—the Editor product line—a concrete deltoid was defined based on an object oriented programming domain. Many concepts are illustrated through this example.

Various aspects of delta semantics were discussed, such as *partial definedness*, *non-determinism* and *correctness* with regard to a relational specification. A number of *algebraic operations*—such as composition, choice and consensus—are introduced in order to allow syntactic reasoning over deltas. Certain expressiveness properties and a refinement relation are then introduced in order to classify deltoids by what they can do. Finally, it is shown how deltas can encode quarks, a similar concept introduced in related literature.

2.10 Related Work

Many other approaches have been described for reaching the goals of feature modularity and separation of concerns. They all have their pros and cons, as discovered by the academic and industrial communities. This section discusses a number of those approaches. Some, however, are directly related to topics we discuss in later chapters. In such cases, their exposition is postponed until then.

In 1997, Prehofer [156] first stated that source-code should treat features explicitly, rather than as an emergent property of traditional object oriented programming methodologies. He called this new approach *Feature Oriented Programming*. Since then, a lot has been done in that direction. Apel and Kästner [13] offer a good overview of the progress between then and 2009 which is, incidentally, the exact year that the research underlying this thesis began. It is therefore an excellent reference-point for “pre-ADM” progress in the field.

Section 2.10.1 discusses approaches directly targeted at software product line variability. Aspect Oriented Programming was not so targeted, but has nonetheless been proposed as a suitable tool on many occasions; a possibility explored in Section 2.10.2. Finally, Section 2.10.3 discusses the object-oriented constructs known as *mixins* and *traits*, which have also been suggested as possible solutions.

2.10.1 Variability Approaches

It was Kastner et al. [108] who first classified variability approaches facilitating automated product derivation for software product lines in the two main directions discussed in Sections 1.2.3 and 1.2.4: annotative and compositional.

Annotative techniques, while allowing automated product derivation on a fine-grained level, offer neither modularity nor separation of concerns. Therefore, we'll discuss them in the **Related Work** section of Chapter 4, which is dedicated to features and the automated generation of specific products.

Compositional approaches, on the other hand, are meant specifically to reach the goals of feature modularity and separation of concerns. They gather all code belonging to a feature—or a closely related set of features—into a single module.

An early description of feature modules as a monoid, with notions of composition and a neutral element (Section 2.6.1) came from Batory and O'malley [30], in a technique which they dubbed GenVoca. A GenVoca codebase consists of a number of core programs and a number of feature modules (which they call *features*), which were applied and composed by model superposition.

GenVoca was later generalized by Batory et al. [31] in an approach called AHEAD, primarily to allow a product line to contain more than just source code, and to satisfy two principles: scalability—the ability to consistently refine different representations belonging to the same program—and uniformity—the ability to represent all of those in the same kind of hierarchical structure. They also introduced tool-support. Apel et al. [15] presented FeatureHouse, which aims to implement AHEAD for actual use in software product line engineering, and Kastner et al. [110] introduced FeatureIDE, an IDE for AHEAD-based development. Work on AHEAD and FeatureHouse lead to the notion of model superimposition by Apel et al. [19] as a way to merge code fragments in the composition of feature modules.

The notion of *program delta* was first introduced by Lopez-Herrejon et al. [90] as a general term to describe modifications to object-oriented programs, such as those by AHEAD. Schaefer et al. [163] built on this and proposed a model-based software framework based on a what they called a core-design and a set of Δ -designs, which play a rôle similar to feature modules and, of course, correspond to what we now call deltas. Source-code composition was achieved using frame technology [181]. The main innovation compared to AHEAD was that Δ -designs and features had a many-to-many relationship, basically separating the two concepts and allowing a module to implement arbitrary combinations of features (something we'll discuss in detail in Chapter 4). A problem with this approach is that variation points have to be annotated in the core product and, therefore, known in advance, losing some of the benefits of the compositional approach. This was addressed later, when Schaefer et al. presented Δ -designs as a way to implement software product line variability [160]. They implemented it for Java [164] and described the practices of using an empty program as the core (Definition 2.58) and having all code introduced by deltas [161, 162].

(Abstract) Delta Modeling has now been extended and analyzed in several directions. For instance, Lienhardt and Clarke [120] introduced a row polymorphic type system which checks whether composition of software deltas, as presented in Section 2.6.4, results in an invalid delta.

2.10.2 Aspect Oriented Programming

Aspect Oriented Programming (AOP) [112, 114] has often been suggested as a way to implement features, because aspects can weave any number of code fragments into specified point-cuts from the outside and thus seem, in that regard, as the right tool for the job.

Generally, however, the AOP model only supports the insertion of statements around identified join-points inside methods and the addition of members to an existing class using *inter-type declarations* [12]. No implementation or formalization known to me supports the manipulation of higher level constructs such as classes and packages. In addition, there has not been much support for coordinating the interaction and composition between different concerns, though there has been work attempting to improve this situation [129]. We also note that, as far as we could discover, AOP is not able to *remove* code from an existing base as software deltas can (Section 2.3).

A well-studied programming language with aspects is AspectJTM [68, 113]. It has been evaluated as a tool for implementing features in a number of publications. Lopez-Herrejon et al. [90] note that it lacks a cohesion mechanism (which would allow feature modularity) and a general model for composition, but found it otherwise flexible. Kästner et al. [107] were generally negative about its suitability for implementing features, stating that most of the unique and powerful features it offers were not useful, and report a decrease in code readability and maintainability as the number of features grows.

There have been attempts to combine aspects with other technologies or otherwise extend them for our purposes, usually with more favorable results. Loughran et al. [126] combined AOP with frame technology. Mezini and Ostermann [133] present an extension to AspectJ that includes dedicated feature oriented approaches. Similar approaches were later taken by Völter and Groher [178] and Apel et al. [16], who note that the two approaches are complementary, aspects being useful on a fine-grained level, but other techniques still being necessary for implementing large-scale software building blocks. The algebraic foundation of finite map spaces and Quarks (Section 2.8) by Batory and Smith [32], and later Apel et al. [17], is based on this combination. On a different note, Noda and Kishi [144] find that AOP as a tool for product line development lacks in reusability, and propose a new mechanism to correct this.

All in all, we conclude that while aspects provide separation of concerns—at least to a certain degree—they were not designed for our purpose, and still have some distance to go before they can be considered as a solution for serious feature oriented development.

2.10.3 Mixins and Traits

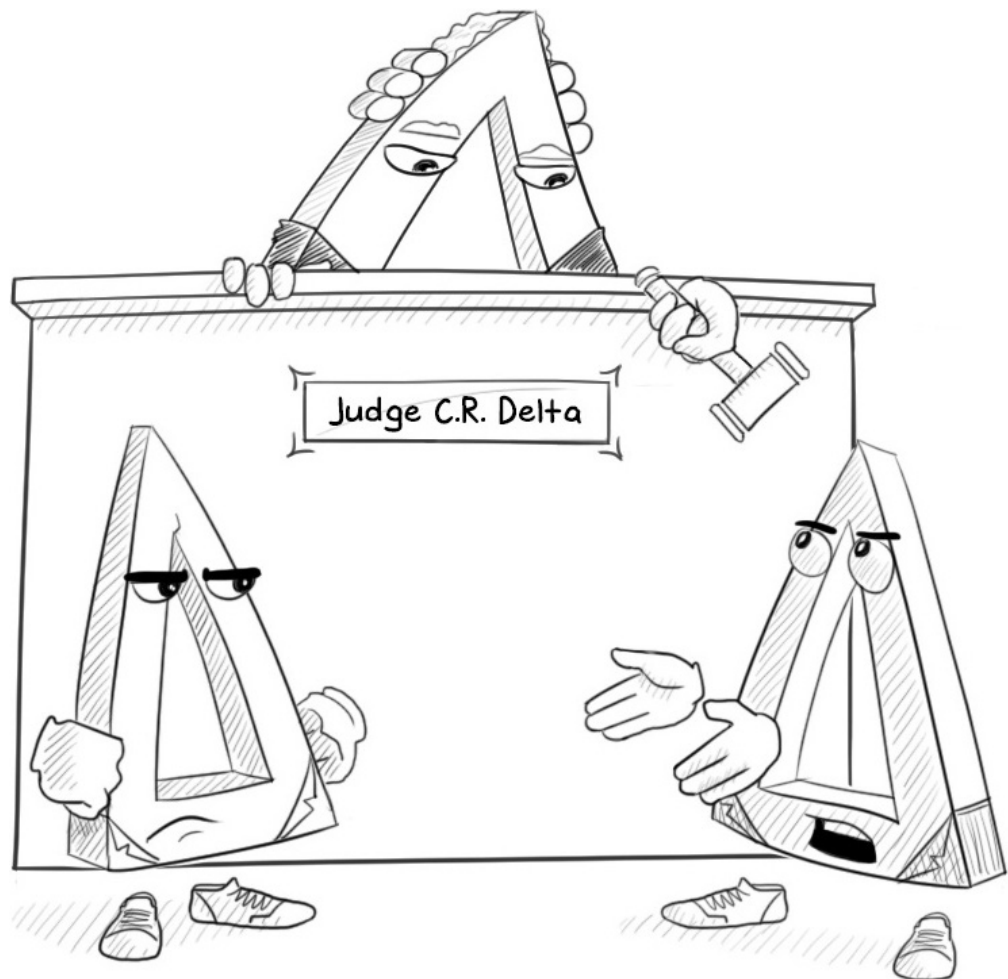
Bracha and Cook [47] first described the general method of mixin-based inheritance, or *mixins*, which basically allows class inheritance to include the addition of extra class members. Smaragdakis and Batory [172] have since proposed a large-scale software refinement technique using mixins. Later, Schärli et al. [67, 165] pointed out a number of shortcomings in mixin-based inheritance and proposed the alternative construct of *traits*, which offer more flexibility with regard to composition and interaction. It appears that traits have since subsumed mixins for reuse purposes.

Bettini et al. [37] describe a way of using traits to implement software product lines. The use of traits for this purpose has also been discussed in the early stages of the HATS project, but the idea was soon dropped in favor of delta modeling. Traits are a mechanism designed for code reuse within a single software product. They are not intended to be the sole mechanism for implementing software product line variability, and appear, in and of themselves, unsuited to the task. Traits are limited to adding methods (and in some formalisms, fields) to new classes. They cannot describe functionality across multiple classes, so they offer insufficient modularity. In addition, they have to be known at the point where a class is first defined and cannot inject functionality from the outside, i.e., they do not support invasive composition.

This is not to say that traits are useless — far from it. [Chapter 9](#) discusses the possibility of regarding deltas and traits as orthogonal and complementary constructs, used together in a single code-base.

Delta Models

On Interaction, Conflict and Conflict Resolution



3.1 Introduction

The previous chapter introduced deltas, the basic feature modules used to build delta-modeling based systems. This chapter explores techniques for organizing a collection of deltas into a robust system. There is a particular focus on the possible *orders* in which a set of modules can be composed. *Delta models* can express design intentions such as priority, interaction and conflict between independently developed deltas by organizing them in a *partial order* (Definition 1.22), an approach not found in previous literature.

3.1.1 Syntax Highlighting and Error Checking

The reason why composition order is so important is the simple fact that feature modules do not necessarily commute (Definition 2.40), and therefore the order in which they are applied can have a direct effect on the final product.

Take, for example, the features *SH* and *EC* from Section 1.4. Both of their implementations have to overwrite the `font(int)` method, each in a different way. So whichever of them is applied last ‘wins’, and the code of the other is (partly) discarded. We need to find a way to properly combine the two.

Goal: *Find a way to mediate between non-commuting feature modules.*

This should be done, of course, without disregarding the goals from the previous chapter: feature modularity and separation of concerns. This is where existing approaches seem to fall short. So to demonstrate the need for a new solution, we’ll attempt to organize the deltas that implement *SH* and *EC* of the Editor product line (Section 1.4) with two existing techniques for feature module organization: AHEAD [31, 32, 124, 125] and ‘pre-ADM’ Delta Oriented Programming [160, 163]. We’ll shine a light on any problems we encounter and, thereby, motivate the work in this chapter. (Since the advent of ADM, both techniques have made some strides in the organizational structure of feature modules; strides which we’ll discuss in Section 3.8.)

3.1.2 AHEAD

The main purpose of AHEAD was to form an algebraic theory of software composition and tools for applying that theory in practice. These are aspects already discussed in Chapter 2. The organization between different modules, though, had been of lower priority.

Batory et al.’s paper on Scaling Step-wise Refinement [31] was strictly algebraic in nature. They recognize the fact that a method cannot be added twice, or removed when absent (as we did in Definition 2.18). They use a similar system of preconditions and postconditions based on earlier work on GenVoca [28], which restricts the application order of feature modules by examining their source-code. But otherwise, application order is not mentioned. The same holds in the later paper by Batory and Smith [32]. Nonetheless, in order to implement tool support [173], a choice had to be made, and it was this: the application order between modules in the AHEAD tool suite is manually supplied on the composer command line [25, 90]. Let us assume that in

practice, each product of interest will have this order encoded in a build-script to avoid having to manually specify it every time. So we can see it as part of the system design.

The conclusion is that AHEAD feature modules are organized in a *total order* (Definitions 1.13 and 1.22), in which later modules can overwrite code from earlier modules. The most naive way of organizing the implementations of the features *EC* and *SH* in AHEAD is illustrated in Figure 3.1a. It doesn't work because the *EC* module discards the *SH* implementation of `font(int)` completely, replacing it with its own.

One solution is to enhance the module implementing *EC* to be aware of *SH* (Figure 3.1b). Rather than add only its own implementation of `font(int)`, it would supplement the existing one left by the *SH* module. If you recall, this is basically what we did manually in the introduction of Chapter 2. The AHEAD toolsuite can do this without duplicating code by using the **super** keyword [31] to reference an implementation from the previous module in the chain.

But there are two problems with this approach. First of all, it breaks separation of concerns; code meant for *EC* is now mixed up with a reference to *SH*, an unrelated feature. Secondly, it is no longer certain that the modules in this code-base can be used to generate a product with *EC* but without *SH*, as the **super** call may no longer make sense. Liu et al. [124] introduced this as the *feature optionality problem*. They later generalized it and dubbed it the *optional feature problem* [125], recognizing that it may be possible for two feature modules to be composable in both orders, but that those orders need not necessarily yield the same result, which is exactly the case for the *EC* and *SH* modules. The optional feature problem has since been thoroughly described by Kastner et al. [111], who summarize a number of possible solutions.

One of those solutions, introduced by Liu et al. [124, 125], was the concept of special *derivative* modules, which contain the code necessary for the interaction between two such modules — though still no mention is made of a more sophisticated organizational structure. A solution using derivative modules is shown in Figure 3.1c. Two main modules independently implement their own feature. A derivative module is applied last to implement their interaction. Separation of concerns has been restored.

But still all is not well. A **super** call can not be used to reference both of the original `font(int)` methods; just the last one in the chain. So it is now necessary to duplicate code, at least from *SH*. Additionally — and more significantly — while it is true that the application order between the two main modules no longer matters, we are forced to choose an order nonetheless. If, at any time in future development, the two modules make another incompatible change, the *SH* feature will be broken again, and this will not be detected. The AHEAD composer is a relatively simple piece of software, not smart enough to realize that the overwrite might be a mistake. The conceptual independence between the *EC* and *SH* modules is simply not expressible with a linear order. We call this *overspecification*.

Goal: Find a way to avoid overspecification of the structural organization between feature modules.

3.1.3 Pre-ADM Delta Oriented Programming

Delta oriented programming —as it was before our work on ADM [160, 163]— had quite a different approach to the problem. Rather than force deltas into a total order, no order could be specified at all. Deltas were applied in an arbitrary order. Therefore, any two deltas that might be applied together were required to commute. Overwriting operations were only allowed to apply to the core product, not to code from other deltas.

This disadvantage was offset by the ability to annotate deltas with very specific application conditions. So any two deltas that did not commute could be given more refined application conditions so that they would never be applied for the same feature selection.

This approach sacrifices some modularity, as the ‘non-conflicting’ parts of the *EC* and *SH* implementations are now separated from their `font(int)` method. But the main disadvantage is code duplication. For the Editor product line, a delta-oriented code base would need to contain three deltas that fully implement the `font(int)` method, one each for the feature configurations $(EC \wedge \neg SH)$, $(\neg EC \wedge SH)$ and $(EC \wedge SH)$. The third configuration leads to the delta selection shown in Figure 3.2.

Goal: Find a way to avoid code duplication through the structural organization between feature modules.

3.1.4 The ADM Solution

ADM was initially developed with the main purpose of solving the optional feature problem, which is what most of this chapter is about. Figure 3.3 shows a preview of the ADM solution, which is situated in between the two polar opposites discussed above: a partial order. Figure 3.3 is best compared with Figure 3.1c. It also uses a derivative module (called a *conflict resolving delta* — a term further explained in Section 3.3). We also make sure it is applied last, so it can overwrite the changes of both d_{EC} and d_{SH} . The two main modules themselves, however, are *not* ordered, as they represent conceptually independent features.

If another conflict ever arises in the future, this can be automatically detected. Furthermore, software deltas can be equipped with a syntax to reference a specific method implementation by name: $\langle \text{delta} \rangle @ \text{method}$, so no code duplication is required to resolve the conflict. A possible implementation would be:

▷ **3.1. Example:** Software delta $d_{SH \wedge EC}$, the “ $d_{SH} \not\leq d_{EC}$ conflict resolver”:

```

1  modify package DeltaEditor {
2      modify class Editor {
3          replace font(c : int) : Font {
4              Font result = new Font();
5              result.setColor      (font@dSH(c).color());
6              result.setUnderlined(font@dEC(c).underlined());
7              return result;
8          };
9      };
10 };

```

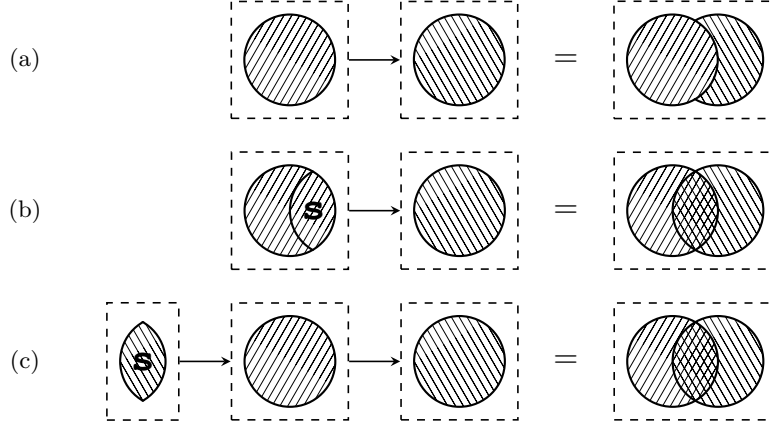


Figure 3.1: Three ways of organizing the EC and SH features into feature modules with AHEAD, in a Venn-diagram representation. The intersection area represents the `font(int)` method. We need the $SH \cap EC$ hybrid version of `font(int)`, so (a) is wrong. A **super** call can reuse code from the next delta in the chain (represented with the **s** symbol). But even so, (b) does not separate concerns, (c) still duplicates code, and both are overspecified.

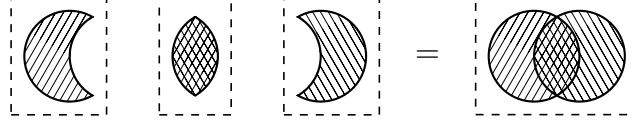


Figure 3.2: One of the ways of organizing the EC and SH features into deltas with pre-ADM delta oriented programming techniques. There is no way to order two deltas that modify the same method, which limits our options.

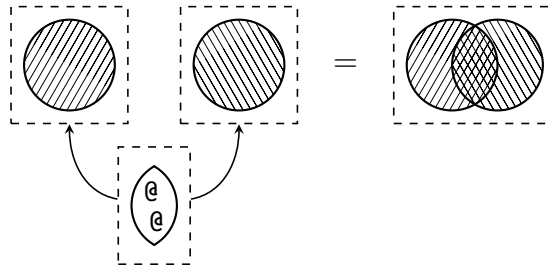


Figure 3.3: The recommended way of organizing the EC and SH features into deltas with ADM. The flexibility of a partial order allows modularity, separation of concerns and avoids overspecification. Internally referencing specific delta implementations from \square with the $@$ operator eliminates code duplication.

3.1.5 Chapter Structure

The remainder of the chapter is structured as follows. Section 3.2 introduces the formal concept of a *delta model*, which embodies a partially ordered organization of deltas, and defines the first of its possible semantics. This semantics requires a *unique derivation*, i.e., the existence of a unique composition compatible with the partial order. In Section 3.3 we discuss the more local concepts of *conflict* and *conflict resolution*, which help ensure a unique derivation.

Section 3.4 revisits the software deltoid of Section 2.3 and enhances it to support fine-grained modification of methods. This will, at times, allow independent deltas to modify the same method without losing commutativity and, therefore, without requiring a conflict resolver. This makes software deltas potentially non-deterministic.

A unique derivation is not always required to obtain a sensible and robust delta model. Section 3.5 introduces *disjunctive semantics* and *conjunctive semantics*: two ways of interpreting a delta model with multiple derivations.

Section 3.6 introduces *nested delta models*, which allow a delta to be a delta model nested inside another, and explains the possible uses of this additional modularization technique.

Sections 3.7 and 3.8 offer concluding remarks and discuss related work.

3.2 The Delta Model

During the remainder of this chapter we assume a given deltoid $Dt = (\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, \llbracket - \rrbracket)$, unless specified otherwise.

This section formally introduces the notion of *delta model*. A delta model contains a *finite set of deltas*, each responsible for modifying a different aspect in the product. In theory, all changes can be encoded in a single delta. But by splitting up the work we may achieve that coveted separation of concerns.

Furthermore, to be able to express certain design intentions such as dependency, interaction and conflict, a delta model organizes these deltas in a *strict partial order* (Definition 1.22), which restricts the order in which they may be applied to a product:

- **3.2. Definition (Delta Model):** A *delta model* is a tuple (D, \prec) , where $D \subseteq \mathcal{D}$ is a finite set of deltas and $\prec \subseteq D \times D$ is their *application order*, a strict partial order on D (Definition 1.22). An ordering $x \prec y$ indicates that x should be applied before y , though not necessarily directly before. The set of all delta models is denoted \mathcal{DM} . If the delta set on which it is based is not clear from context, we attach a subscript as in $\mathcal{DM}_{\mathcal{D}}$. \lrcorner

Figure 3.4 shows a *delta diagram*, which is a representation of a delta model. Figure 1.3 (page 9) also shows a delta diagram, though annotated with additional information. In these diagrams, the deltas are dashed circles (in an abstract setting) or boxes (in a concrete setting) and the partial order is represented by arrows.

The order is intended to capture the intuition that a subsequent delta has full knowledge of (and access to) earlier deltas and more authority over modifications to the product. When two deltas are unordered, such as x and y in Figure 3.4, they represent independent modifications. Neither has priority over the other, but both of them are dominant over w and subordinate to z .

A delta model is applied to a product by sequentially applying each of its deltas in some linear extension of this partial order (Definition 1.24).

As mentioned before, the possibility of setting up a partial application order, rather than a total order [31, 32, 124, 125] or no order at all [160, 163], is important. By allowing these design intentions to be expressed, we set the stage for a formal notion of *conflict* between conceptually independent deltas. We can then talk about how to recognize, avoid and resolve such conflicts (Section 3.3).

The semantics of a delta model are based on its *derivations* — deltas formed by the possible sequential compositions of its partial order:

- **3.3. Definition (Derivation Function):** We define the *derivation function* $\text{derv}: \mathcal{DM} \rightarrow \text{Pow}(\mathcal{D})$, which maps a delta model to the set of its *derivations*, as follows, for all delta models $dm = (D, \prec)$:

$$\text{derv}(dm) \stackrel{\text{def}}{=} \left\{ d_n \cdot \dots \cdot d_1 \mid D = \{d_1, \dots, d_n\} \wedge \forall i, j \in \{1, \dots, n\}: d_i \prec d_j \Rightarrow i < j \right\} \downarrow$$

Figure 3.5 shows the derivations of the delta model in Figure 3.4. Note that if D is empty, then $\text{derv}(dm) = \{\varepsilon\}$, as ε is the identity element of \cdot .

In practice it is often the goal to design a delta model that has exactly one derivation. This corresponds to its deltas being defined and arranged in such a way that they unambiguously specify a product modification together. We typically see this as the mark of a well-designed delta model:

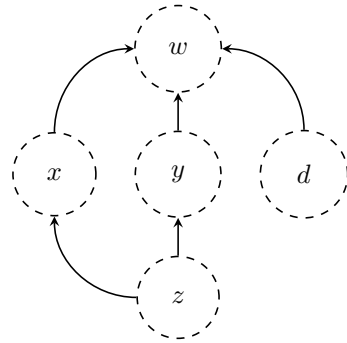


Figure 3.4: A delta diagram of an abstract delta model $dm = (D, \prec)$ with $D = \{d, w, x, y, z\}$ and order $w \prec x \prec z$, $w \prec y \prec z$ and $w \prec d$.

$$\text{derv}(dm) = \left\{ \begin{array}{l} w \cdot d \cdot x \cdot y \cdot z, \\ w \cdot x \cdot d \cdot y \cdot z, \\ w \cdot x \cdot y \cdot d \cdot z, \\ w \cdot x \cdot y \cdot z \cdot d, \\ w \cdot d \cdot y \cdot x \cdot z, \\ w \cdot y \cdot d \cdot x \cdot z, \\ w \cdot y \cdot x \cdot d \cdot z, \\ w \cdot y \cdot x \cdot z \cdot d \end{array} \right\}$$

Figure 3.5: The derivation set of the delta model dm from Figure 3.4.

- **3.4. Definition (Unique Derivation):** A delta model $dm \in \mathcal{DM}$ is said to have a *unique derivation*, denoted $UD(dm)$, iff $d_n \cdot \dots \cdot d_1 = d'_n \cdot \dots \cdot d'_1$ for all pairs of linear extensions (d_1, \dots, d_n) and (d'_1, \dots, d'_n) of \prec . Or equivalently:

$$UD(dm) \stackrel{\text{def}}{\iff} |\text{derv}(dm)| = 1 \quad \lrcorner$$

For example, if deltas x, y and z from Figure 3.4 all commute with each other, and d commutes with z , then all derivations in $\text{derv}(dm)$ from Figure 3.5 are equal, and dm has a unique derivation. The semantics of a delta model with a unique derivation can be formally defined as follows:

- **3.5. Definition (Sole Derivation Semantics):** The semantics of a delta model $dm \in \mathcal{DM}$ with a unique derivation $\text{derv}(dm) = \{d\}$ is defined as follows:

$$\llbracket dm \rrbracket \stackrel{\text{def}}{=} \llbracket d \rrbracket \quad \lrcorner$$

In this situation, applying a delta model to a product just means applying its sole derivation or, in the absence of a composition operator, applying all its deltas in some compatible order. So the ‘apply’ function on the delta model level is easily derived from its delta level counterpart (Definition 2.12, page 37).

- **3.6. Lemma:** If a delta model $dm = (D, \prec)$ has a unique derivation and the deltas in D are all deterministic, $\llbracket dm \rrbracket$ is uniquely defined (Def. 1.13). \square

That means that we are guaranteed a uniquely defined end-product if the delta model is applied to a product that it accepts (Definition 2.24).

It is quite possible for a delta model to have more than one distinct derivation, as we may be working with a noncommutative composition operator. Composition from the software delta algebra (Definition 2.48) is noncommutative, as we have results such as $d_{SH} \cdot d_{EC} \neq d_{EC} \cdot d_{SH}$.

Section 3.5 discusses possible semantics for *ambiguous* delta models. But first, we explore some techniques for maintaining *unambiguity*.

3.3 A Conflict Resolution Model

The property that a delta model has a unique derivation (Definition 3.4) can be checked by brute force. This means generating all derivations (in the worst case, $n!$ derivations for n deltas), and then checking that they all correspond. To allow for a more efficient way to establish this property, and to better reflect developers’ intentions, we introduce the notion of *unambiguous delta model*, which relies on the notions of *conflicting deltas* and *conflict-resolving deltas*.

At this point it is important to explain a crucial distinction. Existing literature on feature oriented programming [29, 124, 140] speaks of *feature interaction*, an undesirable situation in which the final software product will exhibit wrong behavior if two independent features are both implemented. One example is the inclusion of both a fire suppression system and a flood prevention system in a smart building. When a fire is detected, the ceiling sprinklers discharge water. The flood prevention mechanism, sensing an abundance of water on the floor, then proceeds to cut off the water supply.

This thesis recognizes the same notion, though does not comment on the desirability of such feature interaction. The inclusion of both Printing and Syntax Highlighting in the Editor product line, for example, leads us to activate the extra delta $d_{Pr \wedge SH}$ (Section 1.4.2). Initially those two features do not interact, but we *want* them to. It comes down to the same thing. Implementing or preventing this kind of feature interaction is explore more fully in Chapter 7.

In this section we are concerned with *implementational* conflicts, rather than conceptual ones. Such conflicts can always be automatically detected, just as we can automatically detect multiple derivations of a delta model. And while it may require human intervention to resolve a conflict, it can always be detected whether a conflict has indeed been resolved. Neither is possible for feature interactions, which involve complex behavioral notions.

3.3.1 Conflicting Deltas

Two deltas in a delta model are *in conflict* if (1) they do not commute (Definition 2.40, page 50), and (2) no order is imposed between them. Intuitively, two conflicting deltas are independently modifying the same part of a product in different ways, making multiple distinct derivations possible.

- **3.7. Definition (Delta Conflict):** Given a delta model $dm = (D, \prec)$, we introduce the *conflict relation* $\not\prec \subseteq D \times D$. Deltas $x, y \in D$ are said to be *in conflict* iff the following condition holds:

$$x \not\prec y \quad \stackrel{\text{def}}{\iff} \quad y \cdot x \neq x \cdot y \quad \wedge \quad x \not\prec y \quad \wedge \quad y \not\prec x$$

If the delta model is not clear from context, we attach a subscript as in $\not\prec_{dm} \cdot \lrcorner$

The idea, given a concrete deltoid, is to develop a decision procedure for delta commutativity¹ so that conflicts can be automatically detected and developers warned: “two deltas that you deemed mutually independent have conflicting implementations”. If, for example, deltas y and d from Figure 3.4 stop commuting at some point during development, this would not go unnoticed.

When a conflict is present, it may cause multiple distinct derivations. By Definition 3.5 we may only apply a delta model to a product if it has a unique derivation. So for now, let’s assume that we want to create a delta model with a unique derivation. Sections 3.3.2 to 3.3.4 present possible ways to achieve this. More sophisticated methods for dealing with multiple derivations are discussed in Section 3.5.

One way to ensure that conflicts will never occur is to work in a deltoid with an inherently commutative composition operator. But such a deltoid would be severely restricted in the kind of modifications it can express. Any kind of ‘overwriting’ operation, such as those in our software deltoid, would not be possible. Apel et al. [14, 17] explore the possibility of commutative software modifications, and reach the conclusion that this is infeasible for an object oriented domain.

It should be noted that avoiding conflicts by blindly sticking to a total order (with the intention of always having either $x \prec y$ or $y \prec x$; see Definition 1.13) would be akin to sticking our heads in the sand. The fact that some deltas

¹Functionally it would even be enough to have procedures for delta composition and equality, but one dedicated to deciding commutativity can be a great deal more efficient.

ought to act independently would not change. Far from *solving* anything, linearizing such deltas without thinking would *hide* future problems. At any time during development, a delta that is granted priority might begin to inadvertently —and silently— override modifications of other deltas. In software this would cause bugs that are hard to diagnose; a serious issue for approaches such as AHEAD in which a linear order is fundamentally assumed. Such *overspecification* is one of the issues that abstract delta modeling was designed to address.

You might ask: can't we just apply a delta model with a conflict *anyway*? Maybe, but we would still need to choose which of the available derivations to use. A preference of one derivation over another can (and should) be encoded in the delta model itself by adjusting \prec , a solution discussed in Section 3.3.3.

3.3.2 Modifying Conflicting Deltas

It is possible that a conflict arose by a simple lack of communication, and that it could be resolved by making slight modifications to one or both conflicting deltas so that they are no longer in each others way.

- **3.8. Action (Modifying the Conflicting Deltas):** Given a delta model (D, \prec) and a conflict $x \not\prec y$.

$$x, y \rightsquigarrow x', y'$$

Redesign x and/or y so that x' and y' commute. ┘

This solution might apply when two software deltas each introduce a new field into a class, both with a different purpose but using the same identifier. Upon recognizing the problem, either delta's developer could simply switch to a different name, resolving the conflict by making the two deltas commute again.

In practice this action —as well as the actions introduced in the following two subsections— should be sensibly guided. For example, we actually need to include the condition that x' and y' are still individually correct in some sense. Formulating such a constraint is quite involved, and is explored in Chapter 7.

3.3.3 Linearizing Conflicting Deltas

It is possible that one of the two conflicting deltas should rightfully have priority, perhaps because it purposely refines or extends the other's implementation, such as the delta d_{SA} of the Editor product line, which extends d_{EC} — it implements a *subfeature*. It is then appropriate for that delta to be applied later and override the implementation of the other. The two deltas are *conceptually dependent*, and this should be reflected in the partial order.

- **3.9. Action (Linearizing the Conflicting Deltas):** Given a delta model (D, \prec) and a conflict $x \not\prec y$.

$$\prec \rightsquigarrow \prec'$$

Augment the partial order so that $\prec' = \prec \cup \{(x, y)\}$ or $\prec' = \prec \cup \{(y, x)\}$. ┘

This was the resolution technique offered by Schaefer et al. [164] in one of the early papers to apply the partially ordered structure of ADM.

But even *conceptually independent* deltas can have conflicting implementations. In those cases extending the partial order is not advised, even if it appears to resolve the conflict. Further development might introduce additional conflicts, which could then no longer be detected.

As a side-note: This problem might also occur between superfeature and subfeature. If delta d_{SA} , for instance, ever overwrites something in d_{EC} that it was *not* supposed to, this may be an indication that it is doing too much work, and that it should be split up into two deltas, one of them becoming a sibling to d_{EC} to detect similar mistakes in the future.

3.3.4 Introducing a Conflict Resolving Delta

Whenever we encounter conflicts that cannot be adequately resolved by either of the above techniques, we are dealing with the optional feature problem [111, 125]. The noncommutativity of the conflicting deltas is not accidental; they simply need access to the same resource. And imposing an order between them might still not give us the product we need. Take $d_{SH} \not\prec d_{EC}$, for example. If either delta is allowed to fully decide the `font(int)` implementation, the feature of the other will be broken (Figure 3.1b).

Sometimes proper interaction between two deltas simply requires additional effort on the part of the developers; some code that ties the two implementations together the way they should be. It is true that such code could be included in one of the two conflicting deltas — make it aware of the other and order it later. But that would introduce an unnatural dependency and lead to the kind of maintenance problems described at the end of Section 3.3.3.

The proper solution is to allow the conflicting deltas to remain as they are — since they each work fine in isolation — and to create a third delta with the sole purpose of coordinating their interaction. The third delta, in this context, is called a *conflict resolving delta*:

- **3.10. Definition (Conflict Resolution):** Given delta model $dm = (D, \prec)$, we define a *conflict resolution* relation $\triangleleft \subseteq D^2 \times D$ as follows, for all deltas $x, y, z \in D$:

$$(x, y) \triangleleft z \quad \stackrel{\text{def}}{\iff} \quad x, y \prec z \quad \wedge \quad \forall d \in D^*: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y$$

If x and y are in conflict, we say that z *resolves* their conflict. If the delta model is not clear from context, we attach a subscript as in \triangleleft_{dm} . \perp

A conflict resolving delta is applied after the two conflicting deltas, and allows them to commute again. It takes the rôle of what, in existing literature, is called a *lifter* [156], a *derivative module* [111, 124, 125], or *glue code* [67, 165].

- **3.11. Action (Introducing a Conflict Resolving Delta):** Given a delta model (D, \prec) and a conflict $x \not\prec y$.

$$(D, \prec) \rightsquigarrow (D \cup \{z\}, \prec')$$

Design a new delta z such that $\forall d \in D^*: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y$. Augment the partial order so that $\prec' = \prec \cup \{(x, z), (y, z)\}$. \perp

This is what we did for the $d_{SH} \not\prec d_{EC}$ conflict. The conflict resolving delta is $d_{SH \wedge EC}$. It replaces the `font(int)` method with a proper combination of the two conflicting versions (Example 3.1).

How to implement a conflict resolving delta is a genuine design decision and cannot be automated. And unless we are working with a particularly restrictive deltoid, there is almost always more than one way to do it [140].

Lienhardt and Clarke [121], in an extension of their earlier work on row typing for deltas [120], coined the term *hard conflict*, referring to the situation where, for example in Figure 3.4, the deltas y and d are invalid when applied in a certain order, e.g., $y \cdot d = \perp$. They correctly state that a conflict-resolving delta does not help in such a situation, and suggest that the only way to get out of it is to introduce an order between y and d . Indeed, when $y \cdot d = \perp$ and $d \cdot y \neq \perp$, —for example, when $y = \text{“modify X”}$ and $d = \text{“remove X”}$ —, some wrong assumptions must have been made, and applying Action 3.8 or 3.9 is probably the right thing to do.

3.3.5 Unambiguity

When all conflicts in a delta model are resolved, we end up with an *unambiguous* delta model, one which contains a conflict resolving delta for each conflict that still exists:

- **3.12. Definition (Unambiguous Delta Model):** A delta model $dm = (D, \prec)$ is *unambiguous* iff

$$\text{UA}(dm) \stackrel{\text{def}}{\iff} \forall x, y \in D: x \not\prec y \Rightarrow \exists z \in D: (x, y) \triangleleft z \quad \lrcorner$$

And that is the goal we strive for, because a delta model that is unambiguous always has a unique derivation. This is one of the main results of this chapter, as it reduces the effort of checking that all possible derivations are equal to checking that all existing conflicts have a corresponding conflict resolving delta.

- **3.13. Theorem:** Every unambiguous delta model has a unique derivation.

In order to prove this Theorem, we need some intermediate results. Lemma 3.14 states that in an unambiguous delta model, any two deltas in a derivation are either ordered or commutative:

- **3.14. Lemma:** Given an unambiguous delta model $dm = (D, \prec)$ and a derivation $d_2 \cdot y \cdot x \cdot d_1 \in \text{derv}(dm)$ in which $x, y \in D$ and $d_1, d_2 \in D^*$. Then we have either $x \prec y$ or $d_2 \cdot y \cdot x \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$.

Proof: By case distinction on the unambiguity of dm for deltas x, y :

- Case $y \cdot x = x \cdot y$. By associativity of \cdot we have $d_2 \cdot y \cdot x \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$.
- Case $x \prec y$. Immediate.
- Case $y \prec x$. Cannot happen, as $d_2 \cdot y \cdot x \cdot d_1$ is a linear extension of \prec .
- Case $\exists z \in D: (x, y) \triangleleft z$. Firstly, from Definition 3.10 we have $x, y \prec z$. So we have $d_2 = d_2'' \cdot z \cdot d_2'$ for some $d_2', d_2'' \in D^*$. From the remaining condition on z , we have $z \cdot d_2' \cdot y \cdot x = z \cdot d_2' \cdot x \cdot y$, so we finally deduce $d_2 \cdot y \cdot x \cdot d_1 = d_2'' \cdot z \cdot d_2' \cdot y \cdot x \cdot d_1 = d_2'' \cdot z \cdot d_2' \cdot x \cdot y \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$. \square

Next we prove that removing a minimal element (Definition 1.23) from a delta model would preserve its unambiguity. To help us express this we first introduce a new shorthand notation:

- **3.15. Notation:** Given a delta model $dm = (D, \prec)$ and subset $D' \subseteq D$, introduce the following notation, representing dm after removing the deltas in D' :

$$dm \setminus D' \stackrel{\text{def}}{=} \left(D \setminus D', \prec \cap (D \setminus D')^2 \right) \quad \lrcorner$$

- **3.16. Lemma:** If a delta model $dm = (D, \prec)$ is unambiguous, and $w \in D$ is a minimal element of \prec , then $dm \setminus \{w\}$ is also unambiguous.

Proof: From the unambiguity of (D, \prec) we have that $\forall x, y \in D: x \not\prec y \implies \exists z \in D: (x, y) \triangleleft z$. For the *absence* of w to invalidate this property would require that $(x, y) \triangleleft w$ for some $x, y \in D$. But that would also imply $x, y \prec w$, which is impossible because w is a minimal element. So we've proved by contradiction that $dm \setminus \{w\}$ is unambiguous. \square

Finally, we state that a minimal element from an unambiguous delta model can be shuffled to the first position of any derivation without altering its meaning:

- **3.17. Lemma:** For any unambiguous delta model $dm = (\{d_1, \dots, d_n\}, \prec)$, derivation $d_n \cdot \dots \cdot d_1 \in \text{derv}(dm)$ and minimal element d_i with $1 \leq i \leq n$, we have:

$$d_n \cdot \dots \cdot d_1 = d_n \cdot \dots \cdot d_{i+1} \cdot d_{i-1} \cdot \dots \cdot d_1 \cdot d_i$$

Proof: We proceed by induction on i :

- Case $i = 1$. Immediate.
- Case $i > 1$. As d_i is minimal, we have $d_i \not\prec d_{i-1}$. So by Lemma 3.14 they must commute and we can swap their positions:

$$d_n \cdot \dots \cdot d_1 = d_n \cdot \dots \cdot d_{i+1} \cdot d_{i-1} \cdot d_i \cdot d_{i-2} \cdot \dots \cdot d_1.$$

d_i is now in position $i - 1$ so, by induction, we can move it all the way:

$$= d_n \cdot \dots \cdot d_{i+1} \cdot d_{i-1} \cdot \dots \cdot d_1 \cdot d_i. \quad \square$$

We can now prove our main theorem:

Proof of Theorem 3.13: Take unambiguous delta model $dm = (D, \prec)$. We need to prove that $|\text{derv}(dm)| = 1$. We proceed by induction on the size of D :

- Case $|D| = 0$. Immediate, as $\text{derv}(dm) = \{\varepsilon\}$.
- Case $|D| = 1$. Immediate, as $\text{derv}(dm) = D$.
- Case $|D| > 1$. We will prove that any two derivations $d_1, d_2 \in \text{derv}(dm)$ must be equal. Let $d_1 = d'_1 \cdot x$ and $d_2 = d''_2 \cdot x \cdot d'_2$, for some $x \in D$ and $d'_1, d'_2, d''_2 \in D^*$. As x is the rightmost element of d_1 , it must be minimal in \prec . So by Lemma 3.17, $d''_2 \cdot x \cdot d'_2 = d''_2 \cdot d'_2 \cdot x$. By Lemma 3.16, $dm \setminus \{x\}$ is unambiguous. We know from before that $d'_1, (d''_2 \cdot d'_2) \in \text{derv}(dm \setminus \{x\})$ and, therefore, that $d'_1 = d''_2 \cdot d'_2$ by the induction hypothesis. We can now deduce $d_1 = d'_1 \cdot x = d''_2 \cdot d'_2 \cdot x = d'_2 \cdot x \cdot d''_2 = d_2$ and we therefore conclude that $|\text{derv}(dm)| = 1$. \square

In the Editor product line, the deltas d_{SH} and d_{EC} are in conflict over the implementation of `font(int)`, and this conflict is resolved by $d_{SH \wedge EC}$. As mentioned, the situation between d_{Pr} and d_{SH} is a case of desired feature interaction, but there is no conflict. So what about d_{EC} and d_{TI} ? Indeed, looking at Figure 1.3 as a plain delta model, those two would be in conflict regarding `onMouseOver(int)`. As there is no corresponding conflict resolver, the delta model is not unambiguous. But the diagram in question represents a full product line, which is never applied to a core product before a *feature selection* is made, after which irrelevant deltas are filtered out. As the features *EC* and *TI* are mutually exclusive (Figure 1.2), their two deltas will never be applied for the same feature selection, so there won't be a conflict to resolve. We discuss this further in Chapter 4.

3.3.6 Consistent Conflict Resolution

The notion of unambiguous delta model alleviates the task of establishing that a delta model has a unique derivation. However, deciding unambiguity is still somewhat complex, as the test for conflict resolution (Definition 3.10) has us iterating over all elements of D^* . A closer look will indicate that it is enough to check all permutations of subsets of $D \setminus \{x, y, z\}$. We might then eliminate from that set the deltas that cannot be applied between z and $y \cdot x$ because of the partial order. But it would still be a complex endeavour.

Instead, in this subsection, we introduce a new class of deltoid that allows a simpler check for conflict resolution.

If a delta algebra (and, by extension, a deltoid) exhibits *consistent conflict resolution*, then any delta z which can make deltas x and y commute when applied directly after them, will still be able to do so with any number of deltas applied in between:

- **3.18. Definition (Consistent Conflict Resolution):** The class of all delta algebras that exhibit *consistent conflict resolution* is defined as follows:

$$\text{CCR} \stackrel{\text{def}}{=} \left\{ (\mathcal{D}, \cdot) \mid \begin{array}{l} \forall x, y, z \in \mathcal{D}: z \cdot y \cdot x = z \cdot x \cdot y \neq \perp \Rightarrow \\ \forall d \in \mathcal{D}: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y \end{array} \right\}$$

We use the same term for any deltoid or delta model based on such an algebra. ◻

This definition of consistent conflict resolution differs slightly from the one in the original ADM papers [1, 2]. Since the earlier work, which did not include the concept of empty delta, the “ $\neq \perp$ ” condition has been added. Considering the purpose of the property, this would seem to be a reasonable change.

Since the property of consistent conflict resolution is checked at the level of the underlying algebra, rather than for any specific delta model, it has to be established only once and can then be relied upon for checking unambiguity.

To establish the unambiguity of a delta model exhibiting consistent conflict resolution, it is sufficient to check that for each pair of conflicting deltas x and y there exists a conflict resolving delta z such that $x, y \prec z \wedge z \cdot y \cdot x = z \cdot x \cdot y \neq \perp$; there is no need to quantify over intermediate delta sequences. Consequently, the unambiguity of delta models can be established much more efficiently. This is formalized in the next theorem:


- **3.19. Theorem:** For any delta model (D, \prec) exhibiting consistent conflict resolution, deltas $x, y, z \in D$ with $x, y \prec z$ and $z \cdot y \cdot x = z \cdot x \cdot y \neq \perp$, we have $(x, y) \triangleleft z$.

Proof: Assume that delta model (D, \prec) exhibits consistent conflict resolution. Then take arbitrary deltas $x, y, z \in D$. We have the following:

$$\begin{aligned} z \cdot y \cdot x = z \cdot x \cdot y \neq \perp &\implies \forall d \in D: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y \quad (\text{Def. 3.18}) \\ &\implies \forall d \in D^*: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y \quad (\text{Not. 2.41}) \end{aligned}$$

Together with $x, y \prec z$, the result is precisely the definition of $(x, y) \triangleleft z$. \square

This leads us to the following result regarding the running example, which has been proved with the Coq proof assistant:

- **3.20. Theorem:** The software delta algebra (Definition 2.49) exhibits consistent conflict resolution. \square 

3.4 A Fine Grained Software Deltoid

We now take a break from the abstract formalism and discuss the running example, to motivate the next section.

We have mentioned that the software deltas of Definition 2.16 are capable only of coarse grained modifications; they can only make modifications on the level of classes and methods, but cannot work with statements or expressions. This is actually true for many recent compositional techniques [108]. In realistic software development, however, code modifications are rarely so limited, so we shouldn't limit deltas either if we expect them to be used for serious development.

A more specific reason to support fine-grained modifications is something we'll call the *feature initialization problem*. This problem stems from the fact that the traditional object oriented programming model has a single *entry point*; one `main()` method that is invoked when a program starts. To implement a feature, it is not enough to add class-, method- and field-declarations. At some point, every feature will require some code to be *run*—directly or indirectly—by `main()`, to initialize, and, basically, to tell the running program that it exists. For example, software deltas d_{SH} and d_{EC} of the Editor product line need a way to have the new fields `m_syntaxhl` and `m_errorch` instantiated when the application starts. In Example 2.17 (page 40), this is done for *SH* by **replacing** the `Editor.init()` method.

But if the *EC* delta were to do the same, this would introduce a conflict, as was the case when they both replaced `font(int)`. Of course, this conflict could be handled by their conflict-resolving delta; just add another **replace** operation which adds the initialization code for both; problem solved, right? True, but assume that n independent deltas need to add such initialization code, for some large n . This would mean that $2^n - n - 1$ conflict resolving deltas would be required to clean up the mess—one for each combination. The codebase would contain $2^n - 1$ similar—but different!—versions of `Editor.init()` provided by $2^n - 1$ deltas (Figure 3.6).

And there is still another problem with the solution shown in Figure 3.6. Each conflict resolving delta is forced to decide on a specific textual order in which to run the initialization methods of a, b and c. This is so common in

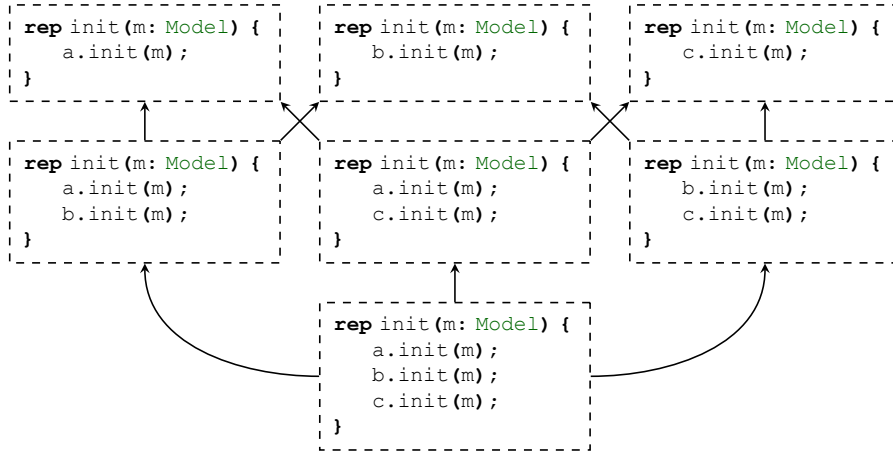


Figure 3.6: A delta model showing the initialization problem on a small scale.

imperative languages—in which sequential composition is ubiquitous—that most programmers wouldn’t look twice. But recall that this kind of overspecification is the reason we found annotative variability approaches lacking. If `a.init()`, `b.init()` and `c.init()` are not as independent as we thought, and one of them accidentally overwrites or otherwise damages the work of another, it is theoretically impossible for an interpreter or compiler to detect this, since it may very well have been intentional. As we are working with a compositional approach, we can do better.

3.4.1 Fine-grained Software Deltas

We now have the machinery to properly model and reason about fine-grained software deltas. As mentioned, previous literature [31, 52, 161] accomplishes some semblance of fine granularity through the **super** (or **original()**) construct, which enables a module to add new code to the beginning or end of an existing method body, somewhat like *around advice* in AOP [113]. To model this we’ll introduce the **prepend** and **append** delta operations which act on the method level, similar to AOP before- and after advice. This only works for single statements, but since those statements can be method calls, that does not reduce expressiveness.

But this doesn’t solve the initialization problem; two deltas that each append (or prepend) a statement to an existing method still do not commute.

If n pieces of initialization code are truly independent, we won’t care in which order they appear in `init()`, since all orders would be semantically equivalent. So we introduce a third method-level delta operation **insert**, which inserts a statement in a nondeterministically chosen position inside a method body. Why does this help? Because the composition of n deltas, each inserting a statement at an arbitrary position, is the same as a single delta which inserts n statements in an arbitrary position. Moreover, all n deltas will commute (Figure 3.7).

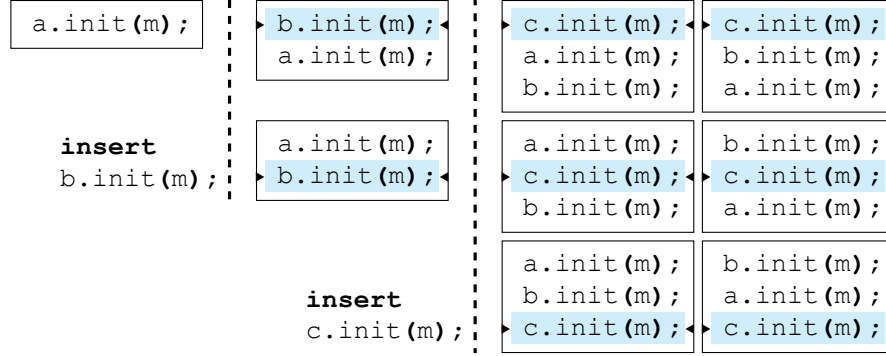


Figure 3.7: The effect of subsequent **insert** operations on a method body. The number of possible output products increases with every insertion. Reversing the order between the two insertions would not affect the final result.

There is a caveat: It is still the responsibility of the developers to ensure that an **inserted** statement is indeed independent to the other statements that may be in the method. It is not trivial to automatically check such a semantic constraint. However, at least the proper intention can now be expressed, so in cases where an inadvertent dependency *can* be detected, it is possible to issue an error message about it.

- ▷ **3.21. Definition (Fine-grained Software Deltas):** Fine-grained software deltas are mostly the same as the software deltas from Definition 2.16 (page 39), but with some additional operations at the method level. This definition only specifies those additions, but implicitly adapts the original sets \mathcal{D}_{pkg} , $\mathcal{OP}_{\text{pkg}}$, \mathcal{D}_{cl} and \mathcal{OP}_{cl} accordingly, and renames them to $\mathcal{D}_{\text{pkg}+}$, $\mathcal{OP}_{\text{pkg}+}$, $\mathcal{D}_{\text{cl}+}$ and $\mathcal{OP}_{\text{cl}+}$. Statement deltas are sequences of statement-level operations:

$$\mathcal{D}_{\text{st}+} \stackrel{\text{def}}{=} \mathcal{OP}_{\text{st}+}^*$$

There are no identifiers at the statement level. Instead, statement operations are performed in the order in which they are sequenced. A statement operation is defined as follows:

$$\mathcal{OP}_{\text{st}+} \stackrel{\text{def}}{=} \left(\begin{array}{l} \{\mathbf{pre}\} \times \mathcal{ST} \cup \\ \{\mathbf{app}\} \times \mathcal{ST} \cup \\ \{\mathbf{ins}\} \times \mathcal{ST} \end{array} \right)$$

A **pre** (**prepend**) operation adds a new statement to the beginning of a method. An **app** (**append**) operation adds one at the end. An **ins** (**insert**) operation adds a statement at an arbitrary position inside a method: at the beginning, the end, or between two existing statements.

Finally, we add a class-level **mod** operation to descend to the method level:

$$\mathcal{OP}_{\text{cl}+} \stackrel{\text{def}}{=} \mathcal{OP}_{\text{cl}} \cup (\{\mathbf{mod}\} \times \mathcal{D}_{\text{st}+}) \quad \lrcorner$$

And here is how method-level operations are evaluated:

- ▷ **3.22. Definition (Fine-grained Software Deltoid):** The *fine-grained software deltoid* ($FgSD$) is a deltoid $Dt_{\text{pkg}+} \stackrel{\text{def}}{=} (\mathcal{PKG}, \mathcal{D}_{\text{pkg}+}, \llbracket - \rrbracket)$ with product set \mathcal{PKG} from Definition 2.5, delta set $\mathcal{D}_{\text{pkg}+}$ from Definition 3.21 and evaluation operator $\llbracket - \rrbracket: \mathcal{D}_{\text{pkg}+} \rightarrow \text{Pow}(\mathcal{PKG} \times \mathcal{PKG})$ as in Definition 2.18, but with additional inference rules: (a) rules for the new statement-level operations **pre**, **app** and **ins**, (b) rules for statement-level deltas, and (c) a **mod** rule to descend from the class-level to the statement-level.

a. Statement Level Operations

First, the meaning of the three statement-level operations is as follows, for all method types $tp \in \mathcal{TP}$, all statement sequences $\overline{st}, \overline{st}_1, \overline{st}_2 \in \mathcal{ST}^*$ and all statements $st' \in \mathcal{ST}$ (recall Definition 2.3, page 31):

$$\frac{}{(tp, \overline{st}) \llbracket \mathbf{pre} \ st' \rrbracket \ (tp, \overline{st'} \frown \overline{st})} \text{ statement prepension}$$

$$\frac{}{(tp, \overline{st}) \llbracket \mathbf{app} \ st' \rrbracket \ (tp, \overline{st} \frown \overline{st'})} \text{ statement appension}$$

$$\frac{}{(tp, \overline{st}_1 \frown \overline{st}_2) \llbracket \mathbf{ins} \ st' \rrbracket \ (tp, \overline{st}_1 \frown \overline{st'} \frown \overline{st}_2)} \text{ statement insertion}$$

The **pre** and **app** operations are deterministic, placing the new statement squarely at the beginning or end of the method. But the **ins** operation is non-deterministic. It can produce a method with the new statement at the very beginning or end, or at any position in between, depending on how the original sequence of statements is apportioned between \overline{st}_1 and \overline{st}_2 . Also, note that none of them change the type of the method. Software deltas have no way of doing so without replacing the entire method. (Though, as mentioned before, this is an intentional simplification.)

b. Statement Deltas

A statement delta applies its operations in sequence. For all methods $mtd, mtd' \in \mathcal{Mtd}$, all statement operations $op \in \mathcal{OP}_{\text{st}+}$ and all trailing sequences of statement operations $\overline{op'} \in \mathcal{OP}_{\text{st}+}^*$:

$$\frac{mtd \llbracket \overline{op'} \rrbracket \circ \llbracket op \rrbracket \ mtd'}{mtd \llbracket op \frown \overline{op'} \rrbracket \ mtd'} \text{ statement delta application (non-empty)}$$

$$\frac{}{mtd \llbracket () \rrbracket \ mtd} \text{ statement delta application (empty)}$$

Pay close attention to the ordering, because sequence concatenation is read from left to right (Definition 1.9, page 19), whereas relation composition is read from right to left (Definition 1.11, page 20).

c. Class Level Operations

Finally, a method modification operation at the class level delegates the work to the statement level delta and then maps to the new result. For all classes $cl \in \mathcal{CL}$, all identifiers $id \in \mathcal{ID}$, all statement level deltas $d_{st+} \in \mathcal{D}_{st+}$ and all methods $mtd \in \mathcal{Mtd}$:

$$\frac{cl(id) \in \mathcal{Mtd} \quad cl(id) \llbracket d_{st+} \rrbracket mtd}{cl \llbracket id \mapsto \mathbf{mod} \ d_{st+} \rrbracket cl[id \mapsto mtd]} \text{ method modification} \quad \lrcorner$$

- ▷ **3.23. Lemma:** The software deltoid of Definition 2.18 (page 41) refines the fine-grained software deltoid, as per Definition 2.63 (page 58):

$$Dt_{\text{pkg}} \sqsupseteq Dt_{\text{pkg}+}$$

Proof: This is trivial by the deltoid homomorphism $\text{id}_{\mathcal{PKG} \times \mathcal{PKG}}$. Every software delta is also a fine-grained software delta, with the same semantics. \square

There are adapted definitions of the algebraic operators to go with this new deltoid, but their full formulation wouldn't add much to the story. Composition of two deltas **modifying** the same method simply concatenates their corresponding lists of statement-level operations. More interesting is the syntactic refinement relation, which can be used to define both equivalence and consensus (Definitions 2.42 and 2.43, page 52). The following is how refinement works on the statement level.

- ▷ **3.24. Definition (Syntactic FgSD Refinement):** We define *syntactic FgSD refinement* $\approx \subseteq \mathcal{D}_{\text{pkg}+}^2 \cup \mathcal{D}_{\text{cl}+}^2 \cup \mathcal{D}_{\text{st}+}^2$ as the smallest preorder satisfying the the conditions of Definition 2.46 (page 54), as well as the following:

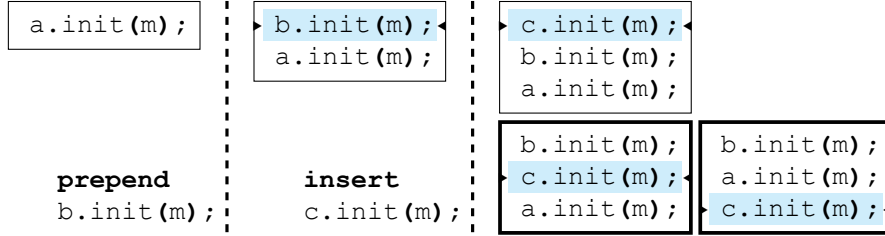
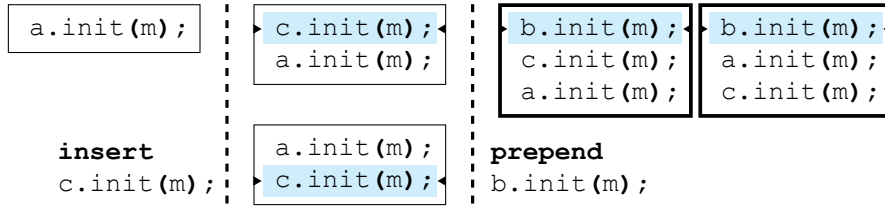
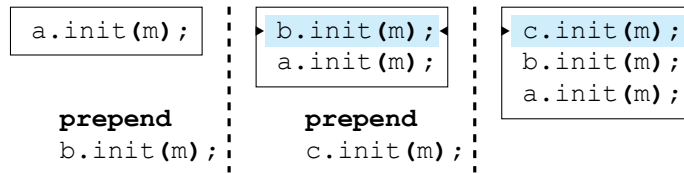
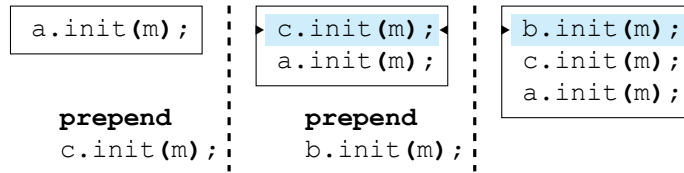
$$\frac{}{(\mathbf{ins} \ st_1) \frown (\mathbf{ins} \ st_2) \frown \overline{op} \ \lesssim \ (\mathbf{ins} \ st_2) \frown (\mathbf{ins} \ st_1) \frown \overline{op}} \text{ by commutative } \mathbf{ins}$$

This rule represents the fact that the order between two insertions does not matter (Figure 3.7). The next two are more interesting:

$$\frac{}{(\mathbf{ins} \ st_1) \frown (\mathbf{pre} \ st_2) \frown \overline{op} \ \lesssim \ (\mathbf{pre} \ st_2) \frown (\mathbf{ins} \ st_1) \frown \overline{op}} \text{ by late prepension}$$

$$\frac{}{(\mathbf{ins} \ st_1) \frown (\mathbf{app} \ st_2) \frown \overline{op} \ \lesssim \ (\mathbf{app} \ st_2) \frown (\mathbf{ins} \ st_1) \frown \overline{op}} \text{ by late appension}$$

The idea is that *after* a **pre** or **app** (Figure 3.8a), there are more potential places to **insert** a new statement than *before* (Figure 3.8b). So a delta is more refined the earlier its **insert** operations occur. \lrcorner

(a) A **prepend** followed by an **insert**.(b) An **insert** followed by a **prepend**.Figure 3.8: The interaction between a **prepend** and an **insert**. The thick borders in the third column indicate the result of their *consensus*.(a) **prepending** first b, then c(b) **prepending** first c, then bFigure 3.9: The interaction between two **prepend** operations. As the two are incompatible (and deterministic), they have an empty consensus.

3.5 Ambiguous Delta Models

In Section 3.3 we explored techniques for ensuring a unique derivation in a delta model. However, there are legitimate cases for the use of delta models with multiple derivations. To that end, this section proposes two semantics that are more flexible than the sole derivation semantics of Definition 3.5.

In previous work [1–3], we gave delta models *disjunctive semantics*, in which a non-deterministic choice is made between available derivations. We describe this semantics in Section 3.5.1. An alternative semantics is proposed in Section 3.5.2: *conjunctive semantics*, which we believe to be a more natural interpretation.

3.5.1 Disjunctive Semantics

In disjunctive semantics, a delta model is seen as providing a source of non-determinism, just like deltas themselves can. This would lead to the following delta model semantics, which applies regardless of the number of derivations:

- **3.25. Definition (Disjunctive Semantics):** The *disjunctive semantics* of a delta model $dm \in \mathcal{DM}$ with $\text{derv}(dm) = \{d_1, \dots, d_n\}$ are defined as follows:

$$\llbracket dm \rrbracket_{\cup} \stackrel{\text{def}}{=} \llbracket d_1 \rrbracket \cup \dots \cup \llbracket d_n \rrbracket$$

Just take the union of the semantic deltas, i.e., a union of the product relations (see Definitions 1.2 and 3.3 and Notation 2.13). \lrcorner

This semantics was used in the original ADM papers [1–3], though not by the same distinctive name, as alternative semantics were not considered at the time.

Disjunctive semantics is the most flexible of the three delta model semantics we present in this chapter, and the easiest to employ, as it simply requires that any applicable derivation be applied. No special constructions or proofs of unambiguity are necessary. For that reason, this semantics is best used to tolerate ambiguity during development until stricter standards can be established.

3.5.2 Conjunctive Semantics

While disjunctive semantics certainly has its uses, it does not, perhaps, properly correspond to a developer’s likely intentions. As a means for introducing nondeterminism, delta models are quite limited. Nondeterminism is much more flexibly introduced by deltas themselves (Sections 2.4 and 3.4). So would it not be better for delta model semantics to instead play a supporting rôle?

Conjunctive semantics is the dual of disjunctive semantics, and has a delta model perform a modification that all derivations agree upon:

- **3.26. Definition (Conjunctive Semantics):** The *conjunctive semantics* of a delta model $dm \in \mathcal{DM}$ with $\text{derv}(dm) = \{d_1, \dots, d_n\}$ are defined as follows:

$$\llbracket dm \rrbracket_{\cap} \stackrel{\text{def}}{=} \llbracket d_1 \rrbracket \cap \dots \cap \llbracket d_n \rrbracket$$

So, take the intersection of the product relations (see Definitions 1.2 and 3.3 and Notation 2.13). \lrcorner

Note, at this point, that both disjunctive and conjunctive semantics are compatible with the sole derivation semantics of Definition 3.5 in the case that a unique derivation exists:

3.27. Lemma: For all delta models $dm \in \mathcal{DM}$ we have:

$$\text{UD}(dm) \implies \llbracket dm \rrbracket_{\cup} = \llbracket dm \rrbracket_{\cap} = \llbracket dm \rrbracket \quad \square$$

Conjunctive semantics is less tolerant to mistakes but offers stronger guarantees. Applying a delta model to a product under conjunctive semantics always results in an end product that might also have resulted from the application of any individual derivation. In other words, a developer may consider a specific derivation $d \in \text{derv}(dm)$ and work under the assumption that d is the derivation that will be chosen to generate the final product, without this leading to contradiction:

3.28. Lemma: For any delta model $dm \in \mathcal{DM}$, any derivation $d \in \text{derv}(dm)$ and any specification $s \in \mathcal{S}$, we have:

$$d \models s \implies \llbracket dm \rrbracket_{\cap} \subseteq s \quad \square$$

The same thing is not true for disjunctive semantics.

That being said, this semantics takes more effort to implement. Disjunctive semantics requires only that single derivations are tried until one is found that is applicable to the product at hand (recall that deltas may be partially defined). Conjunctive semantics, on the other hand, permits no such approach. There is no general procedure for generating all possible outputs of a nondeterministic delta in order to produce the required intersection; in fact, such a set may well be infinite. So implementing delta model application under conjunctive semantics requires a greater understanding of the domain. In particular, it requires an implementation of the consensus operator (Definition 1.34):

► **3.29. Lemma:** For any deltoid $(\mathcal{P}, \mathcal{D}, \sqcap, \cdot, \varepsilon, \llbracket - \rrbracket)$ and delta model $dm \in \mathcal{DM}_{\mathcal{D}}$, we can characterize conjunctive semantics as follows:

$$\llbracket dm \rrbracket_{\cap} = \llbracket \bigcap \text{derv}(dm) \rrbracket$$

Proof: This is easily derived from Definitions 2.38 and 3.26. □

So if we have an effective procedure for delta consensus of a specific deltoid, delta models based on that deltoid can exhibit conjunctive semantics. This can be worth the effort. A conflict model based on delta commutativity, as introduced in Section 3.3, is useful, but can be too strict in the presence of more sophisticated interaction. The concept of conjunctive semantics allows separate developers to express intentions that would otherwise be flagged as a conflict, but can now be reconciled without the need for manual conflict resolution.

For example, imagine two unordered software deltas modifying the same method. One of them **inserts** a statement “`c.init(m)`”. The other one, having stricter requirements, **prepends** a statement “`b.init(m)`” (Figure 3.8). Composing them in two different orders results in two different derivations, but intuitively there should be no conflict. As long as “`b.init(m)`” becomes the

first statement of the method, and “`c.init(m)`” is inserted anywhere else, the intentions of both developers are satisfied. And indeed, this is the result of the consensus between Figure 3.8a and Figure 3.8b, as it should be.

In contrast, two deltas each wanting their own statement to appear first in the method is recognized as a legitimate conflict by the empty consensus between Figure 3.9a and Figure 3.9b.

3.6 Nested Delta Models

This section explores the possibility of deltas that *are* delta models. We then take a particular look at the implications of *nested delta models*. There are a number of reasons we might want delta models to act as deltas inside other delta models, chief among them being the isolated/atomic application of a collection of deltas within a delta model, making sure that any delta outside that collection is applied before or after the entire collection — but not in between.

Let us first establish some terminology:

- **3.30. Definition:** A *nesting delta model* is a delta model that contains another delta model. A *simple delta* is a delta that is not a delta model. A *flat delta model* is a delta model that contains only simple deltas. A *nested delta model* is a delta model contained within another delta model. ┘

Nesting delta models have several uses in the area of modularization:

- Recall from Definitions 3.7 and 3.10 that a conflict is uniquely identified by two conflicting deltas, and that each requires a single delta to resolve it. But a conflict may have several causes. For example, two software deltas may disagree on the implementation of more than one method. Resolution for each of those methods could be modularized as a delta inside a conflict resolving delta model.
- They may also be used for refactoring a single delta into two deltas. Nesting the two together avoids the inadvertent introduction of new conflicts, because the two deltas would still be treated as one (Figure 3.10).
- It may be generally beneficial to structure variability as a hierarchy, i.e., to implement a modification in terms of smaller modifications, in the best traditions of computer programming. Nesting delta models can do this.

As an example of using a nested delta model for refactoring purposes consider Figure 3.10. In this figure, delta d_{SH} from the Editor product line is being refactored into two deltas d_{SH}^1 and d_{SH}^2 , the first handling the fields and initialization of the feature in the `Editor` class and the second handling the actual functionality of configuring the font. To avoid having to introduce extra ordering into the delta model to preserve the original semantics, the two deltas are placed in a nesting delta model which replaces the original d_{SH} .

3.6.1 Semantics Independent Definitions

Let us first look at a way to syntactically extend a set of deltas to include all delta models that could be built from that set:

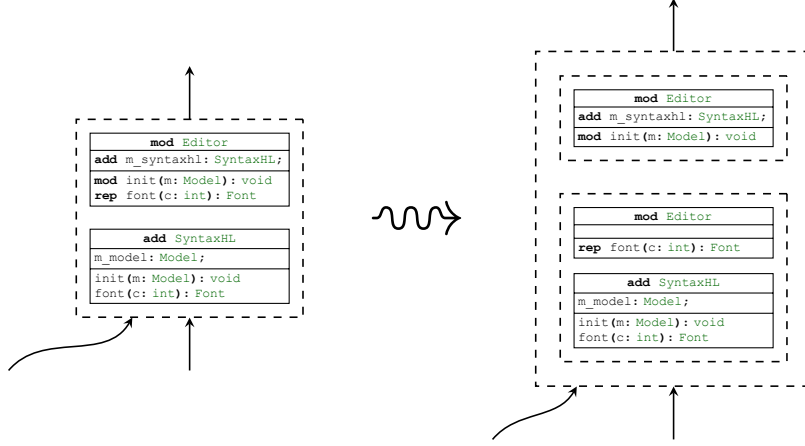


Figure 3.10: Refactoring d_{SH} from Figure 1.3 (page 9) into a nested delta model containing deltas d_{SH}^1 and d_{SH}^2 .

- **3.31. Definition (Delta Model Closure):** Given a delta set \mathcal{D} , we define the following family of delta sets for all natural numbers $n \in \mathbb{N}$:

$$\begin{aligned} \mathcal{D}_{\Delta,0} &\stackrel{\text{def}}{=} \mathcal{D} \\ \mathcal{D}_{\Delta,n+1} &\stackrel{\text{def}}{=} \mathcal{D}_{\Delta,n} \cup \mathcal{DM}_{\mathcal{D}_{\Delta,n}} \end{aligned}$$

We then define the *delta model closure* of \mathcal{D} as follows:

$$\mathcal{D}_{\Delta} \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \mathcal{D}_{\Delta,n}$$

We require that \mathcal{D} did not contain any delta models to begin with. \lrcorner

This definition allows delta models nested at unbounded —but finite— depth. Note that \mathcal{D}_{Δ} can be partitioned into the set of simple deltas \mathcal{D} and the set of delta models $\mathcal{DM}_{\mathcal{D}_{\Delta}}$.

We can then define a nesting aware derivation function, first requiring a straightforward extension of the composition operator to sets of deltas:

- **3.32. Notation:** We extend \cdot to sets as follows, for all delta sets $D_1, D_2 \subseteq \mathcal{D}$:

$$D_2 \cdot D_1 \stackrel{\text{def}}{=} \{d_2 \cdot d_1 \mid d_1 \in D_1 \wedge d_2 \in D_2\} \quad \lrcorner$$

- **3.33. Definition (Nesting-aware Derivation):** Define the *nesting-aware derivation* function $\text{derv}_{\Delta}: \mathcal{D}_{\Delta} \rightarrow \text{Pow}(\mathcal{D})$ as follows, for any simple delta $d \in \mathcal{D}$ and delta model $dm = (D, <) \in \mathcal{DM}_{\mathcal{D}_{\Delta}}$:

$$\begin{aligned} \text{derv}_{\Delta}(d) &\stackrel{\text{def}}{=} \{d\} \\ \text{derv}_{\Delta}(dm) &\stackrel{\text{def}}{=} \bigcup \text{derv}_{\Delta}(d_n) \cdot \dots \cdot \text{derv}_{\Delta}(d_1) \\ &\quad D = \{d_1, \dots, d_n\} \wedge \\ &\quad \forall i, j \in \{1, \dots, n\}: \\ &\quad d_i < d_j \Rightarrow i < j \end{aligned} \quad \lrcorner$$

The algebraic interpretation of a delta model closed deltoid is quite simple. It is a matter of setting up one of the following equivalences:

$$\begin{aligned} dm &\simeq \bigsqcup \text{derv}(dm) \quad (\text{under disjunctive semantics}) \\ dm &\simeq \bigsqcap \text{derv}(dm) \quad (\text{under conjunctive semantics}) \end{aligned}$$

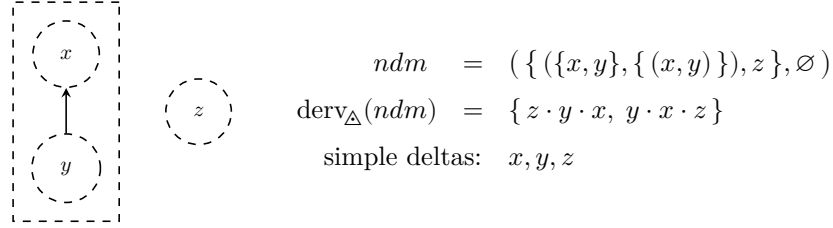
and then to ‘flatten’ delta models to delta expressions containing only simple deltas. A very similar technique is used in Chapter 6 in order to reduce modalities to simple forms.

3.6.2 Expressiveness of Nested Delta Models

The semantics of nesting cannot be captured with flat delta models. Nesting delta models are syntactically more expressive in the following sense:

- **3.34. Theorem:** There are nesting delta models $ndm = (D, \prec) \in \mathcal{DM}_{\Delta}$ that have a derivation set $\text{derv}_{\Delta}(ndm)$ which is inexpressible with any flat delta model $dm' = (D', \prec')$ containing the same simple deltas.

Proof: Consider the following nesting delta model ndm :



To find a flat delta model $dm' = (\{x, y, z\}, \prec')$ s.t. $\text{derv}(dm') = \text{derv}_{\Delta}(ndm)$, consider all possible strict partial orders \prec' over 3 elements:

$$\begin{aligned} \prec' &= \emptyset && \Rightarrow |\text{derv}(dm')| = 6 \\ \prec' &= \{(e, g)\} \quad \text{s.t. } \{e, g\} \subseteq \{x, y, z\} && \Rightarrow |\text{derv}(dm')| = 3 \\ \prec' &= \{(e, g), (g, h)\} \quad \text{s.t. } \{e, g, h\} = \{x, y, z\} && \Rightarrow |\text{derv}(dm')| = 1 \\ \prec' &= \{(e, g), (e, h)\} \quad \text{s.t. } \{e, g, h\} = \{x, y, z\} && \Rightarrow \text{derv}(dm') = \{h \cdot g \cdot e, g \cdot h \cdot e\} \\ \prec' &= \{(e, h), (g, h)\} \quad \text{s.t. } \{e, g, h\} = \{x, y, z\} && \Rightarrow \text{derv}(dm') = \{h \cdot g \cdot e, h \cdot e \cdot g\} \end{aligned}$$

As ndm has two nesting-aware derivations, only the last two cases are relevant. If ndm were expressible via a flat delta model, there would exist a bijection between $\{e, g, h\}$ and $\{x, y, z\}$ such that either $\{h \cdot g \cdot e, g \cdot h \cdot e\} = \{z \cdot y \cdot x, y \cdot x \cdot z\}$ or $\{h \cdot g \cdot e, h \cdot e \cdot g\} = \{z \cdot y \cdot x, y \cdot x \cdot z\}$. But no such bijection exists. Hence, there exists no flat delta model dm' such that $\text{derv}(dm') = \text{derv}_{\Delta}(ndm)$. \square

3.7 Conclusion

In some ways, this chapter describes the most fundamentally novel contribution of ADM: *delta models*, which organize deltas into a *partial application order*. One delta may be dominant over another, or two deltas may be unrelated by the order. This helps developers express their design intentions, and contain the complexity of large system. If two deltas are unrelated, it is still possible that both need access to the same resource, causing a *conflict* if both are applied together, even if each works fine in isolation. To solve this problem and still maintain separation of concerns, *conflict resolving deltas* are introduced.

The chapter then extends the software deltoid to allow *fine-grained* modifications, i.e., manipulating individual statements in methods. This is often neglected in compositional approaches like delta modeling, because unlike classes, methods and fields, statements have no names by which a delta can target their position. *Conjunctive delta model semantics* are introduced to take advantage of fine-grained modifications. The operation of inserting a statement in a non-deterministically chosen location avoids another type of overspecification, representing the intention: “this method should run this statement at some point; I don’t care when”. This reduces the likelihood that two changes to the same method are seen as a conflict.

Finally, the chapter introduces *nested delta models*, which increase expressiveness of a deltoid and offer a new modularization technique.

3.8 Related Work

We now summarize related work that was not mentioned earlier in this chapter, among which some of the advances since we started our work on ADM.

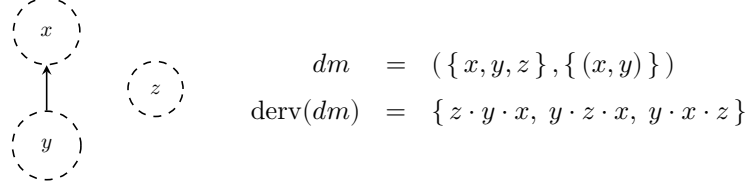
3.8.1 Delta-Oriented Programming

Originally, a delta oriented product line consisted of a single core and a set of incomparable product deltas [160, 163]. Conflicts between deltas applicable for the same feature configuration were prohibited. In order to express all possible products, an additional delta covering the combination of the potentially conflicting deltas had to be specified, leading to code duplication.

Since the work on ADM, Schaefer et al. [162, 164] also introduced a partial order between delta modules. However, it was required that conflicts were removed by changing the deltas or by specifying some linear order (Actions 3.8 and 3.9). They were not allowed to exist and then resolved.

Later work [81, 161, 169] moved away from the partial order in favor of a linearly ordered partition. Delta modules could be freely reordered within a part, but the parts themselves had to be applied in a fixed total order. It is easily proved that this is at least as expressive as Schaefer et al.’s earlier unordered structure as well as the total ordering of AHEAD, as those correspond to the two trivial partitions of the module set. But it is not as expressive as an arbitrary partial order, which can be shown with the following simple example,

inexpressible with a linearly ordered partition:



3.8.2 Feature Interaction Algebra

Two recent papers by Batory et al. [29, 33] describe a new algebraic treatment of feature interaction. This treatment takes place in the setting of CIDE (Colored IDE) [108], a variability tool based on *code painting* — each feature has an associated color, which is used for the annotation of code. When code is painted in more than one color, it is ‘interaction composition code’, similar in function to conflict resolving deltas. The algebra introduces a notation $\#$ for interaction.

A rather confusing aspect of this work is the fact that $\#$ is presented as a *operator*: if A and B are feature implementations, then $A\#B$ is also a feature implementation; their resolution. The operator is commutative, associative, and obeys such distributive laws as $A\#(B + C) = (A\#B) + (A\#C)$, where $+$ is feature composition. But since the interaction between two features is a design decision, it cannot be ‘computed’. And there is almost always more than one way to do it. The operator’s exact meaning and purpose are therefore unclear.

Recently, Apel et al. [18] also refer to the $\#$ notation, but they use it only as a shorthand for the corresponding coordination code; not as an operator.

3.8.3 Other Related Work

Our distinction between feature interaction and implementational conflicts (Section 3.3) is also described by Kästner et al. [109] in a discussion paper on feature modularity. Mosser et al. [140] —in an otherwise fascinating contribution— missed this distinction in their analysis of our original paper on ADM [1].

Our notion of conflict, based on a lack of commutativity, has already been discussed in a number of publications. Mens et al. [132], for instance, describe this phenomenon in the context of refactoring [71]. Their notion of conflict is very similar to ours, though their solution —based on graph transformation and critical pair analysis— is quite different. Apel et al. [14] and Oldevik et al. [145] observe similar notions of conflict, respectively in aspect oriented programming and model transformations. Apel et al. propose that the aspects involved be refactored following a particular scheme — a measure which falls under our Action 3.8. Oldevik et al. analyze the effect of ordering constraints to resolve conflicts — quite like our Action 3.9.

Interestingly, while Darcs patch theory [97] has quite a different purpose from ADM and the aforementioned literature, they deal with a very similar —and naturally occurring— partially ordered structure: that of branches and commits in a version control system. The most significant similarity is that they deal with *conflictors* (entities for resolving conflicts), which are similar to conflict-resolving deltas. Patch theory could certainly offer inspiration to guide future research (Chapter 9).

Product Lines

On Feature Modeling and Delta Selection



4.1 Introduction

A *product line* is usually defined as a set of systems, called *products*, each of which is characterized by the set of *features* it provides. This allows commonality and variability between these products to be well-defined and amenable to formal analysis. Products were introduced in Chapter 2. Delta models, the tools we use for generating new products, were introduced in Chapter 3. We now introduce the final ingredient to ADM-based product line models: features.

Many different definitions of the term “feature” have been given in literature. Griss [75] defines features as follows:

“A feature is a product characteristic that users and customers view as important in describing and distinguishing members of [a] product-line.”

Classen et al. [53] gathered a number of definitions from literature in order to compare them. The above definition is what they would call *problem-oriented*. They also talk about definitions such as “a logical unit of behavior” by Bosch [45] and “an increment in product functionality” by Batory et al. [27], which are geared more towards implementation.

Griss’ definition is especially suitable for us, as our “logical unit of behavior” or “increment in product functionality” is, of course, the delta. We see features more as specifications of product requirements. Deltas and features should not be restricted to a one-to-one relationship; a sentiment first expressed by Schaefer et al. [163].

We are mainly interested in using features as a means of identifying specific products in a product line, allowing us to use features

- to characterize the set of available products through a feature model, which determines the possible *feature configurations*, i.e., the set of feature combinations supported by the product line;
- to formulate *product line implementations* by linking feature symbols to deltas through application conditions, allowing us to select the proper deltas for deriving the implementation of a specific product; and
- to formulate *product line specifications* by associating requirements with each feature.

Product generation is still a time consuming and expensive activity [65]. Ideally, it should be a fully mechanical process, since any manual adjustments after feature selection would need to be repeated whenever the main code-base is changed in some way. This brings us to the main goal of this chapter:

Goal: *Develop a technique for organizing a product line code-base in such a way that product generation can be a mechanical process.*

Such generation process is generally known as *automated product derivation* [1, 2, 65, 84, 163] (Figure 1.1, page 5).

When it comes to automated product derivation, *annotative variability techniques* are undeniably popular. The idea is to take the full code-base and to annotate the code with feature conditions without otherwise altering its structure. For example, take a look at the following annotative implementation of Syntax Highlighting and Error Checking in the Editor product line:

- ▷ 4.1. **Example:** A Delta Editor product annotated with the *SH* and *EC* features. The **if**-conditions are typically resolved at compile-time:

```

1  package DeltaEditor {
2      class Editor {
3          m_model : Model;
4          if (SH) { m_syntaxhl : SyntaxHL; }
5          if (EC) { m_errorch : ErrorChecker; }
6
7          init(m : Model) : void {
8              m_model = m;
9              if (SH) { m_syntaxhl = new SyntaxHL(m); }
10             if (EC) { m_errorch = new ErrorChecker(m); }
11         };
12
13         model() : Model { return m_model; };
14
15         font(c : int) : Font {
16             Font result = new Font();
17             if (SH) { result.setColor(m_syntaxhl.font(c)); }
18             if (EC) { result.setUnderlined(m_errorch.errorOn(c)); }
19             return result;
20         };
21
22         onMouseOver(c : int) : void {
23             if (EC) {
24                 if (m_errorch.errorOn(c)) {
25                     super.showTooltip(m_errorch.errorText(c));
26                 }
27             }
28         };
29     };
30
31     if (SH) {
32         class SyntaxHL {
33             m_model : Model;
34
35             init(m : Model) { m_model = m; };
36
37             font(c : int) : Font { /* something complicated */; };
38         };
39     }
40
41     if (EC) {
42         class ErrorChecker {
43             m_model : Model;
44
45             init(m : Model) { m_model = m; };
46
47             errorOn(c : int) : bool { /* some code */; };
48
49             errorText(c : int) : string { /* more code */; };
50         };
51     }
52 };

```


As these annotations are essentially **if**-statements, this is a technique all programmers will understand, and one that is relatively effortless to set up. A prominent example of the annotative technique in practice is the Linux kernel, an immense collection of C code annotated by `#ifdef` preprocessor directives, which allow conditional compilation [171].

But as discussed before, there are several disadvantages to this approach. There is a notable lack of modularity and separation of concerns. In Example 4.1, the implementations of *SH* and *EC* are mixed together and spread across the core implementation, making it difficult to get a good overview of the structure. Additionally, there is overspecification because of the linear nature of program code, discussed at length in Chapter 3. In this chapter we build on the concept of delta models in an effort to achieve automated product derivation without neglecting these other goals. We also discuss product line level specifications, allowing us to consider product line *correctness*.

Goal: *Develop a formal concept of product line specification, to be used both in verifying product line correctness and in guiding the implementation process.*

This chapter is organized as follows: Section 4.2 reviews *feature modeling*, the discipline of describing product line variability on the high abstraction level of feature symbols. In Section 4.3 we tie features to deltas using *application conditions* and explore a formulation of product line implementations based on delta models. Section 4.4 then introduces product line *specifications*. In Section 4.5 we recognize a possible problem with purely compositional techniques and propose the solution of *parametric deltas* in order to gain some of the benefits of the annotative approach. Finally, Sections 4.7 and 4.8 offer concluding remarks and discuss related work.

4.2 Feature Modeling

Feature-oriented Domain Analysis (FODA) was developed in 1990 in order to study possible *features* of a system to enhance reuse in a particular application domain. *Feature Models* were among its most useful tools, characterized as “the greatest contribution of domain engineering to software engineering” [105], and are still ubiquitous in Software Product Line Engineering today.

Feature models are not concerned with implementation, but with modeling product line commonality and variability on a high level. What kind of features are (should be) available, and what are the relationships between them? The answers to these questions inform both specification and implementation of delta-based product lines. Section 4.2.1 discusses the formal concept of feature. Section 4.2.2 then formalizes feature models.

4.2.1 Features

We first introduce a formal representation for features:

- **4.2. Notation (Features):** We denote *features* by the symbols f , g and h . Finite sets of features are denoted by F , G or \mathcal{F} . ┘

These features are just symbols [56], with no inherent meaning. It is what we do with features next that gives them their semantics. They will play several rôles in our formalism, which will be extended in Chapters 6 to 8.

We assume that each feature represents a discrete Boolean value, i.e., something that can be either on or off. This is a simplification. Czarnecki et al. [58, 59], for example, have worked on cardinality-based and attributed feature models. These allow a product to contain a specific feature more than once, or to contain variations of a feature parametrized with arbitrary data-types. We won't discuss them in detail, though in subsequent chapters we'll occasionally dip our toe in the water. Using the simpler Boolean notion will make this chapter easier to follow, without sacrificing generality.

There are two ways in which we give features meaning in this chapter. First, product line implementations (Section 4.3) conditionally select deltas based on a chosen selection of features — linking features to code. Then, product line specifications (Section 4.4) impose requirements on products that claim to implement given features — linking features to code specifications. This will lead to notions of correctness and refinement for product lines.

4.2.2 Feature Models

Features represent the *variability* and *commonality* of the products in a product line: the ways in which they can differ from each other. This is often expressed in terms of a *feature model* [66, 105, 166], which expresses the relations between features and decides which combinations of them are considered valid or conceptually feasible. Such feature combinations are more commonly called *feature configurations*, a term that is appropriate for simple as well as cardinality-based and attributed feature models.

Many formal descriptions [66, 91, 105] agree that, at the very least, a feature model determines a set of valid feature configurations. And for the moment, that is the only aspect of feature models we are interested in, which motivates the following definition:

- **4.3. Definition (Feature Model):** Given a set of features \mathcal{F} , a *feature model* $\Phi \subseteq \text{Pow}(\mathcal{F})$ is a set of sets of features from \mathcal{F} . Each $F \in \Phi$ is a feature set corresponding to a valid *feature configuration*. ─

Though this is formally a useful notion, it lacks the intuition provided by more diagrammatic descriptions. The most common representation for feature models is the *feature diagram* [35, 60, 166, 167]. We've already seen one in Figure 1.2 (page 8). Their semantics was described extensively by Czarnecki et al. [60] in the aptly named paper “Feature Diagrams and Logics: There and Back Again”. Though they take the concept further, for us it suffices to view a feature diagram as specifying a system of propositional constraints as described in Table 4.1. The corresponding feature model (Definition 4.3) is simply the set of propositional models satisfying those constraints.

- **4.4. Example:** Figure 1.2 represents the following propositional constraints:

$$\begin{array}{lll} Ed & Ed \Leftarrow Pr & Ed \Leftarrow SH \\ Ed \Leftarrow EC & Ed \Leftarrow TI & EC \Leftarrow SA \\ Ed \Rightarrow (EC \wedge \neg TI) \vee (\neg EC \wedge TI) \end{array}$$

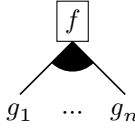
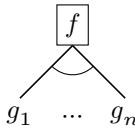


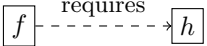
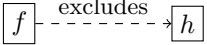
Notation	Meaning
\boxed{f}	root feature f is <i>mandatory</i> f
	f selects <i>at least one</i> branch out of g_1, \dots, g_n $f \Leftrightarrow (g_1 \vee \dots \vee g_n)$
	f selects <i>exactly one</i> branch out of g_1, \dots, g_n $f \Leftarrow (g_1 \vee \dots \vee g_n)$ $f \Rightarrow (g_1 \wedge \neg g_2 \wedge \dots \wedge \neg g_n) \quad \vee$ $\vdots \quad \vee$ $(\neg g_1 \wedge \dots \wedge \neg g_{n-1} \wedge g_n)$
	h is a <i>mandatory subfeature</i> on branch g $g \Leftrightarrow h$
	h is an <i>optional subfeature</i> on branch g $g \Leftarrow h$
	feature f <i>requires</i> feature h $f \Rightarrow h$
	feature f <i>excludes</i> feature h $f \Rightarrow \neg h$

Table 4.1: Compositional semantics for the most common feature diagram notations. More complex feature diagrams are built out of these ingredients by unifying certain features (f, h) and branches (g) without forming directed cycles. A feature can be either a root feature, a mandatory subfeature —● or an optional subfeature —○. A given feature can branch out into subfeatures through any number of groups. Groups can also contain a single branch, in which case exclusive semantics \bigwedge and inclusive semantics \bigvee coincide.

This, in turn, yields the following feature model:

$$\Phi_{\text{Editor}} = \left\{ \begin{array}{l} \{Ed\}, \{Ed, TI\}, \{Ed, EC\}, \{Ed, EC, SA\}, \\ \{Ed, SH\}, \{Ed, SH, TI\}, \{Ed, SH, EC\}, \\ \{Ed, SH, EC, SA\}, \{Ed, Pr\}, \{Ed, Pr, TI\}, \\ \{Ed, Pr, EC\}, \{Ed, Pr, EC, SA\}, \\ \{Ed, Pr, SH\}, \{Ed, Pr, SH, TI\}, \\ \{Ed, Pr, SH, EC\}, \{Ed, Pr, SH, EC, SA\} \end{array} \right\} \quad \lrcorner$$

We do lose some information in this representation; namely, we can't distinguish a feature from any of its mandatory subfeatures. This distinction is not important for the validity of feature configurations, but it does hold intuitive value for developers, and informs the design of the product line architecture. We can represent this extra information by keeping track of a subfeature relation $\Rightarrow \subseteq \mathcal{F} \times \mathcal{F}$. But we won't have any use for this until Chapter 7.

4.3 Product Line Implementation

During the remainder of this chapter we assume a deltoid $Dt = (\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, \llbracket - \rrbracket)$ and a finite set \mathcal{F} of features, unless specified otherwise.

Our goal now is to set up a model for a product line code base with the ability to generate different products for different feature configurations, i.e., with support for automated product derivation.

4.3.1 Product Line Ingredients

To link feature symbols to the implementation layer, each delta in a product line is equipped with an *application condition*, specifying the feature selections for which it should be applied. We map deltas to their application conditions through an *application function*:

- **4.5. Definition (Application Function):** Given a delta set $D \subseteq \mathcal{D}$, an *application function* $\gamma: D \rightarrow \text{Pow}(\text{Pow}(\mathcal{F}))$ expresses, for each delta $d \in D$, the set of feature selections it is applicable to. Thus, $F \in \gamma(d)$ denotes that delta d is applicable to feature selection F . The set $\gamma(d) \subseteq \text{Pow}(\mathcal{F})$ is called the *application condition* of delta d . ┐

In Figure 1.3 (page 9), application conditions are displayed as propositional logic formulas to the bottom right of each delta. While sets of feature configurations are convenient for formal reasoning, the original practice of using propositional formulas [163] is better for developers, as they allow deltas to be annotated with an *open world assumption*:

- **4.6. Example:** Delta $d_{Pr \wedge SH}$ is annotated with $Pr \wedge SH$, the features with which it is concerned. Its application condition is as follows:

$$\gamma(d_{Pr \wedge SH}) = \left\{ \begin{array}{l} \{Ed, Pr, SH\}, \{Ed, Pr, SH, EC, SA\}, \\ \{Ed, Pr, SH, TI\}, \{Ed, Pr, SH, EC\} \end{array} \right\} \quad \lrcorner$$

The advantage of the annotation $Pr \wedge SH$ is that the developer doesn't have to know about the features EC , SA and TI . The propositional annotation continues to be valid even when additional features are added to the feature model; the corresponding set of feature configurations will simply grow with it.

A delta model equipped with an application condition is called *annotated*:

- **4.7. Definition (Annotated Delta Model):** An *annotated delta model* is a tuple $adm = (D, \prec, \gamma)$, where (D, \prec) is a delta model and $\gamma: D \rightarrow 2^{2^{\mathcal{F}}}$ is an application function. The set of all annotated delta models is denoted as \mathcal{aDM} . If the delta set or feature set is not clear from context, we attach a subscript as in $\mathcal{aDM}_{\mathcal{D}, \mathcal{F}}$.

From an annotated delta model we can extract the deltas we need using the chosen feature selection, resulting in the *selected delta model*:

- **4.8. Definition (Selected Delta Model):** Given annotated delta model $adm = (D, \prec, \gamma)$ the delta model *selected* by feature selection $F \in \text{Pow}(\mathcal{F})$ is defined:

$$adm \upharpoonright F \stackrel{\text{def}}{=} (D_F, \prec_F)$$

where the set $D_F = \{d \in D \mid F \in \gamma(d)\}$ contains all applicable deltas, and $\prec_F = (\prec \cap D_F \times D_F)$ is the partial order, restricted accordingly. \dashv

A selected delta model can then be applied to a product as described in Chapter 3 using either sole-derivation, disjunctive or conjunctive semantics (Definitions 3.5, 3.25 and 3.26) to arrive at a target product.

This is the foundation of a *product line implementation*. Each contains a feature model specifying the implemented variations, an *annotated delta model* containing the modifications necessary to obtain them and a *core product* to apply those deltas to:

- **4.9. Definition (Product Line Implementation):** A *product line implementation* is a tuple $PLI = (\Phi, c, D, \prec, \gamma)$, where $\Phi \subseteq \text{Pow}(\mathcal{F})$ is a feature model, $c \in \mathcal{P}$ is the core product, and $adm = (D, \prec, \gamma)$ is an annotated delta model. It is required that the following axioms hold:

- a. All application conditions are valid: $\forall d \in D: \gamma(d) \subseteq \Phi$
- b. All selected delta models are applicable: $\forall F \in \Phi: c \in \text{pre}[\![adm \upharpoonright F]\!]$

The set of all product line implementations is denoted \mathcal{PLI} . If the delta-, product- or featureset is not clear from context, we attach a subscript as in $\mathcal{PLI}_{\mathcal{D}, \mathcal{P}, \mathcal{F}}$. \dashv

So Figures 1.2 and 1.3 (pages 8 and 9) together offer an overview of the whole editor product line implementation, if we take into account that the core product $c = \emptyset$ is just the empty program.

Given a product line implementation, we can generate the end product(s) corresponding to some chosen feature configuration by selecting the correct delta model and applying the result to the core product. We consolidate this in a new notion of semantic evaluation of product line implementations:

- **4.10. Definition (Product Line Evaluation):** *product line evaluation* *Product line evaluation* is a function $\llbracket - \rrbracket: \mathcal{PLJ} \rightarrow \text{Pow}(\text{Pow}(\mathcal{F}) \times \mathcal{P})$ that associates with a given product line implementation PLI a relation $\llbracket PLI \rrbracket \subseteq \text{Pow}(\mathcal{F}) \times \mathcal{P}$ mapping feature selections $F \subseteq \mathcal{F}$ to the products that may be generated by PLI when given F as input. For all product line implementations $PLI = (\Phi, c, adm)$:

$$F \llbracket PLI \rrbracket p \quad \stackrel{\text{def}}{\iff} \quad F \in \Phi \wedge c \llbracket adm \upharpoonright F \rrbracket p$$

The above definition is for sole derivation semantics. Corresponding evaluation functions $\llbracket - \rrbracket_{\cup}$ and $\llbracket - \rrbracket_{\cap}$ for disjunctive and conjunctive semantics (Section 3.5, page 88) are defined analogously. \lrcorner

An effective procedure $\text{prd}: \mathcal{PLJ} \times \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{P})$ corresponding to these semantics —responsible for actually producing specific members of the product line— can be mechanically derived from the delta and delta model application procedures (Sections 3.2 and 3.5).

4.3.2 Unambiguity of Product Lines

Recall that even if deltas are deterministic, a delta model can have multiple distinct derivations, so without any further restrictions we are not guaranteed a unique product for a given feature configuration. When employing sole derivation semantics, it is up to the developers to make sure that all selected delta models are unambiguous (Section 3.3), leading to a unique derivation. We now lift conflict resolution and unambiguity to the product line level.

A product line implementation is unambiguous if every selected delta model is unambiguous. Since not all features are necessarily supposed to work together, we only care about the feature configurations from the embedded feature model:

- **4.11. Definition (Product Line Unambiguity):** A product line implementation $PLI = (\Phi, c, adm)$ is *unambiguous* iff:

$$\text{UA}(PLI) \quad \stackrel{\text{def}}{\iff} \quad \forall F \in \Phi: \text{UA}(adm \upharpoonright F)$$

Recall the UA predicate for delta models from Definition 3.12 (page 79). \lrcorner

We can write out and simplify this condition, which first requires the following definition:

- **4.12. Definition (Joint Application Condition):** Given a set of deltas D' and an application function γ , the set of feature configurations for which all deltas in $D' \subseteq \text{dom}(\gamma)$ are applicable, known as their *joint application condition*, is denoted as follows:

$$\gamma_{\cap}(D') \quad \stackrel{\text{def}}{=} \quad \bigcap_{d \in D'} \gamma(d) \quad \lrcorner$$

- **4.13. Lemma:** Written out and simplified, the unambiguity condition is as follows for a given product line implementation $PLI = (\Phi, c, D, \prec, \gamma)$

$$\underbrace{\forall x, y \in D: x \not\prec y}_{\text{a}} \quad \Rightarrow \quad \underbrace{\forall F \in \gamma_{\cap}(\{x, y\})}_{\text{b}}: \quad \underbrace{\exists z \in D: F \in \gamma(z) \wedge (x, y) \triangleleft z}_{\text{c}}$$

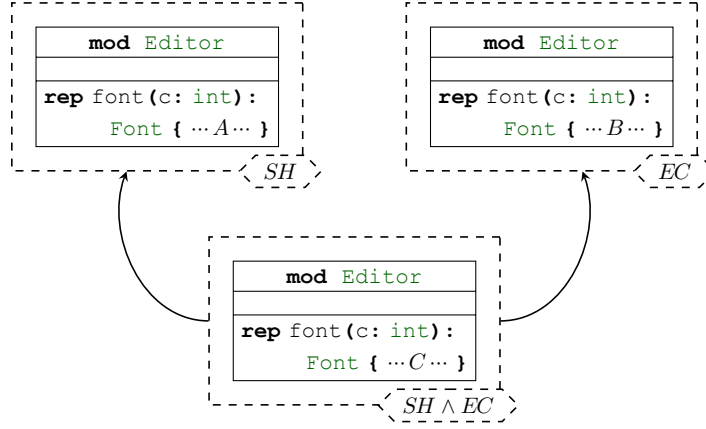


Figure 4.2: We simplify and zoom in on the $d_{SH} \not\leq d_{EC}$ conflict of Figure 1.3 (page 9). The method body C can be found in Example 3.1 (page 71).

This means the unambiguity of a product line implementation can be decided by checking whether (a) for all conflicting deltas $x \not\leq y$, (b) and all feature configurations for which both x and y are selected, (c) that there is a conflict resolving delta z which is also selected for F .

Proof: $UA(PLI)$

$$\begin{aligned}
& \stackrel{1}{\iff} \forall F \in \Phi: UA((D, \prec, \gamma) \upharpoonright F) \\
& \stackrel{2}{\iff} \forall F \in \Phi: \forall x, y \in D_F: x \not\leq y \Rightarrow \exists z \in D_F: (x, y) \triangleleft z \\
& \stackrel{3}{\iff} \forall x, y \in D: \forall F \in \gamma_\cap(\{x, y\}): x \not\leq y \Rightarrow \exists z \in D_F: (x, y) \triangleleft z \\
& \stackrel{4}{\iff} \forall x, y \in D: \forall F \in \gamma_\cap(\{x, y\}): x \not\leq y \Rightarrow \exists z \in D: F \in \gamma(z) \wedge (x, y) \triangleleft z \\
& \stackrel{5}{\iff} \forall x, y \in D: x \not\leq y \Rightarrow \forall F \in \gamma_\cap(\{x, y\}): \exists z \in D: F \in \gamma(z) \wedge (x, y) \triangleleft z
\end{aligned}$$

Steps 1 and 2 apply Definitions 3.12 and 4.11. Step 3 is valid because both before and after the ‘swap’, the second universal quantification is restricted so as to end up with two deltas and a feature configuration for which they are applicable. Step 4 pushes the applicability of z inward by similar justification. Finally, in step 5 the conflict condition $x \not\leq y$ can be pulled out because it is independent from the choice of F . \square

As the set of feature configurations can be exponential in the number of features, this check could be rather expensive. As an alternative, we propose the notion of a *globally unambiguous* product line implementation, a property which implies unambiguity:

- **4.14. Definition (Global Unambiguity):** A product line implementation $PLI = (\Phi, c, D, \prec, \gamma)$ is *globally unambiguous* iff the following holds:

$$\begin{aligned}
GUA(PLI) \quad \stackrel{\text{def}}{\iff} \quad & \forall x, y \in D: (x \not\leq y \wedge \gamma_\cap(\{x, y\}) \neq \emptyset) \Rightarrow \\
& \exists z \in D: \gamma_\cap(\{x, y\}) \subseteq \gamma(z) \wedge (x, y) \triangleleft z \quad \lrcorner
\end{aligned}$$

So a product line implementation is globally unambiguous iff for every pair of conflicting deltas applied together, there is also one conflict resolving delta applied for at least the same feature configurations.

- **4.15. Theorem:** The Editor product line implementation of Section 1.4 is globally unambiguous.

Proof: The only two potential conflicts are $d_{SH} \not\triangleleft d_{EC}$ and $d_{EC} \not\triangleleft d_{TI}$. The former is resolved by $(d_{SH}, d_{EC}) \triangleleft d_{SH \wedge EC}$, with $\gamma_{\cap}(\{d_{SH}, d_{EC}\}) = \gamma(d_{SH \wedge EC})$, as illustrated in Figure 4.2. The latter does not need to be resolved, as $\gamma_{\cap}(\{d_{EC}, d_{TI}\}) = \emptyset$; those two deltas are never selected together. \square

Global unambiguity can be checked by inspecting the product line implementation once and does not require any selected delta models to be generated. The following theorem states that any globally unambiguous product line implementation is also unambiguous:

- **4.16. Theorem:** A product line implementation that is globally unambiguous is also guaranteed to be unambiguous.

Proof: In the following proof, γ_y^x is used as an abbreviation for $\gamma_{\cap}(\{x, y\})$:

$$\begin{aligned}
 & \text{GUA}(PLI) \\
 & \xLeftrightarrow{1} \forall x, y \in D: (x \not\triangleleft y \wedge \gamma_y^x \neq \emptyset) \Rightarrow \exists z \in D: \gamma_y^x \subseteq \gamma(z) \quad \wedge (x, y) \triangleleft z \\
 & \xLeftrightarrow{2} \forall x, y \in D: (x \not\triangleleft y \wedge \gamma_y^x \neq \emptyset) \Rightarrow \exists z \in D: \forall F \in \gamma_y^x: F \in \gamma(z) \quad \wedge (x, y) \triangleleft z \\
 & \xRightarrow{3} \forall x, y \in D: (x \not\triangleleft y \wedge \gamma_y^x \neq \emptyset) \Rightarrow \forall F \in \gamma_y^x: \exists z \in D: F \in \gamma(z) \quad \wedge (x, y) \triangleleft z \\
 & \xLeftrightarrow{4} \forall x, y \in D: x \not\triangleleft y \Rightarrow \forall F \in \gamma_y^x: \exists z \in D: F \in \gamma(z) \quad \wedge (x, y) \triangleleft z \\
 & \xLeftrightarrow{5} \text{UA}(PLI)
 \end{aligned}$$

Steps 1 and 5 apply Definition 4.14 and Lemma 4.13. Step 2 is justified because it is performed in a context where it is known that $\gamma_y^x \neq \emptyset$. In step 4 that condition has become redundant, so it can be removed.

More interesting is the implication of step 3, which clarifies the difference between global unambiguity and general unambiguity. An implementation can be unambiguous without being globally unambiguous if some of its conflicts are resolved by different deltas z for different feature configurations F . \square

It seems to be rare that a practical situation calls for different conflict resolvers for different feature configurations. Global unambiguity is easier to establish. In the delta modeling workflow described in Chapter 7, for example, it is guaranteed by construction.

4.4 Product Line Specification

We now define the concept of a *product line specification*: an abstraction of the desired semantics of a product line. Such a specification can be used both to guide the implementation process (further discussed in Chapter 7) and to verify the correctness of a given implementation.

A product line specification has two ingredients. The first is a feature model, as introduced in Section 4.2, which expresses the set of feature configurations that *should* be supported by the product line. The second ingredient is a *valuation function*, an abstract representation of the desired semantics for every feature. Note, in particular, that deltas don't play a rôle, so we can model specifications for any product line implementation technique.

4.4.1 Valuation Functions

A valuation function is a semantic interpretation of the *requirements* imposed on a product when it should support certain features. It maps a feature selection to the set of products deemed to implement those features correctly:

- **4.17. Definition (Valuation Function):** A *valuation function* $V: \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{P})$ is a function taking a feature selection and returning the set of products that satisfy the required semantics of that feature selection. The following axiom needs to hold for all valuation functions V :

$$\text{a. Compositionality: } \forall F, G \subseteq \mathcal{F}: V(F \cup G) \subseteq V(F) \cap V(G) \quad \lrcorner$$

Valuation functions are a concept originally from modal logic, a context in which we shall revisit them in Chapter 6. Axiom 4.17a represents a reasonably intuitive concept: if a product supports some combination of features, it also supports each feature individually — and every combination in between. The reverse is not generally true. A product can support a number of features individually without properly implementing the requirements of their combination.

- **4.18. Example:** For example, the Editor product $p = (d_{SH} \cdot d_{Pr})(c)$ may implement both Printing and Syntax Highlighting, i.e., $p \in V(\{SH\}) \cap V(\{Pr\})$, but does not implement the combined functionality that we intended; namely, syntax highlighting on the printout: $p \notin V(\{Pr, SH\})$. However, the product $q = (d_{Pr \wedge SH} \cdot d_{SH} \cdot d_{Pr})(c)$ *does* implement the combination: $q \in V(\{Pr, SH\})$. This is what we call *desired feature interaction*. \lrcorner

Even in an abstract setting, we can use the valuation function to express some useful properties. For instance,

$$V(F \cup G) \neq V(F) \cap V(G)$$

means that special interaction is desired between the features of F and G . And

$$\forall F \subseteq \mathcal{F}: V(F) = V(F \cup G)$$

indicates that features in G have no semantics. This sometimes happens when features are used purely to categorize their subfeatures. Chapter 7 includes an example of this in an industrial case study.

We intend a valuation function to be represented *syntactically*, though this is difficult to demonstrate in an abstract setting. The following is an incomplete list of possible representations for the valuation function:

- In an object oriented setting, such as that of our running example, it might contain formal specifications regarding the presence and behavior of packages, classes and methods, to be verified statically. Alas, exploring this option is outside the scope of the thesis.
- In an industrial setting it may be used to support *test driven development* of software product lines. It would be represented as a collection of test cases T annotated with application conditions. The set $V(F)$ would contain the products that pass all test-cases that are annotated with F .

Taking this one step further, it would suggest the practice of maintaining test cases as a delta-based product line implementation, in parallel to the main software product line, and with roughly the same shape. It

would be based on a deltoid such as $(\text{Pow}(T), \text{Pow}(T), \cup)$, where products are sets of test-cases that the corresponding software product needs to pass, and deltas can augment them with additional test-cases.

4.4.2 Product Line Specifications

A product line specification contains a feature model and a valuation function:

- **4.19. Definition (Product Line Specification):** A *product line specification* is a pair $PLS = (\Phi, V)$ where $\Phi \subseteq \text{Pow}(\mathcal{F})$ is a feature model (Definition 4.3) and $V: \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{P})$ is a valuation function (Definition 4.17). We require that the following axiom holds:

- a. No contradictory requirements: $\forall F \in \Phi: V(F) \neq \emptyset$

The set of all product line specifications is denoted \mathcal{PLS} . If the deltoid or feature set is not clear from context, we attach a subscript as in $\mathcal{PLS}_{D_t, \mathcal{F}}$. \lrcorner

Semantically speaking, a product line *implementation* can be seen as a (partial) function performing automated product derivation, taking a feature configuration and returning the corresponding product (Figure 1.1). A *specification*, then, can be seen as the pre- and postcondition for that function. The feature model is the precondition; the valuation function is the postcondition.

This allows us to define a formal notion of product line correctness. That is, correctness of a product line implementation, both partial and total, with regard to a product line specification:

- **4.20. Definition (Product Line Correctness):** A product line implementation PLI is *partially correct* or *totally correct* w.r.t. a product line specification $PLS = (\Phi_S, V)$ respectively iff:

$$\begin{array}{lcl} PLI \models PLS & \stackrel{\text{def}}{\iff} & \underbrace{\forall F \in \Phi_S: F \in \text{pre } \llbracket PLI \rrbracket}_{\text{a}} \Rightarrow \underbrace{\llbracket PLI \rrbracket(F) \subseteq V(F)}_{\text{c}} \\ PLI \models_{\text{tot}} PLS & \stackrel{\text{def}}{\iff} & \underbrace{\forall F \in \Phi_S: F \in \text{pre } \llbracket PLI \rrbracket}_{\text{b}} \wedge \underbrace{\llbracket PLI \rrbracket(F) \subseteq V(F)}_{\text{c}} \end{array}$$

Correctness for disjunctive and conjunctive semantics are defined analogously (Section 3.5). \lrcorner

Note its similarity to Definition 2.29 (page 45): (a) for all feature configurations of the specification, (b) if PLI implements that feature configuration, (c) then any product it might generate is valid according to the valuation function.

The following formulation is equivalent, but may offer more insight because it is at a lower level:

- 4.21. Lemma:** For product line implementation $PLI = (\Phi_I, c, \text{adm})$ and product line specification $PLS = (\Phi_S, V)$ we have:

$$\begin{array}{lcl} PLI \models PLS & \iff & \forall F \in \Phi_S: \left(F \in \Phi_I \Rightarrow \llbracket \text{adm} \upharpoonright F \rrbracket(c) \subseteq V(F) \right) \\ PLI \models_{\text{tot}} PLS & \iff & \forall F \in \Phi_S: \left(F \in \Phi_I \wedge \llbracket \text{adm} \upharpoonright F \rrbracket(c) \subseteq V(F) \right) \end{array}$$

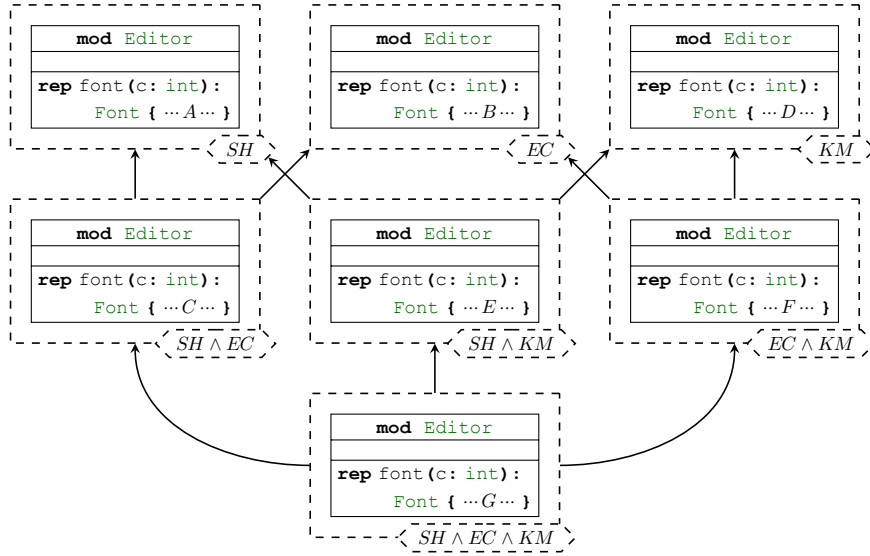


Figure 4.3: An extended version of Figure 4.2, explained in Example 4.22.

Proof: It is a relatively direct translation using Definition 4.10. Just note that

$$F \in \text{pre} \llbracket PLI \rrbracket \iff F \in \Phi_1$$

is a valid equivalence because of Axiom 4.9b. \square

4.5 Parametric Deltas

Now that the foundation for delta-based product lines is in place, we explore a problem with the approach. Conflict resolving deltas may be flexible, but they are also bulky. Section 4.5.1 considers a situation in the Editor product line where three independent deltas are all in conflict with each other, requiring a number of conflict resolving deltas exponential to the number of features.

To accomodate a more lightweight approach, Section 4.5.2 introduces *parametric deltas*, which can pass a chosen feature configuration on to the language of the underlying deltoid. This can give us some of the convenience of annotative variability techniques inside deltas. Parametric deltas do have their downsides, however. To offer some contrast, Section 4.5.3 presents an alternative solution for the Editor problem, based on fine-grained deltas.

Section 4.6 builds on the potential of parametric deltas and uses them to formalize *nested product lines*: product line implementations using nesting annotated delta models as their base.

4.5.1 Combinatorial Explosion of Conflict Resolvers

A potential problem with “a resolver for every conflict” is that an exponential number of them is required if many deltas mutually conflict:

▷ **4.22. Example:** Let’s revisit the conflict in the Editor product line.

The two deltas d_{SH} and d_{EC} are each responsible for implementing a feature: $\gamma(d_{SH}) = \{F \in \Phi \mid SH \in F\}$ and $\gamma(d_{EC}) = \{F \in \Phi \mid EC \in F\}$. They are in conflict: $d_{SH} \not\leq d_{EC}$, and we want to maintain global unambiguity. The idea is to develop a conflict resolving delta $d_{SH \wedge EC}$, applicable when both of those features are selected (Definition 4.14): $\gamma(d_{SH \wedge EC}) = \{F \in \Phi \mid SH, EC \in F\}$. So far so good.

Now what if a new feature is implemented: *Keyword Marking* (KM). It is similar to Syntax Highlighting, but responsible for giving keywords a bold typeface. To realize this feature, an additional delta is designed: d_{KM} . The problem is, d_{KM} has to redefine `font(int)` just like its siblings, so it is in conflict with both of them. Moreover, by the new feature model, all three features are independently selectable.

In order to regain global unambiguity and ensure the proper semantics for each feature configuration, we need to create at least two more conflict resolving deltas. Namely, $d_{SH \wedge KM}$ and $d_{EC \wedge KM}$, resolving $d_{SH} \not\leq d_{KM}$ and $d_{EC} \not\leq d_{KM}$. But it doesn't end there. The three conflict resolving deltas are now in conflict with *each other*. Thankfully, it does end eventually: if any two of the conflict resolving deltas are selected, the third will always be selected too. For example, we have $\gamma(d_{SH \wedge EC}) \cap \gamma(d_{EC \wedge KM}) \subseteq \gamma(d_{SH \wedge KM})$. So to wrap up this *three-way conflict*, one final conflict resolving delta $d_{SH \wedge EC \wedge KM}$ is needed.

Figure 4.3 illustrates that we need four conflict resolving deltas to fully resolve the conflicts arising from three features. The following is method body C , resolving $d_{SH} \not\leq d_{EC}$ (recall the \mathfrak{C} notation from page 71):

```
(C) 1 Font result = new Font();
    2 result.setColor(font@dSH(c).color());
    3 result.setUnderlined(font@dEC(c).underlined());
    4 return result;
```

The following is method body E , resolving $d_{SH} \not\leq d_{KM}$:

```
(E) 1 Font result = new Font();
    2 result.setColor(font@dSH(c).color());
    3 result.setBold(font@dKM(c).bold());
    4 return result;
```

The following is method body F , resolving $d_{EC} \not\leq d_{KM}$:

```
(F) 1 Font result = new Font();
    2 result.setUnderlined(font@dEC(c).underlined());
    3 result.setBold(font@dKM(c).bold());
    4 return result;
```

The following is method body G , resolving the three-way conflict:

```
(G) 1 Font result = new Font();
    2 result.setColor(font@dSH(c).color());
    3 result.setUnderlined(font@dEC(c).underlined());
    4 result.setBold(font@dKM(c).bold());
    5 return result;
```

」

These deltas are not duplicating behavior, as such. They only combine behaviors by referring to the deltas that originally implement them. But they *are* duplicating code; boilerplate code, if you will. In the worst case, n independent features with conflicting implementations require $2^n - 1$ deltas, $2^n - n - 1$ of which are conflict resolvers.

It is possible that each of those feature combinations actually requires a distinct implementation, depending on the product line specification. In that case, this exponential complexity is inherent in the problem and delta models like the one in Figure 4.3 are precisely what we need for full control. But this is obviously not the case for the three-way conflict of the Editor. Indeed, in many practical scenarios such as this one, the glue code follows a uniform pattern which may be much more conveniently expressed in the underlying language.

4.5.2 Parametric Deltas

The way to accomplish the goal as described above is to allow the underlying structure of the delta access to the chosen feature configuration. We first define the abstract notion of such a *parametric delta*. We then instantiate it to software deltas. Finally we show how this solves our three-way conflict problem.

Abstract Parametric Deltas

Intuitively, we want every parametric delta to represent a partial function, accepting a feature selection as parameter and returning a traditional delta. In previous work [5] this was literally the case. But the strict separation of syntax and semantics in this thesis requires a different formulation. A partial function is a semantic concept, and we want to leave the representation of parametric deltas open to be decided for each concrete domain:

- **4.23. Definition (Parametric Deltoid):** A *parametric deltoid* is a deltoid $(\mathcal{P}, p\mathcal{D} \times \text{Pow}(\mathcal{F}), \llbracket -, - \rrbracket)$ with some set $p\mathcal{D}$ of what we call *parametric deltas* and a semantic evaluation function $\llbracket -, - \rrbracket: p\mathcal{D} \times \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{P} \times \mathcal{P})$. \lrcorner

This does not change the basic concept of deltoid (Definition 2.11, page 37); we simply make the set of deltas $\mathcal{D} = p\mathcal{D} \times \text{Pow}(\mathcal{F})$ a set of pairs, each containing a parametric delta and a feature selection (its ‘parameter’). The behavior of a semantic delta $\llbracket pd, F \rrbracket \subseteq \mathcal{P} \times \mathcal{P}$ is based on both.

However, when working with a parametric deltoid, delta models and product lines will be based on $p\mathcal{D}$ rather than \mathcal{D} . We need to adapt their respective semantic evaluation functions to pass on and supply the feature configuration at selection time:

- **4.24. Definition (Parametric Delta Model Evaluation):** Given a parametric deltoid $(\mathcal{P}, p\mathcal{D} \times \text{Pow}(\mathcal{F}), \llbracket -, - \rrbracket)$, *parametric delta model evaluation* $\llbracket -, - \rrbracket: \mathcal{DM}_{p\mathcal{D}} \times \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{P} \times \mathcal{P})$ is defined as follows for all *parametric delta models* $pdm \in \mathcal{DM}_{p\mathcal{D}}$ with a unique derivation $\text{derv}(pdm) = \{pd\}$, and all feature selections $F \subseteq \mathcal{F}$:

$$\llbracket pdm, F \rrbracket \stackrel{\text{def}}{=} \llbracket pd, F \rrbracket$$

Corresponding evaluation functions $\llbracket -, - \rrbracket_{\cup}$ and $\llbracket -, - \rrbracket_{\cap}$ for disjunctive and conjunctive semantics are defined analogously. \lrcorner

- **4.25. Definition (Parametric Product Line Evaluation):** Given a parametric deltoid $(\mathcal{P}, p\mathcal{D} \times \text{Pow}(\mathcal{F}), \llbracket -, - \rrbracket)$, *parametric product line evaluation* $\llbracket - \rrbracket: \mathcal{PLJ}_{p\mathcal{D}} \rightarrow \text{Pow}(\text{Pow}(\mathcal{F}) \times \mathcal{P})$ is defined as follows for all *parametric product line implementations* $pPLI = (\Phi, c, padm) \in \mathcal{PLJ}_{p\mathcal{D}}$:

$$F \llbracket pPLI \rrbracket p \stackrel{\text{def}}{\iff} F \in \Phi \wedge c \llbracket padm \upharpoonright F, F \rrbracket p$$

Corresponding evaluation functions $\llbracket - \rrbracket_{\cup}$ and $\llbracket - \rrbracket_{\cap}$ for disjunctive and conjunctive semantics are defined analogously. \lrcorner

Parametric Software Deltas

Concrete parametric deltas can be realized in any number of ways. For software deltas it would feel most natural to expose the available feature symbols $f \in \mathcal{F}$ as boolean constants in the underlying programming language:

- **4.26. Definition (Parametric Software Deltas):** The set of *parametric software deltas* $p\mathcal{D}_{\text{pkg}+}$ is like the set of fine-grained software deltas $\mathcal{D}_{\text{pkg}+}$, but can contain feature symbols $f \in \mathcal{F}$ in any Boolean context. \lrcorner

Semantic evaluation will substitute truth-values for the feature symbols to get back to the fine-grained software deltoid situation of Definition 3.22:

- **4.27. Definition (Parametric Software Delta Evaluation):** Semantic evaluation for parametric software deltas $\llbracket -, - \rrbracket: p\mathcal{D}_{\text{pkg}+} \times \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{PKG} \times \mathcal{PKG})$ is defined as follows. For all products $p, q \in \mathcal{PKG}$, deltas $pd \in p\mathcal{D}_{\text{pkg}+}$ and feature configurations $F \subseteq \mathcal{F}$:

$$p \llbracket pd, F \rrbracket q \stackrel{\text{def}}{\iff} p \llbracket d \rrbracket q$$

where $\llbracket - \rrbracket$ is fine-grained software delta evaluation (Definition 3.22) and $d \in \mathcal{D}_{\text{pkg}+}$ is the software delta obtained by replacing every occurrence of $f \in F$ in pd with *true* and every occurrence of $f \in (\mathcal{F} \setminus F)$ in pd with *false*. \lrcorner

A Parametric Editor Product Line

The following example, illustrates a possible way to solve the three-way conflict problem of the Editor product line. The four conflict resolving deltas of Example 4.22 are so similar, it would be more sensible to use a single parametric delta:

- **4.28. Example:** The parametric solution to the three-way conflict problem in the Editor product line is shown in Figure 4.4. The following is method body H , combining the three features:

```
(H) 1 Font result = new Font();
    2 if (SH) result.setColor(font@dSH(c).color());
    3 if (EC) result.setUnderlined(font@dEC(c).underlined());
    4 if (KM) result.setBold(font@dKM(c).bold());
    5 return result;  $\lrcorner$ 
```

During the process of automated product derivation, the feature symbols in the implementation are replaced with the proper truth values (Definition 4.27):

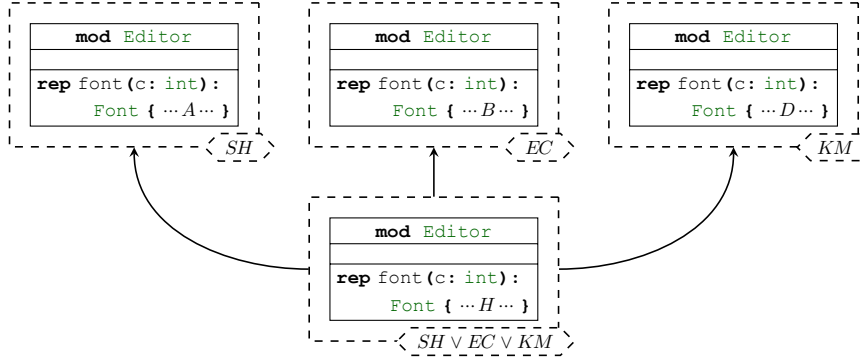


Figure 4.4: A version of Figure 4.3 using a parametric delta. Method body H can be found in Example 4.29.

▷ **4.29. Example:** Given a feature configuration of $F = \{ SH, KM \}$, the method body H from Example 4.28 would become the following:

```
(H') 1 Font result = new Font();
      2 if (true) result.setColor      (font@dSH(c).color());
      3 if (false) result.setUnderlined(font@dEC(c).underlined());
      4 if (true) result.setBold      (font@dKM(c).bold());
      5 return result;
```

We assume that these **if** constructs are resolved at compile-time, so that Line 3 can be discarded. ┘

Note that the Example 4.28 method body is basically using an *annotative* variability technique. Parametric deltas offer a mix of the annotative and compositional approaches [108]. A powerful combination.

But with great power comes great responsibility [174]. In principle, a whole product line could be encoded in a single parametric delta, in which the code is annotated with feature conditions to handle all cases. But, as discussed in Section 1.2.3, annotative approaches do not benefit from modularity or separation of concerns, which were among our main goals. Therefore, parametric deltas are recommended only for resolving multi-way conflicts or feature interactions, and then only when this significantly reduces the amount of code or effort required. Parametric deltas are a double-edged sword.

4.5.3 Interlude: A Fine-grained Software Delta Solution

For the three-way conflict problem in the Editor product line, there is actually an alternative solution using only the facilities of fine-grained software deltas:

▷ **4.30. Example:** The three-way conflict can be avoided altogether by using the fine-grained software delta **insert** operation as shown in Figure 4.5. The following is statement delta I , which inserts the behavior of SH :

```
(I)  insert { result.setColor(SHfont.color()); };
```

The following is statement delta J , which inserts the behavior of EC :

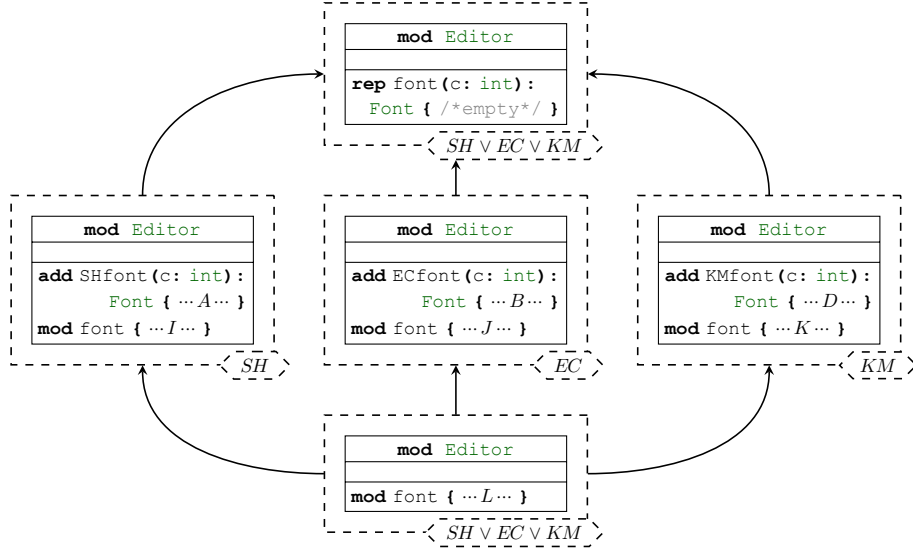


Figure 4.5: An alternative to Example 4.28, explained in Example 4.30.

(J) **insert** { result.setUnderlined(ECfont.underlined()); };

The following is statement delta K , which inserts the behavior of KM :

(K) **insert** { result.setBold(KMfont.bold()); };

The following is statement delta L , which adds the surrounding statements:

(L) 1 **prepend** { Font result = new Font(); };
 2 **append** { return result; }; ┘

This is more modular, but it is also more bulky, and the nondeterministic nature of **insert** needs to be understood well enough to avoid potential pitfalls. Parametric software deltas are based on more generally familiar annotative techniques, and may be preferable in certain situations.

4.6 Nested Product Lines

Section 3.6 discussed nested delta models. At this point the next logical step is to extend them to nested annotated delta models, in order to get *nested product lines*.

There is a way we can achieve something along those lines already, just by applying previously defined concepts. We can equip a product line implementation with an annotated delta model (D, \prec, γ) based on a delta set $D \subseteq \mathcal{D}_{\Delta}$ (Definition 3.31). This results in a nesting delta model of which only the outermost deltas are annotated with an application condition, something we might call a *shallow annotation* (Figure 4.6a).

But if we want to achieve a *deep annotation*, parametric deltas provide a way:

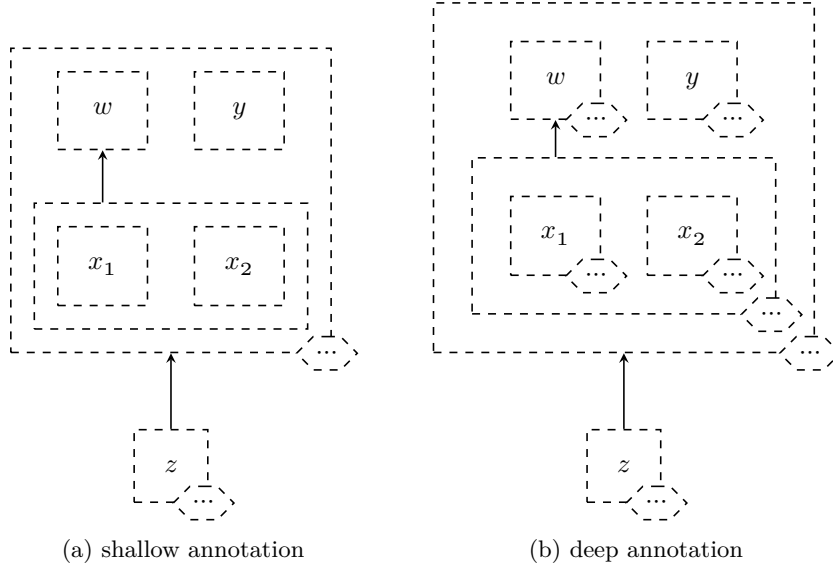


Figure 4.6: The difference between a shallow and a deep annotation.

► **4.31. Definition (Annotated Delta Model Closure):** Define the following family of parametric delta sets for all natural numbers $n \in \mathbb{N}$:

$$\begin{aligned} \mathcal{aD}_{\Delta,0} &\stackrel{\text{def}}{=} \mathcal{D} \\ \mathcal{aD}_{\Delta,n+1} &\stackrel{\text{def}}{=} \mathcal{aD}_{\Delta,n} \cup \mathcal{aDM}_{\mathcal{aD}_{\Delta,n}} \end{aligned}$$

We then define the *annotated delta model closure* of \mathcal{D} as follows:

$$\mathcal{aD}_{\Delta} \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \mathcal{aD}_{\Delta,n}$$

We require that \mathcal{D} didn't contain any annotated delta models to begin with. \lrcorner

Finally, we define an appropriate evaluation function, just as we did for parametric software deltas before (Definition 4.27):

► **4.32. Definition (Deeply Annotated Delta Evaluation):** Given a deltoid $Dt = (\mathcal{P}, \mathcal{D}, (-))$ and parametric deltoid $pDt = (\mathcal{P}, \mathcal{aD}_{\Delta} \times \text{Pow}(\mathcal{F}), \llbracket -, - \rrbracket)$, we define *deeply annotated delta model evaluation* $\llbracket -, - \rrbracket: \mathcal{aD}_{\Delta} \times \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{P} \times \mathcal{P})$ as follows for all simple deltas $d \in \mathcal{D}$, annotated delta models $adm \in \mathcal{aDM}_{\mathcal{aD}_{\Delta}}$ and feature configurations $F \subseteq \mathcal{F}$:

$$\begin{aligned} \llbracket adm, F \rrbracket &\stackrel{\text{def}}{=} \llbracket adm \upharpoonright F, F \rrbracket \\ \llbracket d, F \rrbracket &\stackrel{\text{def}}{=} (d) \end{aligned}$$

Corresponding evaluation functions $\llbracket -, - \rrbracket_{\cup}$ and $\llbracket -, - \rrbracket_{\cap}$ for disjunctive and conjunctive semantics are defined analogously. \lrcorner

A *nested product line implementation* is then simply a *parametric product line implementation* (Definition 4.25) based on a deeply annotated deltoid (Definition 4.32). A resulting annotated delta diagram would look like Figure 4.6b.

4.7 Conclusion

We’ve spoken of features before now, but this chapter is where the concept of feature is formally introduced and integrated into ADM. These features are merely labels, but play a prominent rôle throughout the rest of the story. They are traditionally used in a *feature model* as a way to identify the possible products of a *product line*.

In ADM, producing the product corresponding to a specific feature selection is done by preparing a large delta model, and annotating each delta with an *application condition*. Specifications may be prepared for each significant feature combination; feature combination —not feature— because often, two features that are otherwise independent need to satisfy additional requirements when they are selected together.

Apart from providing a characterisation of product line correctness, this chapter lifts a number of concepts from Chapter 3 to the product line level, such as unambiguity and nesting, and introduces a number of other useful concepts, such as *parametrized deltas*.

4.8 Related Work

Much work related to ADM-based product lines has been discussed in earlier chapters. But there are a number of interesting comparisons left to make with regard to feature modeling and product derivation.

4.8.1 Feature Modeling

Feature-Oriented Domain Analysis (FODA) as a way to model the commonality and the variability of a set of systems on a specification level has been in use since the early 1980’s [105]. In this thesis, as in its corresponding publications, feature models are used (Section 4.2) to express this variability. But feature modeling is not the only studied approach for characterizing different products in a product line. Czarnecki et al. [61] compare feature modeling with *decision modeling*, which is based on setting values for specific Boolean, numerical and enumerated variables. It is interesting to note that if a feature model is simplified to a set of feature configurations, as we do in Definition 4.3, it becomes very similar to a decision model, e.g., one with a set of Boolean variables. According to Czarnecki et al., most of the differences between the two approaches are historical and the two are converging. The biggest remaining difference is that feature modeling has specific support for expressing commonality as well as variability.

4.8.2 Product Derivation

First, we should note that our definition of ‘product’ deviates somewhat from existing literature in feature modeling [26, 27, 53, 66], in which a product is usually uniquely defined for a given feature configuration. For us, the term refers to a specific implementation, multiple of which may be potential candidates for a given selection of features (Definition 4.10).

Automated product derivation has been widely recognized to improve quality and reduce time to market for large software systems [57, 65]. Perrouin et al. [150] note a dichotomy in the way variability approaches support product derivation. On the one hand, they say, approaches that support fully automated product derivation lack the flexibility to adapt to the needs of specific customers. On the other hand, approaches that focus on flexibility lack automation. They propose an approach where a feature configuration directs the *composition* of core assets, the result of which is then *transformed* to obtain a target product.

We would suggest that ADM is fully capable of modeling this approach, as composition and transformation are much the same for ADM. The transformation required to implement ‘special wishes’ for a specific feature configuration can be encoded in a delta. It is then a simple matter to annotate this delta with a specific (i.e., narrow) application condition (Definition 4.5), and make it a maximal element in the application order (Definitions 1.23 and 3.2).

Existing compositional approaches to automated product derivation have been discussed exhaustively in previous chapters. Existing annotative approaches include conditional compilation [171], frames [181] and COLORED FEATHERWEIGHT JAVA (CFJ) [106]. Another noteworthy existing project is CIDE (Colored IDE) [108], a tool which displays annotated code in different colors—each representing a feature—and achieves a kind of visual separation of concerns this way, gaining some of the benefits of compositional techniques.

L^AT_EX Meets Delta Modeling

Deltas for Document Generation and Package Management



5.1 Introduction

In this chapter we take a small break from the Editor example, and discuss the contributions of delta modeling to L^AT_EX, the typesetting language [118].

It has been mentioned before that ADM, being an abstract formalism, applies to a domain wider than just software. Indeed, the principles of ADM may prove quite valuable for preparing families of technical documents. For example, it is often wise to tailor your curriculum vitae (or résumé, if you prefer) to the position that you are applying for. A university course may ask students to buy a textbook, but then only use a small part of it; in fact, many textbooks include course outlines or annotations for level of difficulty [21, 36, 116, 155], already assuming that only a part of it will be perused, basically wasting paper. Save the planet — use delta modeling!

Goal: Implement delta modeling for the L^AT_EX language.

L^AT_EX makes a particularly suitable language to showcase delta modeling. The T_EX language [115], upon which L^AT_EX is based, is a domain specific language meant for preparing technical documents, but it also happens to be Turing complete. At its core it is a macro language capable of manipulating arbitrary strings (more accurately called *token lists*). The relatively recent L^AT_EX3 programming layer [137, 176] includes what is essentially a **while**-language supporting various data-structures.

Moreover, most of the T_EX language can be redefined from within the language itself, giving programmers an inordinate amount of control; a fact famously demonstrated by Carlisle [49] when he wrote the following T_EX program which generates the full lyrics to “The Twelve Days of Christmas”:

```

1 \let~\catcode~`76~`A13~`F1~`j00~`P2jdefA71F~`7113jdefPALLF
2 PA'FwPA;;FPAZZFLaLPA//71F71iPAHHFLPAzzFenPASSFthP;A$$FevP
3 A@@FfPARR717273F737271P;ADDFRgniPAWW71FPATTFvePA**FstRsamP
4 AGGFRruoPAqq71.72.F717271PAY7172F727171PA??Fi*LmPA&&71jfi
5 Fjfi71PAVVFjbigskipRPWGAUU71727374 75,76Fjpar71727375Djifx
6 :76jelse&U76jfiPLAKK7172F7117271PAXX71FVLnOSeL71SLRyadR@oL
7 RrhC?yLRurtKFeLPFovPgaTLtReRomL;PABB71 72,73:Fjif.73.jelse
8 B73:jfiXF71PU71 72,73:PWs;AMM71F71diPAJJFRdriPAQQFRsreLPAI
9 I71Fo71dPA!!FRgiePBt'el@ lTLqdrYmu.Q.,Ke;vz vzLqip.Q.,tz;
10 ;Lql.IrsZ.eap,qn.i. i.eLlMaesLdRcna,;!;h htLqm.MRasZ.ilK,%
11 s$;z zLqs'.ansZ.Ymi,/sx ;LYegseZRyal,@i;@ TLRlogdLrDsW,@;G
12 LcYlaDLbJsw,SWXJW ree @rzchLhzw,;WERcesInW qt.'oL.Rtrul;e
13 doTsW,Wk;Rri@stW aHAHHFndZPpqar.tridgeLinZpe.LtYer.W,:jbye

```

Because of the great expressiveness of the language, L^AT_EX delta modeling can be implemented from within L^AT_EX itself — a practice known as *monkey-patching* [138] — though the operating domain for the deltas will need to be limited to specific subsets of the language.

This great expressiveness can also be a problem at times. Since L^AT_EX has no encapsulation or formal namespacing, L^AT_EX packages written by different people often conflict with each other — a well-known problem with the L^AT_EX ecosystem. Luckily, we know how to deal with conflicts.

Goal: Use ADM principles to manage dependencies and conflicts between independent L^AT_EX packages.

In accordance with the two goals described above, this chapter introduces two L^AT_EX packages. We first look at the `delta-modules` package in Section 5.2, which can be used to specify families of technical documents and includes an implementation of automated document generation. As a case-study we use this very PhD thesis, a member of what we shall call the *Thesis product line*. Other documents belonging to this product line skip certain topics for a selective reading experience. The source and the members of the Thesis product line can be dynamically generated and downloaded from the following URL:

<http://www.mhelvans.net/phd-thesis>

Section 5.3 describes the ADM-based package manager `pkgloader`, thereby addressing the package management problem. Finally, Section 5.4 offers concluding remarks and Section 5.5 briefly discusses related work.

5.2 `delta-modules`: Deltas for Document Generation

We now introduce the L^AT_EX delta modeling functionality that was used to organize the content of this thesis. This functionality is provided by the L^AT_EX package called `delta-modules`. The following subsections describe the process of building a L^AT_EX product line implementation (Section 4.3) using this package, with the structure of this thesis as an example. The underlying deltoid is implied by the available L^AT_EX commands, rather than defined formally.

5.2.1 Building the Feature Model

Users of the package declare a feature model using the `\DeclareFeature` command:

- ▷ 5.1. **Definition (`\DeclareFeature`):** The `\DeclareFeature` command offers the following syntax for the declaration of features and feature-relations:

```

declare-f  ::=  \DeclareFeature [ * ] { <fid> } { <f-relation> }
<f-relation> ::= extends { <fid> { , <fid> } }
               | requires { <fid> { , <fid> } }
               | excludes { <fid> { , <fid> } }

```

where `<fid>` represents a feature name, which can be an arbitrary string, apart from some L^AT_EX-specific exceptions. ┘

Each use of the command declares a feature and its relation to other features. **extends** indicates that the left-hand feature is a subfeature of the right-hand feature. **requires** and **excludes** take their respective meanings from feature diagram terminology (Table 4.1). The optional asterisk to the right of the command name indicates that the declared feature is mandatory, relative to its superfeatures, if any.

- ▷ 5.2. **Example (Thesis Feature Model):** The following was used to specify the feature model of the thesis:

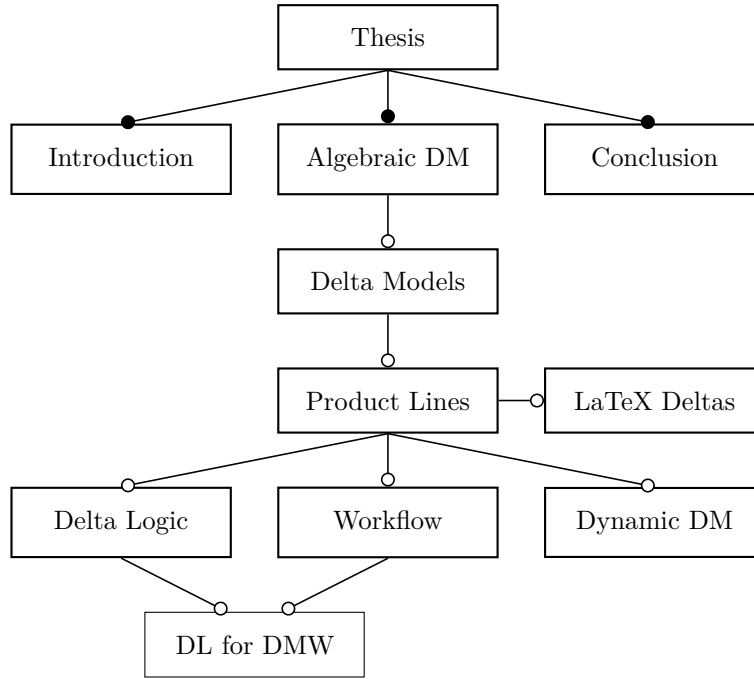


Figure 5.1: The feature model of the Thesis product line.

```

1 \DeclareFeature*{Thesis}
2 \DeclareFeature*{Introduction} extends {Thesis}
3 \DeclareFeature*{Algebraic DM} extends {Thesis}
4 \DeclareFeature {Delta Models} extends {Algebraic DM}
5 \DeclareFeature {Product Lines} extends {Delta Models}
6 \DeclareFeature {LaTeX Deltas} extends {Product Lines}
7 \DeclareFeature {Delta Logic} extends {Product Lines}
8 \DeclareFeature {Workflow} extends {Product Lines}
9 \DeclareFeature {Dynamic DM} extends {Product Lines}
10 \DeclareFeature*{Conclusion} extends {Thesis}
11 \DeclareFeature {DL for DMW} extends {Delta Logic,
12                                     Workflow}

```

The mandatory Thesis feature represents the basic document structure, specified outside of any delta, i.e., in the core product. The Introduction and Conclusion features represent their respective chapters, both also mandatory. All other features except DL for DMW represent research topics, each of which also has a chapter to itself, but the selection of which also influences the content of other chapters. DL for DMW represents the option of using Delta Logic (Chapter 6) in the formalization of the Delta Modeling Workflow (Appendix A). It is the first feature we have seen with more than one direct superfeature. Altogether, the thesis product line supports 22 products.

The corresponding feature diagram is shown in Figure 5.1. Note that it closely follows the suggested reading order on Page 14. But the conclusion, while being last in the narrative structure, should be included regardless of which other features are selected.

5.2.2 Building the Delta Modules

Deltas take the form of a L^AT_EX environment. This means that their main content is delimited by a `\begin` and an `\end`:

▷ **5.3. Definition (DeltaModule):** L^AT_EX delta modules have the following syntax:

```

⟨d-module⟩ ::= \begin{DeltaModule} {⟨did⟩}
               { ⟨condition⟩ | ⟨order⟩ }
               { ⟨operation⟩ }
               \end{DeltaModule}

⟨condition⟩ ::= if { ⟨ϕ⟩ }

⟨ϕ⟩ ::= ⟨fid⟩ | ⟨ϕ⟩ && ⟨ϕ⟩ | ⟨ϕ⟩ || ⟨ϕ⟩ | !⟨ϕ⟩ | (⟨ϕ⟩)

⟨order⟩ ::= before { ⟨did⟩ { , ⟨did⟩ } }
           | after  { ⟨did⟩ { , ⟨did⟩ } }
```

where *⟨did⟩* represents a delta name. Deltas occupy a separate namespace from features, so a delta and a feature can share the same name. ┘

Every delta has a unique name *⟨did⟩*. An *application condition* is specified through the *⟨condition⟩* clause. Its Boolean expressions use the syntax established by L^AT_EX3 [176], which is the same as for the C language. If multiple *⟨condition⟩* clauses are used for a single delta, their disjunction is used. The *application order* is specified through the *⟨order⟩* clause, using **before** and **after** to indicate the delta order. An error message is displayed if the eventual application order is not a strict partial order, i.e., if it contains a cycle.

5.2.3 L^AT_EX Delta Operations

We now specify the possible *⟨operation⟩*s. What can a L^AT_EX delta module do? The answer is: anything L^AT_EX can do. The full language is available within a delta module, though no content should be typeset directly, since delta modules are usually specified in the document preamble (before `\begin{document}`). Rather, they should modify commands that are later used to typeset content.

Even so, if we want to be able to reason about delta commutativity (Definition 2.40) —to perform conflict analysis— or take a delta consensus (Section 2.6.2), the full L^AT_EX language is too rich and unpredictable. We would need to restrict delta modules to simpler operations with more predictable behavior. Since L^AT_EX processes token lists (which can contain both code and data), the package introduces some delta-aware token list operations, similar to the method- and statement-level operations of fine-grained software-deltas (Section 3.4):

- ▷ **5.4. Definition (Delta Aware Operations):** The delta-aware operations available as of writing this are the following:

```

(operation) ::= \Replace <command> with { <tokenlist> }
              | \Prepend { <tokenlist> } to <command>
              | \Append { <tokenlist> } to <command>
              | \Insert { <tokenlist> } into <command>

```

A *tokenlist* is a fragment of L^AT_EX code, which can later be interpreted as either code or data. A *command* is not delimited by braces and is of the form `\<commandname>`. (`\Replace`, `\Prepend`, etc. are themselves commands, but they are prohibited from modifying themselves to safeguard the consistency of the running package.) ┘

Even though the delta-modules package currently employs disjunctive semantics (Definition 3.25), the `\Insert` command is already provided for when conjunctive semantics (Definition 3.26) is implemented in the future.

Apart from tracking modifications for conflict analysis, the first three operations map quite directly to L^AT_EX3 token list commands. `\Insert`, however, needs some special attention. It would not be useful to insert new material between two arbitrary tokens; every letter and command constitutes a token. Instead, `\Insert` interprets the content of *command* as a comma-separated list, and inserts the given material at an arbitrary position in that list (including an extra comma). If a comma should not be interpreted as a separator, it can be protected with braces: `{,}`.

- ▷ **5.5. Example:** Each of the nine deltas implementing a specific thesis chapter look something like this:

```

1 \begin{DeltaModule} {<fid> Delta} if {<fid>} after { ... }
2   \Insert {<fid>} into \vpFeatureList
3   :
4 \end{DeltaModule}

```

The `\Insert` operation above is how we can say that the document you are reading was generated with the features Thesis, Introduction, Algebraic DM, Delta Models, Product Lines, LaTeX, Delta Logic, Workflow, Dynamic DM, and Conclusion. The previous sentence was generated using the following code:

```

1 \FormatSequence \vpFeatureList { and } {, } {, and }

```

which interprets the given token list as a comma separated sequence, discards redundant commas, then joins the list together using the specified separators.

- ▷ **5.6. Example:** This is the delta module integrating the LaTeX Deltas feature:

```

1 \begin{DeltaModule} {LaTeX Deltas} if {LaTeX Deltas}
2   after {Product Lines}
3   \Insert {LaTeX Deltas} into \vpFeatureList
4   \Insert {\include{chap:latex-deltas}} into \vpChapters
5   \Insert {
6     \DescribeChapter{chap:latex-deltas}

```

```

7      This chapter demonstrates the \LaTeX\
8      implementation of delta modeling by
9      documenting two new \LaTeX-packages.
10     :
11   } into \vpChapterSummaries
12   \Insert {
13     \ChapterSummarySubsection{chap:latex-deltas}
14     Several publications on ADM make the claim
15     that deltas can be used to modularize any
16     kind of artefact – not just source code.
17     :
18   } into \vpConclusionSections
19   \Insert {
20     \subsection {\LaTeX\ Deltas}
21     Future work related to the \LaTeX\ packages is
22     likely to be of a \emph{development-} rather than
23     a research nature. As the code is open source,
24     :
25   } into \vpFutureWorkSections
26   :
27   \end{DeltaModule}

```

The `\vpChapters` command on line 4, as one of the most coarse-grained variation points, inserts the actual chapters into the thesis structure. The `\vpChapterSummaries` command (line 11) inserts the chapter summaries of Section 1.5.3. `\vpConclusionSections` and `\vpFutureWorkSections` (lines 19 and 12) contain subsections for Chapter 9. ┘

5.2.4 Parametrized Delta Modules

L^AT_EX delta modules are parametrized, as described in Section 4.5. Anywhere in the document, the `\IfFeatureSelected(TF)` commands can be used:

- 5.7. **Definition (`\IfFeatureSelected(TF)`):** The following commands allow inline branching based on the feature selection:

```

if-f-selected ::= \IfFeatureSelectedTF {\phi} {true} {false}
                  \IfFeatureSelectedT  {\phi} {true}
                  \IfFeatureSelectedF   {\phi} {false}

```

┘

For the thesis, this is used for the odd crossreference to an optional chapter. It provides more clarity than using an abstractly named variation point.

5.2.5 Choosing a Feature Configuration and Generating the Document

The final delta-related command in the document preamble should be the `\SelectFeatures` command, making the final selection:

- ▷ **5.8. Definition (`\SelectFeatures`):** The feature selection is made with the following syntax:

$$\langle select-f \rangle ::= \backslash \text{SelectFeatures} \{ \langle fid \rangle \{ , \langle fid \rangle \} \} \quad \lrcorner$$

Similar commands are provided for reading the feature selection from a file or to request it by standard input from the command-line.

At this point an error message is displayed if the given selection is not a valid feature configuration or if there is no acyclic delta derivation. If no problems arise, the applicable deltas are executed in the proper order. After this, the normal \LaTeX compilation process can resume.

- ▷ **5.9. Example:** The following command was used in the generation of this thesis:

```

1 \SelectFeatures {Thesis, Introduction,
2                 Algebraic DM, Delta Models,
3                 Product Lines, LaTeX,
4                 Delta Logic, Workflow,
5                 Dynamic DM, Conclusion}
```

The code above is also different for each thesis product (besides possibly not being numbered 5.9, which is handled by \LaTeX natively). Making sure the code sample is generated with nice formatting for all feature configurations took some effort. Future versions of the package will be equipped with commands specifically meant to make that sort of task easier. \lrcorner

The application of delta modeling to document preparation may offer a number of practical benefits. The introduction to this chapter introduced the idea of a line of textbooks, each tailored to a specific curriculum or level of difficulty, as well as the idea of maintaining a number of specifically targeted versions of ones CV. Another is the preparation of technical manuals that come with many consumer products. The dominant practice right now is to include the same manual for a range of products, confusing customers as to the features of their new purchase. The preparation of the various versions of this thesis is meant as a proof-of-concept for such use-cases.

5.3 pkgloader: An ADM-based Package Manager

L^AT_EX can be extended by loading packages, which can add new definitions, and remove and modify existing ones. The `delta-modules` package described in Section 5.2 is one example. Packages can implement domain specific languages, monkey-patch the core language to hook into existing commands, and even change the meaning of individual symbols. Put simply, L^AT_EX packages have free rein. This power can be quite useful, but makes it too easy for independent package authors to step on each others' toes. CTAN (the Comprehensive T_EX Archive Network [168]) is full of conceptually independent packages that cannot be loaded together, or break if they are not loaded in a specific order.

This problem sounds familiar. Let us look at it from a delta modeling perspective. We can see the runtime state of the L^AT_EX language as a product, package-names as features and package implementations as deltas.¹ From this new perspective, we can describe the problem in more familiar terms. The L^AT_EX eco-system is suffering from the optional feature problem [111]. But there is no automated package management to speak of. Document authors are told to avoid certain package combinations, or to load packages in some specific order (Action 3.9, page 77). Some of the larger packages are designed to test for the presence of other packages in order to circumvent known conflicts (Action 3.8, page 77). But solving these problems on a case-by-case basis takes time and effort for both document and package authors. It pollutes the code, makes maintenance more difficult, and confuses new users. This is an opportunity for delta modeling to shine. Enter `pkgloader`.

5.3.1 Package Description

L^AT_EX packages are generally loaded with either the `\usepackage` command or the `\RequirePackage` command. Similarly, document classes are loaded with `\documentclass` or `\LoadClass`. Normally when such a command is reached, the corresponding file is loaded right away. The idea behind `pkgloader` is to make it the very first file you load: before the document class, and before any other package. It can then intercept all document class and package loading requests, treat them as a feature selection and load them in the proper order.

▷ 5.10. Example: The main file for a L^AT_EX document using `pkgloader`:

```

1  \RequirePackage{pkgloader}
2      :
3      \documentclass{article}
4      :
5      \usepackage{algorithm}
6      \usepackage{hyperref}
7      \usepackage{float}
8      :
9  \begin{document}
10     :
11 \end{document}
```

} any order

¹The analogy goes further. Both plain T_EX and L^AT_EX are really extensions of the primitive language INITEX, the initial product (Definition 2.58). L^AT_EX packages are loaded after the main language extension, which we could reflect in the application order (Definition 3.2).

The area between lines 1 and 9 is called the *pkgloader area*. Inside this area, the loading of all packages and document classes is postponed. It may also be closed explicitly with the `\LoadPackagesNow` command, so that additional code can be run in the preamble. At line 9, a selected delta model is generated (Definition 4.8) and everything is loaded in some valid order, during which conflict resolving code may also be run. If the Example 5.10 code were compiled without `pkgloader`, the given order between `algorithm`, `hyperref` and `float` would cause an error. The main advantage to this approach is that the complexity of dealing with package conflicts is moved to the `pkgloader` package and handled in a systematic manner, taking this burden off the shoulders of the average user. If the package becomes well-used, package authors will be able to develop in a more modular fashion.

5.3.2 Conflict Analysis

Here is the main difficulty: in Section 5.2 we were able to depend on the delta operations of Definition 5.4, safe in the knowledge that there are no primitive `TeX` commands that might mess things up. But package authors are not delta authors. They can make use of the full `TeX` language. So the `pkgloader` package does *not* analyze the actual code of each package in order to detect conflicts. Package conflicts are technically what we would call *bad interactions* (Section 3.3), so they cannot be detected automatically.

The package manager is backed by a database of *rules* for recognizing and resolving known conflicts.

▷ 5.11. **Example:** The following are examples of such rules:

```

1 \Load {float} before {hyperref}
2 \Load {algorithm} after {hyperref}
3 \Load {fixltx2e} always early
4   because {it fixes some imperfections in LaTeX2e}
5 \Load error if {algorithms && pseudocode}
6   because {they provide the same functionality
7             and conflict on many command names}      」

```

The first two rules encode some workarounds for the `hyperref` package, which is notorious for causing conflicts. The first one says that `float` must be loaded before `hyperref`. The second rule ensures that `hyperref` is loaded before `algorithm`. These are the rules that would allow the code of Example 5.10 to compile without problems. Note that neither rule actually loads any packages. They simply tell the package manager how to treat certain pairs of packages, should they ever be requested together in a single document.

The third rule states that `fixltx2e` must always be loaded, and must be loaded early. The fourth rule states that the `algorithms` and `pseudocode` packages should never be loaded together. These two rules also include a textual reason, to document the rule, and to include in certain error messages.

5.3.3 \Load Rules

The feature model, partial order and application function for the collective set of packages are built up manually through the **\Load** command. Each invocation sets up a rule. All rules together form the product line implementation (Section 4.3). In contrast to L^AT_EX delta modules, these rules can come from any number of different sources. A central registry will be maintained by the community, specifying well-known conflicts and resolutions. Individual package authors can supply their own rules, as can document authors. Though ideally, for the average document author, things should ‘just work’.

▷ **5.12. Definition (\Load):** The **\Load** command expects the following syntax, some of which inherits from Definition 5.3:

$$\begin{aligned}
 \langle load-pkg \rangle &::= \textbf{\textbackslash Load} (\langle package \rangle \mid \langle error \rangle) [\langle reason \rangle] \\
 \langle package \rangle &::= [\textbf{class}] \{ \langle id \rangle \} \{ \langle p-condition \rangle \mid \langle p-order \rangle \} \\
 \langle error \rangle &::= \textbf{error} \{ \langle condition \rangle \} \\
 \langle p-condition \rangle &::= \textbf{if} \{ \langle \phi \rangle \} \mid \textbf{always} \mid \textbf{if loaded} \\
 \langle p-order \rangle &::= \langle order \rangle \mid \textbf{early} \mid \textbf{late} \\
 \langle reason \rangle &::= \textbf{because} \{ \langle text \rangle \}
 \end{aligned}$$

Package names play the rôle of both features and deltas (*fid* and *did* in Definition 5.3). └

We look at each of the clauses of the **\Load** command one by one.

It usually contains a *package description*, consisting of a name, a set of options and a minimal version, just like the **\usepackage** command.

```
1 \Load [options] {package-name} [version]
```

The application condition of every delta *id* is a propositional disjunction which is initialized to *id*, i.e., a package is loaded if it is selected. (To decide otherwise would contradict the expected behavior of **\usepackage**.) The disjunction can be extended by the *condition clause*:

```
1 \Load {pkgA} if {pkgB pkgC && !pkgD}
```

Alternatively, the condition clause can be **always**, indicating that the rule should be applied under any conditions. Finally, the keywords **if loaded** can be used to apply the rule only if the package named in the package description is requested anyway. This is the default behavior, but the keywords can be included to make it explicit.

There is one exception to the structure described above. Instead of a package description, a rule can contain the **error** keyword, followed by a condition clause, to describe conditions that should never occur — usually invalid package combinations. This refines the feature model. Initially all package combinations are viable. But if two packages are irredeemably incompatible, their combination can be made to generate an error message as follows:

```
1 \Load error if {pkgA && pkgB}
```

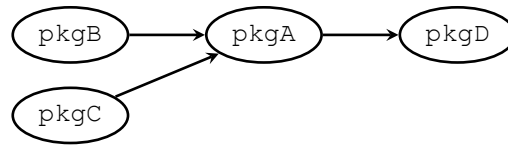
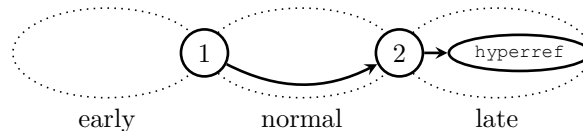


Figure 5.2: A delta diagram showing an example package-loading order.

Figure 5.3: A global delta-diagram view of the pkgloader package loading order. it shows the two placeholder packages used to impose order between **early** and **late** packages.

A non-error rule may contain an *order clause*, which forms the application order of the delta model. The **before** and **after** keywords come from Definition 5.3 and have the same meaning as they do there.

```
1 \Load {pkgA} after {pkgB,pkgC} before {pkgD}
```

This particular rule ensures that if package pkgA is ever loaded, it is never loaded before pkgB or pkgC, and never after pkgD, as illustrated in Figure 5.2.

That can take care of specific known package ordering conflicts. But the set of L^AT_EX packages is constantly growing, and it appears that some big packages should almost always be loaded early in the process, and others should almost always be loaded late. Therefore the **early** and **late** stages are provided as a fallback mechanism. If two packages are not related by the application order, their loading order may still be decided by their relative stages: **early** before ‘normal’ before **late**. That way, conflicts are avoided in a majority of cases.

▷ **5.13. Example:** A typical example is the hyperref package, which should almost always be loaded late in the run:

```
1 \Load {hyperref} late ┘
```

The **early** and **late** clauses work by ordering the package relative to one of two placeholder packages in the loading order (Figure 5.3).

These two nodes are always present in the graph. Ordering a package **early** is intuitively the same as ordering it ‘**before {1}**’. And ordering it **late** is the same as ordering it ‘**after {2}**’. All packages that are, after considering all rules, not (indirectly) ordered ‘**before {1}**’ or ‘**after {2}**’ are automatically ordered ‘**after {1} before {2}**’. A rule can have any number of order clauses, and all are taken into account when one of the conditions of the rule is satisfied.

Finally, a rule can be annotated with the *reason* it was created. This text should be semantically and grammatically correct when following the words “This rule was created because ...”. It can also be used for citing relevant sources.

▷ **5.14. Example:** The reason clause can be used as follows:

```
1 \Load {comicsans} always because {that font is awesome!} »
```

This does not have any effect on the behavior of the rule. It is meant for human consumption, though should not be formatted in any way. It is used in certain pkgloader error messages (Section 5.3.5) and may eventually be used to generate documentation.

5.3.4 Rulesets

Rules can be placed directly inside the pkgloader area, but they can also be bundled in a file. By default, pkgloader loads a recommended set of rules, allowing the average user to get started without any hassle. But this behavior can be overwritten using package options:

```
1 \RequirePackage[recommended=false,  
2             my-better-rules=true]{pkgloader}  
3     :  
4 \LoadPackagesNow
```

This means: the recommended rules that are usually preloaded by default should *not* be loaded for this document. Instead, load the my-better-rules rule-set. Any user can create rules for their own documents, or distribute custom rulesets, e.g., through CTAN. But primarily, we expect two groups of people to author pkgloader rules:

The L^AT_EX community: The recommended ruleset would, ideally, be populated further through the efforts of anyone who diagnoses and solves package conflicts.

Package authors: pkgloader will eventually be directly usable for package authors just as for document authors, to include their own rules from right inside their packages. Rather than manually scanning for and fixing potential conflicts, they could leverage pkgloader.

5.3.5 Error messages

There are two types of error messages that may be generated by pkgloader.

The first type of error message happens when an **error** rule is triggered. It looks like this:

```
| A combination of packages fitting the  
| following condition was requested:  
|   (condition)  
| This is an error because (reason).
```

The second type of error message is a bit more interesting. Since rules can effectively come from any source, the package loading order may not be an order at all; it may be an arbitrary transitive relation (Section 1.7.6).

▷ **5.15. Example:** A cycle can occur when contradictory ordering rules are specified:


```

1 \Load {pkgX} always before {pkgY}
2     because {pkgX is better}
3 \Load {pkgY} always before {pkgX}
4     because {pkgY is better}

```

In practice this could happen if the authors of `pkg1` and `pkg2` independently discover a conflict, and both try to solve it by patching their code and having their own package be loaded last. ┘

A potential circular ordering is not necessarily a problem, so long as both rules are never applied in the same run. But taken literally, Example 5.15 generates the following error message:

```

There is a cycle in the requested
package loading order:
      pkgX
--1--> pkgY
--2--> pkgX
The circular reasoning is as follows:
(1) 'pkgX' is to be loaded before
    'pkgY' because pkgX is better.
(2) 'pkgY' is to be loaded before
    'pkgX' because pkgY is better.

```

Whenever this happens, the user may want to reconsider one of their included rulesets, or file a bug-report to the responsible party or parties — especially if the circularity comes from the recommended ruleset.

5.3.6 Obtaining these Packages

The two main \TeX distributions, `TeXlive` and `MikTeX`, only come out with new versions periodically. The `delta-modules` and `pkgloader` packages can be downloaded from CTAN with full documentation. They are dependent on two other new packages: `withargs` and `lt3graph`.

5.4 Conclusion

Several publications on ADM make the claim that deltas can be used to modularize any kind of artefact — not just source code. An example occasionally brought up is documentation. Indeed, the abstract nature of ADM should allow this, but it had not yet been demonstrated.

So what better language to implement and demonstrate deltas for than the one used to write this very thesis? \TeX is a fascinating language; functional by nature, but with the unusual characteristic that practically the entire language can be redefined from within. This brings two opportunities. First, it is a way for deltas to hook into document generation without requiring outside tools: deltas can just be defined in a \LaTeX package. Second, the power of the language has caused a number of problems in the \LaTeX ecosystem: conflicts between independent packages that access the same resources. The conflict and dependency model of ADM can be adapted to mediate between such packages and, hopefully, alleviate much frustration in the \LaTeX community.

This chapter described the L^AT_EX packages implementing these ideas. The first part of the chapter introduced `delta-modules`, used to modularize the development of technical documents. The second part introduced `pkgloader`, which manages the L^AT_EX package loading process to resolve conflicts.

5.5 Related Work

The T_EXbook by Knuth [115] is a fantastic resource for a grounding in the basics of T_EX as a language. The standard book on L^AT_EX is Lamport's [118], though it does not go much beyond the basics.

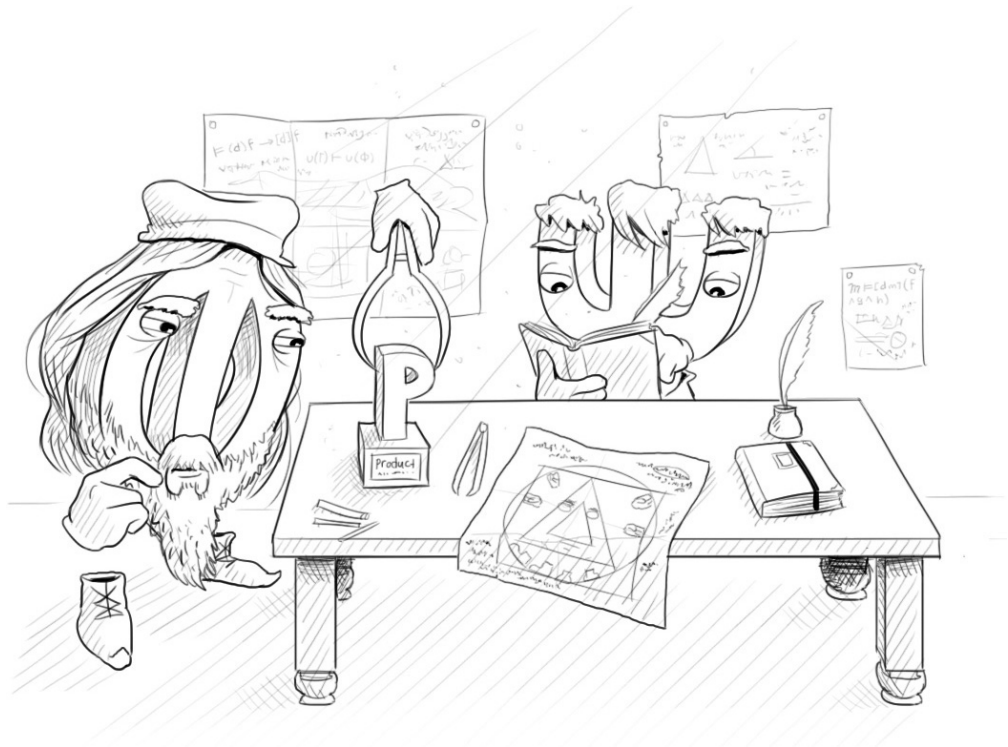
As far as we have been able to discover, this is the first time any sort of product line principles have been applied to these languages. There are, however, some packages that attempt to make specific other packages work together. An example is `interfaces` [70], which also provides a consistent interface across the packages it supports. However, none attempt to provide a general solution.

The ability to selectively include certain chapters is offered by the core L^AT_EX command `\includeonly`. But this command is meant purely to save compilation time during development —cross-references to excluded chapters and page numbers are preserved— and `\include` operates on a very coarse-grained level — it forcibly starts a new page in the document and is meant for full chapters.

The `delta-modules` and `pkgloader` packages depend on two other packages I have written: `withargs` [87] and `lt3graph` [86]. The former provides a construct for anonymous functions, providing more convenient access to the L^AT_EX3 argument parsing facilities. The latter implements a graph datastructure for L^AT_EX3, supporting cycle detection, transitive and reflexive closure generation and vertex iteration in topological order. This is particularly important for the implementation of delta models.

Delta Logic

Reasoning About Products and Delta Effects



6.1 Introduction

At its core, ADM is about deltas that can transform one product into another product. But the algebraic notation of the previous chapters is not ideal for specifying and reasoning about the semantics of deltas, and what effect they have on the properties of a product. We want to be able to specify that a delta implements a specific new feature or that a delta refrains from breaking some existing feature, without talking about products. Similarly, we want to prove that certain local constraints on deltas ensure desirable global properties. This chapter introduces a modal logic tailored to this goal. For a brief introduction to modal logic, see Section 1.7.10 (page 25).

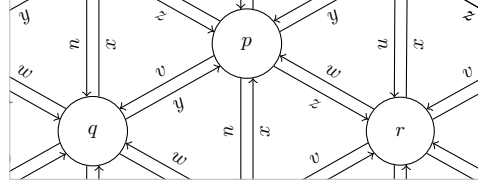


Figure 6.1: Example view of a delta frame with products p, q, r and deltas u, v, w, x, y, z currently visible.

Goal: Create a modal logic for reasoning syntactically about the semantics of deltas and their effects on product properties.

Basically, we take the set of products as the set of *worlds* in a frame (Figure 6.1). We then model deltas as binary relations on this set by applying semantic evaluation. The result is something very similar to dynamic logic [69]. In this logic, we want to be able to make judgments such as

$$\models \langle d \rangle k \qquad \models [d]k,$$

meaning “delta d *may* implement property k ” (left) and “delta d *definitely* implements property k ” (right). Or perhaps, if stated for all formulas ψ ,

$$\models \langle d \rangle \psi \rightarrow [d] \psi \qquad \models [d] \psi \rightarrow \langle d \rangle \psi,$$

meaning “delta d is deterministic” (left) and “delta d is fully defined” (right). These formulas implicitly quantify over all products that d may be applied to, but such judgments may also be made with regard to specific products or models. We will also use delta models as modalities, in order to make judgments such as

$$\models [dm](f \wedge g \wedge h),$$

meaning that, if it applies, delta model dm implements features f , g and h in all possible products. To that purpose, we allow the possibility of nested delta models (Section 3.6).

Section 6.2 specifies the modal language. Section 6.3 explores the modal logic on a Kripke frame level, specifying the proof theory, proving its completeness and extending it to a proof system for delta correctness. Section 6.4 explores the logic on a Kripke model level, addressing a problem in proving judgments about specific propositions. Finally, Sections 6.5 and 6.6 offer concluding remarks and discuss related work.

6.2 A Multimodal Language

One of the primary goals of this chapter is to reason about abstract delta modeling using the language and techniques of modal logic. A necessary starting point, before moving on to an axiomatic characterization (in which we are concerned with issues such as completeness), is to describe a modal language.

The first pair of example formulas on page 135 reference a property k . It comes from a set of propositional variables:

- **6.1. Notation (Propositional Variables):** We denote *propositional variables* by the symbols k, l, m . Sets of propositional variables are denoted by $PROP$. \perp

We then define the language that will form the basis of our logic. The intention is to describe properties of (sets of) products in a syntactic manner, by the propositions that hold there, or those that hold in products reachable through the application of certain deltas. The language is a multimodal language (Definition 1.36) based on the specific artefacts of ADM:

- **6.2. Definition (Product Formulas):** Given a set of deltas (or delta models) \mathcal{D}_Δ (Section 3.6) and a set of propositional variables $PROP$, we define a multimodal language of *product formulas* with the following grammar:

$$\Psi \quad \ni \quad \varphi \quad ::= \quad \top \mid k \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle d \rangle \varphi$$

where $k \in PROP$ is a propositional variable and $d \in \mathcal{D}_\Delta$ is an expression resolving to a delta (model) (Sections 2.6 and 3.6). We introduce the following formulas as abbreviations, so we need only be concerned with the minimal grammar above in further analysis. For all formulas $\varphi, \psi \in \Psi$:

$$\begin{aligned} \perp &\stackrel{\text{def}}{=} \neg\top \\ [d]\varphi &\stackrel{\text{def}}{=} \neg\langle d \rangle \neg\varphi \\ \varphi \wedge \psi &\stackrel{\text{def}}{=} \neg(\neg\varphi \vee \neg\psi) \\ \varphi \rightarrow \psi &\stackrel{\text{def}}{=} \neg\varphi \vee \psi \\ \varphi \leftrightarrow \psi &\stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \end{aligned}$$

To resolve ambiguity we assume the traditional set of precedence rules (e.g. \wedge binds stronger than \vee) and allow parentheses to override those rules.

If the set of deltas or propositional variables is not clear from context, we attach a subscript as in $\Psi_{\mathcal{D}, PROP}$. \perp

6.3 Kripke Frames

This section defines the ‘delta version’ of Kripke frames and models (Definitions 1.37 and 1.39), and discusses proof theory on the frame level.

6.3.1 Kripke Semantics

Defining the Kripke semantics for a given deltoid is not difficult, because a deltoid (Definition 2.11) is already a Kripke frame (Definition 1.37). To work with delta models, a delta model closed deltoid (Definition 3.31) is assumed:

- **6.3. Notation (Delta Kripke Frame):** A *delta Kripke frame* \mathfrak{F} is a deltoid $Dt = (\mathcal{P}, \mathcal{D}_\Delta, \llbracket - \rrbracket)$, where the set of products \mathcal{P} is the set of worlds, the set of deltas \mathcal{D}_Δ is the set of modal labels and the semantic evaluation operator $\llbracket - \rrbracket: \mathcal{D}_\Delta \rightarrow \text{Pow}(\mathcal{P} \times \mathcal{P})$ maps each delta to a corresponding accessibility relation.

For the sake of brevity, we will just write Dt or $(\mathcal{P}, \mathcal{D}_\Delta, \llbracket - \rrbracket)$ when a delta Kripke frame is expected.

The class of all *disjunctive* delta Kripke frames is denoted $\Delta\mathcal{U}\mathcal{F}$. The class of all *conjunctive* delta Kripke frames is denoted $\Delta\mathcal{R}\mathcal{F}$. Both are classes of frames with an underlying set of delta (model) expressions as modalities, following their respective policies for delta model semantics (Section 3.5). \lrcorner

Figure 6.1 shows part of an infinite deltoid Kripke frame.

To reason about product properties, we need a *valuation function* (Definition 1.38), mapping proposition letters to the set of worlds in which they are true. A delta Kripke model is a delta Kripke frame with a valuation function:

- **6.4. Notation (Delta Kripke Model):** A *delta Kripke model* is a tuple $\mathfrak{M} = (Dt, V) = (\mathcal{P}, \mathcal{D}_\Delta, \llbracket - \rrbracket, V)$ — a deltoid Kripke frame equipped with a valuation function $V: \text{PROP} \rightarrow \text{Pow}(W)$. \lrcorner

The semantics of product formulas (Definition 6.2) can, of course, be given in the traditional manner for modal formulas: by defining a forcing relation \Vdash (Definition 1.40). But in the trend set by the previous chapters, we do it instead by extending the semantic evaluation operator $\llbracket - \rrbracket$ so it can map product formulas (a syntactic notion) to sets of products (a semantic notion), which is quite compact and intuitive:

- **6.5. Definition (Formula Semantics):** Given a Kripke model $\mathfrak{M} = (\mathcal{P}, \mathcal{D}_\Delta, \llbracket - \rrbracket, V)$, we extend semantic evaluation to product formulas as follows. For all propositional variables $k \in \text{PROP}$, formulas $\varphi, \psi \in \Psi$ and deltas $d \in \mathcal{D}_\Delta$ we define $\llbracket - \rrbracket: \Psi \rightarrow \text{Pow}(\mathcal{P})$ by induction on the shape of the formula:

$$\begin{aligned} \llbracket \top \rrbracket &\stackrel{\text{def}}{=} \mathcal{P} \\ \llbracket k \rrbracket &\stackrel{\text{def}}{=} V(k) \\ \llbracket \varphi \vee \psi \rrbracket &\stackrel{\text{def}}{=} \llbracket \varphi \rrbracket \cup \llbracket \psi \rrbracket \\ \llbracket \neg \varphi \rrbracket &\stackrel{\text{def}}{=} \llbracket \top \rrbracket \setminus \llbracket \varphi \rrbracket \\ \llbracket \langle d \rangle \varphi \rrbracket &\stackrel{\text{def}}{=} \llbracket d \rrbracket^{-1}(\llbracket \varphi \rrbracket) \end{aligned}$$

As always, the proper subscripts can be added to disambiguate between disjunctive and conjunctive semantics. \lrcorner

This essentially gives us a way to describe sets of products by which properties they satisfy, including properties of which products could result if certain deltas are applied. We could reintroduce the traditional forcing relation \Vdash (Definition 1.40, page 26) as follows:

- 6.6. Lemma:** Given a deltoid Kripke model $\mathfrak{M} = (\mathcal{P}, \mathcal{D}_\Delta, \llbracket - \rrbracket, V)$, product $p \in \mathcal{P}$ and formula $\varphi \in \Psi$, we have:

$$\mathfrak{M}, p \Vdash \varphi \iff p \in \llbracket \varphi \rrbracket$$

In other words, Definition 6.5 corresponds to traditional modal semantics. \square

6.3.2 Proof Theory

Now that we have a multimodal language with Kripke semantics, we define the logic that can be used to reason purely in that language, and prove that this logic is sound and complete. This then allows us to reason about the effects of deltas without resorting to semantic evaluation.

- **6.7. Definition (Delta Logic):** The *delta logics* $\mathbf{K}\Delta\cup$ and $\mathbf{K}\Delta\cap$ are normal modal logics¹ (Definition 1.43) generated by the following axiom schemas, which encode the laws of our algebraic operators.² For all delta (model) expressions $x, y \in \mathcal{D}_{\Delta}$ and all formulas $\varphi \in \Psi$:

- the composition axiom: $\langle y \cdot x \rangle \varphi \leftrightarrow \langle x \rangle \langle y \rangle \varphi \in \mathbf{K}\Delta\cup, \mathbf{K}\Delta\cap$
- the choice axiom: $\langle x \sqcup y \rangle \varphi \leftrightarrow (\langle x \rangle \varphi \vee \langle y \rangle \varphi) \in \mathbf{K}\Delta\cup, \mathbf{K}\Delta\cap$
- the consensus axiom: $\langle x \sqcap y \rangle \varphi \leftrightarrow (\langle x \rangle \varphi \wedge \langle y \rangle \varphi) \in \mathbf{K}\Delta\cup, \mathbf{K}\Delta\cap$
- the neutral delta axiom: $\varphi \leftrightarrow \langle \varepsilon \rangle \varphi \in \mathbf{K}\Delta\cup, \mathbf{K}\Delta\cap$
- the empty delta axiom: $[\perp] \varphi \in \mathbf{K}\Delta\cup, \mathbf{K}\Delta\cap$

Then, depending on policy regarding delta model semantics, one of the following axiom schemas should be added:

- the delta model axiom $\Delta\cup$: $\langle dm \rangle \varphi \leftrightarrow \bigvee_{d \in \text{derv}(dm)} \langle d \rangle \varphi \in \mathbf{K}\Delta\cup$
- the delta model axiom $\Delta\cap$: $\langle dm \rangle \varphi \leftrightarrow \bigwedge_{d \in \text{derv}(dm)} \langle d \rangle \varphi \in \mathbf{K}\Delta\cap$ ┘

For disjunctive semantics, a different formulation for the disjunctive delta model axiom follows straightforwardly from Definition 3.25:

- **6.8. Theorem:** For nonempty delta model $dm = (D, <)$ and all formulas φ :

$$\begin{aligned} \Vdash_{\Delta\cup F} \langle (\emptyset, \emptyset) \rangle \varphi &\leftrightarrow \varphi \\ \Vdash_{\Delta\cup F} \langle dm \rangle \varphi &\leftrightarrow \bigvee_{d \neq} \langle d \rangle \langle dm \setminus \{d\} \rangle \varphi \end{aligned}$$

where $d \neq$ quantifies over all minimal deltas in dm (Notation 1.12, page 20; and Definition 1.23, page 22).

Proof: Induction on the size of D . □

- **6.9. Corollary:** For nonempty delta model $dm = (D, <)$ and all formulas φ :

$$\begin{aligned} \Vdash_{\Delta\cup F} [(\emptyset, \emptyset)] \varphi &\leftrightarrow \varphi \\ \Vdash_{\Delta\cup F} [dm] \varphi &\leftrightarrow \bigwedge_{d \neq} [d] [dm \setminus \{d\}] \varphi \end{aligned}$$

by taking the inverse of Theorem 6.8. □

¹The original paper [3], which did not consider conjunctive semantics, presented $\mathbf{K}\Delta\cup$ under the name $\mathbf{K}\Delta$.

²From the available relation algebra operators (Section 2.6), the original paper [3] included only axioms for composition \cdot and choice \sqcup . We have added axioms for the other operators fundamental to this thesis. This does not include negation \neg , the full delta τ or converse \smile . Defining converse as a modal operator is rather involved [74], though interesting, and deserves more than a hasty treatment in a small part of this chapter. The other two are simply not that important for us, as well as not constructive (Section 2.6.2).

It is worthwhile to note that the above theorem and corollary are similar to what is known as the *expansion law* of the process algebra CCS [136]. The fact that it works explains why a delta model under disjunctive semantics can be applied to a product simply by applying its deltas in an arbitrary order compatible with \prec . A similar law does *not* exist for conjunctive semantics.

Next, we'll use the delta logics as proof systems by looking at their provability relations $\vdash_{\mathbf{K}\Delta_{\cup}}$ and $\vdash_{\mathbf{K}\Delta_{\cap}}$ (Definition 1.44), and prove their completeness.

6.3.3 Completeness

It is not hard to see that the delta logics are sound with respect to their respective frames (Definition 1.45, page 27). More interesting is the issue of their completeness (Definition 1.46). It turns out they are strongly complete. Except for the delta model axioms, the presented logic is a subset of dynamic logic [69]; one without iteration. Because there is no iteration axiom, and because delta models are finite and do not contain cycles, modalities can be completely reduced to simple deltas. We define a translation function 'kt':

- **6.10. Definition:** Given a set of simple deltas \mathcal{D} and a set of propositional variables $PROP$, we define a translation function $\text{kt}: \Psi \rightarrow \Psi$ such that for all propositional variables $k \in PROP$, all simple deltas $d \in \mathcal{D}$, all delta models $dm \in \mathcal{DM}_{\Delta}$, all delta (model) expressions $x, y \in \mathcal{D}_{\Delta}$ and all formulas $\varphi, \psi \in \Psi$:

$$\begin{aligned}
 \text{kt}(k) &\stackrel{\text{def}}{=} k \\
 \text{kt}(\neg\varphi) &\stackrel{\text{def}}{=} \neg\text{kt}(\varphi) \\
 \text{kt}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \text{kt}(\varphi) \vee \text{kt}(\psi) \\
 \text{kt}(\langle y \cdot x \rangle \varphi) &\stackrel{\text{def}}{=} \text{kt}(\langle x \rangle \langle y \rangle \varphi) \\
 \text{kt}(\langle x \sqcup y \rangle \varphi) &\stackrel{\text{def}}{=} \text{kt}(\langle x \rangle \varphi \vee \langle y \rangle \varphi) \\
 \text{kt}(\langle x \sqcap y \rangle \varphi) &\stackrel{\text{def}}{=} \text{kt}(\langle x \rangle \varphi \wedge \langle y \rangle \varphi) \\
 \text{kt}(\langle \varepsilon \rangle \varphi) &\stackrel{\text{def}}{=} \text{kt}(\varphi) \\
 \text{kt}(\langle \perp \rangle \varphi) &\stackrel{\text{def}}{=} \perp \\
 \text{kt}(\langle d \rangle \varphi) &\stackrel{\text{def}}{=} \langle d \rangle \text{kt}(\varphi)
 \end{aligned}$$

With one of the following depending on delta model semantics:

$$\begin{aligned}
 \text{kt}(\langle dm \rangle \varphi) &\stackrel{\text{def}}{=} \bigvee_{d \in \text{derv}(dm)} \langle d \rangle \varphi && \text{(disjunctive semantics)} \\
 \text{kt}(\langle dm \rangle \varphi) &\stackrel{\text{def}}{=} \bigwedge_{d \in \text{derv}(dm)} \langle d \rangle \varphi && \text{(conjunctive semantics)} \quad \lrcorner
 \end{aligned}$$

The idea behind this function is to translate any formula into an equivalent formula in which all unary modalities are labeled only by simple deltas. This enables us to forget about arbitrary algebraic expressions and delta models, and to construct our completeness proof in terms of the completeness of \mathbf{K} with regard to the class of all frames (Theorem 1.47, page 27).

► **6.11. Lemma:** For all sets of formulas Γ and all formulas φ , we have:

- a. $\Gamma \vdash_{\mathbf{K}\Delta\cup} \varphi \iff \Gamma \vdash_{\mathbf{K}\Delta\cup} \text{kt}(\varphi)$
- b. $\Gamma \Vdash_{\Delta\cup\mathbf{F}} \varphi \iff \Gamma \Vdash_{\Delta\cup\mathbf{F}} \text{kt}(\varphi)$
- c. $\Gamma \Vdash_{\Delta\cup\mathbf{F}} \text{kt}(\varphi) \iff \Gamma \Vdash \text{kt}(\varphi)$

as well as the same for $\mathbf{K}\Delta\cap$ and $\Delta\cap\mathbf{F}$.

Proof: (a) and (b) can be proved by induction (on the complexity of formulas as well as that of delta terms); (c) follows from the observation that for any translated formula, only the relations corresponding to simple deltas are used: hence, we are simply treating our delta frame as a regular frame. \square

► **6.12. Theorem:** $\mathbf{K}\Delta\cup$ (resp. $\mathbf{K}\Delta\cap$) is strongly complete w.r.t. the class of delta kripke frames $\Delta\cup\mathbf{F}$ (resp. $\Delta\cap\mathbf{F}$).

Proof: This amounts to saying that, for any Γ and φ , if $\Gamma \Vdash_{\Delta\cup\mathbf{F}} \varphi$, then $\Gamma \vdash_{\mathbf{K}\Delta\cup} \varphi$. If $\Gamma \Vdash_{\Delta\cup\mathbf{F}} \varphi$ then, by Lemma 6.11b, we have $\Gamma \Vdash_{\Delta\cup\mathbf{F}} \text{kt}(\varphi)$ and by Lemma 6.11c, we have $\Gamma \Vdash \text{kt}(\varphi)$. Completeness of \mathbf{K} now gives $\Gamma \vdash_{\mathbf{K}} \text{kt}(\varphi)$ and, because $\mathbf{K} \subseteq \mathbf{K}\Delta\cup$, we also get $\Gamma \vdash_{\mathbf{K}\Delta\cup} \text{kt}(\varphi)$. Finally, Lemma 6.11a yields $\Gamma \vdash_{\mathbf{K}\Delta\cup} \varphi$. \square

It is possible to extend this completeness result in simple and straightforward ways, because any formula in \mathbf{K} yields a complete axiomatization for the class of frames it defines [42].

6.13. Example: Consider the class of deterministic deltoid Kripke frames $\Delta\cup\mathbf{dF}$ (resp. $\Delta\cap\mathbf{dF}$), in which all simple deltas are deterministic (Definition 2.26). This class of frames can be characterized by the following axiom schema. For all simple deltas d and formulas φ :

$$\langle d \rangle \varphi \rightarrow [d] \varphi$$

We call the delta logic generated by that axiom schema $\mathbf{K}\Delta\cup\mathbf{d}$ (resp. $\mathbf{K}\Delta\cap\mathbf{d}$). \lrcorner

6.14. Theorem: The logic $\mathbf{K}\Delta\cup\mathbf{d}$ (resp. $\mathbf{K}\Delta\cap\mathbf{d}$) is strongly complete with regard to the class of deterministic delta frames $\Delta\cup\mathbf{dF}$ (resp. $\Delta\cap\mathbf{dF}$). \square

6.3.4 Delta Contracts

Our modal product formulas are essentially syntactic representations of product sets. But they also allow us to syntactically characterize deltas based on their effect on such product sets. We can formulate *delta contracts* reminiscent of Hoare triples:

► **6.15. Definition (Delta Contracts):** A *delta contract* is a pair of product formulas $(\varphi, \psi) \in \Psi \times \Psi$, where φ is the *precondition* and ψ is the *postcondition*. \lrcorner

The following is a way to prove, in a fully syntactic manner, that a specific delta satisfies a specific delta contract:

- **6.16. Definition (Contract Provability):** A given delta $d \in \mathcal{D}_{\Delta}$ is *provably correct* with regard to delta contract $(\varphi, \psi) \in \Psi \times \Psi$ iff the following holds:

$$\begin{array}{ccc} d \vdash (\varphi, \psi) & \stackrel{\text{def}}{\iff} & \varphi \vdash_{\mathbf{K}\Delta\cup} [d]\psi \\ d \vdash_{\text{tot}} (\varphi, \psi) & \stackrel{\text{def}}{\iff} & \underbrace{\varphi \vdash_{\mathbf{K}\Delta\cup}}_{\text{a}} \underbrace{\langle d \rangle \top}_{\text{b}} \wedge \underbrace{[d]\psi}_{\text{c}} \end{array}$$

An analogous definition can be given for conjunctive semantics. \lrcorner

Basically, a delta d is said to be provably correct with regard to a contract (φ, ψ) iff (a) given a product satisfying the premise φ , (b) delta d is applicable to that product (for total correctness), and (c) all products resulting from the application of delta d satisfy ψ .

This simple proof system is sound and complete in the following sense:

- **6.17. Theorem:** The way of using the $\mathbf{K}\Delta\cup$ proof system from Definition 6.16 is sound and complete —with regard to all delta frames— in the following sense:

$$\begin{array}{ccc} d \vdash (\varphi, \psi) & \iff & d \models [\varphi] \times [\psi] \\ d \vdash_{\text{tot}} (\varphi, \psi) & \iff & d \models_{\text{tot}} [\varphi] \times [\psi] \end{array}$$

where \models and \models_{tot} represent delta correctness (Definition 2.29, page 45).

Proof: The following proves the total correctness version:

$$\begin{array}{ll} d \vdash_{\text{tot}} (\varphi, \psi) & \\ \stackrel{1}{\iff} & \varphi \vdash_{\mathbf{K}\Delta\cup} \langle d \rangle \top \wedge [d]\psi \\ \stackrel{2}{\iff} & \varphi \Vdash_{\Delta\cup\mathbf{F}} \langle d \rangle \top \wedge [d]\psi \\ \stackrel{3}{\iff} & \forall p \in \mathcal{P}: (p \Vdash \varphi) \implies (p \Vdash \langle d \rangle \top \wedge [d]\psi) \\ \stackrel{4}{\iff} & \forall p \in \mathcal{P}: p \in [\varphi] \implies p \in [\langle d \rangle \top \wedge [d]\psi] \\ \stackrel{5}{\iff} & \forall p \in \mathcal{P}: p \in [\varphi] \implies p \in [\langle d \rangle \top] \cap [[d]\psi] \\ \stackrel{6}{\iff} & \forall p \in \mathcal{P}: p \in [\varphi] \implies (p \in \text{pre}[[d]]) \wedge ([d](p) \subseteq [\psi]) \\ \stackrel{7}{\iff} & \forall p \in [\varphi]: \emptyset \subset [[d](p)] \subseteq [\psi] \\ \stackrel{8}{\iff} & d \in ([\varphi] \Rightarrow_{\text{tot}} [\psi]) \\ \stackrel{9}{\iff} & d \models_{\text{tot}} [\varphi] \times [\psi] \end{array}$$

Step 1 applies Definition 6.16. Step 2 applies the completeness result of Theorem 6.12. Step 3 applies Definition 1.42 (page 26) of local consequence. Step 4 twice applies Lemma 6.6. Steps 5 and 6 apply Definition 6.5 (though some steps are skipped). Step 7 applies a number of general simplifications.

Steps 8 and 9 apply Definition 2.31 and Lemma 2.32, confirming what the reader possibly already suspected after seeing the use of the Cartesian product in the theorem: delta contracts are syntactic representations of *delta derivations* (Section 2.4.3). \square

This tells us something about the power of the delta logics presented in this chapter. Though they give us valuable insight into the behavior of deltas, they are actually rather limited when it comes specifying the behavior of ‘practical’

deltas. For example, while they would be able to specify that software delta (**remove class** C) only accepts products that have a class C, and that it is guaranteed to yield a product without such a class, they are unable to express that the delta leaves all other artefacts the way they are.

One way to express this would be to use a modal language that can refer back to the original world in the frame: a *hybrid language* [22, 40]. This is presented as future work in Chapter 9.

6.4 Kripke Models

As we can now reason on the frame level with the proof system of Section 6.3, we would also like to reason on the level of models.

Recall that a Kripke model is a Kripke frame augmented with a valuation function, which maps propositional variables to the set of worlds in which they are true. Our worlds are products from \mathcal{P} . What we'd actually like to reason about is the *features* that are implemented by those products; or more accurately, the *feature combinations*. We want to prove properties about the effects deltas can have on products that satisfy specific feature combinations. So we state that $\text{Pow}(\mathcal{F}) \subseteq \text{PROP}$. This is in line with Definition 4.17 on page 107, where it is also explained why we need to handle feature combinations explicitly: it is possible to implement multiple features without implementing their combination.

6.4.1 Proof System Soundness

The ultimate goal here is to formulate some axioms about specific features (i.e., propositional variables in a Kripke model), and then to prove properties about the effects of deltas on those features. However, the proof system for the frame level Definition 1.44 is not sound with respect to global semantic entailment on models. For example, consider the following ‘proof’:

- (1) $F \rightarrow \langle d \rangle G$ axiom
- (2) $F \rightarrow \langle d \rangle \neg G$ uniform substitution on G

So we have $F \rightarrow \langle d \rangle G \vdash_{\mathbf{K}\Delta\mathcal{U}} F \rightarrow \langle d \rangle \neg G$, but at the same time the (global) semantic consequence

$$F \rightarrow \langle d \rangle G \Vdash_{\Delta\mathcal{U}\mathbf{F}}^g F \rightarrow \langle d \rangle \neg G$$

is easily seen to be false. The culprit is our use of uniform substitution. The initial axiom in our false proof is not meant to be a tautology that is “true for all G ”. It is meant as a statement about the feature G specifically. We can’t take away the uniform substitution rule, however. We still need it to prove such truths as:

- (1) $k \vee \neg k$ propositional tautology
- (2) $[d] F \vee \neg[d] F$ uniform substitution on k

The trick is to allow uniform substitution only on newly produced proposition-letters, but not on the original features in our axioms. This is accomplished by first transforming all propositions in our axioms and formulas to *nullary modalities* [41], on which uniform substitution does not apply. We can then prove valid formulas in the system of frames.

So we now introduce nullary modalities, which may be seen as propositional constants, into the modal language (Definition 1.36). A nullary modality labeled with a propositional variable k is denoted \textcircled{k} . This extends frames with a set of predicates on worlds. A nullary modality \textcircled{k} corresponds to a predicate P_k in a frame:

- **6.18. Definition (Nullary Modality Semantics):** Given a Kripke frame $\mathfrak{F} = (W, M, U, R)$ —which now includes a function $U: PROP \rightarrow \text{Pow}(W)$, mapping each propositional variable $k \in PROP$ to a corresponding predicate $P_k \subseteq W$ —, a nullary modality \textcircled{k} has the following semantics:

$$\mathfrak{M}, p \Vdash \textcircled{k} \stackrel{\text{def}}{\iff} p \in U(k)$$

Or equivalently, as an extension to Definition 6.5:

$$\llbracket \textcircled{k} \rrbracket \stackrel{\text{def}}{=} U(k) \quad \lrcorner$$

We define the following function to translate propositional variables to corresponding nullary modalities:

- **6.19. Definition:** Define the function $u: \Psi \rightarrow \Psi$, which transforms all propositional variables in a formula into nullary modalities. For all propositional variables $k \in PROP$ and all formulas $\varphi \in \Psi$:

$$\begin{aligned} u(k) &\stackrel{\text{def}}{=} \textcircled{k} \\ u(\neg\varphi) &\stackrel{\text{def}}{=} \neg u(\varphi) \\ &\vdots \end{aligned}$$

For the other shapes of formulas the ‘u’ translation is simply propagated down to the propositional variables, leaving everything else unchanged. We also lift the function ‘u’ to sets of formulas in the expected manner. \lrcorner

We extend this function to translate from models to frames (overloading the name ‘u’):

- **6.20. Definition:** Extend translation function ‘u’ to take a model $\mathfrak{M} = (W, M, R, V)$ and return a frame:

$$u(\mathfrak{M}) \stackrel{\text{def}}{=} (W, M, U, R)$$

Where U maps propositional variables to the set of worlds in which they are true. So essentially we take $U \stackrel{\text{def}}{=} V$. \lrcorner

The following translation lemma holds:

► **6.21. Lemma:** For all models \mathfrak{M} , worlds w and sets of formulas Γ , we have:

- a. $\mathfrak{M}, w \Vdash \Gamma \iff u(\mathfrak{M}), w \Vdash u(\Gamma)$
- b. $\mathfrak{M} \Vdash \Gamma \iff u(\mathfrak{M}) \Vdash u(\Gamma)$

Proof: Proof of (a) is by induction on the complexity of (sets of) formulas. The base case trivially follows from our construction of nullary modalities in terms of propositional variables. (b) follows trivially from (a). \square

This lemma enables us to prove the following soundness result with regard to global truth on the model level:

► **6.22. Theorem:** For all sets of formulas Γ and all formulas φ :

$$u(\Gamma) \vdash u(\varphi) \implies \Gamma \Vdash^g \varphi$$

Proof: Assume $u(\Gamma) \vdash u(\varphi)$. Let \mathfrak{M} be a model (based on a delta frame) such that $\mathfrak{M} \Vdash \Gamma$. Then, by Lemma 6.21b, we have $u(\mathfrak{M}) \Vdash u(\Gamma)$. Now let Λ be the logic of the class of delta frames

$$\{ \mathfrak{F} \mid \mathfrak{F} \Vdash u(\Gamma) \}.$$

Because Λ is a normal modal logic, it is closed under proof rules, and hence it follows from $u(\Gamma) \vdash u(\varphi)$ combined with the fact that $u(\Gamma) \subseteq \Lambda$, that $u(\varphi) \in \Lambda$. It follows that $u(\varphi)$ is valid on this class of frames, so we have:

$$u(\mathfrak{M}) \Vdash u(\varphi).$$

Lemma 6.21b now gives us $\mathfrak{M} \Vdash \varphi$ and hence $\Gamma \Vdash^g \varphi$. \square

Note that this result is valid for all normal modal logics and corresponding frames. It is not specific to delta logics. But we'll now demonstrate it by proving a delta modeling result.

6.4.2 Example

We now illustrate the use of $\mathbf{K}\Delta\mathcal{U}$ through an example proof. Say we have the feature model as shown in Figure 6.2. The features F , G and H are implemented by the delta model dm in Figure 6.3. The feature T is satisfied in some empty core product, on which we'd like to apply those deltas.

We now introduce a set of basic axioms valid in this model:

6.23. Axiom (Delta Model Axioms): The following are assumed to hold:

- | | |
|--|---|
| <ul style="list-style-type: none"> (1) $F \leftrightarrow T$ (2) $G \leftrightarrow F$ (3) $H \leftrightarrow F$ (4) $G \leftrightarrow [y] G$ (5) $H \leftrightarrow [x] H$ | <ul style="list-style-type: none"> (6) $T \leftrightarrow [w] F$ (7) $F \leftrightarrow [x] G$ (8) $F \leftrightarrow [y] H$ (9) $G \leftrightarrow [z] G$ (10) $H \leftrightarrow [z] H$ |
|--|---|
- ┐

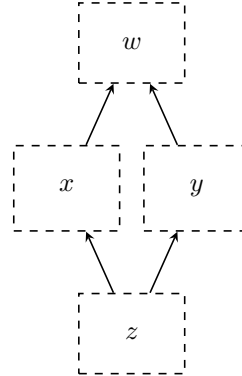
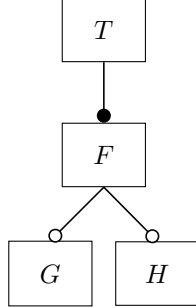


Figure 6.2: Example feature model Figure 6.3: Example delta model dm

Axioms (1), (2) and (3) are due to the feature model shown in Figure 6.2. It is generally the case that when a subfeature is implemented its superfeature is implemented as well. (4) and (5) are due to a property we assume the underlying deltoid to have, called non-interference [5], which states that commuting deltas cannot interfere with each others features. (6) to (10) are by design of the deltas: they were developed such that w , x and y implement the features F , G and H (6, 7 and 8), taking into account only the deltas ‘above’ them, and that conflict resolving delta z does not break the features implemented by the previous deltas (9 and 10).

Axioms (6) to (10) are enforced by the developers of the product line if they follow the workflow to be described in Chapter 7. It ensures desirable global properties by design if local constraints such as axioms (6) to (10) are met. Now say we have a core product $c \in \mathcal{P}$ with $c \models T$. For our example, we’d like to prove the following global property about delta model dm :

6.24. Lemma: $c \models [dm](T \wedge F \wedge G \wedge H)$

In order to prove this property more succinctly, we introduce the following auxiliary proof rules:

6.25. Lemma: For all formulas φ , ψ and χ , and for all box modalities $[d_1], \dots, [d_n]$, we have:

$$\varphi \rightarrow [d_1] \dots [d_n] \psi, \quad \psi \rightarrow \chi \quad \vdash \quad \varphi \rightarrow [d_1] \dots [d_n] \chi$$

Proof: By induction on n . □

6.26. Lemma: For all formulas φ and ψ and all box modalities $[d]$, we have:

$$\vdash \quad ([d] \varphi \wedge [d] \psi) \leftrightarrow [d] (\varphi \wedge \psi)$$

Proof: See [42, Example 1.40]. □

The numbers 1 to 10 in the proof of Lemma 6.24 refer to the ‘u’ translation of the corresponding item from Axiom 6.23.

Proof of Lemma 6.24:

- | | |
|---|----------------------------------|
| (11) $\mathbb{T} \leftrightarrow [w][x]\mathbb{G}$ | lem 6.25: 6, 7 |
| (12) $\mathbb{T} \leftrightarrow [w][x](\mathbb{F} \wedge \mathbb{G})$ | lem 6.25: 11, 2 |
| (13) $\mathbb{T} \leftrightarrow [w][x](\mathbb{F} \wedge \mathbb{G} \wedge [y]\mathbb{H})$ | lem 6.25: 12, 8 |
| (14) $\mathbb{T} \leftrightarrow [w][x](\mathbb{G} \wedge [y]\mathbb{H})$ | lem 6.25: 13, 2 |
| (15) $\mathbb{T} \leftrightarrow [w][x]([y]\mathbb{G} \wedge [y]\mathbb{H})$ | lem 6.25: 14, 4 |
| (16) $\mathbb{T} \leftrightarrow [w][x][y](\mathbb{G} \wedge \mathbb{H})$ | lem 6.26: 15 |
| (17) $\mathbb{T} \leftrightarrow [w][x][y]([z]\mathbb{G} \wedge \mathbb{H})$ | lem 6.25: 16, 9 |
| (18) $\mathbb{T} \leftrightarrow [w][x][y]([z]\mathbb{G} \wedge [z]\mathbb{H})$ | lem 6.25: 17,10 |
| (19) $\mathbb{T} \leftrightarrow [w][x][y][z](\mathbb{G} \wedge \mathbb{H})$ | lem 6.26: 18 |
| (20) $\mathbb{T} \leftrightarrow [w][x][y][z](\mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | lem 6.25: 19, 2 |
| (21) $\mathbb{T} \leftrightarrow [w][x][y][z](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | lem 6.25: 20, 1 |
| (22) $\mathbb{T} \leftrightarrow [w][x][y][dm_1](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | lem 6.25: 21, $\Delta\mathbb{U}$ |
| (23) $\mathbb{T} \leftrightarrow [w][x][dm_2](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | lem 6.25: 22, $\Delta\mathbb{U}$ |

Formula (24) is derived in a manner symmetric to formula (23).

- | | |
|--|----------------------------------|
| (24) $\mathbb{T} \leftrightarrow [w][y][dm_3](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | symmetric |
| (25) $\mathbb{T} \leftrightarrow [w][x][dm_2](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$
$\quad \wedge [w][y][dm_3](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | I_\wedge : 23,24 |
| (26) $\mathbb{T} \leftrightarrow [w]([x][dm_2](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$
$\quad \wedge [y][dm_3](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H}))$ | lem 6.26: 25 |
| (27) $\mathbb{T} \leftrightarrow [w][dm_4](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | lem 6.25: 26, $\Delta\mathbb{U}$ |
| (28) $\mathbb{T} \leftrightarrow [dm](\mathbb{T} \wedge \mathbb{F} \wedge \mathbb{G} \wedge \mathbb{H})$ | lem 6.25: 27, $\Delta\mathbb{U}$ |

where

$$\begin{array}{llll} dm_1 & = & dm \setminus \{w, x, y\} & dm_3 & = & dm \setminus \{w, y\} \\ dm_2 & = & dm \setminus \{w, x\} & dm_4 & = & dm \setminus \{w\} \end{array}$$

Then, by $c \Vdash \mathbb{T}$, we have our result. \square

Many steps are skipped in this proof, mostly those concerned with invoking propositional tautologies and applying modus ponens. We have kept only the more interesting steps — those that directly use our axioms.

Since satisfiability for the normal multimodal logic is decidable (in fact, it is PSPACE-complete [41]), and the special modal operators of delta logic can be trivially translated away (Definition 6.10), proofs such as this one can be automated.

6.5 Conclusion

Much of ADM is dedicated to the goal of developing syntactic languages and techniques for semantic concepts. Deltas are syntactic. But products (from the ADM point of view) are semantic concepts. Consequently, reasoning about the semantics of deltas requires semantic proof machinery.

This chapter describes how modal logic can solve this problem. Given any kind of decidable specification language for the product domain, wrapping a multi-modal logic around it enables us to prove that certain deltas implement certain features, that they do not break existing features, and so on. The result is a language reminiscent of dynamic logic, but lacking a construct for iteration, making the logic decidable.

The chapter shows that the modal proof system can be used to prove certain kinds of delta correctness, but in Section 6.3.4 we discover that it cannot be used in delta postconditions to refer back to the original product. They are therefore unable, for instance, to specify that software delta (`remove class C`) does not modify any classes other than C. Chapter 9 briefly discusses how an extension to hybrid logic [22, 40] may be used to overcome this without losing decidability.

6.6 Related Work

Completeness proofs in modal logic have a long-standing history, closely tied to the history of relational semantics based on Kripke frames. A comprehensive survey of this history can be found in e.g. [42, Section 1.8].

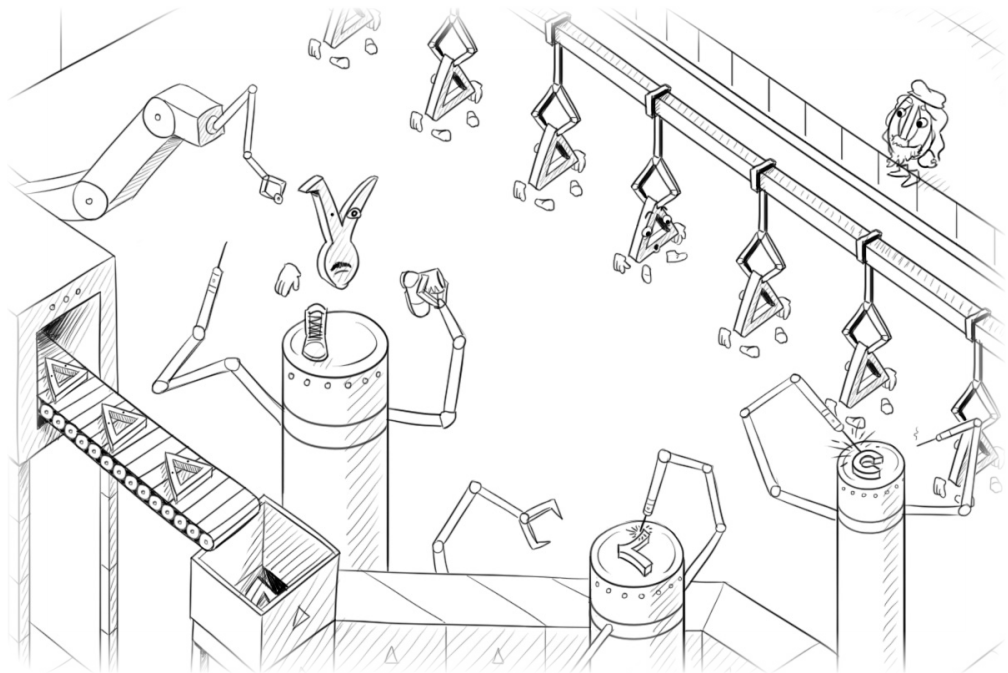
The modal logic presented in this chapter has a flavour very reminiscent of dynamic logics such as PDL [42, 69]. A crucial difference, however, is that the logic presented here is simpler (and hence, easier to work with) due to the absence of iteration. Due to this simplicity, complex modalities can be easily unraveled into simpler ones, enabling the main results from Sections 6.3 and 6.4.

Partial motivation for the presented delta logics is to make formal properties in the Delta Modeling Workflow (Chapter 7) more transparent. The proof of Lemma 6.24 is just an example of a proof of product line completeness for a specific case.

Finally, it is worth noting that the typesystem described by Lienhardt and Clarke [120], unlike the logic of this chapter, is able to specify that deltas in an object oriented setting do not modify unmentioned artefacts, by regarding them as polymorphic functions.

Delta Modeling Workflow

On the Development of Delta-based Product Lines



7.1 Introduction

Until now we have seen what is *possible* with Abstract Delta Modeling; what the various formal artefacts look like and what they mean. But it may still not be clear how they should actually be used in practice. If a team of developers started out with only a product line specification, how would they actually build and organize the product line implementation? How should the deltas be ordered and what should be their application conditions and content, for maximal reuse of code and isolated, concurrent development of features?

Goal: *Describe how delta-based product lines might be built.*

This chapter proposes a specific development workflow for ADM, dubbed *Delta Modeling Workflow (DMW)*. The structured and flexible nature of ADM lends itself quite naturally to a systematic approach to building product lines. This chapter stays at the same level of abstraction as before, but approaches the topic from the other side. It describes, step-by-step, how to build a product line from scratch. At the moment, it is far from a practical guide, as it requires a fully defined product line specification in advance — an unrealistic requirement in modern engineering practices. But it is the first step in guiding the proper use of the delta modeling concepts introduced in previous chapters.

Of course, there may be many ways to use delta modeling to good effect. Indeed, many tools are eventually put to innovative uses that were initially unintended. Let's just say that ADM lends itself naturally to a certain way of working which happens to exhibit favorable properties. Following it leads to a well-structured product line that automatically exhibits two desirable properties: global unambiguity (Definition 4.14, page 105) and total correctness with regard to the specification (Definition 4.20, page 108). The work is split up into well-defined jobs derived from that specification.

Most importantly, the workflow naturally supports concurrent development. Multiple developers can work on parts of a non-trivial product line implementation at the same time and in isolation without breaking global unambiguity or correctness. An important reason for this is the concept of *delta model locality*: any delta under development need only take into account the existing deltas that occupy *subordinate positions* in the delta model.

Of course, a development workflow, of all things, should be evaluated in practice. Formal proofs, while valuable, are not enough to guarantee a good practical experience. Therefore, the DMW should be evaluated based on its application to an industrial scale system.

Goal: *Test the delta modeling workflow on an industrial scale system in order to evaluate its practical applicability.*

To that end, the workflow was used to model the replication system of the *Fredhopper Access Server (FAS)* product line, in one of the industrial scale case studies of the HATS project. This was done using the *Abstract Behavioral Specification (ABS)* language of the HATS project, in which delta modeling is an integrated component.



Figure 7.1: Two different feature diagrams representing the same Definition 4.3 style feature model $\Phi = \{\{f, g\}, \{f, g, h\}\}$.

The chapter is organized as follows: Section 7.2 enriches the specification of product lines to include a subfeature relation. Section 7.3 introduces the fundamental notion of locality, setting the stage for Section 7.4 to describe the workflow itself. Section 7.5 describes the ABS language and provides a succinct DMW description in concrete ABS terms. Following that, Section 7.6 discusses its application to the replication system product line of FAS. Finally, Sections 7.7 and 7.8 offer concluding remarks and discuss related work.

In Appendix A (page 210), the beneficial properties of DMW are proved formally. It includes a formulation of the workflow using operational semantics.

7.2 The Subfeature Relation

From this point on, assume that a deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, \llbracket - \rrbracket)$ and a feature set \mathcal{F} are given. Assume also that the deltoid exhibits consistent conflict resolution (Definition 3.18).

Feature models as formalized in Section 4.2 are not as useful for developers as they could be. When we view a feature model as the set of all possible feature configurations, we disregard the intended hierarchical structure between features. Compared to a traditional feature model [66, 105, 166], the “ Φ ” representation from Definition 4.3 (page 100) lacks some useful information. For instance, we lose the distinction between the two feature models in Figure 7.1, which would both have $\Phi = \{\{f, g\}, \{f, g, h\}\}$.

Since the feature diagram notation is quite common in product line engineering [166], it is a sensible structure to base the workflow on. To capture the hierarchy represented by feature diagrams, we introduce the following binary relation into the product line specification tuple (Definition 4.19, page 108):

- **7.1. Definition (Subfeature):** The binary *subfeature* relation is a strict partial order $\Rightarrow \subseteq \mathcal{F} \times \mathcal{F}$. Write $f \Rightarrow g$ when g is a subfeature of f and f is a *superfeature* of g . ┘

Note that this definition allows one feature to have multiple direct superfeatures (a ‘join’ in the feature diagram), something that is not standard in feature modeling, but useful, as you may remember from Section 5.2 (page 120).

The subfeature relation recognizes both mandatory and optional subfeatures. We only use it to formally introduce the feature diagram structure. Information on optionality, grouping, implication, exclusion and more can be derived from the combination of Φ and \Rightarrow . For example, if $f \Rightarrow g$ and there exists a feature configuration that includes f but not g , we know that g must be an optional subfeature rather than a mandatory one.

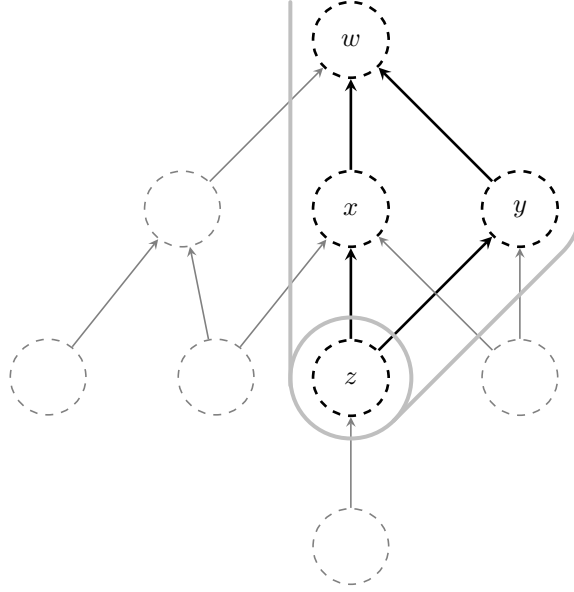


Figure 7.2: The local delta model of z in the context of a larger delta model.

7.3 Locality

When developing a specific delta in the (annotated) delta model of a product line, it would be inconvenient if we had to consider all other deltas to make sure the combined whole works properly — this would defeat the purpose of using delta modeling for modularity and separation of concerns in the first place. Thus the workflow should only require that local constraints are met, and then guarantee beneficial properties for the entire product line implementation by construction. But what does ‘local’ mean in the context of ADM?

- **7.2. Definition (Local Delta Model):** Given a delta model $dm = (D, \prec)$, the *local delta model* of a given delta $d \in D$ is defined as follows:

$$\downarrow d \stackrel{\text{def}}{=} (D', \prec \cap D' \times D')$$

where $D' = \{x \in D \mid x \prec d\}$ (known as the *principal ideal* of d in \prec). If the delta model is not clear from context, we attach a subscript as in \downarrow_{dm} . \lrcorner

This concept embodies a basic principle of the DMW: when engaging in the implementation of a new delta z , you already know which position in the delta model it will occupy. During development and maintenance you only have to know about the deltas that are to be applied earlier —those that z has control over (Figure 7.2)— and you need to establish certain correctness properties only over this local delta model.

Conversely, during long term maintenance, when any delta x is changed, it will always be clear which other deltas may now be out-of-date and in need of attention: all deltas $z \succ x$, which have x in their local delta model.

But in order for this to be enough to guarantee global properties, we need to place one restriction regarding the balance between the deltoid and the valuation function. Namely, we need to ensure that deltas that are *not* related

through the application order, and are therefore not in each others local delta model, cannot influence each others semantic effects on the final product; or, if they do, that this can be automatically detected by them being in conflict (Definition 3.7, page 76). We now assume this property of *non-interference* (formally described in Definition A.2, page 211). It is independent of any specific product line, though does depend on the kind of features that need to be implemented. Systems that break this restriction might include deltas that can add advice in aspect-oriented languages [114] that have effects beyond the entities they overwrite, or features with mutually exclusive specifications.

7.4 Workflow Description

The goal of the workflow is to start with a product line specification and implement from this a product line, by implementing all features, resolving all conflicts and implementing all desired feature interaction in an iterative process, maximally exploiting parallelism in the development.

7.4.1 Input

The input to the workflow is a product line specification and subfeature relation. The feature model Φ indicates which feature configurations need to be derivable. The order of the workflow steps is guided by the subfeature relation \Rightarrow . The valuation function V , in a sense, guides the implementation of each individual delta.

7.4.2 Output

The output of the DMW is a product line implementation $(\Phi, c, D, \prec, \gamma)$. The goal is for this implementation to be totally correct with regard to the specification (Definition 4.20, page 108). One of the ways we make this easier is by ensuring global unambiguity. This means we'll be able to work with sole derivation semantics for delta models (Definition 3.5, page 75). Taking advantage of ambiguous semantics (Section 3.5, page 88) is planned as future work.

The feature model Φ of the implementation will be the same as that of the specification. While it is true that a product line implementation is allowed to implement more feature selections than are specified —while maintaining total correctness—, that is not the goal of this workflow.

We make the core product an initial product $c = 0$ (Definition 2.58, page 57) and do everything with deltas, a practice that has been dubbed *pure delta oriented programming* [162], or, in this case, *pure delta modeling*. While this is not required —i.e., it is possible to implement mandatory features in the core product— the choice simplifies the workflow description. Note, however, that optional features should never be implemented in the core product (with the intention of selectively ‘removing’ them with deltas) as this is incompatible with the workflow and can be said to be less flexible and robust.

The annotated delta model (D, \prec, γ) is initialized as empty and built up during the workflow.

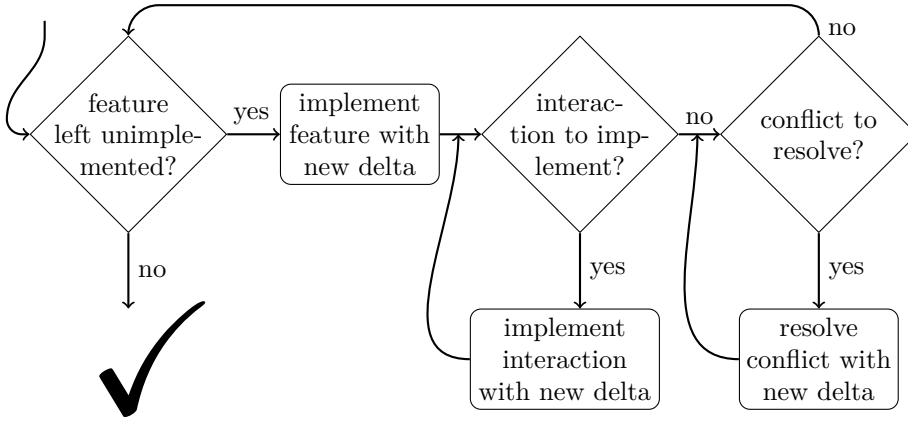


Figure 7.3: An intuitive overview of the development workflow.

7.4.3 A Sequential View

An overview of the workflow process is shown in Figure 7.3. Regard it as the flowgraph that a single developer would follow to implement a product line:

1. Is there a feature f that still needs to be implemented?
If not, go to step 7.
2. Implement feature f with new *feature implementation delta* d_f .
3. Is there a required interaction between a set of implemented features F with $f \in F$ that still needs to be implemented? If not, skip to step 5.
4. Implement this interaction with a new *feature interaction delta* d_F .
Then go back to step 3.
5. Is there an unresolved conflict $x \not\leq y$ involving any of the deltas introduced in this iteration? If not, go back to step 1.
6. Resolve the conflict with a new *conflict resolving delta* $d_{\{x,y\}}$.
Then go back to step 5.
7. The product line implementation is finished.

This was the workflow description used in the first DMW papers [5, 7]. It gives a good intuition as to what the workflow is all about. But it is not ideal for describing concurrent development by multiple engineers. Therefore, we break up this flowgraph into its constituent steps, and set up a proper dependency model.

7.4.4 Jobs

We now introduce the higher-level concept of *jobs*, each of which involve the development of a specific delta to place into the annotated delta model in progress. We distinguish between two kinds of job, each with a specific purpose:

- *Feature implementation jobs*, identified by a feature set $F \subseteq \mathcal{F}$, are to develop a delta responsible for either the implementation of a single feature —when $F = \{f\}$ for some f — or the interaction between a set of features. Conceptually these two cases are really the same, so it is more elegant to drop the distinction.

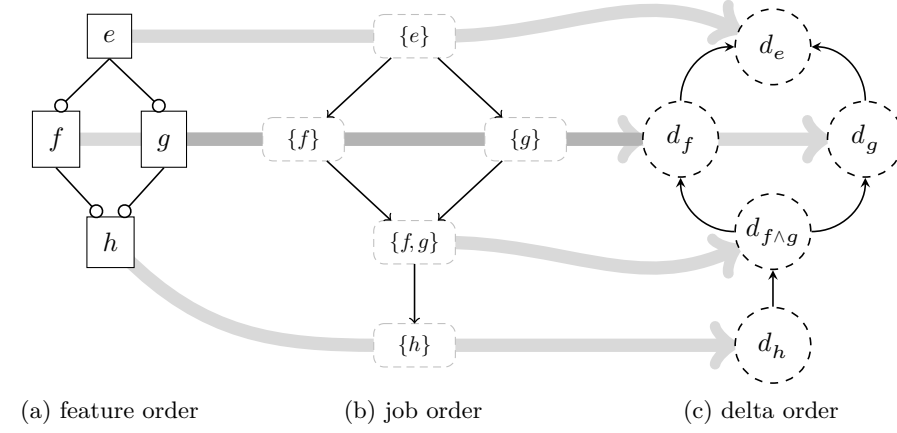


Figure 7.4: How the subfeature relation \Rightarrow informs the job order and how that, in turn, informs the delta application order \prec . In this example, features f and g require additional implementation effort, and their mutual subfeature h is implemented last.

- *Conflict resolution jobs*, identified by a finite set of already developed deltas $C \subseteq \mathcal{D}$, implement a delta to resolve the conflict(s) between the deltas in C .

Both the old and new workflow descriptions resemble an algorithm [116], but fail to be one for a simple reason: neither type of job can generally be *automated*. Both feature implementation and conflict resolution require creativity and domain-knowledge. So rather than provide well-defined instructions, a job imposes requirements on the local delta model of the new delta (Definition 7.2). The delta needs to be developed in a way that satisfies those requirements.

Although there are many jobs that can be performed concurrently, there are some that need to be performed in a specific order. Characterizing this order is one of the main contributions of this chapter. It is a strict partial order, and it is reflected in the following ways (Figure 7.4):

- the \Rightarrow relation, strict partial order of the feature diagram,
- the order in which the jobs are to be performed, and
- the \prec relation, strict partial order of the delta model under development.

Simply put: the feature order \Rightarrow informs the job order which, in turn, informs the application order \prec .

Let's first discuss how feature implementation jobs are ordered, and forget about conflict resolving jobs for now. The main idea is that individual features f and g are implemented in the strict partial order of the subfeature relation \Rightarrow . That is, if $f \Rightarrow g$, then f (the superfeature) will be implemented before g (the subfeature). This is reasonable; as base functionality should naturally be in place before it is extended by subfeatures.

But in the general case, feature implementation jobs are feature *sets*, sometimes larger than one. We extend the subfeature relation \Rightarrow to sets of features, so we can use it to order all such jobs:

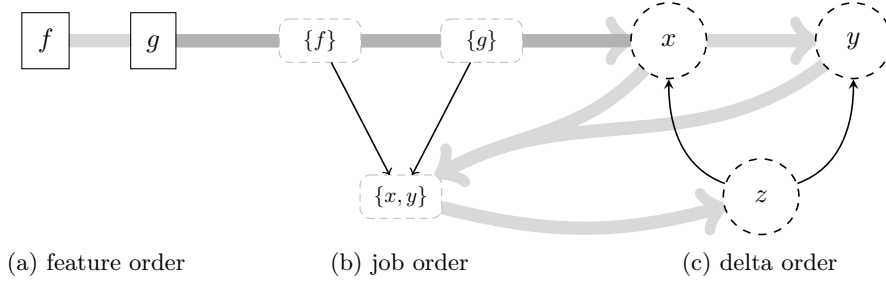


Figure 7.5: How conflicts between existing deltas introduce new jobs.

- **7.3. Definition (Feature Combination Order):** We extend subfeature relation \Rightarrow to sets of features. The resulting *feature combination order* is the smallest strict partial order $\Rightarrow \subseteq \text{Pow}(\mathcal{F}) \times \text{Pow}(\mathcal{F})$ such that, for all features $f, g \in \mathcal{F}$ and feature combinations $F, G \subseteq \mathcal{F}$ with $f, g \notin F \cup G$, the following axioms hold:

$$\begin{aligned} \text{a. } F \subset G &\implies F \Rightarrow G \\ \text{b. } f \Rightarrow g &\implies F \cup \{f\} \Rightarrow G \cup \{g\} \end{aligned}$$

When $F \Rightarrow G$ we say that F is *weaker* and that G is *stronger*. \lrcorner

- **7.4. Example:** Figure 7.4 illustrates how the subfeature relation guides the order in which the job transitions are allowed to take place. For example, the features $e, f, g, h \in \mathcal{F}$ have $e \Rightarrow f, g \Rightarrow h$. So, with regard to feature combinations, we have $\{e\} \Rightarrow \{f\}, \{g\} \Rightarrow \{f, g\} \Rightarrow \{h\}$, as shown in Figure 7.4b. \lrcorner

Conflict resolving deltas are implemented through conflict jobs $C \subseteq D$. Such jobs are introduced from analysis on the current state of the product line implementation, rather than from analysis of the specification. This is illustrated in Figure 7.5. When existing deltas are in conflict (Figure 7.5c), a conflict job can be introduced (Figure 7.5b) for the implementation of a new delta to resolve the conflict (back to Figure 7.5c).

7.5 The Abstract Behavioral Specification Language

The *Abstract Behavioral Specification (ABS)* language [8, 52, 100] was developed within the FP7 EU project HATS, the project that started and guided my PhD research [80]. This is one of the only languages with delta modeling integrated into its core design (if not the only). As a member of the HATS project, I collaborated on the implementation of delta modeling in ABS and described the delta modeling workflow in terms of ABS constructs.

Section 7.5.1 provides a short background on the ABS language. Section 7.5.2 describes the concrete ‘ABS delta modeling workflow manual’ [8].

7.5.1 Background

The ABS language is designed for formal modeling and specification of concurrent, component-based systems at a high level, though is perfectly capable of producing executable programs. Particularly, it is targeted at complex software systems that exhibit a high degree of variation, such as software product lines.

Figure 7.6 describes the layered architecture of the ABS language. At its most fundamental levels, it provides pure functional programming constructs, algebraic data types, an object model and imperative language constructs. The main contributions of the HATS project are built on top: concurrency constructs based on the concept of COGs, a language layer for behavioral specification as well as module and component structures.

The top left layer is of interest to us. Delta modeling in ABS consists of four languages: μ TVL, DML, CL and PSL.

μ TVL is a feature description language based on a subset of TVL [46]. It is used to describe the variability of a product line in terms of (attributed) feature models (Section 4.2).

The delta modeling language DML is used to develop the delta modules containing modifications of a core ABS model. A delta module in ABS is similar to the software deltas of Section 2.3, and can modify classes, methods and fields on a course-grained level. The previous implementation of a method in the derivation can be invoked by using the **original()** call (similar to the **super** keyword of AHEAD [31]), though it is not possible to invoke specific method versions through the name of the delta.² As a bonus, delta modules can be parametrized by specific values as well as the ‘feature Booleans’ (Section 4.5).

The configuration language CL links μ TVL feature models with the DML delta modules that implement the corresponding behavioral modifications, and also specifies the order in which those delta modules should be applied. Therefore, CL specifications fulfil the role of annotated delta models (Definition 4.7). The language provides **when** and **after** keywords, which work the same way as their equivalents **if** and **after** in the L^AT_EX packages of Section 5.2 (page 120).

Finally, the product selection language PSL is used to give names to specific feature configurations, by which the corresponding products of an ABS product line implementation can then be generated. A PSL script contains a feature selection, a set of values for the relevant attributes, and an *initialization block*,

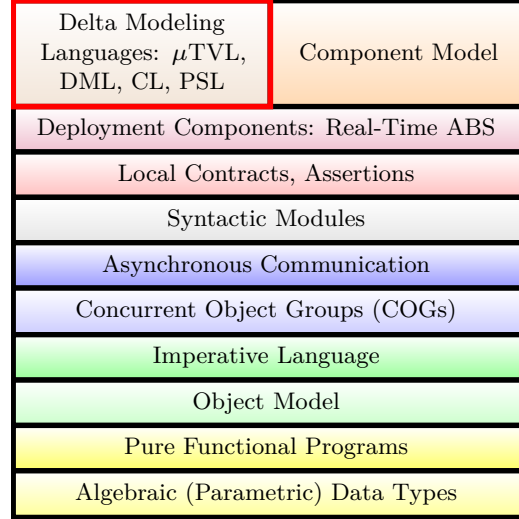


Figure 7.6: Layered Architecture of ABS¹

¹This figure was designed by Reiner Hähnle for the joint Architecture paper [8].

²Actually, in Section 7.5.2, we’ll pretend that it is.

which is often just a call to an appropriate *main* method, though it may also contain configuration code. After the extraction of the proper selected delta model (Definition 4.8) and its application to the core program, the initialization block is added to the result to be the first code to run.

A more detailed account of these languages may be found in the second report on ABS [52]. The content of this section is based on my work in the third report [8].

7.5.2 DMW for ABS

The following workflow description gives step-by-step instructions for development of an ABS software product line, specifying the proper code templates to use for each of the steps of Figure 7.3.

It often makes sense to put basic code common to all products into the core product directly. In the case of ABS, this means at least the following:

```
1 class Main { Unit run() {} } { new Main(); }
```

We start with a *Main* class with an empty *run* method. We then create a new *Main* instance, implicitly calling the *run* method, which will later be modified by deltas. It is possible to put mandatory features into the core product. But, as mentioned, it is recommended that all features are implemented with deltas, as this makes the product line more robust to evolution, and promotes the separation of concerns.

Also, we begin with a minimal ABS product line configuration: the list of features and the list of desired products. The latter can be empty.

```
1 productline (name) { features  $f_1, f_2, \dots, f_n$ ; }
```

In the following workflow description, we'll use a subset of the Editor product line example of Section 1.4, one containing only the *Ed*, *PR*, *SH* and *EC* features. Now we specify each step of the flowchart from Figure 7.3.

Step 1: Feature left unimplemented?

In this stage of the workflow, we choose the next feature to implement. Essentially we walk through the subfeature hierarchy of the feature model in a topological order, i.e., base features first, subfeatures later. If all features have been implemented, we are finished.

For the example, we would have to start with the Editor (*Ed*) feature. Any of the three features on the second level may be chosen next.

Step 2: Implement feature with new delta

Having chosen a feature f , we now write a “feature delta” d_f to implement it:

```
1 delta  $d_f$  { ... }
```

The delta may add, remove or modify any classes and methods necessary to realize the functionality of f , while preserving the functionality of all superfeatures. The developer only has to consider the local delta model: the core product and the deltas implementing superfeatures of f . The following four feature deltas implement the four individual features of the Editor product line (some details are left out for the sake of brevity):

```

1  delta D_Ed { // Editor Delta
2      adds class Model { ... }
3      adds class Font { ... }
4      adds class Editor (Model m) {
5          Model m_model;
6          Font m_plain_font;
7          { init(m); }
8          Unit init(Model m) {
9              m_model = m;
10             m_plain_font = new Font();
11         }
12         Model model() { return m_model; }
13         Font font(int c) { return m_plain_font; }
14         Unit onMouseOver(int c) { /* nothing */ }
15     }
16     modifies class Main {
17         modifies Unit run() { new Editor(new Model()); }
18 } }

1  delta D_Pr { // Printing Delta
2      modifies class Editor {
3          adds Printer m_printer;
4          modifies Unit init(m: Model) {
5              original(m);
6              m_printer = new Printer();
7          }
8          adds Unit print() { /* print the plain text */ }
9  } }

1  delta D_SH { // Syntax Highlighting Delta
2      adds class SyntaxHL (Model m) {
3          Model m_model;
4          { m_model = m; }
5          Color color(int c) { ... }
6      }
7      modifies class Editor {
8          adds SyntaxHL m_syntaxhl;
9          modifies init(Model m) {
10             original(m);
11             m_syntaxhl = new SyntaxHL( model() );
12         }
13         modifies font(int c) {
14             Font f = D_Ed.original(c);
15             f.setColor( m_syntaxhl.color(c) );
16             return f;
17     } } }

1  delta D_EC { // Error Checking Delta
2      adds class ErrorCh (Model m) {
3          Model m_model;
4          { m_model = m; }
5          Bool errorOn(int c) { ... }
6          String errorText(int c) { ... }
7      }

```

```

8   modifies class Editor {
9       adds ErrorCh m_errorch;
10      modifies init(Model m) {
11          original(m);
12          m_errorch = new ErrorCh( model() );
13      }
14      modifies Font font(int c) {
15          Font f = D_Ed.original(c);
16          f.setUnderlined( getModel().isError(c) );
17          return f;
18      }
19      modifies onMouseOver(int c) { ... }
20 }

```

Finally, we add the following line to the ABS product line configuration:

```

1 delta  $d_f$  when  $f$  after  $d_s$ ;

```

where d_s is the delta implementing the superfeature of f . If f has no superfeature, the **after** clause may be omitted. Our example requires the following product line configuration:

```

1 productline PL_Editor {
2     features Ed, Pr, SH, EC;
3     delta D_Ed when Ed;
4     delta D_Pr when Pr after D_Ed;
5     delta D_SH when SH after D_Ed;
6     delta D_EC when EC after D_Ed;
7 }

```

Step 3: Interaction to implement?

At the feature modeling and specification level, two features f and g may be independently realizable, but require extra functionality when both are selected. This behavior is not implemented by the feature deltas, so a new delta needs to be created. In our example, this is the case for the features Printing and Syntax Highlighting. When printing, we would like the syntax highlighting colors to be used.

Step 4: Implement interaction with new delta

The new delta $d_{f,g}$ must implement the required interaction without breaking the features f and g or their superfeatures. It may change anything introduced by feature deltas d_f and d_g . When overwriting methods, it may also access the original methods using the syntax d_f .**original**() and d_g .**original**(). In our example:

```

1 delta D_Pr_SH { // Pr + SH Interaction Delta
2     modifies class Editor {
3         modifies Unit print() {
4             // print as before, but use
5             // colors of D_SH.font(c)
6         } } }

```

Then we add the following to the ABS product line specification:

```
1 delta  $d_{f,g}$  when  $f$  &&  $g$  after  $d_f, d_g$ ;
```

In our example:

```
1 delta D_Pr_SH when Pr && SH after D_Pr, D_SH;
```

This may be generalized to interaction between more than two features.

Step 5: Conflict to resolve?

By adding new deltas, we may have introduced an implementation conflict between two deltas d_1 and d_2 that are independent, but modify the same method in a different way. In our example, this is the case for D_SH and D_EC, as they both modify the `font(int)` method in a different way, and are not ordered in the product line configuration. For each such conflict, we write a delta to resolve it.

Step 6: Resolve conflict with new delta

The conflict resolving delta $d_{1,2}$ must overwrite the methods causing the conflict, while not breaking the features implemented by d_1 or d_2 , or their super-features. Typically, $d_{1,2}$ invokes $d_1.\text{original}()$ and $d_2.\text{original}()$ to combine the functionality of the conflicting deltas. In our example:

```
1 delta D_SH_EC { // SH + EC Conflict Resolving Delta
2   modifies class Editor {
3     modifies Font font(int c) {
4       Font result = D_Ed.original(c);
5       result.setColor(D_SH.original(c).color());
6       result.setU...lined(D_EC.original(c).u...lined());
7       return result;
8   } } }
```

We then add the following to the ABS product line specification:

```
1 delta  $d_{1,2}$  when (  $\gamma(d_1)$  ) && (  $\gamma(d_2)$  ) after  $d_1, d_2$ ;
```

where $\gamma(d)$ is the **when** clause of delta d . In our example:

```
1 delta D_SH_EC when (SH) && (EC) after D_SH, D_EC;
```

Step 7: Done

This means the product line implementation is finished, and it enjoys total correctness by construction.



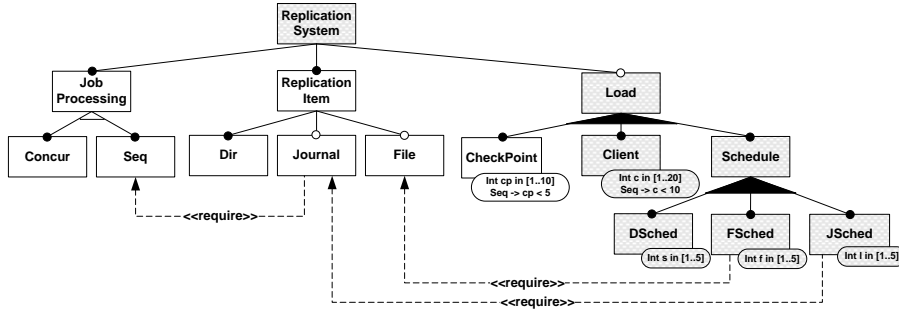


Figure 7.7: Feature diagram of the FAS replication system

7.6 The Fredhopper Access Server

This section discusses the use of the delta modeling workflow for modeling the industrial case study of the Fredhopper Access Server (FAS) product line. FAS, developed by Fredhopper B.V.³, is a distributed service-oriented software system for Internet search and merchandising. In 2012, we used the workflow to model FAS's *replication system*, which ensures data consistency across a FAS deployment. The FAS product line is modeled using the ABS language (Section 7.5).

First, Section 7.6.1 briefly describes the FAS case study. Then we discuss the results of our modeling efforts in Section 7.6.2. We don't discuss the implementation itself in this thesis, as this would duplicate quite some information. Details can be found in the corresponding paper [7].

7.6.1 FAS Overview

The *Fredhopper Access Server (FAS)* is a component-based and service-oriented distributed software system. It provides search and merchandising services to e-Commerce companies such as large catalogue traders and travel agencies. Without going into too much detail: FAS tries to provide full throughput of data across many different clients by cleverly replicating data across a network of nodes. As part of the *FAS product line*, there are several variants of this *replication system*. We've implemented these variants in ABS using the delta modeling workflow.

We let the product line specification be that of the replication system. The feature model, also shown in Figure 7.7, is expressed in μ TVL as follows:

```

1 root RS {
2   group allof {
3     JobProcessing { ... },
4     ReplicationItem { ... },
5     opt Load {
6       group [1..3] {
7         Client { Int c in [1..20]; Seq -> c < 10; },
8         CheckPoint { ... },

```

³<http://www.fredhopper.com>

Metrics	Java	ABS
Number of lines of code	~6400	5000
Number of classes	44	40
Number of interfaces	2	43
Number of user-defined functions	<i>n/a</i>	80
Number of user-defined data types	<i>n/a</i>	17
Number of features	<i>n/a</i>	15
Number of deltas	<i>n/a</i>	10 (7)
Number of products	<i>n/a</i>	12108 (96)

Table 7.8: Metrics on the FAS replication system code

```

9      Schedule {
10          group [1..3] {
11              DSched { Int s in [1..5]; },
12              FSched { Int f in [1..5]; require: File; },
13              JSched { Int l in [1..5]; require: Journal; }
14      } } } } }

```

For reasons of space and to focus on the application of the workflow, rather than the replication system itself, we considered only the modeling of the features RS, Load, Client, Schedule, DSched, FSched and JSched as the representative parts of the replication system variability. These are the features that are shaded in Figure 7.7. The other features are also omitted from the μ TVL code above.

7.6.2 Results

The existing FAS product line was implemented in Java, and had over 150,000 lines of code. Table 7.8 shows some metrics about the existing implementation and the ABS model of the replication system only. In particular, if parametrized deltas are used to resolve the three-way conflict between DSched, FSched and JSched, the number of deltas reduces from 10 to 7. If feature attributes are ignored, the number of possible feature configurations reduces from 12108 to 96.

We now discuss our experiences while applying the DMW to the implementation of the FAS case study. This case study not only raised discussion points about the pros and cons of the DMW, but also guided the development of DMW while its practical applicability was put to the test.

Correctness Following the DMW we were able to systematically implement all features in the feature model in a top-down fashion to obtain a product line implementation of the replication system. We were also able to systematically implement all necessary feature interaction and resolve implementation conflicts between deltas, since the workflow directed us to consider every situation. So we avoid accidentally forgetting to implement some functionality from a complex feature model.

Collaboration During the case study, the workflow description of the original paper [5] was used, which did not yet focus on concurrent development. We were therefore unsure how to apply DMW in a collaborative development environment. Feedback from this case study has led to the new job-based description and the formalization found in Appendix A, in which concurrent development is an explicit benefit.

Evolution The original DMW description assumed the core product to be the empty program. In the case study we relaxed this assumption to facilitate product line evolution. In practice it is often the case that a product line will not be implemented from scratch, but will be built on legacy code, which lends itself to be incorporated as the core product. As a result, the formal DMW description no longer requires an empty core.

Overall, the conclusion was that DMW offers a useful guideline for systematically traversing the feature model and implementing its features to arrive at a software product line which is globally unambiguous and correct. And in retrospect, I can add that the feedback gathered during this experience was an invaluable tool in improving the workflow description.

7.7 Conclusion

The formalisation of ADM thus far had been *descriptive*, describing what deltas are, how they work and how they are selected. The other side of the story is *prescriptive*. This chapter describes useful patterns for the implementation of product lines. It delineates a workflow to show insight in how the delta modeling constructs may be used to good effect.

The main contribution is insight in how independent features can be implemented concurrently and in isolation. Important to this is the concept of *locality*. Any delta under development need only take into account the existing deltas that occupy subordinate positions in the delta model — the ones the new delta has control over. In effect, the product line specification is split up and localized. If each delta is developed to satisfy local constraints, the resulting product line will exhibit total correctness. In Appendix A, the workflow is formalized with an operational semantics, and this is proved as a theorem.

Finally, the chapter describes the delta modeling workflow for the *Abstract Behavioral Specification (ABS)* language, which was developed for the HATS project. Then, its application to the *Fredhopper Access Server* is discussed — the results, and the lessons we learned. This industrial scale case study helped validate and improve the workflow.

7.8 Related Work

Software product lines have existed in industry for quite a while, and many useful lessons have been learned from this experience [54, 65, 117, 122, 155]. Reuse in software as a way to improve quality and time to market has been an important theme since the 1960s [65], and software product lines in particular see their origins in the early 1980s, though not yet by that name [65].

However, it has been a predominantly empirical field; it is only recently that formal methods have started being applied, with initiatives such as the HATS project [80]. As such, though various building blocks and algebras of

software composition have been thoroughly formalized over the last decade or so (Sections 2.10, 3.8 and 4.8). We have been unable to find much previous work applying a similar level of formalism to the development process itself, though a number have recently emerged [62, 142]. The work on the delta modeling workflow [5, 7, 8, 2] —and therefore this thesis chapter— are, in part, an effort to fill this gap. But they are also a way to tell a part of the delta modeling story that didn't fit in the original ADM papers.

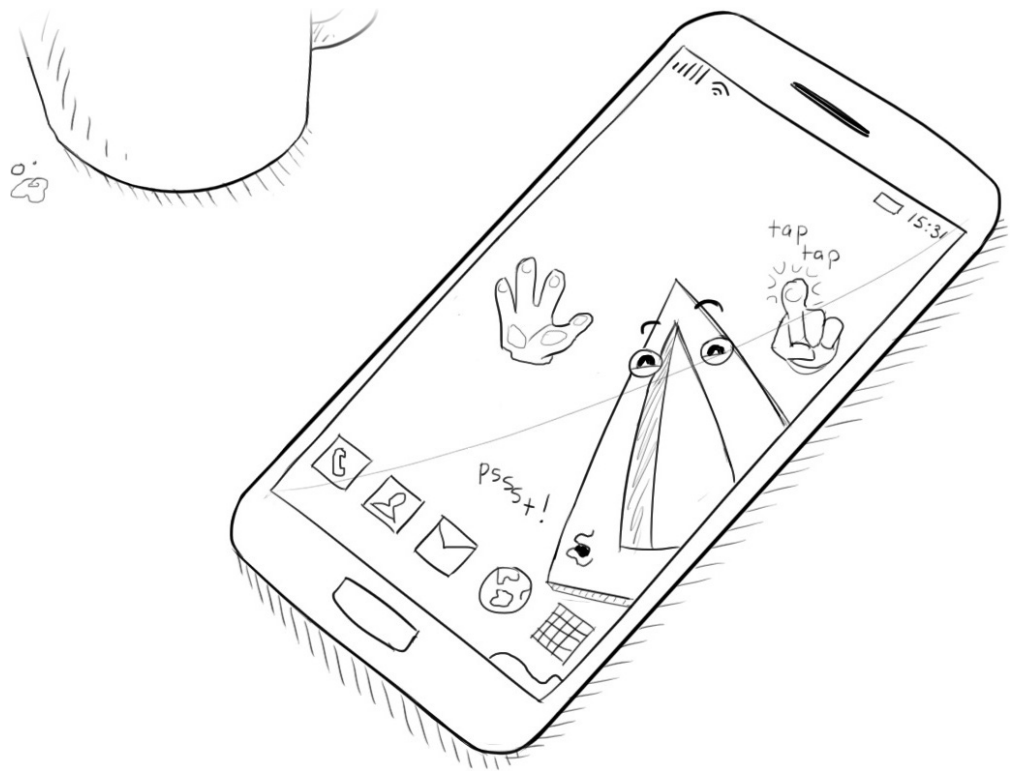
Our formalization of feature models (Definition 4.3) is actually quite close to the practice of *decision modeling*, as, for example, described by Czarnecki et al. [61]. They note that an essential ingredient of feature modeling, in contrast to decision modeling, is the subfeature hierarchy. And indeed, this information seems to express design intentions essential for the development of a properly modular product line. The addition of the subfeature relation in Section 7.2 and its use in guiding the development workflow is in recognition of this.

In a short survey paper, Krueger [117] makes a lot of points relevant to this chapter. For one, he stressed the importance of *mass customization* over application engineering: the practice of working on all product line members at once rather than on one at a time. This is done through the feature-oriented development style of DMW. It is set up so that every product that needs a certain feature gets this feature from the same delta. He also notes the importance of encapsulation among the various implementation artefacts: “[If] any feature can impact any core asset and any core asset may be impacted by any feature, [this] has all of the software engineering comprehension drawbacks as global variables in conventional programming languages.” ADM addresses this problem with its partially ordered module structure and conflict resolution model (Chapter 3). The notions of locality and non-interference introduced in Section 7.3 aim to complement these, in order to reduce the combinatorial complexity of software product lines.

Finally, he makes the following point in which DMW still falls short: it is rare that product lines are developed from scratch. Usually, legacy code is already in place, and discarding it to start over is too disruptive to project schedules. Similarly, with the rising popularity of *agile development* methodologies [130], it is now considered unwise to demand a full specification in advance, preferring short cycles of development and adaptation, making that aspect of the workflow description decidedly non-agile. However, the core principles of the workflow do have great potential in that regard. This is discussed in more detail in Chapter 9.

Dynamic Product Lines

On the Runtime and Context-aware Application of Deltas



8.1 Introduction

Traditionally, a feature configuration (Section 4.2.2, page 100) is chosen once at build-time. Its corresponding product is then generated and deployed, after which the chosen feature configuration can no longer change. That is sometimes limiting, as it could be advantageous for products to be able to adapt to dynamic conditions at runtime [24]. *Dynamic software product lines* [82] are product lines for which the feature configuration is *not* fixed. It can be changed dynamically in order to meet changing requirements for continuously running systems, upon which the running product can adapt accordingly.

Damiani et al. have discussed delta-based dynamic software product lines before [63, 64]. As a way of offering additional insight, the motivation of this chapter is based on their work. They explore several of the problems encountered in an object-oriented setting. In particular, they introduce a **reconfigure** statement to the programming language, which, when reached at runtime, offers the system the opportunity to adapt the running product to the newest feature configuration. A developer places this statement wherever it is deemed safe for the system to do so without creating inconsistencies — a sensible precaution. However, the semantics of **reconfigure** is never formalized. From the perspective of structural operational semantics [153], the corresponding inference rule might take the following shape:

$$\frac{\langle \text{premises} \rangle}{\langle p, H, \mathbf{reconfigure}; st, \sigma \rangle \longrightarrow \langle p', H', st, \sigma \rangle}$$

The next statement st and current state σ are classical in structural operational semantics. The current product (code) p and current heap H are needed to capture the meaning of the **reconfigure** statement, because both need to be modified during reconfiguration.

The work of Damiani et al. focuses specifically on the modification of the heap, but does not discuss modification of the product. It is true; if functional correctness is the only concern, the product can simply be generated from scratch each time, using existing techniques (Chapter 4). But if efficiency is a factor, this approach won't suffice. Nor is it feasible to store every possible configuration of the code; this number can be exponential in the number of features, and the approach can't ever scale to the more complex requirement of adaptation to unanticipated change [146]. This chapter explores some strategies that are potentially better:

Goal: *Formulate efficient strategies for reconfiguration of the running product in an ADM-based dynamic product line.*

The strategies explored in this chapter are based on the assumption that the difference between two subsequent running products will be small, relative to their individual size. So rather than build them from scratch every time, lightweight deltas can be derived at build-time, which can then be used to perform the proper transformation at runtime.

To reason about the correctness and efficiency of this approach to dynamic reconfiguration, we introduce a new operational semantics. We develop models to represent dynamic product lines in an abstract context and explore different

strategies for ‘running’ them. These models are defined in terms of *Mealy machines* [131]: finite state machines with an input symbol and an output symbol on each transition. In our case, the input symbol corresponds to a feature (or features) that has been turned on or off by external events and the output symbol corresponds to the delta that has to be applied to bring the current product up to date.

Besides being inherently more efficient than the naive approach, it affords us the opportunity to apply a particular kind of optimization. We assume that monitoring specific features for change has a certain cost—different for each feature—such as powering a sensor or polling a server. We introduce a *cost model* to express these costs, and *optimize* dynamic product lines by disregarding costly features until they become relevant. This is modeled by selectively removing transitions from the Mealy machine.

As this chapter does not offer any contributions regarding the heap or control flow issues of dynamic *software* product lines, it uses an alternative domain so as not to distract from the main contribution. A novel case-study is presented: the development of a mobile application for automated profile management, which is used as a running example throughout the chapter. By monitoring personal data such as time, location and schedule, a smartphone can automatically adjust its internal settings based on user defined rules, essentially operating as a dynamic product line. This allows us to explore strategies for reconfiguring running products without having to consider software-specific issues.

Goal: *Develop a profile management app for Android based on the dynamic product line strategies explored in this chapter.*

The rest of the chapter is structured as follows: Section 8.2 introduces the case study that will be used to illustrate the theory of the chapter. We then develop the operational semantics and Mealy machine model in Section 8.3—the main section of the chapter—and introduce the feature-based cost-model and optimization techniques in Section 8.4. Section 8.5 ties up loose ends by showing how the new operational semantics may be integrated with the classical programming language semantics and, finally, Sections 8.5 and 8.6 offer concluding remarks and discuss related work.

8.2 Automated Profile Management

The running example of this chapter is a mechanism for *automated profile management* on modern mobile devices. Smartphones and tablets, such as those based on Android [11], iOS [20] or Windows Phone [134], have access to a great variety of data concerning the current circumstances of their user: the current time and physical location, their scheduled appointments, which application is currently running, and so on. Privacy issues aside, that sort of information can be used to automatically adjust the devices settings based on user defined rules, such as: “when I’m at the movies, mute all sound” or “when my battery is running low, turn down screen brightness”.

This example addresses a real practical need. Smartphones are ubiquitous these days, and a number of applications already provide automated profile management. However, they suffer from various limitations, and are often so complex that one has to be a programmer to use them. I felt I could improve upon this.

The theory in this chapter has lead to the development of a new profile management application for Android [85]. Besides offering a great deal of power with an intuitive user experience, the application also serves to illustrate the versatility of ADM and the theory of its dynamic counterpart.

8.2.1 A Mobile Device

We start by introducing a simplified model of a mobile device. We are interested in two distinct aspects: *quantities* and *settings*. Quantities are what we want a device to monitor, such as ‘location’, ‘schedule’, ‘weather forecast’ and ‘battery level’. Settings are all aspects of the device that the user has control over, such as ‘volume’, ‘brightness’, ‘chat status’, ‘alarm’, and so on.

▷ **8.1. Definition (Device):** First, assume a universal set of identifiers \mathcal{I} and a universal set of values \mathcal{V} . A *device* is a triple $(ID_q, ID_s, \text{type})$ where:

- $ID_q \subseteq \mathcal{I}$ is a finite set of names for all *quantities* the device can monitor.
- $ID_s \subseteq \mathcal{I}$ is a finite set of names for all *settings* the device can modify.

The names of quantities and the names of settings are disjoint: $ID_q \cap ID_s = \emptyset$.

- The function $\text{type}: ID_q \cup ID_s \rightarrow \text{Pow}(\mathcal{V})$ maps a quantity or setting to its set of possible values. For example, we’d have $\text{type}(\text{battery level}) = \{0\%, \dots, 100\%\}$ and $\text{type}(\text{chat status}) = \{\text{available}, \text{busy}, \text{offline}\}$. \lrcorner

From this point on, for the rest of the chapter, assume that some device $DEV = (ID_q, ID_s, \text{type})$ is given.

We call a complete mapping of quantity values a device’s *environment*, and a collection of its current settings its *profile*. The environment is ‘read-only’. Through user-defined rules, specific environmental conditions can trigger a modification to the profile. We define the notion of profile explicitly:

▷ **8.2. Definition (Profile):** Define the set of *profiles* \mathcal{P}_{DEV} as a map of all of a device’s settings to values of the proper type:

$$\mathcal{P}_{DEV} \stackrel{\text{def}}{=} ID_s \rightarrow \mathcal{V}$$

such that $p(id) \in \text{type}(id)$ for all $p \in \mathcal{P}_{DEV}$ and all $id \in ID_s$. \lrcorner

▷ **8.3. Example:** The following is a profile $p_x \in \mathcal{P}_{DEV}$:

$$p_x = \left\{ \begin{array}{ll} \text{volume} & \mapsto 10, \\ \text{bluetooth} & \mapsto \text{on}, \\ \text{brightness} & \mapsto 3, \\ \text{foreground app} & \mapsto \text{clock}, \\ & \vdots \end{array} \right\}$$

Since the number of settings is usually quite large, we show only relevant ones. \lrcorner

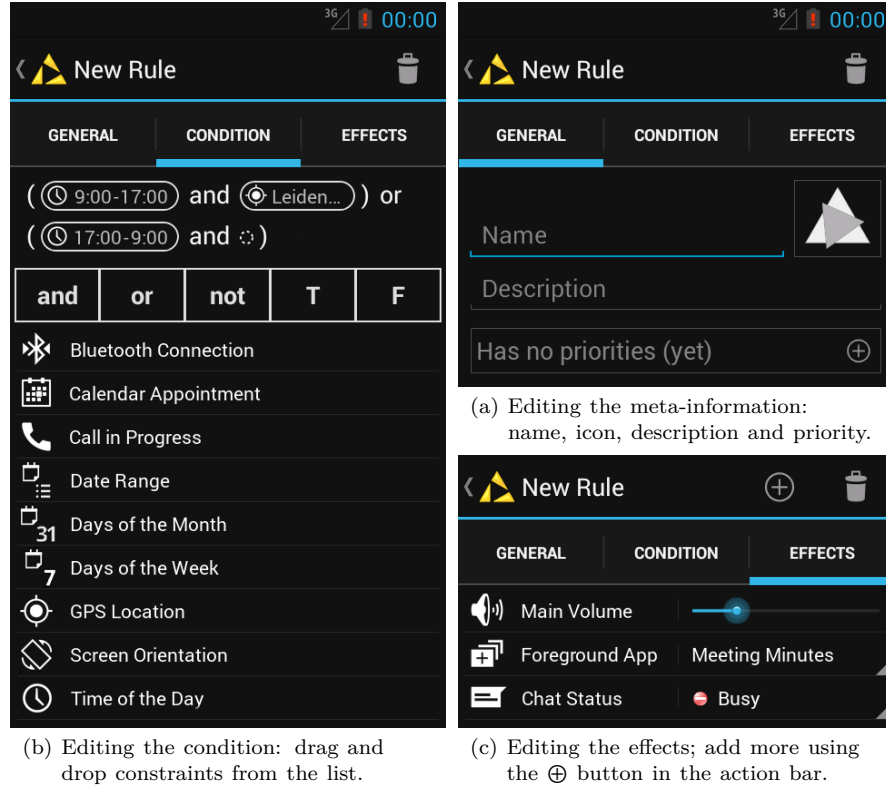


Figure 8.1: Screenshots of the Android interface. These controls are used for editing profile management rules.

Profiles play the rôle of products (Notation 2.9, page 36) in a device specific deltoid which will be defined shortly.

8.2.2 Rules

The idea behind the profile manager application is that the user manually inputs a set of *rules* using the graphical interface (Figure 8.1). A rule consists of an environmental *condition* and an *effect specification*, which contains new settings. A condition is entered as a formula containing constraints on specific quantities (Figure 8.1b). A constraint is formally defined as follows:

- ▷ 8.4. **Definition (Constraint):** A *constraint* is a dependent pair $\langle id, VAL \rangle$ where $id \in ID_q$ is the name of a quantity and $VAL \subseteq \text{type}(id)$ is the set of values to which id is constrained. The set of all possible constraints is denoted C_{DEV} .

Constraints play the rôle of features (Notation 4.2), since environmental constraints ultimately decide what the current profile should be.

An *effect specification* is formally similar to a profile (Definition 8.2), as both map settings to values. But profiles are total functions, whereas effect specifications are partial; they represent only the *changes* to a profile:

- ▷ **8.5. Definition (Effect Specification):** Define the set of *effect specifications* \mathcal{D}_{DEV} as a map of *some* settings to new values:

$$\mathcal{D}_{DEV} \stackrel{\text{def}}{=} ID_s \multimap \mathcal{V}$$

such that $d(id) \in \text{type}(id)$ for all $d \in \mathcal{D}_{DEV}$ and all $id \in ID_s$. Settings that are not to be modified are not mapped. \lrcorner

- ▷ **8.6. Example:** The following is an effect specification d_x :

$$d_x = \left\{ \begin{array}{ll} \text{volume} & \mapsto 5, \\ \text{foreground app} & \mapsto \text{calendar} \end{array} \right\}$$

Settings that are not mentioned are not mapped. \lrcorner

Effect specifications, of course, play the rôle of deltas (Notation 2.10, page 36). They form a monoid:

- ▷ **8.7. Definition (Profile Delta Monoid):** The *profile delta monoid* $(\mathcal{D}_{DEV}, \cdot, \varepsilon)$ has a composition operator $\cdot : \mathcal{D}_{DEV} \times \mathcal{D}_{DEV} \rightarrow \mathcal{D}_{DEV}$ defined as follows, for all deltas $x, y \in \mathcal{D}_{DEV}$ and identifiers $id \in \mathcal{ID}$:

$$(y \cdot x)(id) \stackrel{\text{def}}{=} \begin{cases} y(id) & \text{if } id \in \text{pre}(y) \\ x(id) & \text{if } id \in \text{pre}(x) \\ \perp & \text{otherwise} \end{cases}$$

The neutral profile delta $\varepsilon = \emptyset$ is the “everything undefined” function, mapping no identifiers at all. \lrcorner

The profile deltoid is functional (Definition 2.66, page 59). As such, we’ll simplify the type of the evaluation operator:

- ▷ **8.8. Definition (Profile Deltoid):** The *profile deltoid* for a device DEV is a deltoid $Dt_{DEV} \stackrel{\text{def}}{=} (\mathcal{P}_{DEV}, \mathcal{D}_{DEV}, \cdot, \varepsilon, \llbracket - \rrbracket)$, with product set \mathcal{P}_{DEV} from Definition 8.2, delta monoid $(\mathcal{D}_{DEV}, \cdot, \varepsilon)$ from Definitions 8.5 and 8.7 and semantic evaluation operator $\llbracket - \rrbracket : \mathcal{D}_{DEV} \rightarrow (\mathcal{P}_{DEV} \rightarrow \mathcal{P}_{DEV})$ defined as follows, for all profile deltas $d \in \mathcal{D}_{DEV}$, profiles $p \in \mathcal{P}_{DEV}$ and identifiers $id \in ID_s$:

$$\llbracket d \rrbracket(p)(id) \stackrel{\text{def}}{=} \begin{cases} d(id) & \text{if } id \in \text{pre}(d) \\ p(id) & \text{otherwise} \end{cases} \quad \lrcorner$$

- ▷ **8.9. Example:** For example, applying delta d_x from Example 8.6 to profile p_x from Example 8.3 results in the following profile:

$$\llbracket d_x \rrbracket(p_x) = \left\{ \begin{array}{ll} \text{volume} & \mapsto 5, \\ \text{bluetooth} & \mapsto \text{on}, \\ \text{brightness} & \mapsto 3, \\ \text{foreground app} & \mapsto \text{calendar}, \\ & \vdots \end{array} \right\} \quad \lrcorner$$

In conclusion, the domain of profile management gives rise to many deltoids — one for every device DEV .

Based on such a deltoid, a set of user-defined rules is defined as follows:

- ▷ **8.10. Definition (Rule-set):** A *rule-set* is a triple (D, \prec, γ) where $D \subseteq \mathcal{D}_{DEV}$ is a set of profile deltas representing the effects of the rules, $\prec \subseteq D \times D$ is a strict partial order representing rule-priority and the function $\gamma: D \rightarrow \text{Pow}(\text{Pow}(C_{DEV}))$ maps effect specifications to the condition under which they should be applied. \lrcorner

Rule-sets, then, play the rôle of annotated delta models (Definition 4.7, page 103). Intuitively, a rule is an instruction to the profile manager: “Whenever $\gamma(d)$ holds, ensure that the device is set to the values in d .” A condition $\gamma(d) \subseteq \text{Pow}(C_{DEV})$ is a set of sets of constraints, but should be thought of as a formula in disjunctive normal form, i.e., a disjunction of conjunctions of constraints (Figure 8.1b).

8.2.3 Defining Rules

I now present a typical scenario of a user entering some new rules, resulting in an example rule-set (D_x, \prec_x, γ_x) . We use these rules as a running example throughout the remainder of the chapter. We first specify each rule in an informal manner and follow up with their formalization.

The user enters the first rule:

- ▷ **8.11. Rule (At Work):** Whenever I am within 1 km of the Leiden University computer science building between 9:00 and 17:00, I want volume set to 5:

$$\begin{array}{ll} \gamma_x(x_x) & \stackrel{\text{def}}{=} \left\langle \begin{array}{l} \text{time, between 9:00 and 17:00} \\ \text{location, } < 1 \text{ km of } +52^\circ 10' 10'', \\ & +4^\circ 27' 24'' \end{array} \right\rangle \quad \left. \vphantom{\begin{array}{l} \text{time, between 9:00 and 17:00} \\ \text{location, } < 1 \text{ km of } +52^\circ 10' 10'', \\ & +4^\circ 27' 24'' \end{array}} \right\} \text{condition} \\ x_x & \stackrel{\text{def}}{=} \{ \text{volume} \mapsto 5 \} \quad \left. \vphantom{\text{volume} \mapsto 5} \right\} \text{effect} \quad \lrcorner \end{array}$$

The user then proceeds to enter the second rule:

- ▷ **8.12. Rule (In a Meeting):** During a scheduled meeting, I want the volume set to 0 and the ‘meeting minutes’ app brought to the foreground:

$$\begin{array}{ll} \gamma_x(y_x) & \stackrel{\text{def}}{=} \langle \text{meeting, } \{\text{true}\} \rangle \\ y_x & \stackrel{\text{def}}{=} \{ \text{volume} \mapsto 0, \\ & \quad \text{foreground app} \mapsto \text{‘meeting minutes’} \} \quad \lrcorner \end{array}$$

Both rules are entered through the interface shown in Figure 8.1. A name, description and icon can be associated with each rule, but those are not relevant to the formalism.

Upon entering the second rule, the user receives a warning from the application (Figure 8.2). The two rules have overlapping conditions—which means both can be true at the same time—but they disagree about the proper volume setting. So if the user ever attends a scheduled meeting at work during the designated working hours, the profile manager will not know whether the volume should be set to 0 or to 5. To break the tie, the user is given a choice:

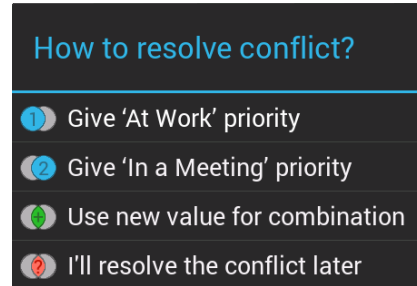


Figure 8.2: I found a conflict.

1. “always grant priority to the first rule (i.e., set the volume to 5)”,
2. “always grant priority to the second rule (i.e., set the volume to 0)”,
3. “use a third value, specifically for the combination $\gamma(x_x) \wedge \gamma(y_x)$ ”, or
4. “deactivate the rule-set for now; I’ll correct the problem later”.

In this case, the user chooses option 2 to give the second rule priority:

$$x_x \prec_x y_x$$

In a different situation, an alternative resolution might have been more appropriate. Perhaps a combination of two or more conditions requires specific consideration, and rather than give priority to either rule, a third alternative is required. Option 3 would automatically create a rule z_x such that $x_x, y_x \prec_x z_x$ with a preset default value for the volume setting to override the conflict. Option 4 gives the user the opportunity to manually correct the problem at leisure; perhaps by editing one or both rules. These options correspond roughly to Actions 3.9, 3.11 and 3.8 (pages 77 and 78).

8.2.4 Rule-sets as Product Line Implementations

Next, we create a dynamic product line from a given rule-set. ‘Profile features’, as noted earlier, are environmental constraints (Definition 8.4). We name the constraints of the running example above as follows:

- ▷ **8.13. Example (Profile Features):** The set of features $\mathcal{F}_x \subseteq C_{DEV}$ relevant to the example rule-set of Section 8.2.3 is $\{t, l, m\}$, where:

$$\begin{array}{lll} t & \stackrel{\text{def}}{=} & \langle \text{time, between 9:00 and 17:00} \rangle \\ l & \stackrel{\text{def}}{=} & \langle \text{location, } < 1 \text{ km of } +52^\circ 10' 10'', +4^\circ 27' 24'' \rangle \\ m & \stackrel{\text{def}}{=} & \langle \text{meeting, } \{\text{true}\} \rangle \quad \lrcorner \end{array}$$

A constraint $\langle id, VAL \rangle$ is essentially a predicate over quantity id , and represents a single feature; features are no longer just symbols. One could argue that reasoning in terms of ‘on-or-off’ features at all is impractical here, and a more flexible model should be used; perhaps a simple mapping between quantities and values. However, a feature model gives us a discrete and finite state-space, just expressive enough to distinguish the conditions provided by the user. A more realistic representation might well involve continuous and infinite domains, depending on the environmental quantities involved.

Mapping predicates to propositional symbols, as we are doing now, is a trick from SMT (SAT Modulo Theory) [143], allowing us to reason about them propositionally. This technique requires us to impose some restrictions on possible feature configurations, because some combinations of constraints will —by their very nature— exclude or imply others. For instance, $\langle \text{time, between 9:00 and 12:00} \rangle$ and $\langle \text{time, between 13:00 and 17:00} \rangle$ would never appear in the same feature configuration. But the presence of either would also ensure the presence of t (Example 8.13). We can encode such restrictions in a feature model. For a source-code based product line, a feature model is set up manually, based on which features ‘make sense’ together, and which conceptually exclude or include each other (Section 4.2). A ‘profile feature model’ is fixed for a given set of constraints, derived from their respective theories (the T in SMT). We give the following definition for interests sake, but we will not require these details further in the chapter:

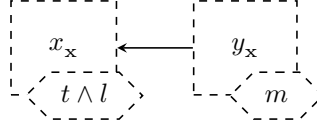


Figure 8.3: The Section 8.2.3 rule-set diagram.

8.14. Definition (Profile Feature Model): Given a set of profile features \mathcal{F} , the corresponding feature model Φ contains exactly all sets $F \subseteq \mathcal{F}$ such that

$$\begin{array}{c} \text{included constraints are satisfiable} \\ \hline \forall id \in ID_q: \quad \emptyset \subset \underbrace{\left(\text{type}(id) \cap \bigcap_{\substack{\langle id, VAL \rangle \\ \in F}} VAL \right)}_{\text{and implied constraints are not excluded.}} \not\subset \underbrace{\left(\bigcup_{\substack{\langle id, VAL \rangle \\ \in \mathcal{F} \setminus F}} VAL \right)}_{\text{and implied constraints are not excluded.}} \end{array} \quad \lrcorner$$

The only thing missing from our product line implementation now is a core product (Definition 4.9). To determine what a ‘core profile’ is, we first realize a simple truth: a smartphone application never has full control over the settings. The user can also manipulate them manually. So the core profile can basically be anything. There are a number of ways to capture this in the formalism, but we go for the simplest approach:

- ▷ **8.15. Definition (Manual Profile):** Introduce the value ‘manual’ $\in \mathcal{V}$, which is included in all types, i.e., we have ‘manual’ $\in \text{type}(id)$ for all settings $id \in ID_s$. Then define the *manual profile* $c \in \mathcal{P}_{DEV}$ as the constant that maps all identifiers to ‘manual’. ⌋
- ▷ **8.16. Definition (Rule-set Implementation):** A *rule-set implementation* is a product line implementation $(\Phi, c, D, \prec, \gamma)$, fully defined by Definitions 8.10, 8.14 and 8.15. ⌋

The implementation of the Section 8.2.3 ruleset is depicted in Figure 8.3.

8.2.5 Product Line Specifications

Now what of product line specifications (Section 4.4)? They are a valuable concept in this chapter too. Just as *static* product line implementations were validated against them in Chapter 4 (page 96), so will *dynamic* product line implementations be validated against them in the next section.

However, since users of the profile manager express their requirements directly in the form of delta models, the static notion of product line correctness (Section 4.4.2) loses some meaning. The valuation function is defined directly in terms of the implementation — $V(F) = \llbracket PLI \rrbracket(F)$ — making rule-set implementations correct by definition. But things become more interesting as we consider the correctness of dynamic product lines instead.

8.3 An Operational Semantics

For the remainder of the chapter we assume that some product line specification $PLS = (\Phi, V)$ is given.

In this section we work on defining the structure and semantics of ADM-based *dynamic product line implementations*. We start by stating the problem we need to solve. We then proceed step-by-step as we explore possible solutions, using the profile management application as an example.

8.3.1 The Problem

The problem is as follows: Say we are running a dynamic product line. It is currently ‘occupying’ feature configuration $F_e \in \Phi$, as imposed by the environment, and exhibiting the behavior of a product $p \in V(F_e)$. In other words, the product we are running is correct with regard to the *environmental feature configuration*. So far so good.

The environment could then impose a new feature configuration F_e' . This triggers our reconfiguration process, which is then responsible for updating the product p to some product $p' \in V(F_e')$. Our goal is to find the best possible strategy for doing so; preferably one that is (potentially) efficient, since we are now in a runtime setting, where time and space matter.

For the Section 8.2 profile manager, the environmental feature configuration would change whenever the truth value of a constraint is ‘flipped’ by an environmental quantity receiving a new value. For example, in the Section 8.2.3 rule-set, if we have $F_e = \emptyset$ and it becomes 9:00, we would switch to feature configuration $F_e' = \{t\}$.

8.3.2 An Operational Semantics

In order to reason about different strategies for updating the running product, we develop an operational semantics; one that might be ‘inserted’ at the point where an imperative program reaches the **reconfigure** statement described in the Introduction. However, we’ll develop the semantics in the abstract setting of ADM, so we will not track memory state or control flow.

We first need to choose a *configuration space*, as this choice will determine the kind of properties we can express about the dynamic system. If not memory and control flow, what *do* we need to track? It might make sense if our configurations were *feature configurations*. After all, a dynamic product line is all about moving from one feature configuration to another. But then we would not be able to express the property that the running product is continually correct with regard to that feature configuration. The configurations need, at least, to contain that product too. So a minimal configuration space would be as follows:

- 8.17. Definition (Minimal Configurations):** A *minimal configuration* is a pair $\langle F_e, p \rangle \in \Phi \times \mathcal{P}$ representing a dynamic product line state. F_e is the environmental feature configuration and p is the current product. \dashv

Side-note: The lack of a ► marker to the left of this definition indicates that it is not part of our final solution. We purposely explore a number of approaches in this section that turn out to be impractical. Doing so allows us to demonstrate useful concepts to build upon, and motivates the work that follows.

Do not let the term ‘configuration’ confuse you. The product p represents the actual running code (or, as the case may be, the actual running profile).

We make a distinction between *stable* and *unstable* configurations:

8.18. Definition (Configuration Stability): If a given configuration $\langle F_e, p \rangle$ has the property that $p \in V(F_e)$, we call that configuration *stable*. Otherwise we call it *unstable*. ┘

Recall that for the profile manager, this is the same as stating that a configuration is stable iff $F_e \llbracket PLI \rrbracket p$, because rule-set specifications are defined directly in terms of rule-set implementations (Section 8.2.5).

► **8.19. Example:** An example of a stable configuration would be

$$\langle \{t, l\}, \{ \text{volume} \mapsto 5, \dots \} \rangle$$

and an example of an unstable configuration would be

$$\langle \{t, l, m\}, \{ \text{volume} \mapsto 1, \dots \} \rangle$$

because all profiles in $V(\{t, l, m\})$ need volume set to 0. ┘

To define a transition relation, we introduce *inference rules* (Notation 1.15, page 21). They are important reference points in the chapter, so we distinguish them typographically by printing their names in SMALL-CAPS and placing them inside a solid box where they are defined.

We distinguish between two different ‘kinds’ of transitions. There are *environmental transitions* \xrightarrow{e} , in which the environment switches to a new feature configuration, and *local transitions* $\xrightarrow{\ell}$, in which the product is updated in an attempt to regain a correct state. The full transition relation \longrightarrow is defined as the smallest relation satisfying both an environmental inference rule and a local inference rule.

After an environmental transition, we will generally end up in an unstable configuration and will need to update the product:

8.20. Definition (Environmental Inference Rule):

ENV-MIN

$$\frac{p \in V(F_e)}{\langle F_e, p \rangle \xrightarrow{e} \langle F_e', p \rangle}$$

┘

Note that before we allow any environmental transition, we require that the current configuration is stable. This assumption simplifies the formalism, and is reasonable because we can expect to reach local stability in relatively short amounts of time. It allows us to think about the entire process as an alternation between two distinct phases. In the first phase the environmental feature configuration changes; this always takes one environmental transition.

In the second phase we update the product, using zero or more local transitions to achieve stability. We know that a local phase will not be interrupted by environmental transitions:

$$\overbrace{\langle F_e, p \rangle \xrightarrow{e} \langle F'_e, p \rangle}^{\text{environmental}} \underbrace{\xrightarrow{\ell^*} \langle F'_e, p' \rangle}_{\text{local}} \xrightarrow{e} \langle F''_e, p' \rangle \xrightarrow{\ell^*} \langle F''_e, p'' \rangle \xrightarrow{e} \dots$$

8.3.3 Correctness

So let us now restate our goal more formally: we need to find a local inference rule. We know that we have a good one if it leads to a stable configuration in finite time. We call such a local inference rule *correct* with regard to the product line specification. This is the dynamic counterpart of ‘static’ product line correctness (Definition 4.20).

We distinguish between two levels of correctness, as we do for static product lines (Section 4.4.2). *Partial correctness* means that *if* the local transition relation ever gets stuck, it will be in a stable configuration (so an environmental transition can take place). *Total correctness* means that a local transition is *guaranteed* to get stuck in a stable configuration within finite steps.

- **8.21. Definition (Partial Correctness):** A given local inference rule is *partially correct* iff, for all configurations $\langle cn \rangle$, we have:

$$\langle cn \rangle \not\xrightarrow{\ell} \quad \implies \quad \langle cn \rangle \text{ is stable} \quad \lrcorner$$

We could also state Definition 8.21 as follows:

“a local inference rule is partially correct iff the transition relation $\xrightarrow{e} \cup \xrightarrow{\ell}$ never gets stuck”

but the current formulation is closer to the traditional meaning of partial correctness, and it allows us to refrain from referring to ENV-MIN.

- **8.22. Definition (Total Correctness):** A local inference rule is *totally correct* iff it is partially correct and there is no infinite local transition path (Definition 1.50, page 28):

$$\langle cn \rangle \not\xrightarrow{\ell}^\infty \quad \lrcorner$$

8.3.4 A Local Inference Rule: LOC-PRD

We now try to find a correct local inference rule. Let us first get an obvious (but naive) idea out of the way. We could use the same process we used to generate a product statically. We would assume that a static product line implementation *PLI* is given (Definition 4.10), totally correct with respect to *PLS* (Definition 4.20), and define a local inference rule as follows:

8.23. Definition (Local Inference Rule):

LOC-PRD

$$\frac{\overbrace{p \notin V(F_e)}^{\text{a}} \quad \overbrace{F_e \llbracket PLI \rrbracket p'}^{\text{b}}}{\langle F_e, p \rangle \xrightarrow{\ell} \langle F_e, p' \rangle} \quad \lrcorner$$

This local transition can take place (a) from any unstable configuration (b) to a configuration with a product p' built from scratch to correspond with feature configuration F_e . We can prove that this rule is totally correct (Definitions 8.21 and 8.22):

8.24. Theorem: The LOC-PRD rule is totally correct.

Proof: First, assume that a given configuration $\langle F_e, p \rangle$ is locally stuck. This means we have the negation of LOC-PRD's premise: $p \in V(F_e)$ or $\exists p': F_e \llbracket PLI \rrbracket p'$. The latter is untrue by our static correctness assumption. By the former, all configurations that are locally stuck must also be stable. This gives us partial correctness.

If it is not stable, and therefore not stuck, we have $\langle F_e, p \rangle \xrightarrow{\ell} \langle F_e, p' \rangle$ with $F_e \llbracket PLI \rrbracket p'$ by LOC-PRD. By the assumed correctness of *PLI* we can conclude $p' \in \llbracket PLI \rrbracket(F_e) \subseteq V(F_e)$. Since we use at most one transition to gain this stability, there is clearly no infinite local transition path, which gives us total correctness. \square

But generating a new product on the fly this way will turn out to be too inefficient for non-trivial product lines. Recall that we need local transitions to be fast. Storing all possible products in memory beforehand and then dynamically switching to the correct one is also infeasible. In general the number of products will be exponential in the number of features. If we want to model industrial-scale dynamic product lines, we need to do better.

8.3.5 Difference-based Configurations

An alternative approach is to take the current product and transform it into a new one incrementally. This may be a lot more efficient, since we would be reusing the parts of the product that do not need to change. A transformation of a product is, of course, a delta. But how do we decide which delta to apply at every change? We are currently rather limited by the information in our configuration tuples. If we want to do this we need to add some bookkeeping. In particular, if we want to keep track of what changed in the environment, we'll need to store a *local feature configuration*. We can compare it to the environmental feature configuration and use their 'difference' to determine the delta or deltas that need to be applied (Figure 8.4):

- **8.25. Definition (Difference-based Configurations):** *Difference-based configurations* are triples $\langle F_e, F_\ell, p \rangle \in \Phi \times \Phi \times \mathcal{P}$ with an environmental feature configuration F_e , a *local feature configuration* F_ℓ and a current product p . \lrcorner

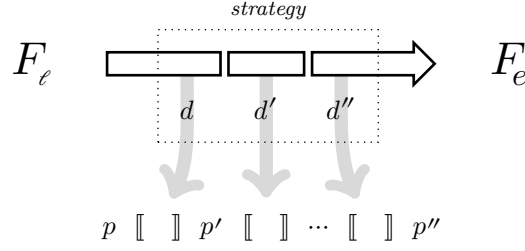


Figure 8.4: Illustrating the rôle of the local feature configuration in $\langle F_e, F_\ell, p \rangle$. If $p \notin V(F_e)$, one or more deltas are derived from the difference between F_e and F_ℓ . These deltas are then used to transform the running product into valid product p'' . This is presumably faster than building p'' from scratch. How to derive the proper sequence of deltas depends on our *strategy*.

Damiani et al. [63, 64] have a similar concept. They call the local feature configuration `CurrentConfiguration` and the environmental feature configuration `NextConfiguration`.

The difference between two feature configurations is their set-theoretic *symmetric difference* (Definition 1.1). We use it to measure their distance and how far we have progressed from one to the other:

- **8.26. Example (Symmetric Difference):** Take, as an example, two feature configurations $\{t\}, \{m\} \in \Phi_x$. If we currently occupy $F_\ell = \{t\}$ and intend to reach $F_e = \{m\}$, we need to ‘remove’ feature t and ‘add’ feature m in the implementation of the current product. We represent this required work with $F_\ell \ominus F_e = \{t, m\}$. (We preserve the distinction between adding and removing by remembering the context of the operation.)

Conversely, if we start at $F_\ell = \{t\}$ and perform the work described by $F_\Delta = \{t, l\}$, we reach the state $F_\ell \ominus F_\Delta = \{l\}$. The symmetric difference operator has the interesting property that $F \ominus (F \ominus G) = G$, so we can use it for both types of operation. \lrcorner

Before we try to translate such a difference to a delta, we need to restate Definitions 8.18 and 8.20 to work with our new configurations. They are trivial changes, just adding and disregarding the additional element:

- **8.27. Definition (Configuration Stability):** If a given configuration $\langle F_e, F_\ell, p \rangle$ has the property that $p \in V(F_e)$ then we call that configuration *stable*. Otherwise we call it *unstable*. \lrcorner
- **8.28. Definition (Environmental Inference Rule):** ENV-DIFF

$$\frac{p \in V(F_e)}{\langle F_e, F_\ell, p \rangle \xrightarrow{e} \langle F_e', F_\ell, p \rangle} \quad \lrcorner$$

When it comes to stability and environmental transitions, we do not care about our new bookkeeping element; it is left alone. Note that we do not need to redefine Definitions 8.21 and 8.22 because they were presented in a sufficiently general manner.

8.3.6 Dynamic Product Lines as Mealy Machines

We now need to decide, given a feature configuration difference, how to derive the delta or deltas that can transform the current product into a valid target product. We call this a *strategy* (Figure 8.4). We describe different strategies with a new model based on *Mealy machines*. Please have a look at Definition 1.52 (page 28) for the formal definition. This representation will be quite useful for describing as well as visualizing different strategies for running dynamic product lines.

It is worth noting that this type of graph (Figure 1.6, page 29) offers a completely different view of a product line than a delta diagram does. A delta diagram can be said to represent the design space, whereas Mealy machines represent the dynamic state-space.

We define local transitions of the operational semantics in terms of Mealy machine transitions. For this we use the following syntax for the Mealy-machine transition relation, so we need not use T and O directly:

- **8.29. Notation (Mealy Machine Transition Relation):** A Mealy machine tuple $(S, \Sigma, \Lambda, T, O)$ induces a quaternary *Mealy machine transition relation* $\xrightarrow{i/o} \subseteq S \times \Sigma \times \Lambda \times S$ as follows. For all states $s, s' \in S$, input symbols $i \in \Sigma$ and output symbols $o \in \Lambda$:

$$s \xrightarrow{i/o} s' \quad \stackrel{\text{def}}{\iff} \quad T(s, i) = s' \wedge O(s, i) = o \quad \lrcorner$$

We can now define a local inference rule in terms of the $\xrightarrow{i/o}$ relation of a Mealy machine. But how do we define such a *DPL Mealy machine*? The tuple has five ingredients. The first three are simple enough:

- $S = \Phi$: The states of our machine are feature configurations. The local feature configuration F_ℓ is the *current state*. F_e is the *target state*. In a diagram we annotate these two states as in Figure 8.5.
- $\Sigma = \text{Pow}(\mathcal{F})$: The input we want to process at each transition is the difference—or part of the difference—between feature configurations F_ℓ and F_e . We denote such a difference by $F_\Delta \subseteq \mathcal{F}$.
- $\Lambda = \mathcal{D}$: As we move F_ℓ towards F_e in the machine, we'd like to get, as output, a light-weight delta (or deltas) to update the product.

The output function O requires some thought. Given a current product and feature difference, which delta do we use to update the product?

Since a delta d can be non-deterministic, its application to the current product p could result in more than one possible next product $p' \in \llbracket d \rrbracket(p)$. We at least need all of those to be correct: $\llbracket d \rrbracket(p) \subseteq V(F_e)$. But at build time we can't be sure what product p is. It could itself be any of the products generated by the previous transition, and so on. In other words, if we want to reason locally, we'll need to choose an invariant on our configurations $\langle F_e, F_\ell, p \rangle$ to give us more information about our current product. The invariant will need to hold in the initial configuration and every transition will need to maintain it. We will use the *local consistency* invariant:

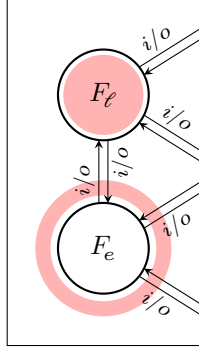


Figure 8.5: Local and environmental marking of states in a DPL Mealy machine diagram. Local feature configuration F_ℓ is shaded and environmental feature configuration F_e has a ring around it. During a local transition, imagine our general strategy as the local state trying to reach the ring by taking any direct path that leads in the right direction.

- **8.30. Definition (Local Consistency):** A difference-based configuration $\langle F_e, F_\ell, p \rangle$ is *locally consistent* iff $p \in V(F_\ell)$. \lrcorner

For specific systems there may be stronger invariants that are more appropriate, but in general, local consistency is the best we can do. We know now that every configuration has a product consistent with F_ℓ and we can use that to choose the right delta. We need to derive one that transforms *any* product from a locally consistent configuration to a correct target product, i.e., we need an effective procedure for delta derivation (Section 2.4.3):

- **8.31. Definition (Derived Delta Operator):** A *derived delta operator* is a binary operator $\mapsto: \text{Pow}(\mathcal{P}) \times \text{Pow}(\mathcal{P}) \rightarrow \mathcal{D}$ that returns a derived delta for all product sets $P, P' \subseteq \mathcal{P}$:

$$(P \mapsto P') \in (P \Rightarrow_{\text{tot}} P')$$

where \Rightarrow_{tot} specifies a set of derived deltas (Definition 2.31). \lrcorner

From now on we assume that such a procedure is implemented for the given deltoid. We can define one for the profile deltoid as follows:

- **8.32. Definition (Derived Profile Delta Operator):** The *derived profile delta operator* $\mapsto_{DEV}: \text{Pow}(\mathcal{P}_{DEV}) \times \text{Pow}(\mathcal{P}_{DEV}) \rightarrow \mathcal{D}_{DEV}$ takes a finite set of source profiles and a nonempty finite set of target profiles and produces a profile delta that transforms any profile from the source set into an arbitrary profile from the target set. For all product sets $P \subseteq \mathcal{P}_{DEV}$, products $p' \in \mathcal{P}_{DEV}$ and identifiers $id \in ID_s$:

$$(P \mapsto_{DEV} \{p', \dots\})(id) \stackrel{\text{def}}{=} \begin{cases} p'(id) & \text{if } \exists p \in P: p(id) \neq p'(id) \\ \perp & \text{otherwise} \end{cases}$$

The resulting delta remains undefined for the settings on which P and p' agree, and favors p' for the rest. The product p' is chosen arbitrarily from the right-hand operand. \lrcorner

Generally, given such a procedure, we can define an output function O that produces an appropriately derived delta $O(F_\ell, F_\Delta) = V(F_\ell) \mapsto V(T(F_\ell, F_\Delta))$. We assume that this is done at build-time for all relevant feature differences, so they can simply be looked up at run-time.

As for our final ingredient: the definition of the transition function T is what determines our further strategy. The output function O will remain fixed, though its preimage will adapt to correspond with T . So we can now define the concept of a *DPL Mealy machine* parametrized on T :

- **8.33. Definition (DPL Mealy Machine):** Given a *transition function* $T: \Phi \times \text{Pow}(\mathcal{F}) \rightarrow \Phi$, we define the corresponding *DPL Mealy machine*

$$\text{MM}(T) \stackrel{\text{def}}{=} (\Phi, \text{Pow}(\mathcal{F}), \mathcal{D}, T, O)$$

where $O: \Phi \times \text{Pow}(\mathcal{F}) \rightarrow \mathcal{D}$ is defined as

$$O(F_\ell, F_\Delta) \stackrel{\text{def}}{=} \begin{cases} V(F_\ell) \mapsto V(T(F_\ell, F_\Delta)) & \text{if } (F_\ell, F_\Delta) \in \text{pre}(T) \\ \perp & \text{otherwise} \end{cases} \quad \lrcorner$$

And finally, we define a local inference rule in terms of a DPL Mealy machine, also parametrized on T :

- **8.34. Definition (Local Inference Rule):**

$$\frac{\overbrace{p \notin V(F_e)}^{\text{a}} \quad \overbrace{\emptyset \subset F_\Delta \subseteq F_\ell \ominus F_e}^{\text{b}} \quad \overbrace{F_\ell \xrightarrow{F_\Delta/d} F_\ell'}^{\text{c}} \quad \overbrace{p \llbracket d \rrbracket p'}^{\text{d}}}{\langle F_e, F_\ell, p \rangle \xrightarrow{\ell} \langle F_e, F_\ell', p' \rangle} \quad \boxed{\text{LOC-DIFF}(T)} \quad \lrcorner$$

with $\xrightarrow{i/o}$ from Mealy machine $\text{MM}(T)$ (Definition 8.33). \lrcorner

(a) Starting from an unstable configuration $\langle F_e, F_\ell, p \rangle$, (b) some nonempty subset of $F_\ell \ominus F_e$ is chosen as input symbol F_Δ . (c) Given that input symbol from current state F_ℓ , the DPL Mealy machine reaches a state $F_\ell' \in \Phi$ and generates a delta $d \in \mathcal{D}$ as output symbol. (d) This delta can transform product p into product p' , forming a possible next configuration $\langle F_e, F_\ell', p' \rangle$.

At this point a general recapitulation is in order: The behavior of a dynamic product line is modeled by a configuration space and a transition relation on that space (Notations 1.48 and 1.49). We define the transition relation based on environmental inference rule ENV-DIFF and local inference rule $\text{LOC-DIFF}(T)$ (Definitions 8.28 and 8.34). The local inference rule is based on a Mealy machine, parametrized on its transition function T (Definition 8.33). The remaining sections will be spent trying to find the optimal function T .

Before we explore the first candidate, we prove a number of useful properties about the local inference rule, which will help us in the upcoming correctness proofs. First, we prove that it maintains local consistency, purely by our fixed choice of output function:

- **8.35. Lemma (Local Consistency by LOC-DIFF):** Given any transition function T , the transition relation defined by $\text{LOC-DIFF}(T)$ maintains the local consistency invariant (Definition 8.30).

Proof: Assume that $\langle F_e, F_\ell, p \rangle$ is locally consistent, so $p \in V(F_\ell)$. Take any transition $\langle F_e, F_\ell, p \rangle \xrightarrow{\ell} \langle F_e, F_\ell', p' \rangle$. We know from Definition 8.33 that $p \llbracket d \rrbracket p'$ for some delta $d \in V(F_\ell) \Rightarrow_{\text{tot}} V(F_\ell')$. So by our assumption $p \in V(F_\ell)$ and by Definition 8.31, we have $p' \in V(F_\ell')$. That means $\langle F_e, F_\ell', p' \rangle$ is also locally consistent. \square

Next, there is a certain property all our choices of transition function *should* exhibit. They will all take a *direct path* from the local to the environmental feature configurations. This means that the difference between the two sets strictly decreases in size from one configuration to the next:

- **8.36. Definition (Direct Path):** A transition function $T: \Phi \times \text{Pow}(\mathcal{F}) \rightarrow \Phi$ takes a *direct path* iff, for all configurations $\langle F_e, F_\ell, p \rangle$ and a transition relation $\xrightarrow{\ell}$ defined by $\text{LOC-DIFF}(T)$, we have:

$$\langle F_e, F_\ell, p \rangle \xrightarrow{\ell} \langle F_e, F_\ell', p' \rangle \implies F_e \ominus F_\ell \supset F_e \ominus F_\ell' \quad \lrcorner$$

This will help us prove total correctness by use of the following lemma:

- **8.37. Lemma (Direct Path Convergence):** Any transition function that takes a direct path (Definition 8.36), allows no infinite local transition path:

$$\nexists \langle F_e, F_\ell, p \rangle \xrightarrow{\ell}^\infty$$

Proof: By wellfoundedness of \subset . The finite set $F_e \ominus F_\ell$ can only shrink until it is empty. \square

So basically, for any partially correct local inference rule that takes a direct path, we get total correctness for free.

8.3.7 A Local Inference Rule: $\text{LOC-DIFF}(T_f)$

The first obvious strategy is to take the difference between the local and environmental feature configurations $F_\Delta = F_e \ominus F_\ell$ directly as input symbol for the Mealy machine. So, if $F_\ell = \{t\}$ and $F_e = \{m\}$ we take a single transition with input symbol $F_\Delta = \{t, m\}$:

- 8.38. Definition:** A local inference rule $\text{LOC-DIFF}'(T)$ is the same as rule $\text{LOC-DIFF}(T)$ (Definition 8.34), but with the additional premise that $F_\Delta = F_e \ominus F_\ell$. \lrcorner

This premise is only temporary, because the strategy will turn out to be impractical. Nonetheless, a brief exploration of it will be instructive. We can define the transition function as follows:

- 8.39. Definition (Full Difference Transition Function):** Define the *full difference transition function* $T_f: \Phi \times \text{Pow}(\mathcal{F}) \rightarrow \Phi$ as follows, for all $F_\ell \in \Phi$ and $F_\Delta \subseteq \mathcal{F}$:

$$\begin{aligned} T_f(F_\ell, F_\Delta) &\stackrel{\text{def}}{=} F_\ell \ominus F_\Delta \\ \text{pre}(T_f) &\stackrel{\text{def}}{=} \{ (F_\ell, F_\Delta) \mid F_\ell \ominus F_\Delta \in \Phi \} \end{aligned}$$

J

There are two almost separate aspects to defining a transition function: defining its output and defining its preimage. The output specified above makes it clear that the new local transition rule $\boxed{\text{LOC-DIFF}'(\text{T}_f)}$ (Definitions 8.38 and 8.42) moves from F_ℓ to F_e in a single step after every environment change. We chose the preimage so that the output is guaranteed to be a valid feature configuration, rather than some arbitrary feature set.

8.40. Lemma (Direct Path by $\text{LOC-DIFF}'(\text{T}_f)$): The inference rule $\text{LOC-DIFF}'(\text{T}_f)$ takes a direct path (Definition 8.36).

Proof: The difference F_Δ provided as an input symbol is a nonempty set (Definition 8.34b). By Definition 8.39 the new local feature configuration is $F_\ell' = F_\ell \ominus F_\Delta = F_\ell \ominus (F_\ell \ominus F_e) = (F_\ell \ominus F_\ell) \ominus F_e = F_e$. In short, $F_\ell' = F_e$, so the new difference $F_e \ominus F_\ell' = \emptyset$ is empty, making it strictly smaller than F_Δ . \square

With this results we prove total correctness as defined by Definitions 8.21 and 8.22:

8.41. Theorem: The rule $\text{LOC-DIFF}'(\text{T}_f)$ is totally correct.

Proof: Assume that a given configuration $\langle F_e, F_\ell, p \rangle$ is stuck. Since $\text{LOC-DIFF}'(\text{T}_f)$ (Definitions 8.38 and 8.39) is our only local inference rule, we have the negation of one of its premises. So we have one of the following:

- a. The configuration is already stable: $p \in V(F_e)$,
- b. there is no F_Δ , because $F_\ell = F_e$,
- c. the T_f function accepts none: $\nexists \emptyset \subset F_\Delta \subseteq F_\ell \ominus F_e: (F_\ell, F_\Delta) \in \text{pre}(\text{T}_f)$, or
- d. the generated delta does not accept the current product: $\llbracket d \rrbracket(p) = \emptyset$.

By Definition 8.39 it cannot be (c), and by Definition 8.33 it cannot be (d). So by the process of elimination we have $p \in V(F_e) \vee F_\ell = F_e$. The former would give us our result directly. Given the latter, we have $p \in V(F_\ell)$ by local consistency (Lemma 8.35), and therefore $p \in V(F_e)$. So we know that when $\text{LOC-DIFF}'(\text{T}_f)$ is stuck on a configuration, that configuration must be stable, giving us partial correctness.

We have total correctness by the direct path property (Lemmas 8.37 and 8.40). \square

This is a pattern of proof we will use more often:

- Prove that a stuck configuration is stable by invoking local consistency and the negation of one of the premises of Definition 8.34. This gives us partial correctness.
- Prove that the transition function takes a direct path (Definition 8.36), giving us total correctness.

A Mealy machine diagram for $\text{LOC-DIFF}(\text{T}_f)$ would be quite unreadable. It has an excessive number of transitions: $|\text{pre}(\text{T}_f)| = |\Phi|^2 - |\Phi| = 2^{2|\mathcal{F}|} - 2^{|\mathcal{F}|}$, quadratic in the number of feature configurations, so exponential in the number of features. This is not surprising, given that the input alphabet we chose is $\text{Pow}(\mathcal{F})$. This also means that we would have to store a lot of deltas. Too many.

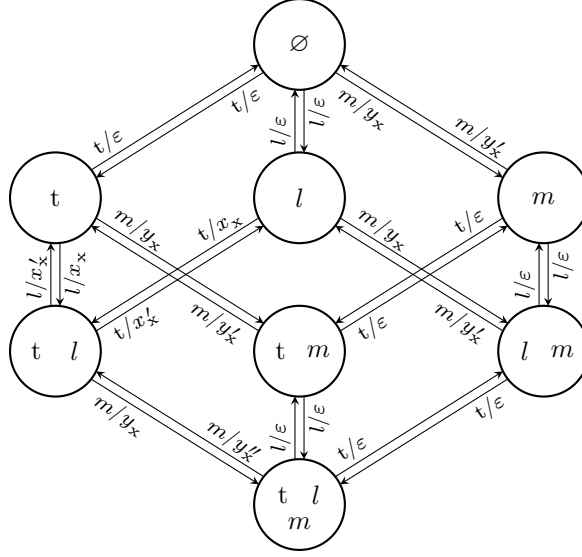


Figure 8.6: The DPL Mealy machine $\text{MM}(\text{T}_s)$ for the Section 8.2.3 example.

The key insight here is that we do not need to reach the target product in a single transition. We can use multiple local transitions to get there. And with the concepts introduced up to this point, it will be relatively simple to define a new rule for that.

8.3.8 A Local Inference Rule: **LOC-DIFF-ND**(T_s)

We now reduce the number of Mealy-machine transitions by taking one local transition per *feature* rather than one per *feature-set*. This is at the cost of using multiple local transitions during a single phase if necessary. It will bring the number of transitions in the Mealy machine down from $(2^{|\mathcal{F}|} - 2^{|\mathcal{F}|})$ to $(|\mathcal{F}| \times 2^{|\mathcal{F}|})$, a significant difference in practice, even though it is still exponential. So for the profile manager, if $F_\ell = \{t\}$ and $F_e = \{m\}$ we take one local transition for ‘it became 17:00’ and one for ‘a meeting has started’, even if both occur simultaneously.

To maintain consistency, we do not change the input alphabet to \mathcal{F} , but will simply use singleton sets. The transition function is otherwise similar to T_f :

8.42. Definition (Singleton Transition Function): Define the *singleton transition function* $\text{T}_s: \Phi \times \text{Pow}(\mathcal{F}) \rightarrow \Phi$ as follows, for all $F_\ell \in \Phi$ and $f_\Delta \in \mathcal{F}$:

$$\begin{aligned} \text{T}_s(F_\ell, \{f_\Delta\}) &\stackrel{\text{def}}{=} F_\ell \ominus \{f_\Delta\} \\ \text{pre}(\text{T}_s) &\stackrel{\text{def}}{=} \{(F_\ell, \{f_\Delta\}) \mid F_\ell \ominus \{f_\Delta\} \in \Phi\} \quad \lrcorner \end{aligned}$$

The DPL Mealy machine $\text{MM}(\text{T}_s)$ (Definitions 8.33 and 8.42) for our running example is shown in Figure 8.6. Observe that the features t, l, m each represent one ‘dimension’ in the diagram. For readability we omitted the ‘set braces’,

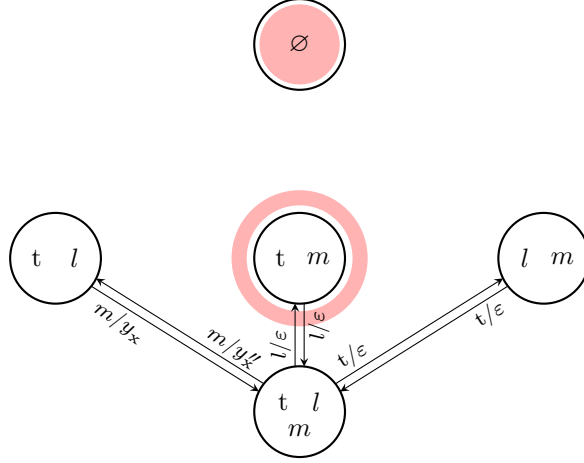


Figure 8.7: A DPL Mealy machine $MM(T_s)$ based on an some unrestricted feature model. Note that $F_e = \{t, m\}$ cannot be reached from $F_\ell = \emptyset$.

since we only use singleton input sets for T_s . Three known deltas are used in this machine: ε , x_x and y_x (Section 8.2.3). The new deltas x'_x , y'_x and y''_x are:

$$\begin{aligned} x'_x &= \{ \text{'volume'} \mapsto \text{'manual'} \}, \\ y'_x &= \left\{ \begin{array}{l} \text{'volume'} \mapsto \text{'manual'}, \\ \text{'foreground app'} \mapsto \text{'manual'} \end{array} \right\}, \\ y''_x &= \left\{ \begin{array}{l} \text{'volume'} \mapsto 5, \\ \text{'foreground app'} \mapsto \text{'manual'} \end{array} \right\}. \end{aligned}$$

They reverse the effects of their counterparts. Since semantic profile deltas are not surjective—they just overwrite any value that was there before—they do not have an inverse without taking their context into account. That's why we need both y'_x and y''_x .

Let's investigate local inference rule $\boxed{\text{LOC-DIFF}(T_s)}$ (Definitions 8.34 and 8.42). As it turns out, there is a problem with it: it is not correct for arbitrary feature models. Now that we are taking small 'feature-sized' steps through the Mealy machine, it is no longer certain all states are reachable. If $\Phi \neq \text{Pow}(\mathcal{F})$ we would be missing some intermediate states we need to land on. We cannot allow Mealy machines such as the one in Figure 8.7, for example. So for this strategy we need to restrict the feature model to $\Phi = \text{Pow}(\mathcal{F})$. The transition function *does* take direct path:

8.43. Lemma (Direct Path by $\text{LOC-DIFF}(T_s)$): The inference rule $\text{LOC-DIFF}(T_s)$ takes a direct path (Definition 8.36).

Proof: The difference $\{f_\Delta\}$ provided as an input symbol is obviously a nonempty set. By Definition 8.42 we have $F_{\ell'} = F_\ell \ominus \{f_\Delta\}$, so the new difference is $F_e \ominus F_{\ell'} = F_e \ominus (F_\ell \ominus \{f_\Delta\}) = (F_e \ominus F_\ell) \ominus \{f_\Delta\}$. Since $\{f_\Delta\} \subseteq F_e \ominus F_\ell$, we have $(F_e \ominus F_\ell) \ominus \{f_\Delta\} = (F_e \ominus F_\ell) \setminus \{f_\Delta\}$, strictly smaller than $F_e \ominus F_\ell$. \square

And this leads to total correctness, much as before, as long as we restrict the feature model as described above:

8.44. Theorem: For feature model $\Phi = \text{Pow}(\mathcal{F})$, the rule $\text{LOC-DIFF}(\text{T}_s)$ is totally correct.

Proof: This proof proceeds much as the one for Theorem 8.41, so we leave some simple steps out. Assume that a given configuration $\langle F_e, F_\ell, p \rangle$ is stuck. By negating the premises, we have either of the following:

- a. The configuration is already stable: $p \in V(F_e)$,
- b. there is no F_Δ because $F_\ell = F_e$, or
- c. the T_s function accepts none: $\nexists f_\Delta \in F_\ell \ominus F_e: (F_\ell, \{f_\Delta\}) \in \text{pre}(\text{T}_s)$.

It cannot be (c), because even though T accepts only singleton sets, we are assuming a complete feature model Φ , so all singleton sets are valid input symbols (Definition 8.42). We therefore have $p \in V(F_e) \vee F_\ell = F_e$, giving us partial correctness as before.

And as before, we get total correctness from Lemmas 8.37 and 8.43. \square

We also know that any local transition-path starting from $\langle F_e, F_\ell, p \rangle$ will always reach a configuration $\langle F_e, F_e, p' \rangle$ in exactly $|F_e \ominus F_\ell|$ steps.

8.3.9 A Local Inference Rule: $\text{LOC-DIFF}(\text{T}_m)$

Our next goal is to drop the restriction on the feature model imposed in Section 8.3.8. So we want to go from F_ℓ to F_e , but somewhere on an otherwise direct path we are missing an intermediate state we need to pass through to reach the target state. We are going to add extra transitions to solve this problem — just enough to regain reachability. Those extra transitions will have $|F_e \ominus F_\ell| > 1$.

We are entitled to ask, however: can we still keep using singleton sets as *input symbols*? As long as a single feature unambiguously determines the right direction, we could always define T to jump any additional distance required, i.e., so that $F_\ell \xrightarrow{\{f_\Delta\}/d} F_\ell'$ with $\{f_\Delta\} \subset F_\ell \ominus F_\ell'$. But the answer is no. There are situations where a single feature cannot unambiguously determine a transition. Take, for example, a feature model $\Phi'_x = \{\emptyset, \{t, l\}, \{t, m\}, \{l, m\}, \{t, l, m\}\}$ with the Mealy machine from Figure 8.7. With $F_\ell = \emptyset$ and $F_e = \{t, m\}$, choosing either of $t, m \in (F_\ell \ominus F_e)$ as the sole input symbol would not be enough to determine the next state. We need the full information $\{t, m\}$ for that.

The following transition function has a preimage that is minimal, unique and preserves reachability:

► **8.45. Definition (Minimal Transition Function):** Define the *minimal transition function* $\text{T}_m: \Phi \times \text{Pow}(\mathcal{F}) \rightarrow \Phi$ as follows, for all $F_\ell \in \Phi$ and $F_\Delta \subseteq \mathcal{F}$:

$$\begin{aligned} \text{T}_m(F_\ell, F_\Delta) &\stackrel{\text{def}}{=} F_\ell \ominus F_\Delta \\ \text{pre}(\text{T}_m) &\stackrel{\text{def}}{=} \left\{ (F_\ell, F_\Delta) \mid \begin{array}{l} F_\ell \ominus F_\Delta \in \Phi \quad \wedge \\ \nexists \emptyset \subset F_\Delta' \subset F_\Delta: F_\ell \ominus F_\Delta' \in \Phi \end{array} \right\} \end{aligned}$$

J

The additional restriction, when compared to Definition 8.39, ensures that the only transitions that are preserved are those that the feature model does not allow taking in smaller steps. If there *are* smaller steps, those will become transitions themselves. So Definition 8.45 covers the ‘one feature difference’ transitions we had before, as well as new transitions required to bridge larger gaps.

We lift the restrictions from our input symbols as well as our feature model by using local inference ruleset $\boxed{\text{LOC-DIFF}(\mathbf{T}_m)}$ (Definitions 8.33 and 8.45). Now for the necessary correctness proofs:

- **8.46. Lemma (Direct Path by LOC-DIFF(\mathbf{T}_m)):** The inference rule LOC-DIFF(\mathbf{T}_m) takes a direct path (Definition 8.36).

Proof: Almost identical to the Lemma 8.43 proof; just replace $\{f_\Delta\}$ with F_Δ . □

- **8.47. Theorem:** The LOC-DIFF-ND(MM(\mathbf{T}_m)) rule is totally correct.

Proof: Again, this proof is quite similar to those for Theorems 8.41 and 8.44. To summarize, if \mathbf{T}_m can always accept some non-empty input symbol $F_\Delta \subseteq F_\ell \ominus F_e$ (Definition 8.34c), then a stuck state is also stable (Definition 8.34a and 8.34b), giving us partial correctness. We then get total correctness from the direct path property as before.

In this case, Definition 8.45 only excludes an input symbol if an equivalent string of smaller ones is also available. So there is always at least one. This gives us our desired result. □

We also know that any local transition-path starting from $\langle F_e, F_\ell, p \rangle$ will always reach a configuration $\langle F_e, F_e, p' \rangle$ in at most $|F_e \ominus F_\ell|$ steps.

The Mealy machine from Figure 8.7 would now be constructed as in Figure 8.8. Note that \mathbf{T}_m gives us the necessary transitions to bridge the distance. If we want to get from $F_\ell = \emptyset$ to $F_e = \{t, l, m\}$, any of the three possible local transition paths will take us there properly:

$$\langle F_e, \emptyset, p \rangle \left\{ \begin{array}{ll} \xrightarrow{\ell} \langle F_e, \{t, l\}, \llbracket x_x \rrbracket(p) \rangle & \xrightarrow{\ell} \langle F_e, \{t, l, m\}, \llbracket y_x \cdot x_x \rrbracket(p) \rangle \\ \xrightarrow{\ell} \langle F_e, \{t, m\}, \llbracket y_x \rrbracket(p) \rangle & \xrightarrow{\ell} \langle F_e, \{t, l, m\}, \llbracket \varepsilon \cdot y_x \rrbracket(p) \rangle \\ \xrightarrow{\ell} \langle F_e, \{l, m\}, \llbracket y_x \rrbracket(p) \rangle & \xrightarrow{\ell} \langle F_e, \{t, l, m\}, \llbracket \varepsilon \cdot y_x \rrbracket(p) \rangle \end{array} \right.$$

Note that $y_x \cdot x_x = y_x = \varepsilon \cdot y_x$ (Section 8.2.3), so we reach the same result regardless.

8.4 Cost and Optimization

We now allow an unrestricted feature model and have reduced the number of transitions to a reasonable amount. In this section, we examine a technique for optimizing the Mealy machine even further, in a way that is particularly effective when the activity of ‘monitoring’ the environment for change carries with it a certain cost that needs to be minimized.

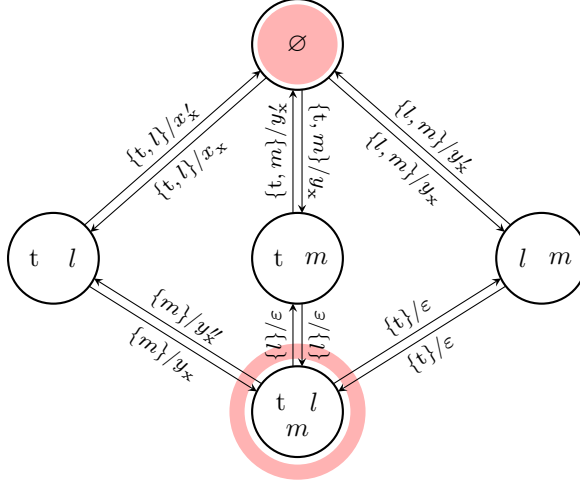


Figure 8.8: The DPL Mealy machine $MM(T_m)$ based on the same feature model as $MM(T_s)$ from Figure 8.7. We now have full reachability.

8.4.1 Cost

The cost of occupying a state in a DPL Mealy machine is that of monitoring the features from the accepted input-symbols for change. I posit that monitoring some features can be more expensive than monitoring others.

For example, it is more draining to the battery of a smartphone to constantly monitor GPS location (l) than it is to intermittently check the calendar for meetings (m), since the GPS receiver needs to constantly receive signals and the calendar is internal. But checking the calendar is still more costly keeping track of the time (t). The operating system does that anyway, and can notify our app through an alarm-subscription service.

- **8.48. Definition (Cost):** Assume some *cost domain* C measured over time, with an additive neutral element 0. Given DPL Mealy machine $(\Phi, \text{Pow}(\mathcal{F}), \mathcal{D}, T, O)$ we introduce a function $\text{cost}: \Phi \times \mathcal{F} \rightarrow C$. The value $\text{cost}(F_\ell, f)$ represents the cost of monitoring feature f from state F_ℓ . A feature is only monitored from a state if that state has an outgoing transition with f in its input symbol. So if the current state does not have such a transition, the cost is 0:

$$\nexists F_\Delta \subseteq \mathcal{F}: f \in F_\Delta \wedge (F_\ell, F_\Delta) \in \text{pre}(T) \implies \text{cost}(F_\ell, f) = 0 \quad \lrcorner$$

We want to maintain generality in the definition, but for the profile manager, the cost-domain is usually *power* in watt, i.e. joules per second. It is also likely that the cost of monitoring a profile feature depends solely on which quantity is being constrained (Definition 8.4), and is independent from the local feature configuration F_ℓ and the set of values of the constraint. When that is the case we can use a shorter notation:

- **8.49. Notation (Cost of Monitoring Quantities):** For all device constraints $\langle id_q, VAL \rangle \in C_{DEV} = \mathcal{F}$ and feature configurations $F \in \Phi$:

$$\text{cost}(id_q) \stackrel{\text{def}}{=} \text{cost}(F, \langle id_q, VAL \rangle) \quad \lrcorner$$

We minimize the cost of running a dynamic product line by removing costly transitions from our Mealy machine through additional restrictions on $\text{pre}(T)$, but only so far as we can maintain partial correctness (i.e., so far as we can avoid getting stuck in unstable configurations). Features only need to be monitored when they become relevant.

In our example product line (Figure 8.6) we need to apply delta x_x only when we are both in a certain GPS location (l) and at a certain time (t). Either constraint satisfied on its own does not modify the profile. So it makes sense to only start monitoring GPS (the more costly quantity), when it is already the right time. For this to work, we just have to check the GPS immediately when we reach the proper time, since the transition event may have occurred without the device observing it.

8.4.2 Optimization through Refinement

The trick to optimization is to realize that we do not need to reach a configuration where $F_\ell = F_e$, even though that has always been our goal in Section 8.3. There is another way to get a stable configuration. It is sufficient if we occupy a locally consistent configuration $\langle F_e, F_\ell, p \rangle$ with $V(F_\ell) \subseteq V(F_e)$. Such an F_ℓ is a state from which we might have $F_\ell \xrightarrow{F_\Delta/\varepsilon} F_e$, i.e. get the neutral delta ε as an output symbol if we actually did make the transition to F_e . Applying ε to a product does not change it, so we can sometimes remove transitions like that to avoid having to monitor the features in F_Δ .

In general, $V(F_\ell) \subseteq V(F_e)$ does not imply $V(F_e) \subseteq V(F_\ell)$. However, for a device rule-set that yields an unambiguous static product line implementation, it *does*. For every feature configuration there is only one profile that satisfies it, so we can set up an equivalence relation between feature configurations:

- ▷ **8.50. Definition (Equivalence):** Two feature configurations $F, G \in \Phi$ are *equivalent*, denoted $F \equiv G$, iff $V(F) = V(G)$. ┘

This equivalence can be decided at build-time and is represented in diagrams by a gray background which marks equivalence classes (Figure 8.9). This makes optimization through refinement more intuitive: the system only needs to reach a state in the same equivalence class as F_e .

8.4.3 Optimization through Redundancy

There is another kind of transition we could eliminate. Recall that we explained in Definition 4.10 why we could not go back to using singleton sets as input symbols. Sometimes we simply need all information in $F_\ell \ominus F_\ell'$ to unambiguously find the next state in the Mealy machine. So we kept defining T to expect the full difference as an input symbol, i.e., for every $F_\ell \xrightarrow{F_\Delta/d} F_\ell'$ we had $F_\Delta = F_\ell \ominus F_\ell'$.

But that is not required. It would be enough if the input symbol was *included* in the state-difference: $F_\Delta \subseteq F_\ell \ominus F_\ell'$. This is another opportunity to reduce the cost of a Mealy machine, because we may only need to monitor *some* of the features in $F_\ell \ominus F_\ell'$ to trigger the full transition.

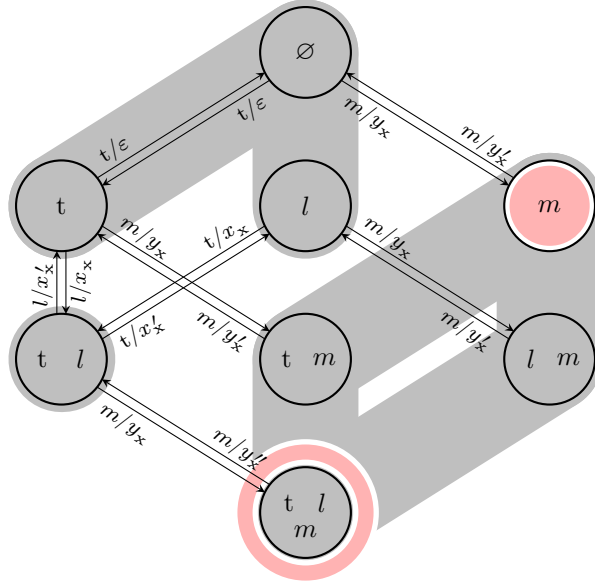


Figure 8.9: A DPL Mealy machine with a transition function T_o . Equivalence classes are marked: $\emptyset \equiv \{t\} \equiv \{l\}$ and $\{m\} \equiv \{t, m\} \equiv \{l, m\} \equiv \{t, l, m\}$.

8.4.4 A Local Inference Rule $\text{LOC-DIFF}(T_o)$

But this is where we hit a roadblock, because in an abstract setting there is not just one transition function to find. Finding the best T_o is an *optimization problem*. The goal is to choose one that minimizes the cost of running the dynamic product line, while preserving the property that $\text{LOC-DIFF}(T_o)$ only gets stuck on configurations $\langle F_e, F_\ell, p \rangle$ with $V(F_\ell) \subseteq V(F_e)$.

The correctness-proof of such a transition rule will be very similar to those already covered, except that this time, the usual process of elimination will leave us with only the negation of Definition 8.34a: $p \in V(F_e)$.

For the more specific domain of the profile manager, we can provide an example of T_o . Let us make a couple of assumptions:

- The cost of monitoring f depends solely on the quantity being monitored.
- During the average lifetime of the dynamic product line, all states are occupied for roughly the same amount of time.
- The following holds for our device (Notation 8.49):

$$\text{cost}(\text{time}) < \text{cost}(\text{meeting}) < \text{cost}(\text{gps})$$

So to define T_o , we start with T_m . We first remove l/ε transitions, then m/ε transitions, then t/ε transitions, so long as the T_o reachability between equivalence classes is preserved.

As you can see in Figure 8.9, we are able to remove ten transitions as compared to Figure 8.6, significantly reducing the overall monitoring cost. Intuitively, the transitions between \emptyset and l could be removed because the GPS position does not become relevant until it is the right time. The other eight transitions could be removed because y_x completely overwrites the effect of x_x , so it does not matter what happens with t and l during a meeting.

▷ **8.51. Example:** We show this with an example walk through Figure 8.9:

$$\begin{aligned}
\langle \emptyset, \quad \emptyset, \quad p \rangle &\xrightarrow{e,1} \langle \{l\}, \quad \emptyset, \quad p \rangle \xrightarrow{e,2} \\
\langle \{l,m\}, \quad \emptyset, \quad p \rangle &\xrightarrow{l,3} \langle \{l,m\}, \{m\}, p_m \rangle \xrightarrow{e,4} \\
\langle \{t,l,m\}, \{m\}, p_m \rangle &\xrightarrow{e,5} \langle \{t,l\}, \{m\}, p_m \rangle \xrightarrow{l,6} \\
\langle \{t,l\}, \quad \emptyset, \quad p \rangle &\xrightarrow{l,7} \langle \{t,l\}, \{t\}, p \rangle \xrightarrow{l,8} \\
\langle \{t,l\}, \quad \{t,l\}, p_{t,l} \rangle
\end{aligned}$$

This is the story:

1. We arrive at work before 9:00 for an early meeting. Nothing changes.
2. The meeting starts. We have an unstable configuration.
3. $p_m = \llbracket y_x \rrbracket(p)$. Our phone is automatically muted and the meeting minutes app is put on the screen.
4. It turns 9:00 during the meeting, but our phone does not have to respond. (This is the configuration marked in Figure 8.9.)
5. The meeting ends. We have another unstable configuration.
6. ... updating ($p = \llbracket y'_x \rrbracket(p_m)$) ...
7. ... updating ...
8. ... done. $p_{t,l} = \llbracket x_x \rrbracket(p)$. The phone is set to volume 5.

Note how the GPS module was not required until transition 8. When we physically arrived at work (F_e), our phone (F_ℓ) was unaware of it. Still, our phone was always operating with a proper profile without requiring a large drain on the battery. \perp

8.5 Conclusion

This chapter developed, step by step, an operational semantics for the reconfiguration of a running product in a dynamic product line. There has been some previous effort towards keeping objects in the heap up to date with the latest feature configuration. There has been a noticeable lack of work, however, in coming up with strategies for keeping the running product itself up to date. By tracking both the local and environmental feature configurations, a set of light-weight deltas can apply just enough changes to the running product to bring it up to date without having to regenerate it from scratch. The number of necessary deltas has been reduced to a minimal level, and we have proof that the system maintains correctness of the product before and after every reconfiguration. The software deltoid defined for the main example of this thesis was intended for structural modification, not to reason about running programs. It has no syntax defined below the statement level, let alone a memory model. Instead, this chapter introduced a new deltoid based on profile management. The main case study is a mobile application for managing the settings of a smartphone based on user-defined rules and input from its sensors.

That being said, we can now get a clearer picture of what the concrete operational semantics may look like around the **reconfigure** inference rule of page 167. Recall, this statement was introduced by by Damiani et al. [63,

64] so that developers can indicate when it is safe to reconfigure the product. That means the entire process described in this chapter will have to occur while the control flow waits on that statement.

We'll describe a new operational semantics; a hybrid of the classical imperative program semantics and the semantics developed in this chapter, embodied by a new transition relation \longrightarrow . We'll also include reconfiguration of the heap, as explored by Damiani et al. They describe *reconfiguration translations*, modeled by an automaton much like our Mealy machines. We'll abstract from the details, and encapsulate their technique into a function $\text{rh}: \Phi \times \Phi \times \mathcal{H} \rightarrow \mathcal{H}$, representing an effective procedure that performs those translations. It takes a local feature configuration, an environmental feature configuration and a heap, and returns a reconfigured heap. The hybrid configurations $\langle F_e, F_\ell, p, H, st, \sigma \rangle$ contain the environmental feature configuration F_e , the local feature configuration F_ℓ , the current product p , the current heap H , the next statement st and the current state σ .

The inference rules of the hybrid system would be as follows. First, environmental transitions can happen at any time during normal execution, but not while reconfiguration is taking place:

$$\frac{p \in V(F_e) \quad \langle F_e, F_\ell, p \rangle \xrightarrow{e} \langle F_e', F_\ell, p \rangle}{\langle F_e, F_\ell, p, H, st, \sigma \rangle \longrightarrow \langle F_e', F_\ell, p, H, st, \sigma \rangle}$$

Until a **reconfigure** statement is encountered, the program just runs like it normally would, following the imperative semantics:

$$\frac{st \neq \mathbf{reconfigure}; st'' \quad \langle st, H, \sigma \rangle \longrightarrow \langle st', H', \sigma' \rangle}{\langle F_e, F_\ell, p, H, st, \sigma \rangle \longrightarrow \langle F_e, F_\ell, p, H', st', \sigma' \rangle}$$

When control flow reaches a **reconfigure** statement, what happens next depends on whether the running product is out-of-date. If so, control is released to the dynamic product line semantics:

$$\frac{p \notin V(F_e) \quad \langle F_e, F_\ell, p \rangle \xrightarrow{\ell} \langle F_e, F_\ell', p' \rangle \quad \text{rh}(F_\ell, F_\ell', H) = H'}{\langle F_e, F_\ell, p, H, \mathbf{reconfigure}; st, \sigma \rangle \longrightarrow \langle F_e, F_\ell', p', H', \mathbf{reconfigure}; st, \sigma \rangle}$$

Note that the heap is updated along with the product at every step.

If/when the product is up to date, a **reconfigure** statement can be discarded and control released to the (now modified) program:

$$\frac{p \in V(F_e)}{\langle F_e, F_\ell, p, H, \mathbf{reconfigure}; st, \sigma \rangle \longrightarrow \langle F_e, F_\ell, p, H', st, \sigma \rangle}$$

Further formalization —and implementation— of this idea would be a fascinating topic for future research.

8.6 Related Work

Hallsteinsen et al. [82] describe several properties that constitute a dynamic software product line. The approach presented in this chapter, and the papers on DDM [6, 1], allow several of these, such as ‘dynamic variability’, ‘changes binding several times over lifetime’ and ‘context awareness’, but does not yet model others, such as ‘variation point change during runtime’ and ‘deals with unexpected changes during runtime’. In approach of this chapter, even though the environmental feature configuration can change during runtime, the set of available feature configurations is still fixed at build time.

Though ADM was designed from a software product line engineering perspective, the profile management case study of Section 8.2 is, of course, not a software product line, as it does not model the variability of software. It does, however, bear resemblance to a *Context-aware Program* [38, 103] or a *Self Adaptive System* [50, 146, 180]. Self-adaptive systems in particular have been linked with software product lines in recent literature. A number of papers aim to implement self-adaptive systems with dynamic software product line techniques [73, 170].

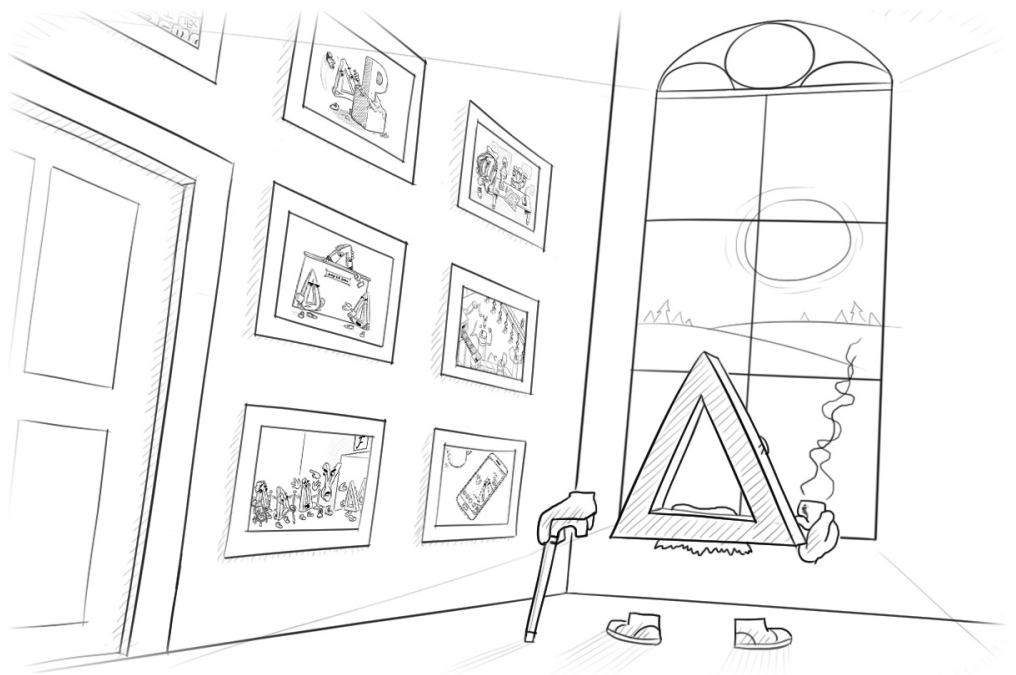
In self-adaptivity terms, DDM is *closed-adaptive*, as it is not able to cope with unexpected changes, in contrast to *open-adaptive* systems [146]. According to a recent survey by Weyns et al. [180], the vast majority of papers on these topics do focus specifically on development of flexible and reliable self-adaptive *software*. In comparison, the profile management model is relatively simplistic. As such, to call the profile management app a self-adaptive system would do the field (which has been concerned with self-driving cars and unmanned air vehicles behind enemy lines) a disservice. It is safe to say that DDM has not yet proved itself in those terms.

A number of recent publications, though, *have* explored dynamic software product lines in terms of delta modeling. Damiani et al. [63, 64] apply delta oriented programming to the problem, and focus on control flow, heap reconfiguration and type safety, as explained in the introduction to this chapter. Additionally, Muschevici et al. [141] recently extended the ABS language (Section 7.5) for the implementation of dynamic systems. They do mention the issue of dynamic product reconfiguration, and propose that certain deltas not already present in the original delta model (e.g., deltas x'_x , y'_x and y''_x in Figure 8.6) should be manually developed. One of the messages of Section 8.3.6 is that this may in fact be automated, given a correctly implemented delta derivation procedure, something Muschevici et al. have proposed as future work.

Interestingly, both groups mention the unanticipated runtime evolution necessary for open-adaptive systems. The idea is that adding, removing and modifying deltas while the product is still running is valid, so long as those changes have no impact on the deltas used in generating the currently running product. In particular, Muschevici et al. discuss MetaABS, an API based on reflection used for this very purpose.

Conclusion

A Look Back and a Look Forward



9.1 A Look Back

This is the final chapter. It is time to reflect on what has been achieved and to determine the best way to go forward from here. We first revisit the goals and contributions of Chapters 2 to 8. This section stays away from formalization and focuses instead on motivation and summary.

Chapter 2: Algebraic Delta Modeling

This chapter introduced the basic building blocks of delta modeling. One of those building blocks is the *product*. This rather abstract concept represents the kind of artefact that needs to be manufactured. In practice, a product is built up out of many smaller artefacts. Common examples are packages, classes, methods and fields, in an object oriented programming language, together forming a program. Since this is the original motivation behind delta modeling, and a concept well understood by the target audience of this thesis, the running example of the thesis is based upon this sort of product.

The problem is that the artefacts in such a product almost never map directly to the higher level concept of ‘feature’. Indeed, a feature can relate to many classes, and a class can relate to many features. This brings us to the goal of *feature modularity*:

Goal: Find a way to ‘group together’ code related to the same feature.

To accomplish feature modularity, *deltas* are introduced. Deltas are an abstract concept embodying the changes to a product —necessary to implement certain functionality— that a developer might make. When a developer needs to implement a feature in a product, he or she is able to modify any number of artefacts in order to do so. Similarly, deltas, too, should be able to specify modifications that break encapsulation and artefact boundaries.

That way, all code related to a feature can be gathered in one place: the delta. In this vision, the rôle of the human developer would change from making changes to the product to writing deltas that do. A separate but related goal is *separation of concerns*:

Goal: Find a way to ‘separate’ code belonging to different features.

Deltas are to be true units of functionality, in that they should not implement more than one piece of functionality; that is, they would ideally contain the smallest set of changes that make sense in isolation —and do something useful— but no more, and this way achieve a separation of concerns. This carefully phrased ideal allows for scenarios in which some features extend or depend on others, or are conceptually independent but require access to the same resource. Chapter 3 was dedicated to these kinds of issues.

This chapter thoroughly explores the interaction between deltas and the interaction between a delta and a product, thereby introducing the fundamentals of *Abstract Delta Modeling (ADM)*, built upon by chapters to follow. The notion of *deltoid* is introduced, which contains the full sets of products and deltas representing a specific domain, as well as the semantics

of deltas: how they modify products. By working abstractly, ADM is ready to encode any domain, not limited to any specific programming language, nor even to software itself.

To jumpstart the running example introduced in Section 1.4—the Editor product line—a concrete deltoid was defined based on an object oriented programming domain. Many subsequent concepts were illustrated through this example.

Various aspects of delta semantics were discussed, such as *partial definedness*, *non-determinism* and *correctness* with regard to a relational specification. A number of *algebraic operations*—such as composition, choice and consensus—are introduced in order to allow syntactic reasoning over deltas. Certain expressiveness properties and a refinement relation are then introduced in order to classify deltoids by what they can do. Finally, it is shown how deltas can encode quarks, a similar concept introduced in related literature.

Chapter 3: Delta Models

In some ways, this chapter described the most fundamentally novel contribution of ADM: *delta models*, which organize deltas into a strict partial *application order*. One delta may be dominant over another, or two deltas may be unrelated by the order. This helps developers express their design intentions, and to contain the complexity of large system.

Goal: *Find a way to mediate between non-commuting feature modules.*

Let's say there are two feature modules (the more general term for what we call deltas), each implementing a different feature in a way that preserves modularity and separates concerns. It is possible that both need access to the same artefact, causing a *conflict* if both are applied together, even if each works fine in isolation. It is for this reason that separation of concerns is not easy to achieve. This chapter proposes three possible ways of mediating such a conflict:

1. Make (minimal) changes to one or both deltas.
2. Impose an application order between the two deltas. The 'dominant' delta is applied last so it can override some of the changes. It should be designed to expect the other delta to go first.
3. Write a *conflict resolving delta*, ordered last so it can make the appropriate changes allowing the original two deltas to work together.

Each is applicable in different situations. For instance, if the conflict is merely an accidental name-collision caused by a lack of communication, the issue is quickly solved by making minimal changes.

Perhaps one of the two features is really a *subfeature* of the other, and it rightly *should* override modifications performed by the other. In that case the deltas should be ordered.

But often enough, neither applies. In this case a conflict-resolving delta is the only way to resolve a conflict properly. It allows the original deltas to remain as they are, and introduces the necessary 'glue code' to facilitate their coexistence. How exactly this is done is a design choice. Development of a conflict resolving delta cannot generally be automated.

Goal: *Find a way to avoid overspecification of the structural organization between feature modules.*

Many features in a system can be *conceptually independent*. This means that they make sense—and should work—in isolation. Ideally, these are even *developed* in isolation (more on this in Chapter 7). When the implementations of two such features are in conflict, this is known as the *optional feature problem*.

Unfortunately, a number of existing systems and formalisms do not have a partially ordered structure between modules, but a linearly ordered one: between any two modules, one can override the changes of the other. If two features are conceptually independent, one of their modules overwriting the changes of the other is most likely a *bug*. Forcing such modules into a linear order is called *overspecification*, and can obscure such bugs. After all, an automated system assembling these feature modules can hardly be expected to know the difference between an accidental overwrite and an intentional one.

By allowing two deltas to be *unordered*, it becomes possible to express that they implement conceptually independent functionality. If there is ever a conflict, developers can be warned.

Goal: *Find a way to avoid code duplication through the structural organization between feature modules.*

Other existing systems, perhaps in an effort to avoid the overspecification problem, take the opposite approach and do not allow *any* feature module to interfere with any other.

The way to resolve a conflict in such a system is to completely extract the artefacts that clash, and put them into a dedicated module, in such a way that both features work. But in doing so, both modularity and separation of concerns are violated. And when creating a product line, code needs to be duplicated between modules that implement the same artefact for different configurations. Because delta models allow deltas to be ordered when necessary, they do not share this problem.

Apart from introducing the general concepts of delta model, conflict and conflict resolver, this chapter introduced conditions based on these concepts to ensure *unambiguous* product generation.

It then extends the software deltoid to allow *fine-grained* modifications, i.e., manipulating individual statements in methods. This is often neglected in compositional approaches like delta modeling, because unlike classes, methods and fields, statements have no names by which a delta can target their position. *Conjunctive delta model semantics* are then introduced to take advantage of fine-grained modifications. The operation of inserting a statement in a non-deterministically chosen location avoids another type of overspecification by representing the intention: “this method should run this statement at some point; I don’t care when”. It reduces the likelihood that two changes to the same method are seen as a conflict. Finally, the chapter introduced *nested delta models*, which increase expressiveness of a deltoid and offer a useful new modularization technique.

Chapter 4: Product Lines

We’ve spoken of features before now, but this chapter is where the concept of *feature* is formally introduced and integrated in ADM. These features are merely labels, but play a prominent rôle throughout the rest of the story. They are traditionally used in a *feature model* as a way to identify the possible products of a *product line*, which is defined as a set of products with well-defined commonalities and variabilities. Ideally, a product line should be able to produce any of these products, given only the desired feature selection.

Goal: *Develop a technique for organizing a product line code-base in such a way that product generation can be a mechanical process.*

Given that deltas are our feature modules, producing the product corresponding to a specific feature selection is really just a matter of applying the right set of deltas. In ADM, this is done by annotating each delta with an *application condition*: a propositional formula representing the set of feature selections for which it is applicable.

The main challenge here is that each delta must be able to deal gracefully with the presence and absence of other deltas. So developing a product line in which every product behaves properly is the ultimate test of modularity and separation of concerns, because if those principles are adhered to, robust deltas should be an automatic consequence. This does beg the question: what does ‘behave properly’ even mean?

Goal: *Develop a formal concept of product line specification, to be used both in verifying product line correctness and in guiding the implementation process.*

The naive way of giving a product line specification would be to give a separate specification for each of its products. But specifications should be modular and compositional, just like deltas. It is better to write a separate specification for each *feature*. However, an important observation made in this chapter is that it is more realistic to write specifications for *feature combinations* instead. Often, two features that are otherwise independent need to satisfy additional requirements when they are selected together. This is not about conflicts; those are purely an implementation issue. This is about features that inherently interact but shouldn’t, or don’t inherently interact but should. (Formally speaking there is little difference between the two.)

Apart from providing a characterisation of product line correctness, this chapter lifted a number of concepts from Chapter 3 to the product line level, such as unambiguity and nesting, and introduced a number of other useful concepts. Of particular note is that of *parametric deltas*. Sometimes the delta language is much better at resolving conflicts and implementing interaction than the delta model structure. For those eventualities, deltas can be given access to the feature symbols to be used as Boolean constants. This brings some of the power of annotative variability approaches to the compositional technique of delta modeling. However, caution is advised in using this technique, as annotative techniques have their disadvantages.

Chapter 5: L^AT_EX Meets Delta Modeling

Several publications on ADM make the claim that deltas can be used to modularize any kind of artefact — not just source code. An example occasionally brought up is documentation. Indeed, the abstract nature of ADM should allow this, but it had not yet been demonstrated. So what better language for which to implement and demonstrate deltas than the one used to write this very thesis? T_EX is a fascinating language; functional by nature, but with the unusual characteristic that practically the entire language can be redefined from within. This brings two opportunities. First, it is a way for deltas to hook into document generation without requiring outside tools; deltas can just be defined in a L^AT_EX package. Second, the power of the language has caused a number of problems in the L^AT_EX ecosystem: conflicts between independent packages that access the same resources. The conflict and dependency model of ADM can be adapted to mediate between such packages and, hopefully, alleviate much frustration in the L^AT_EX community.

Goal: Implement delta modeling for the L^AT_EX language.

The first part of the chapter introduced delta-modules, a new L^AT_EX package that brings the main ADM concepts —deltas, partial application orders, feature models— to the L^AT_EX language, and supports operations quite similar to those of fine-grained software deltas. The package is introduced in a software documentation style, with crosslinks to the relevant formal concepts of the thesis. And what better case study than the thesis itself? (I *am* a fan of self-reference. [95]) In practice, the package may become useful for preparing families of text-books, tech manuals and résumé.

Goal: Use ADM principles to manage dependencies and conflicts between independent L^AT_EX packages.

The solution to this problem also takes the form of a package. It is called `pkgloader`, and is similar to delta-modules in many respects. But this package has two additional challenges to overcome. First, package authors are not delta authors. We cannot rely on the fact that packages in the wild will limit their tampering to specialized delta operations. And with the full power of a Turing complete language behind them, this means that the problem of detecting conflicts is undecidable. Second, document authors should not be bothered with product line concepts. Ideally, they would just load `pkgloader`... and that's it; things should just work.

To address the first challenge, a centralized knowledge-base is maintained with known package conflicts and resolutions. L^AT_EX has an active community that can be relied upon to keep such a database up to date. To address the second challenge, the package takes control of the ubiquitous `\usepackage` and `\RequirePackage` commands. Document authors are already familiar with those, and use them to load packages. When they do, this will be interpreted as selecting a feature. By default, this just loads the package. But with a well-maintained database, the loading order between packages may be changed, and specific glue-code inserted where necessary.

Chapter 6: Delta Logic

Much of ADM is dedicated to the goal of developing syntactic languages and techniques for semantic concepts. Deltas are syntactic. But products (from an ADM point of view) are semantic concepts. Consequently, reasoning about the semantics of deltas requires semantic proof machinery.

Goal: *Create a modal logic for reasoning syntactically about the semantics of deltas and their effects on product properties.*

Modal logic fits this problem like a glove. Given any kind of decidable specification language for the product domain, wrapping a multi-modal logic around it enables us to prove that certain deltas implement certain features, that they do not break existing features, and so on. Modal logic was invented specifically to reason about labeled relations, and deltas fit the bill. It is also possible to reason about the algebraic delta operators introduced in Chapter 2. The result is a language reminiscent of dynamic logic, but lacking a construct for iteration. This turns out to be a great advantage, because it keeps the logic decidable. A proof of *strong completeness* is given based on a straightforward translation to a plain multi-modal language, allowing us to simply invoke the completeness of **K** with regard to the class of all frames.

The other two contributions of the chapter are these: First, a proof system for *delta correctness* with regard to modal formulas in the form of Hoare triples, including a proof of its soundness and strong completeness. Second, a proof system for reasoning about specific features on the level of *Kripke models*. The proof system on the Kripke frame level cannot be used because of uniform substitution. We solve this with a translation to *nullary modalities*.

Chapter 7: Delta Modeling Workflow

The formalisation of ADM thus far has been *descriptive*, describing what deltas are, how they work and how they are selected. The other side of the story is *prescriptive*: how are delta models intended to be used? In what way and in which order should a product line be developed so that the full advantage of delta modeling is exploited?

Goal: *Describe how delta-based product lines might be built.*

There may be many ways to use delta modeling to good effect. Indeed, many tools are eventually put to innovative uses that were initially unintended. Let's just say that ADM lends itself naturally to a certain way of working which happens to exhibit favorable properties. This chapter presents that workflow under the moniker of *delta modeling workflow (DMW)*. It is split up into well-defined jobs derived from the product line specification.

This formulation makes it explicit that independent features can be developed concurrently and in isolation, allowing for maximal throughput. This way of working counts on the eventual collaborative development of conflict resolving deltas and feature interaction deltas to integrate these

individual efforts. If local constraints are respected, two properties are guaranteed to hold by construction: global unambiguity (i.e., all conflicts are resolved) and total correctness with regard to the specification.

An important concept of the chapter is that of *locality*. Any delta under development need only take into account the existing deltas that occupy subordinate positions in the delta model. Those are the ones the new delta has control over and no other knowledge is required to satisfy local constraints.

In Appendix A, the states and progression of the workflow are represented with a *structural operational semantics*. This is used to prove its beneficial properties.

Goal: *Test the delta modeling workflow on an industrial scale system in order to evaluate its practical applicability.*

As a member of the HATS project, I had the opportunity to describe the delta modeling workflow for the *Abstract Behavioral Specification (ABS)* language. I was also in a position to work together with Fredhopper on the *Fredhopper Access Server*, an industrial scale case study which helped validate and improve the workflow. Lessons learned from the case study include a confidence in the thoroughness of the workflow — no features, conflicts or interactions fell through the cracks. Collaboration was possible with the workflow, but this was not yet apparent from the formalism, which was revised accordingly. Flexibility is still a problem. Therefore, adaptation to a more agile approach is planned as future work.

Chapter 8: Dynamic Product Lines

The penultimate chapter of the thesis takes ADM to *runtime*, as deltas are used to update a product to new feature configurations while it is still running. This had already been discussed in previous work. In particular, there has been some effort towards keeping objects in the heap up to date with the latest feature configuration. There has been a noticable lack of work, however, in coming up with strategies for keeping the running product itself up to date. Doing it in the ‘static way’, applying all deltas every time the feature configuration changes, is too slow. And storing every possible product in advance would require too much memory, as the number of products is generally exponential in the number of features.

Goal: *Formulate efficient strategies for reconfiguration of the running product in an ADM-based dynamic product line.*

First, an operational semantics is set up as a framework in which to discuss possible strategies. The abovementioned ‘static-style’ strategy is formulated and proved correct as an example. A case is made for keeping track of the *differences* between subsequent feature configurations, allowing the system to figure out the minimal delta that needs to be applied to bring the product up to date. However, this still leaves a lot of possibilities. A Mealy machine

model is introduced to compare the pros and cons between various ‘difference-based’ strategies. In this model, each state represents a feature configuration and each transition represents both a feature configuration difference and a corresponding delta to be applied to the running product.

Eventually, this leads to a strategy that balances the number of stored deltas with the desired runtime efficiency. This strategy is subsequently proved correct using a number of techniques introduced step-by-step throughout the chapter.

Finally, a specialized optimization opportunity is presented. In order to detect feature configuration changes, the environment needs to be monitored. This is naturally modeled with the Mealy machine, in which each feature configuration difference (represented as an input symbol on certain transitions) represents a set of ‘sensors’ that need to be engaged when occupying certain states. The optimization technique consists of discarding certain transitions from the Mealy machine that are irrelevant, saving energy for the average state occupation of the model. This allows us to segue into the final goal:

Goal: *Develop a profile management app for Android based on the dynamic product line strategies explored in this chapter.*

The software product model defined for the running example of the thesis were intended for structural modification, not to reason about running programs. It has no syntax defined below the statement level, let alone a memory model. It is therefore not a useful example in this chapter. Additionally, the main contribution of the chapter is separate from any issues specific to software. Such issues were already explored by other researchers. In the trend set by the rest of the thesis, the dynamic delta modeling formalism is abstract by nature and can potentially support any domain. I therefore chose a model that is formally simple, yet able to directly illustrate the practical use of the theory.

The case study used to illustrate the formal concepts in this chapter is a mobile application for managing the settings of a smartphone based on any kind of sensory input. While somewhat untraditional as an example of a product line, it actually fits the mold quite readily. Features are represented by predicates over specific environmental quantities, such as GPS location, battery level and calendar appointments. Products are represented by the possible configurations (or *profiles*) of the settings on the phone, such as volume, screen brightness and chat status. By having deltas applying changes to the running profile based on specific environmental conditions (i.e., feature configurations) specified by the user, we essentially have a simple dynamic product line running on the smartphone, as well as a case study with actual practical value: the idea was developed into a working Android application. The ‘energy saving’ optimization techniques are employed to preserve battery life.

9.2 A Look Forward

This thesis, rather than focusing deeply on any one topic, covers a broad area of research and application. As such there is a great deal of potential for future work. This section shares a glimpse of the possibilities.

9.2.1 Darcs Patch Theory

There is a lot of similarity between delta modeling and Darcs patch theory [97], yet they have very different purposes.

Deltoids can be designed with smart, domain specific operations tailored to the product domain, allowing deltas to be more robust under changing circumstances. This approach might add something to patch theory and version control systems. Conversely, a fundamental aspect of patch theory is that the application of any patch can be reversed. This relates to one algebraic operator that wasn't well-covered in Section 2.6: the converse operator \smile . Studying the impact of this idea on delta modeling could yield useful results.

Additionally, Darcs patch theory deals with a naturally occurring, partially ordered structure very similar to delta models: that of branches and merges in a version control system. The most significant similarity is that it deals with *conflictors*, which are entities for resolving conflicts. They are quite similar to conflict-resolving deltas, though they seem to have a more complex set of properties due to the added structure of their core setting.

All in all, making a more detailed comparison promises to be a worthwhile pursuit.

9.2.2 A Constructive Relation Algebra

Section 2.6.2 briefly discussed the constructivism of the algebraic operators of the relation algebra pioneered by Tarski [101, 102, 175]. Relation algebras (Definition 1.35, page 24) are not constructive, because they contain Boolean algebras (Definition 1.34), which, in turn, contain a non-constructive axiomatisation for the negation operator $-$ and the full element \top .

Slightly weaker than Boolean algebras, and widely known to be constructive, are *Heyting algebras* [92]. They still have a negation operator and a top element, but not as fundamental ingredients. They contain an *implication* operator $\Rightarrow: S \times S \rightarrow S$ instead. The semantics of $-$ and \top are weakened to what may be deduced from the axioms $e \Rightarrow g = e^- \sqcup g$ and $\perp^- = \top$.

For delta modeling, however, we suspect that a different approach would be more useful. Rather than use an implication operator as a fundamental ingredient, a *difference operator* $-: S \times S \rightarrow S$ could be used. This structure is called a *co-Heyting algebra* (or *Brouwer lattice*) [34, 177]. Co-Heyting algebras are the dual of Heyting algebras, and employ the axiom $e - g = e \sqcap g^-$. The difference operator seems to have an intuitive interpretation for deltas, semantically corresponding to set difference \setminus .

We have not been able to discover any work applying this idea to full relation algebras, i.e., forming 'co-Heyting relation algebras' and exploring the implications, particularly with regard to the converse operator \smile . Such research would have a direct and potentially large impact on delta modeling.

9.2.3 Deltas and Traits

Section 2.10 compares deltas with *traits* as a means of implementing product line features. The conclusion was reached that traits are not suitable for the task, at least not in and of themselves. They were designed to enhance code reuse as an alternative to the (inappropriate) use of class inheritance.

Software deltas, on the other hand, were designed to contain the code implementing a specific feature (combination). They were *not* meant for code reuse — at least not in the same sense (sharing code across different products in a product line might also be called reuse). It may be valuable to look at traits and deltas as solving orthogonal goals, and to consider combining them, e.g., to allow deltas to manipulate and insert traits.

9.2.4 A General Development Framework

Now follows one of the more ambitious future work proposals: the implementation of a *general development framework* for building and analyzing delta-based software. This framework should address two problems in particular.

The first problem is that in a compositional approach such as delta modeling, a lot of code will need to be written outside the context where it is eventually applied. This decoupling is a great advantage in the fight against complexity, but programmers are not used to going back and forth between various modules to understand the behavior of a single class or method. They require tool-support to help them reason about a modification in any desired context.

The second problem is that many existing AOP and compositional SPL variability tools only work on a single programming language at a time. Many software features, however, are expressed in multiple languages. For example: HTML for the logical structure of an interface, CSS for its styling and Javascript for its behavior. But current approaches to feature-based modularity require a team to either restrict themselves to one language (per feature) or to manage the variability for each language separately — a maintenance nightmare.

The development framework would likely take the form of a plugin for an existing IDE, such as Eclipse [139] or IntelliJ IDEA [99]. It should include the feature of *code views*: when editing a delta, the programmer would be able to edit it as a whole (as it is stored) or to edit fragments of it directly in the context where they apply. This would bring one of the main benefits of annotative variability techniques, and address the first problem. This concept of code views is reminiscent of CIDE [108] as well as of ‘hyperplanes’ [148]. But we expect the concept to be much more powerful when applied to the more expressive structure of ADM, leveraged to implement features, coordinate interaction and resolve implementation conflicts in a way more intuitive than has been possible before.

Figure 9.1 shows a mockup of what an Eclipse interface for code views might look like. The controls marked “Delta” show which code artifact is currently being edited (in this case, the RulePriorities delta). If it is a delta, a “Code View” can also be chosen, indicating the context in which to edit that delta, consisting of the ‘core’ code artifact (in this case the EditGeneralFragment class) already modified by a chosen set of deltas

from its *local delta model* (a concept described in Chapter 7; in this case, 3 other deltas). A code view is visible in the editor. The editable fragments of the delta appear in white blocks nested in their proper context.

A development effort like this is likely to be the most valuable contribution to delta modeling that could be made right now.

9.2.5 L^AT_EX Deltas

Future work related to the L^AT_EX packages is likely to be of a *development*-rather than a research nature. As the code is open source, anyone and everyone is encouraged to contribute. One idea is to implement *conjunctive semantics* (Definition 3.26) for the delta-modules package.

The pkgloader package is still quite limited and there is much that can be done to improve it, though little having to do with delta modeling. But if pkgloader becomes widely used in the future, it would become worthwhile to start thinking about the creation of *delta-aware commands* for package authors to use.

9.2.6 Delta Logic

As explained in Section 6.3.4, the delta logics of Chapter 6 are too limited for practical use because the postcondition in a delta contract is not able to refer back to the original product, leaving delta contracts at the level of expressiveness of delta derivation (Section 2.4.3, page 46).

One way to solve this would be a *hybrid language* [22, 40]. Hybrid logics rely on a set of *nominals*, which are propositional variables that are true in exactly one world. They also offer one or more hybrid operators. For instance, the best known hybrid logic is $\mathcal{H}(@, \downarrow)$, which offers a *satisfaction operator* $@_{nom}$ for each nominal nom , acting as a sort of modality to travel to the world characterized by nom , and an operator $\downarrow nom$, which can dynamically bind a nominal symbol nom to the current world. We could use a nominal to characterize the original world, then travel back to it from the resulting world to make certain comparisons.

It turns out that $\mathcal{H}(@, \downarrow)$ is undecidable [43] (though still weaker than first order logic). Fortunately, we wouldn't need to dynamically bind nominals to worlds. For our purpose it would be sufficient that nominals are under implicit universal quantification, as all propositional variables are. The logic $\mathcal{H}(@)$ is decidable [39, 72, 149], making it a perfect candidate to explore in future work.

Additionally, a more concrete exploration of the concept is called for; one that applies delta logic techniques to the verification of actual software product lines. This will illuminate the challenges ahead in embedding other logics into the modal logic, to specify specific software properties.

9.2.7 Delta Modeling Workflow

The delta modeling workflow is really just a first step in defining good development practices for delta modeling. The creation of a development framework as discussed above would help enormously. But in the mean time there are a few improvements that can be made to the formalism.

The workflow assumes a sole-derivation semantics right now. A possibility for future work is to define a good workflow taking advantage of conjunctive semantics (Section 3.5).

Section 7.8 points out that the DMW does not yet conform to the modern practices of *agile development* [130]. In particular, demanding a full specification in advance is now considered unwise. Even so, a number of the core principles of ADM and DMW have great potential for an agile development workflow. The fact that features are isolated and modularized in the first place makes it easy to try new things—start on new features—while confident that it cannot have permanent impact on the code base. Deltas can simply remain inactive until they are deemed ready for production (without the hassle of branching and merging in a text-centric version control system). Moreover, features can be thoroughly developed and tested individually before developers have to worry about testing their interaction, making it easier to divide the work into clear steps. Recall the test driven development scenario described on page 108. As for DMW principles: enforcement of delta locality (Section 7.3) ensures that developers are warned automatically when one of the modules needs to be updated because of changes higher up the delta model, making delta models a safe environment for experimentation.

Creating solid refinement and refactoring theory for delta models is probably the best place to start in adopting agile values and principles into the DMW. This would also help in converting legacy code bases into delta models through a gradual process.

9.2.8 Dynamic Delta Modeling

The hybrid operational semantics for dynamic software product lines presented in Section 8.5 should be much more thoroughly explored. At the moment, the work of Damiani et al. [63, 64] addresses the heap without addressing the product, and DDM addresses the product without addressing the heap. And a good comparison with the work of Muschevici et al. [141] has not yet been made either. Another interesting direction is to develop DDM support for open-adaptivity.

Finally, generation of the Mealy machine based on the static product line implementation is still done in a very roundabout way, assuming an implemented delta derivation operator. A more promising approach, perhaps, is to implement the delta converse operator \smile . This may help in effectively ‘navigating’ the product line delta model at runtime.

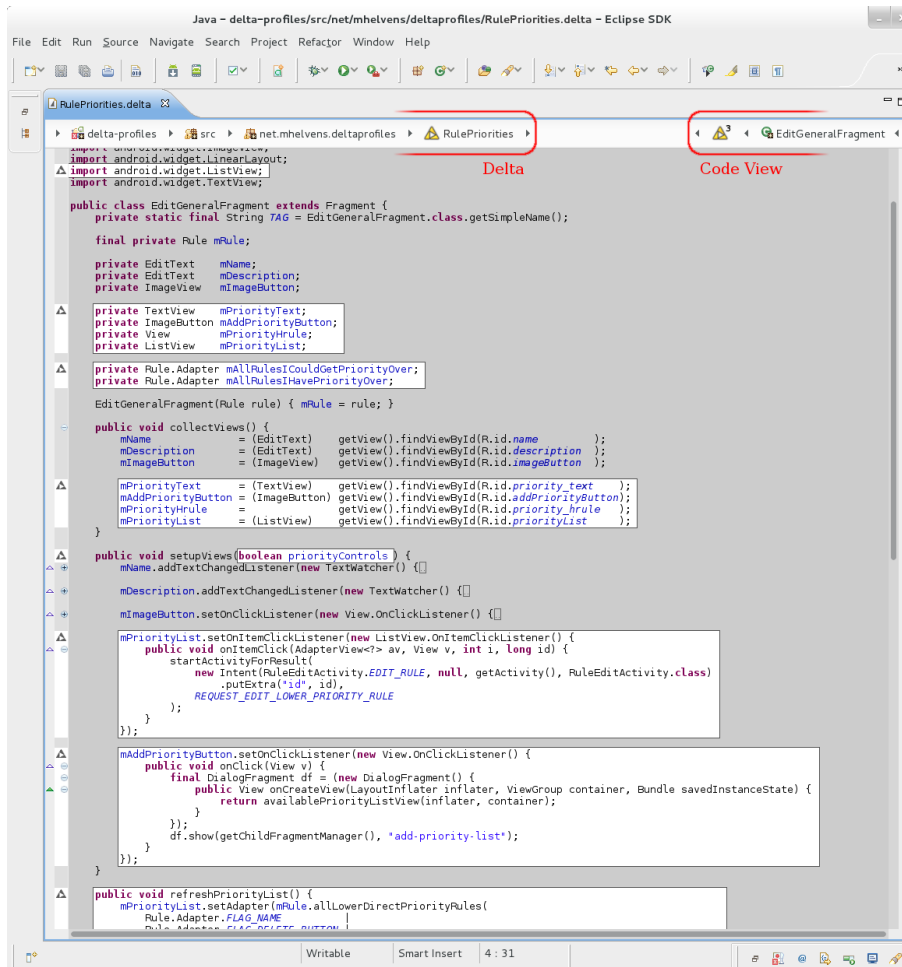


Figure 9.1: A mockup of a possible Eclipse interface for delta modeling with code views. (Incidentally, the code displayed here is part of the Android profile management application from Chapter 8.)

DMW Operational Semantics

Formalization of the Workflow and Proofs of its Properties

A.1 The Subfeature Relation

From here on, assume a deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, \llbracket - \rrbracket)$ that exhibits consistent conflict resolution (Definition 3.18) and a feature set \mathcal{F} .

Based on the subfeature relation defined in Section 7.2 (page 150), we extend the specification of a product line as follows:

- **A.1. Definition (Structured Product Line Specification):** A *structured product line specification* is a triple $sPLS = (\Phi, \Rightarrow, V)$ where (Φ, V) is a product line specification (Definition 4.19) and $\Rightarrow \subseteq \mathcal{F} \times \mathcal{F}$ is a direct subfeature relation. For all such specifications we require the following two properties to hold for all features $f, g \in \mathcal{F}$, all feature configurations $F \in \Phi$:

$$\begin{aligned} \text{a. } f \Rightarrow g &\implies (g \in F \implies f \in F) \\ \text{b. } f \Rightarrow g &\implies (V(\{g\}) \subseteq V(\{f\})) \end{aligned}$$

Namely, (a) that the selection of any feature implies the selection of its superfeatures and (b) a product's support for any feature implies support for its superfeatures. The set of all structured product line specifications is denoted $s\mathcal{PLS}$. If the deltoid or feature set is not clear from context, we attach a subscript as in $s\mathcal{PLS}_{Dt, \mathcal{F}}$. \lrcorner

A.2 Non-interference

The *non-interference* property described in Section 7.3 (page 151) is formally defined as follows:

- **A.2. Definition (Non-Interference):** A given deltoid $Dt = (\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, \llbracket - \rrbracket)$ and valuation function $V: \text{Pow}(\mathcal{F}) \rightarrow \text{Pow}(\mathcal{P})$ jointly exhibit the property of *non-interference* iff for all deltas $x, y, z \in \mathcal{D}$, products p and feature selections F :

$$\underbrace{z \cdot y \cdot x = z \cdot x \cdot y}_{\text{a}} \quad \Rightarrow \quad \underbrace{\llbracket z \cdot x \rrbracket(p) \subseteq V(F) \Rightarrow \llbracket z \cdot y \cdot x \rrbracket(p) \subseteq V(F)}_{\text{b}} \quad \lrcorner$$

So we can make use of locality if (a) for all delta-pairs x, y that commute in some context z (which may or may not be resolving a conflict between them), (b) in that same context, any property introduced to the final product by x cannot be broken by the presence or absence of y .

From this point on, we assume a structured product line specification $sPLS = (\Phi, \Rightarrow, V)$ which exhibits non-interfere with the earlier assumed deltoid Dt (Definition A.2).

A.3 An Operational Semantics

The job-based model introduced in Section 7.4.4 (page 153) is now described as an *operational semantics* (Section 1.7.11), in which the steps are transitions (Notation 1.49).

A.3.1 Job Status

A job can be in one of three stages. We define a type of mapping to keep track of the status of all jobs:

- **A.3. Definition (Job Status Map):** Given some set of already developed deltas $D \subseteq \mathcal{D}$, a *job status map* $J: (\text{Pow}(\mathcal{F}) \cup \text{Pow}(D)) \rightarrow (\{\text{av}, \text{ip}\} \cup D)$ is a finite partial function (Definition 1.17) mapping each job $j \in \text{Pow}(\mathcal{F}) \cup \text{Pow}(D)$ to its current status. Either:

- it is not recognized as a viable job (yet): $J(j) = \perp$,
- it is available: $J(j) = \text{av}$,
- it is in progress: $J(j) = \text{ip}$, or
- it is finished, and has resulted in delta $d \in D$: $J(j) = d$ \lrcorner

A.3.2 Configurations

Each state of the workflow is represented by a configuration as follows:

- **A.4. Definition (Workflow States):** A *workflow state* is a configuration:

$$ws = \langle adm, J \rangle$$

where $adm = (D, \prec, \gamma)$ is the annotated delta model in progress and the function $J: (\text{Pow}(\mathcal{F}) \cup \text{Pow}(D)) \rightarrow (\{\text{av}, \text{ip}\} \cup D)$ is a job status map (Definition A.3) used for bookkeeping. The whole configuration space is denoted WS . \lrcorner

The initial state of the workflow is simple. The annotated delta model is still empty and no jobs have been formulated yet:

- **A.5. Definition (Initial State):** The *initial state* of the workflow is defined as follows:

$$ws_0 \stackrel{\text{def}}{=} \langle adm_0, J_0 \rangle = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

with an empty annotated delta model $(\emptyset, \emptyset, \emptyset)$ —parentheses are omitted—, and an empty set of initial jobs \emptyset . \lrcorner

Steps 1 to 6 of the workflow description in Section 7.4.3 are each represented as inference rules (Notation 1.15), which define valid state transitions $\rightarrow \subseteq WS \times WS$. The whole development process can then be represented as a chain of n transitions:

$$\langle adm_0, J_0 \rangle \rightarrow \dots \rightarrow \langle adm_n, J_n \rangle$$

The workflow is finished after $\frac{1}{3}n$ jobs $j \in \text{Pow}(\mathcal{F}) \cup \text{Pow}(D)$, each of which goes through 3 transitions:

1. Identifying the job (steps 1, 3 and 5), which inserts it into the job status map J with status ‘available’ (av),
2. starting a job, which gives it the status ‘in progress’ (ip), and
3. finishing a job, which results in a new delta (steps 2, 4 and 6).

- **A.6. Notation:** We identify each specific transition t by its job and transition number, e.g., $t = \{f\}_3$ is the transition that takes place upon finishing the implementation of feature f . We sometimes annotate the transition arrow with this information: $\xrightarrow{\{f\}_3}$. \lrcorner

By splitting each job up this way, we explicitly allow interleaving its transitions with the transitions of other jobs, while keeping important updates as atomic operations. This makes it clear which jobs can be performed concurrently.

A.3.3 Inference Rules

The job order described in Section 7.4.4 (page 153) is formally encoded in the inference rules of the operational semantics. We now define these inference rules. Together they define the transition relation \rightarrow . We need to make sure that the workflow eventually terminates at the configuration representing a correct product line $(\Phi, 0, adm_n)$.

Identifying New Jobs

First, define the inference rule for introducing a new feature implementation job. But to do so, we first have to determine exactly which feature combinations will (eventually) get a specific delta to implement them:

- **A.7. Definition (Viable Feature Combination):** A *viable feature combination* is a feature set $F \subseteq \mathcal{F}$ such that:

$$\text{vf}(F) \stackrel{\text{def}}{\iff} \underbrace{\exists F' \in \Phi: F \subseteq F'}_{\text{a}} \quad \wedge \quad \underbrace{V(F) \neq \bigcap_{E \twoheadrightarrow F} V(E)}_{\text{b}} \quad \lrcorner$$

A feature combination is viable as a feature implementation job iff (a) all of its features can be selected together and, (b) when selected, present requirements that are not already presented by some combination of weaker subsets. If not for these conditions, we would have to ‘implement’ many deltas that do nothing.

Next, we’ll define a much used shorthand notation:

- A.8. Notation:** The *viable feature combination order* $\twoheadrightarrow_v \subseteq \mathcal{F} \times \mathcal{F}$ is the feature combination order (Definition 7.3) restricted to viable feature combinations:
 $\twoheadrightarrow_v \stackrel{\text{def}}{=} \twoheadrightarrow \cap \text{vf}^2$ \lrcorner

Now for the inference rule itself:

- **A.9. Definition (Workflow Inference Rule):**

NEW-FEATURE-JOB

$$\frac{\underbrace{\text{vf}(F)}_{\text{a}} \quad \underbrace{\forall E \twoheadrightarrow_v F: J(E) \in D}_{\text{b}}}{\langle D, \prec, \gamma, J[F \mapsto \perp] \rangle \xrightarrow{F \mapsto} \langle D, \prec, \gamma, J[F \mapsto \text{av}] \rangle} \quad \lrcorner$$

To recognize F as a valid job from the current state, it needs (a) to be viable, and (b) any weaker viable feature combinations must be already implemented. That way, functionality is implemented in the proper order.

We need this particular ordering because deltas implementing stronger feature combinations should have knowledge and control over deltas implementing weaker ones. After writing deltas to implement the features SH , EC and KM from Section 4.5.1, for example, we would like the resultant annotated delta model to look like the one in Figure 4.3. We need the $\{SH, EC, KM\}$ job to be available only after the ‘smaller’ jobs are finished. This is assuming the general case that each combination needs special consideration. In simpler cases, parametric deltas may be used (which were the topic of Section 4.5), but that is out of scope for the workflow description of this chapter.

Now to define an inference rule for identifying conflict resolution jobs. Formally, a conflict occurs between two deltas, as discussed in Section 3.3. However, when there is a set of deltas with many (related) conflicts, we will also want to introduce conflict-resolving deltas for some larger sets, if they have a non-empty joint application condition, in order to cover all combinations. We again define a predicate to help us:

- **A.10. Definition (Viable Conflict Set):** Given an annotated delta model $adm = (D, \prec, \gamma)$, a set $C \subseteq D$ is a *viable conflict set* iff:

$$vc(C) \stackrel{\text{def}}{\iff} \underbrace{\gamma_{\cap}(C) \neq \emptyset}_{\text{a}} \wedge \underbrace{\forall x \in C: \exists y \in C: x \not\prec y \wedge \nexists z \in D: \gamma_{\cap}(\{x, y\}) \subseteq \gamma(z) \wedge (x, y) \triangleleft z}_{\text{b}}$$

If the delta model is not clear from context, we attach a subscript as in vc_{adm} or vc_{\prec} . \lrcorner

A delta set is a viable as a conflict resolution job iff (a) all of its deltas can be selected together (Definition 4.12), and (b) all are in unresolved conflict with at least one other delta in the set. Now, the inference rule itself:

- **A.11. Definition (Workflow Inference Rule):**

NEW-CONFLICT-JOB

$$\frac{\overbrace{vc(C)}^{\text{a}} \quad \overbrace{vc(C') \Rightarrow \gamma_{\cap}(C') \not\subseteq \gamma_{\cap}(C)}^{\text{b}} \wedge \overbrace{\gamma_{\cap}(C') = \gamma_{\cap}(C) \Rightarrow C' \subseteq C}^{\text{c}}}{\langle D, \prec, \gamma, J[C \mapsto \perp] \rangle \xrightarrow{C_1} \langle D, \prec, \gamma, J[C \mapsto \text{av}] \rangle}$$

 \lrcorner

To recognize C as a valid new job, it (a) needs to be viable, (b) may not have a stronger joint application condition than any other viable set and (c) must be the largest of all viable sets that share the same joint application condition. Condition (b) ensures that more generally applicable conflict resolvers are developed first. Condition (c) ensures that no duplicate work is performed, and that the workflow eventually terminates (Appendix A.4.1).

Starting a Job

Starting a job is the simplest inference rule, but having it is important to make the possibility of concurrent development explicit, i.e., that more than one job can be in progress at the same time.

- **A.12. Definition (Workflow Inference Rule):**

STARTING-JOB

$$\frac{}{\langle adm, J[j \mapsto \text{av}] \rangle \xrightarrow{j_2} \langle adm, J[j \mapsto \text{ip}] \rangle}$$

 \lrcorner

When a job is started, its status is simply set from ‘available’ to ‘in progress’, to prevent a job from being started more than once.

Finishing a Job

We now present the final two inference rules, responsible for validating the correctness of a developed delta and integrating it into the annotated delta model. First, the rule for finishing a feature implementation job:

► **A.13. Definition (Workflow Inference Rule):**

FINISHING-FEATURE-JOB

$$\begin{array}{c}
\overbrace{\quad\quad\quad}^{\text{a}} \quad \overbrace{\quad\quad\quad}^{\text{c}} \\
\frac{\prec_{\Delta} = \{ (J(E), d) \mid E \twoheadrightarrow_v F \} \quad (\forall E \twoheadrightarrow_v F: \llbracket \downarrow J(E) \rrbracket(c) \subseteq V(E)) \Rightarrow \llbracket \downarrow d \rrbracket(c) \subseteq V(F)}{\begin{array}{c} \langle \underbrace{D \setminus \{d\}}_{\text{b}}, \prec, \underbrace{\gamma}_{\text{a}}, \underbrace{J[F \mapsto \text{ip}]}_{\text{e}} \rangle \xrightarrow{F_3} \\ \langle \underbrace{D \cup \{d\}}_{\text{b}}, \prec \cup \prec_{\Delta}, \underbrace{\gamma[d \mapsto \{F' \in \Phi \mid F \subseteq F'\}]}_{\text{d}}, \underbrace{J[F \mapsto d]}_{\text{e}} \rangle \end{array}} \\
\text{J}
\end{array}$$

To implement a feature or feature interaction, a new delta d is developed (a) to be applied later than the deltas that implement weaker viable feature combinations. (b) It is added to the delta set. (c) It needs to have a local delta model that satisfies the requirements of F , with the assumption that all weaker viable feature combinations similarly satisfy their own requirements. (d) It is applied whenever all features in F are selected. Finally, (e) we map the job F to the delta d that implements it. (As you can see, we may use this mapping later to (a) order subsequent feature implementation deltas.)

And last but not least, the rule for finishing a conflict resolution job:

► **A.14. Definition (Workflow Inference Rule):**

FINISHING-CONFLICT-JOB

$$\begin{array}{c}
\overbrace{\quad\quad\quad}^{\text{a}} \quad \overbrace{\quad\quad\quad}^{\text{b}} \\
\frac{\forall x, y \in C: z \cdot y \cdot x = z \cdot x \cdot y \quad \forall x \in C: \llbracket \downarrow z \rrbracket(c) \subseteq \llbracket \downarrow x \rrbracket(c)}{\begin{array}{c} \langle \underbrace{D \setminus \{z\}}_{\text{c}}, \prec, \underbrace{\gamma}_{\text{a}}, \underbrace{J[C \mapsto \text{ip}]}_{\text{e}} \rangle \xrightarrow{C_3} \\ \langle \underbrace{D \cup \{z\}}_{\text{c}}, \prec \cup \{ (d, z) \mid x \preceq d \vee y \preceq d \}, \underbrace{\gamma[z \mapsto \gamma(C)]}_{\text{e}}, \underbrace{J[C \mapsto z]}_{\text{f}} \rangle \end{array}} \\
\text{J}
\end{array}$$

To properly resolve the conflict between all deltas in C , a new delta z is developed that (a) allows all deltas in C to commute and (b) has a resulting local delta model that preserves the requirements that were preserved by each individual delta in C . It is (c) added to the delta set, (d) to be applied later than the conflicting deltas, (e) whenever those are applied too. Finally, (f) we map the job C to the delta z that implements it.

The Full Transition Relation

And that concludes the formulation of the abstract delta modeling workflow:

► **A.15. Definition (DMW Transition Relation):** The transition relation of the delta modeling workflow operational semantics is the smallest relation $\rightarrow \subseteq WS \times WS$ characterized by Definitions A.9, A.11, A.12, A.13 and A.14. J

A.4 Analysis

This section presents proofs of three main theorems about the workflow. First, Appendix A.4.1 proves termination. Then, Appendices A.4.2 and A.4.3 prove that any product line created through the workflow is unambiguous and totally correct with respect to the product line specification that was used as input.

A.4.1 Termination

First, we show that the workflow eventually terminates. This being an operational semantics, the workflow is finished when we reach a configuration that is stuck, i.e., from which there are no valid transitions left to take.

- **A.16. Theorem:** The workflow is guaranteed to terminate, i.e., starting from initial state ws_0 (Definition A.5), there is no infinite transition path: $ws_0 \not\rightarrow^\infty$ (Definition 1.50, page 28).

Proof: Two kinds of job exist. Feature implementation jobs are identified by sets of features, and generated directly from the product line specification (Figure 7.4). Since there are only a finite number of features in a feature model (Notation 4.2 and Definition 4.3), these jobs can never be a source of divergence, even if every possible combination would require a separate delta.

Conflict resolution jobs, however, are generated not from the specification, but from the implementation itself. They add a new conflict resolution delta to the set D . Conflict resolution deltas may cause new conflicts themselves, so there is a potentially infinite source of new conflicts. Figure 7.5 shows the feedback loop in question.

To prove termination we show that \rightarrow is well-founded, but for simplicity we'll only consider conflict resolution jobs:

$$\begin{aligned} \langle D, \prec, \gamma, J \rangle &\xrightarrow{C_1} \langle D, \prec, \gamma, J[C \mapsto \text{av}] \rangle \\ &\xrightarrow{C_2} \langle D, \prec, \gamma, J[C \mapsto \text{ip}] \rangle \\ &\xrightarrow{C_3} \langle D', \prec', \gamma', J[C \mapsto z] \rangle \end{aligned}$$

In particular, we assign a value from a well-founded set to each delta d , and show that the value assigned to z is strictly smaller than that assigned to the deltas $x \in C$ of the conflict-set it resolves. This value is the pair $(\gamma(d), \not\leq(d))$, where

$$d \not\leq d' \iff \gamma(d) = \gamma(d') \wedge d \not\prec d' \wedge d' \not\prec d$$

is a symmetric relation between deltas that share the same application condition and can potentially be in conflict with each other. The pair reduces lexicographically from x to z if either:

- $\gamma(z) \subset \gamma(x)$, i.e., z has a stronger application condition than x , or
- $\gamma(z) = \gamma(x) \wedge \not\leq(z) \subset \not\leq(x)$, i.e., z has an application condition equal to x , but there are strictly fewer deltas it can potentially conflict with.

By Definition A.14, we have $\gamma(z) = \gamma_\cap(C)$, so the application condition of z is always equal to or stronger than that of $x \in C$ (Definition 4.12). If $\gamma(z) \subset \gamma(x)$, we are done. If $\gamma(z) = \gamma(x)$, then we can show that $\not\leq(z) \subset \not\leq(x)$:

- Take a delta $d \not\leq z$.
 - (a) We have $\gamma(d) = \gamma(z) = \gamma(x)$.

- (b) Because $d \not\prec z$ and $x \prec z$ we also have $d \not\prec x$.
 - (c) Assume $x \prec d$. Without loss of generality we can assume d was the result of a conflict resolution job C' with $x \in C'$ (it may in fact be a feature implementation delta, but this can only happen finitely often, so we dismiss it as a source of divergence). This leads to contradiction, as by Definition A.11, C' would be largest conflict set with the same joint application condition, so we'd have $C \subseteq C'$, and by Definition A.14, d would have resolved all conflicts in C already, making it inviable as the current job. So we have $x \not\prec d$.
 - (d) From (a), (b) and (c) we conclude that $\mathcal{S}(z) \subseteq \mathcal{S}(x)$.
 - By Definition A.11, we have $|C| > 1$, so there is clearly at least one delta $y \not\prec x$ with $\neg(y \not\prec z)$. So $\mathcal{S}(z) \neq \mathcal{S}(x)$, and therefore $\mathcal{S}(z) \subset \mathcal{S}(x)$.
- So by this measure, $(\gamma(z), \mathcal{S}(z))$ is strictly smaller than $(\gamma(x), \mathcal{S}(x))$. As there is clearly a smallest value (\emptyset, \emptyset) , the described relation is well-founded, and there cannot be an infinite decreasing chain of conflict resolutions. \square

In short, if any conflict resolving delta is in conflict itself, it requires a new conflict resolving delta with a lower ‘value’, and this can only happen a finite number of times. As an extreme example, there could be one conflict resolving delta z , with $d \prec z$ for all other deltas $d \in D$, giving $\mathcal{S}(z) = \emptyset$.

A.4.2 Unambiguity

The product line implementation PLI resulting from the workflow is supposed to be totally correct with regard to the specification $sPLS$. This is proved in Appendix A.4.3. But first, we need an intermediate result: unambiguity (Definition 4.11, page 104). We prove that every feature configuration gives rise to an unambiguous selected delta model.

- **A.17. Theorem:** Given a stuck configuration of the workflow $\langle D, \prec, \gamma, J \rangle$, the corresponding product line implementation $PLI = (\Phi, c, D, \prec, \gamma)$ is unambiguous.

Proof: We'll prove by contradiction that PLI is *globally unambiguous*, which implies that it is generally unambiguous (Theorem 4.16, page 106).

Assume that PLI is not globally unambiguous. This means that there exists a pair of deltas $x, y \in D$ with all of the following properties:

- They are in conflict: $x \not\prec y$,
- They have a non-empty joint application condition: $\gamma_{\cap}(\{x, y\}) \neq \emptyset$
- There is no delta $z \in D$ such that $\gamma_{\cap}(\{x, y\}) \subseteq \gamma(z)$ and $(x, y) \triangleleft z$

Then by Definition A.10, $\{x, y\}$ is a viable conflict set. Therefore, the inference rule **NEW-CONFLICT-JOB** (Definition A.11) is applicable to configuration $\langle D, \prec, \gamma, J \rangle$, meaning it is not stuck and the workflow is not finished.

This contradiction proves the original statement: a stuck configuration yields a product line implementation that is globally unambiguous, and, therefore, unambiguous as per Definition 4.11. \square

A.4.3 Total Correctness

Finally, we prove that the resulting product line implementation is totally correct w.r.t. the given structured product line specification as defined in Definition 4.20 (page 108):

- **A.18. Theorem:** Given a final (stuck) configuration of the workflow $\langle D, \prec, \gamma, J \rangle$, the corresponding product line implementation $PLI = (\Phi, c, D, \prec, \gamma)$ is totally correct with regard to the given product line specification $sPLS = (\Phi, \Rightarrow, V)$.

Proof: Call the annotated delta model $adm = (D, \prec, \gamma)$.

Take an arbitrary feature configuration $F \in \Phi$. We name the selected delta model $dm_F = (D_F, \prec_F) = adm \upharpoonright F$. We then name the set $VF = \{G \subseteq F \mid \text{vf}(G)\}$ of viable feature combinations that are a subset of F (Definition A.7).

By Definition A.9, each $G \in VF$ becomes a new feature job. Moreover, by Definition A.13a, the job map J is a homomorphism from $(VF, \Rightarrow_v \cap VF^2)$ to (D_F, \prec_F) , i.e., for all feature combinations $G_1, G_2 \subseteq F$:

$$G_1 \Rightarrow_v G_2 \iff J(G_1) \prec_F J(G_2)$$

This is also illustrated in Figures 7.4 and 7.5.

We can prove that $V(F) = \bigcup_{G \in VF} V(G)$. While this is not true in the general case (as stated in Section 4.4.1), it is now true by construction. If it were not, there would need to still be a feature combination $E \subseteq F$ with $\text{vf}(E)$ and $E \notin VF$. But if there was, **NEW-FEATURE-JOB** would still apply, and our ‘final’ configuration would not be stuck. But it is.

We can prove that for all $G \in VF$, we have $\llbracket \downarrow J(G) \rrbracket(c) \subseteq V(G)$, by induction on the strength of G . Both the base and inductive case are proved rather straightforwardly by using Definition A.13c.

That being true, we have $\llbracket dm_F \rrbracket(c) \subseteq V(F)$ if none of the deltas outside of the local delta model $\downarrow J(G)$ break the introduced functionality. There are two kinds of such deltas: deltas d with $J(G) \prec_F d$ and deltas unordered with $J(G)$. The former type cannot interfere: because of Definitions A.14b and A.13c, developers have to obey local constraints not to break features of delta’s before them. The latter type also cannot interfere: because PLI is unambiguous (Theorem A.17), all pairs of deltas $x, y \in D_F$ are either ordered by \prec_F , or they commute in the context of the full derivation $d_1 \cdot y \cdot x \cdot d_2 = d_1 \cdot x \cdot y \cdot d_2 \in \text{derv}(dm_F)$, and we assumed a deltoid with non-interference (Definition A.2).

This leads to our desired result:

$$\forall F \in \Phi: \llbracket adm \upharpoonright F \rrbracket(c) \subseteq V(F) \quad \square$$

Summary

Programming is an activity very prone to human error. As more and more features are implemented in a software system by different programmers, progress will often slow to a crawl. It is all too easy for programmers to lose overview of what their code is doing when it is spread across the code base surrounded by the code of others. This can result in bugs and, inevitably, much time will need to be spent on maintenance. This, in turn, results in more expensive software that takes longer to reach the user.

To prevent a large software system from collapsing under its own complexity, its code needs to be well-structured. Manny Lehman (remembered as the Father of Software Evolution) stated the following as his second law of software evolution:

“As a program is evolved its complexity increases unless work is done to maintain or reduce it.”

Ideally we want all code related to a certain *feature* (sometimes called *concern*) to be grouped together in one module —which is called *feature modularization*— and code belonging to different features not be mixed together — which is called *separation of concerns*. But many concerns cannot be easily captured by existing abstractions. They are known as *cross-cutting concerns*. By their very nature their implementation needs to be spread around the code base, so modularization and separation of concerns are still elusive.

This thesis is about *Abstract Delta Modeling (ADM)*, a formal framework developed to achieve modularity and separation of concerns in software.

The software engineering discipline that has the most to gain from those properties is *Software Product Line Engineering (SPLE)*, a relatively new development. To quote van der Linden, Schmid and Rommes:

“Software product lines represent perhaps the most exciting paradigm shift in software development since the advent of high-level programming languages.”

SPLE is concerned with the development and maintenance of *multiple* software systems at the same time, each possessing a different (but often overlapping) set of features — a form of *mass customisation*. This gives rise to an additional need. It is no longer enough that the code for a given feature is separated and modular; it also need to be *composable* and able to deal

gracefully with the presence or absence of other features. We need to be able to make a selection from a set of available features and have the corresponding software mechanically generated for us — a process known as *automated product derivation*. This is another area where ADM can help out.

This thesis is a product of the European HATS project. It presents a formal foundation for the techniques of *delta modeling*, which was the main approach to variability used by HATS. To do this, it employs (among other things) abstract algebra, modal logic, operational semantics and Mealy machines, and lays the bridges between the different disciplines as we go. Its chapters provide a broad overview of the ADM framework and its possibilities, as well as a number of existing practical applications, laying a foundation for further research and development.

Samenvatting

Programmeren is een zeer foutgevoelige activiteit. Naarmate er in een software systeem meer en meer features geïmplementeerd worden, zal de vooruitgang van dat systeem steeds langzamer worden. Programmeurs verliezen snel het overzicht als hun code over het hele project verspreid is, en omringd door de code van anderen. Hierdoor worden sneller fouten gemaakt, en is het onvermijdelijk dat de meeste programmeertijd in onderhoud gaat zitten. Dit leidt tot duurdere software die later op de markt komt.

Om te voorkomen dat een software systeem bezwijkt onder zijn eigen complexiteit, zal de code een duidelijke structuur moeten volgen. Manny Lehman (herinnerd als de Vader van Software-evolutie) gaf het volgende als zijn tweede wet van software-evolutie:

“Naarmate een programma evolueert zal de complexiteit ervan toenemen, tenzij deze actief gehandhaafd of verminderd wordt.”

Idealiter willen we alle code met betrekking tot een bepaalde *feature* (ook wel *concern* genoemd) samenvoegen tot één module — genaamd *feature modularizatie*— en code die tot verschillende features behoort van elkaar scheiden — genaamd *separation of concerns*. Maar vele features kunnen niet makkelijk uitgedrukt worden in bestaande programmeer-abstracties. Zulke features noemen we *cross-cutting concerns*. Hun implementatie moet nu eenmaal tot naar verschillende locaties in het project verspreid worden. Modularisatie en ‘separation of concerns’ zijn dus niet makkelijk tot stand te brengen.

Dit proefschrift gaat over *Abstract Delta Modeling (ADM)*, een formele beschrijving die ons helpt deze eigenschappen in software te behalen.

De software engineering discipline die hier het meest bij te winnen heeft is *Software Product Line Engineering (SPLE)*, een relatief nieuwe ontwikkeling. Ik citeer van der Linden, Schmid en Rommes:

“Software product lines vertegenwoordigen misschien wel de spannendste paradigmaverschuiving in de software-ontwikkeling sinds de komst van ‘high-level’ programmeertalen.”

SPLE houdt zich bezig met de ontwikkeling en het onderhoud van *meerdere* software systemen tegelijk, elk in bezit van een andere (maar vaak overlap-pende) verzameling van features — een vorm van *mass customization*. Aan

de implementatie van een product line stellen we wel extra eisen. Het voldoet niet meer dat de code van een feature gescheiden en modulair is; het moet ook *componeerbaar* zijn, en goed omgaan met de aanwezigheid of afwezigheid van andere features. We moeten uit een verzameling beschikbare features een selectie kunnen maken, en de bijbehorende software mechanisch voor ons kunnen laten genereren. Dit proces staat bekend als *automated product derivation*. Dit is een ander gebied waarin ADM van dienst kan zijn.

Dit proefschrift komt uit het Europese HATS project. Het representeert een formele basis voor *delta modeling*, de techniek die gekozen was door HATS voor het uitdrukken van software variabiliteit. Voor dit doel gebruikt het (onder andere) abstracte algebra, modale logica, operationele semantiek en Mealy machines, en legt het geleidelijk de bruggen tussen deze verschillende disciplines. De hoofdstukken van het proefschrift geven een breed overzicht van het ADM framework, zowel als de mogelijkheden van dit framework en verscheidene praktische applicaties, en legt hiermee een fundering voor verder onderzoek en ontwikkeling.

Curriculum Vitæ

- 1997–1998 • MBO Praktische Informatiekunde 1 at Stichting Facta
- 1999–2000 • MBO Algemene Ondernemers Vaardigheden at the NHA
- 1998–2004 • Voorbereidend Wetenschappelijk Onderwijs at Alfrink College Zoetermeer
- 2004–2007 • Bachelor of Science in Informatica at Leiden University (*cum laude*)
- 2007–2009 • Master of Science in Computer Science at Leiden University (*cum laude*)
- 2009–2014 ◦ PhD in Computer Science at the CWI and Leiden University

Index

A

ABS, *see* Abstract Behavioral Specification
absorption, **23**
Abstract Behavioral Specification, **155**
Abstract Delta Modeling, **6**
accessability relation, **25**
ADM, *see* Abstract Delta Modeling
advice, **5**
AHEAD, **65**
algebra, **24**
 Boolean, **24**
 quotient, **24**
 relation, **24**
algebraic signature, **24**
algebraic structure, **24**
annotated delta model, **103**
annotated delta model closure, **115**
annotative variability, **6**
antisymmetry, **21**
AOP, *see* Aspect Oriented Programming
application condition, **102**
 joint, **104**
application function, **102**
application order, **73**
arity, **23**
around advice, **83**
aspect, **5**
Aspect Oriented Programming, **5**
associativity, **23**
asymmetry, **21**
automated product derivation, **97**
automated profile management, **168**

B

bijectivity, **21**
Boolean algebra, **24**

C

cardinality, **18**
carrier set, **24**
Cartesian product, **19**
 n -ary, **19**
choice axiom, **138**
CIDE, **94**
class, **31**
closure
 annotated delta model, **115**
 delta model, **91**
 reflexive transitive, **21**
 transitive, **21**
coarse-grained, **40**
code view, **206**
codomain, **22**
commutativity, **23**
composition axiom, **138**
compositional variability, **6**
concatenation, sequence, **19**
concern, *see* feature
 cross-cutting, **4**
conclusion, **21**
configuration, **27**
 difference-based, **178**
 minimal, **175**
 stable, **179**
 stuck, **28**
 unstable, **179**
configuration space, **27**
conflict, *see* delta conflict
conflict relation, **76**
conflict resolution job, **154**

conflict resolution relation, **78**
 conflict resolving delta, **78**
 conjunctive delta Kripke frame, **137**
 conjunctive delta model evaluation,
 88
 conjunctive semantics, **88**
 consensus axiom, **138**
 consistent conflict resolution, **81**
 constant, **23**
 constraint, **170**
 constructivism, **52**
 context-aware program, **194**
 core product, **103**
 correctness
 delta, **45**
 dynamic product line, **177**
 product line, **108**
 cost domain, **189**
 cross-cutting concern, **4**
 current state, **180**

D

decision modeling, **116**
 deeply annotated delta model, **114**
 deeply annotated delta model eval-
 uation, **115**
 delta, **36**
 conflict resolving, **78**
 deterministic, **44**
 empty, **51**
 feature implementation, **153**
 feature interaction, **153**
 fine-grained software, **84**
 fully defined, **44**
 invalid, *see* delta, undefined
 \LaTeX , **122**
 neutral, **50**
 non-deterministic, **44**
 parametric, **111**
 parametric software, **112**
 partially defined, **44**
 semantic, **37**
 semantic equivalence, **48**
 semantic refinement, **47**
 simple, **90**
 software, **39**
 software class, **39**
 software package, **39**
 statement, **84**
 undefined, **44**

delta application, **37**
 delta choice, **51**
 delta commutativity, **50**
 delta composition, **50**
 delta conflict, **76**
 hard, **79**
 three-way, **110**
 delta consensus, **51**
 delta contract, **140**
 delta contract provability, **141**
 delta converse, **53**
 delta correctness, **45**
 delta derivation, **46**
 delta diagram, **73**
 delta evaluation, **37**
 parametric software, **112**
 delta Kripke frame, **137**
 delta Kripke model, **137**
 delta logic, **138**
 delta model, **73**
 annotated, **103**
 axiom $\Delta\cap$, **138**
 axiom $\Delta\cup$, **138**
 deeply annotated, **114**
 flat, **90**
 local, **151**
 nested, **90**
 nesting, **90**
 parametric, **111**
 selected, **103**
 unambiguous, **79**
 delta model closure, **91**
 delta model evaluation, **75**
 deeply annotated, **115**
 parametric, **111**
 delta modeling, **6**
 Delta Modeling Workflow, **149**
 delta oriented programming, **71**
 delta postcondition, **140**
 delta precondition, **140**
 delta specification, **45**
 delta-aware operation, **122**
 deltoid, **37**
 fine-grained software, **85**
 fully expressive, **56**
 functional, **59**
 \LaTeX , **120**
 maximally expressive, **57**
 parametric, **111**
 parametric software, **112**

- partially functional, **59**
- profile, **171**
- quark, **63**
- quotient, **38**
- relational, **58**
- software, **41**
- stone carving, **57**
- deltoid homomorphism, **58**
- deltoid refinement, **58**
- derivation, **74**
 - nesting-aware, **91**
 - unique, **75**
- derivation function, **74**
- derivative module, **78**
- derived delta operator, **181**
- derived profile delta operator, **181**
- desired feature interaction, **107**
- deterministic delta, **44**
- device, **169**
- difference-based configuration, **178**
- direct path, **183**
- discreteness, **21**
- disjunctive delta Kripke frame, **137**
- disjunctive delta model evaluation, **88**
- disjunctive semantics, **88**
- distribution, **23**
- DMW, *see* Delta Modeling Workflow
- DMW transition relation, **215**
- domain, **22**
- DPL Mealy machine, **182**
- dynamic product line, **167**
- dynamic product line correctness, **177**
- dynamic product line implementation, **175**

E

- Editor product line, **7**
- effect specification, **171**
- element, **18**
 - absorbing, **23**
 - identity, **23**
 - maximal, **22**
 - minimal, **22**
- empty delta, **51**
- empty delta axiom, **138**
- empty introduction, **61**
- empty quark, **62**

- empty set, **19**
- environment, **169**
- environmental feature configuration, **175**
- environmental transition, **176**
- equivalence class, **23**
- equivalence relation, **22**
- expansion law, **139**
- extension, **23**

F

- FAS, *see* Fredhopper Access Server
- feature, **4**, **99**
- feature combination order, **155**
- feature configuration, **100**
 - environmental, **175**
 - local, **178**
- feature configuration equivalence, **190**
- feature diagram, **100**
- feature implementation delta, **153**
- feature implementation job, **153**
- feature initialization problem, **82**
- feature interaction, **75**
- feature interaction delta, **153**
- feature locality, *see* feature modularity
- feature model, **100**
- feature modularity, **34**
- feature module, **6**
- feature optionality problem, **70**
- Feature Oriented Programming, **64**
- Feature Oriented Software Development, **5**
- feature structure tree, **60**
- Feature-oriented Domain Analysis, **99**
- FgSD, *see* fine-grained software deltoid
- field, **31**
- fine-grained, **40**
- fine-grained software delta, **84**
- fine-grained software deltoid, **85**
- flat delta model, **90**
- FODA, *see* Feature-oriented Domain Analysis
- forcing relation, **26**
- FOSD, *see* Feature Oriented Software Development
- Fredhopper Access Server, **161**

Fredhopper Access Server product line, **161**
 FST, *see* feature structure tree
 full difference transition function, **183**
 full quark, **62**
 full-definedness, **21**
 fully defined delta, **44**
 fully expressive deltoid, **56**
 function, **22**
 partial, **22**
 updated, **22**
 functional deltoid, **59**
 functionality, **21**

G

generalization, **26**
 global consequence, **26**
 global quark, **62**
 global unambiguity, **105**
 glue code, **78**
 granularity, **40**

H

hard delta conflict, **79**
 Heyting algebra, **205**
 hybrid language, **207**

I

idempotency, **23**
 identifier, **31**
 identity, **23**
 identity modification, **61**
 identity relation, **20**
 image, **20**
 inclusive order, **22**
 incremental application, **37**
 inference rule, **21**
 infinite transition path, **28**
 initial product, **57**
 initial state, **212**
 initialization block, **156**
 injectivity, **21**
 integers, **19**
 introduction, **61**
 introduction equivalence, **61**
 introduction sum, **61**
 invalid delta, *see* undefined delta
 invalid software delta, **43**
 invasive composition, **6, 40**

irreflexivity, **21**

J

job, **153**
 conflict resolution, **154**
 feature implementation, **153**
 job status map, **211**
 joint application condition, **104**

K

Kripke frame, **25**
 delta, **137**
 Kripke model, **25**
 delta, **137**

L

L^AT_EX, **119**
 L^AT_EX deltoid, **120**
 lattice, **52**
 lifter, **78**
 linear extension, **23**
 local consequence, **26**
 local consistency, **181**
 local delta model, **151**
 local feature configuration, **178**
 local quark, **62**
 local transition, **176**

M

magma, **62**
 manual profile, **174**
 mapping, **22**
 mass customization, **4**
 maximal element, **22**
 maximally expressive deltoid, **57**
 Mealy machine, **28**
 DPL, **182**
 Mealy machine transition relation, **180**
 method, **31**
 minimal element, **22**
 minimal transition function, **187**
 mixin, **66**
 modal label, **25**
 modality, **25**
 modification, **61**
 identity, **61**
 modification application, **61**
 modification product, **61**
 modus ponens, **26**

monkey-patching, **119**
 monoid, **24**
 profile delta, **171**
 multimodal language, **25**

N

naive set theory, **18**
 n -ary Cartesian product, **19**
 n -ary operation, **23**
 n -ary relation, **20**
 natural numbers, **19**
 positive, **19**
 nested delta model, **90**
 nested product line, **114**
 nested product line implementation, **115**
 nesting delta model, **90**
 nesting-aware derivation, **91**
 neutral delta, **50**
 neutral delta axiom, **138**
 nominal, **207**
 non-deterministic delta, **44**
 non-interference, **211**
 normal modal logic, **26**
 n th Cartesian power, **19**
 n -tuple, **19**
 nullary modality, **143**
 nullary modality semantics, **143**

O

one-to-oneness, **21**
 open world assumption, **102**
 operation, **23**
 associative, **23**
 commutative, **23**
 distributing, **23**
 idempotent, **23**
 n -ary, **23**
 operator, *see* operation
 optimization problem, **191**
 optional feature problem, **70**
 order, **22**
 overspecification, **70**

P

package, **32**
 pair, **19**
 parametric delta, **111**
 parametric delta model, **111**

parametric delta model evaluation, **111**
 parametric deltoid, **111**
 parametric product line evaluation, **112**
 parametric product line implementation, **112**
 parametric software delta, **112**
 parametric software delta evaluation, **112**
 partial function, **22**
 partial order, **22**
 partially defined delta, **44**
 partially functional deltoid, **59**
 patch, **53**
 pkgloader area, **127**
 pointcut, **5**
 positive natural numbers, **19**
 powerset, **18**
 predicate, **20**
 preimage, **20**
 premise, **21**
 preorder, **22**
 principal ideal, **151**
 product, **36**
 core, **103**
 initial, **57**
 product acceptance, **44**
 product family, *see* product line
 product formula, **136**
 product formula evaluation, **137**
 product line, **97**
 dynamic, **167**
 Editor, **7**
 Fredhopper Access Server, **161**
 Thesis, **120**
 product line correctness, **108**
 product line evaluation, **104**
 parametric, **112**
 product line implementation, **103**
 nested, **115**
 parametric, **112**
 product line implementation, dynamic, **175**
 product line specification, **108**
 structured, **210**
 product line unambiguity, **104**
 profile, **169**
 manual, **174**
 profile delta monoid, **171**

profile deltoid, **171**
 profile feature model, **174**
 program delta, **65**
 proper relation algebra, **49**
 propositional variable, **25**
 provability relation, **27**
 pure delta modeling, **152**
 pure delta oriented programming,
 152

Q

quadruple, **19**
 quantification and weaving, **60**
 quantity, **169**
 quark, **62**
 empty, **62**
 full, **62**
 global, **62**
 local, **62**
 simple, **62**
 quark deltoid, **63**
 quark model, **60**
 quintuple, **19**
 quotient algebra, **24**
 quotient deltoid, **38**
 quotient set, **23**

R

real numbers, **19**
 reconfiguration translation, **193**
 reflexivity, **21**
 relation, **20**
 antisymmetric, **21**
 asymmetric, **21**
 bijective, **21**
 discrete, **21**
 equivalence, **22**
 fully defined, **21**
 functional, **21**
 identity, **20**
 injective, **21**
 irreflexive, **21**
 n -ary, **20**
 one-to-one, **21**
 reflexive, **21**
 surjective, **21**
 symmetric, **21**
 total, **21**
 transitive, **21**
 uniquely defined, **21**

 well defined, **21**
 relation algebra, **24**
 proper, **49**
 relation algebra semantics, **49**
 relation composition, **20**
 relation diagram, **22**
 relation image, **20**
 relation inverse, **20**
 relation preimage, **20**
 relational deltoid, **58**
 rule-set, **172**
 rule-set implementation, **174**

S

satisfaction operator, **207**
 selected delta model, **103**
 self adaptive system, **194**
 semantic delta, **37**
 semantic delta equivalence, **48**
 semantic delta refinement, **47**
 semantics, **38**
 separation of concerns, **35**
 sequence, **19**
 sequence concatenation, **19**
 set, **18**
 set complement, **18**
 set difference, **18**
 set intersection, **18**
 set membership, **18**
 set union, **18**
 setting, **169**
 simple delta, **90**
 simple quark, **62**
 singleton transition function, **185**
 software class delta, **39**
 software delta, **39**
 empty, **54**
 invalid, **43**
 neutral, **54**
 software delta algebra, **56**
 software delta application, **43**
 software delta composition, **55**
 software delta consensus, **54**
 software delta equivalence, **44**
 software deltoid, **41**
 software deltoid evaluation, **41**
 software package delta, **39**
 software product line engineering,
 4
 sole derivation semantics, **75**

soundness, **27**
 SPLE, *see* software product line
 engineering
 stable configuration, **179**
 statement, **31**
 statement delta, **84**
 stone carving deltoid, **57**
 strategy, **180**
 strict order, **22**
 strong completeness, **27**
 structured product line specifica-
 tion, **210**
 stuck configuration, **28**
 subfeature, **150**
 subset, **18**
 superfeature, **150**
 superimposition, **60**
 superset, **18**
 surjectivity, **21**
 symmetric difference, **18**
 symmetric set difference, **18**
 symmetry, **21**
 syntactic delta equivalence, **53**
 syntactic delta refinement, **52**
 syntactic FgSD refinement, **86**
 syntactic software delta refinement,
 54
 syntax, **38**

T

target state, **180**
 test driven development, **107**
 T_EX, **119**
 Thesis product line, **120**
 three-way delta conflict, **110**
 token list, **119**
 totality, **21**
 trait, **66**
 transition, **28**
 environmental, **176**
 local, **176**
 transition function, **182**
 full difference, **183**
 minimal, **187**
 singleton, **185**
 transition relation, **28**
 DMW, **215**
 Mealy machine, **180**
 transitive closure, **21**
 transitivity, **21**

triple, **19**
 tuple, **19**
 type, **31**

U

unambiguity
 delta model, **79**
 unambiguous delta model, **79**
 undefined, **22**
 undefined delta, **44**
 uniform substitution, **26**
 unique derivation, **75**
 unique-definedness, **21**
 unstable configuration, **179**
 updated function, **22**

V

valuation function, **25, 107**
 variability, **5**
 viable conflict set, **214**
 viable feature combination, **213**
 viable feature combination order,
 213

W

well-definedness, **21**
 workflow state, **211**
 world, **25**

Bibliography of My Publications

- [1] D. Clarke, M. Helvensteijn, and I. Schaefer. “Abstract Delta Modeling”. In: 9th international conference on Generative Programming and Component Engineering. Vol. 46. SIGPLAN Notices. Eindhoven, The Netherlands: ACM, Oct. 10, 2010, pp. 13–22. ISBN: 9781-450-3015-4-1.
- [2] D. Clarke, M. Helvensteijn, and I. Schaefer. “Abstract Delta Modeling”. In: *Accepted to MSCS special issue* (2012).
- [3] F. d. Boer, M. Helvensteijn, and J. Winter. “A Modal Logic for Abstract Delta Modeling”. In: 16th International Software Product Line Conference. Vol. 2. SPLC ’12. Salvador, Brazil: ACM, Sept. 2, 2012, pp. 45–52. ISBN: 978-1-4503-1095-6.
- [4] M. Helvensteijn. “Abstract Delta Modeling: My Research Plan”. In: 16th International Software Product Line Conference. Vol. 2. SPLC ’12. Salvador, Brazil: ACM, Sept. 2, 2012, pp. 217–224. ISBN: 978-1-4503-1095-6.
- [5] M. Helvensteijn. “Delta Modeling Workflow”. In: 6th International Workshop on Variability Modelling of Software-intensive Systems. ACM International Conference Proceedings Series. Leipzig, Germany: ACM, 2012, pp. 129–137. ISBN: 978-1-4503-1058-1.
- [6] M. Helvensteijn. “Dynamic Delta Modeling”. In: 16th International Software Product Line Conference. Vol. 2. SPLC ’12. Salvador, Brazil: ACM, Sept. 2, 2012, pp. 127–134. ISBN: 978-1-4503-1095-6.
- [7] M. Helvensteijn, R. Muschevici, and P. Wong. “Delta Modeling in Practice, a Fredhopper Case Study”. In: 6th International Workshop on Variability Modelling of Software-intensive Systems. ACM International Conference Proceedings Series. Leipzig, Germany: ACM, 2012, pp. 139–148. ISBN: 978-1-4503-1058-1.
- [8] R. Hähnle et al. “HATS Abstract Behavioral Specification: The Architectural View”. In: *Formal Methods for Components and Objects*. Ed. by B. Beckert et al. Vol. 7542. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 109–132. ISBN: 978-3-642-35886-9, 978-3-642-35887-6.
- [9] D. Clarke, M. Helvensteijn, and I. Schaefer. *Abstract Delta Modeling (Technical Report)*. CW592. Dept. Computer Sciences, Katholieke Universiteit Leuven, Aug. 2010.

- [10] M. Helvensteijn. “The pkgloader and lt3graph Packages: Toward simple and powerful package management for LaTeX”. In: *TUGboat: The Communications of the TeX Users Group* Volume 35 (Issue 1 2014). Ed. by B. Beeton, pp. 47–51.

Main Bibliography

- [11] *Android*. In collab. with Google. 2008. URL: <http://www.android.com>.
- [12] S. Apel, D. Batory, and M. Rosenmüller. “On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?” In: GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPL). 2006.
- [13] S. Apel and C. Kästner. “An Overview of Feature-Oriented Software Development”. In: *Journal of Object Technology* 8 (5 July 2009), pp. 49–84. ISSN: 1660-1769.
- [14] S. Apel, C. Kästner, and D. Batory. “Program Refactoring Using Functional Aspects”. In: *Proceedings of the 7th international conference on Generative programming and component engineering*. GPCE ’08. New York, NY, USA: ACM, 2008, pp. 161–170. ISBN: 978-1-60558-267-2.
- [15] S. Apel, C. Kästner, and C. Lengauer. “FEATUREHOUSE: Language-Independent, Automated Software Composition”. In: IEEE 31st International Conference on Software Engineering. Vancouver, BC, May 16, 2009, pp. 221–231.
- [16] S. Apel, T. Leich, and G. Saake. “Aspectual Feature Modules”. In: *IEEE Transactions on Software Engineering* 34 (2 2008), pp. 162–180. ISSN: 0098-5589.
- [17] S. Apel et al. “An Algebraic Foundation for Automatic Feature-Based Program Synthesis”. In: *Science of Computer Programming* 75 (11 Nov. 1, 2010), pp. 1022–1047. ISSN: 0167-6423.
- [18] S. Apel et al. “Exploring Feature Interactions in the Wild”. In: (2013).
- [19] S. Apel et al. “Model Superimposition in Software Product Lines”. In: *Theory and Practice of Model Transformations*. 2nd International ICMT Conference. Ed. by R. F. Paige. Vol. 5563. Lecture Notes in Computer Science. Zurich, Switzerland: Springer Berlin Heidelberg, June 29, 2009, pp. 4–19. ISBN: 978-3-642-02407-8, 978-3-642-02408-5.
- [20] Apple. *iOS*. 2007. URL: <http://www.apple.com/ios>.
- [21] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Dordrecht: Springer, 2009. 528 pp. ISBN: 9781848827448 184882744X.

- [22] C. Areces, P. Blackburn, and M. Marx. “Hybrid Logics: Characterization, Interpolation and Complexity”. In: *The Journal of Symbolic Logic* 66 (3 Sept. 2001), p. 977. ISSN: 00224812.
- [23] U. Aßmann. *Invasive Software Composition*. Springer, Feb. 27, 2003. 364 pp. ISBN: 9783540443858.
- [24] L. Baresi and C. Ghezzi. “The Disappearing Boundary Between Development-time and Run-time”. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. New York, NY, USA: ACM, 2010, pp. 17–22. ISBN: 978-1-4503-0427-6.
- [25] D. Batory. “A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite”. In: *Generative and Transformational Techniques in Software Engineering*. Lecture Notes in Computer Science (2006). Ed. by R. Lämmel, J. Saraiva, and J. Visser, pp. 3–35. ISSN: 0302-9743.
- [26] D. Batory. “Feature Models, Grammars, and Propositional Formulas”. In: *Proc. Int’l Software Product Line Conference (SPLC)*. 9th International Software Product Line Conference. Ed. by H. Obbink and K. Pohl. Vol. 3714. Lecture Notes in Computer Science. Rennes, France: Springer Berlin Heidelberg, 2005, pp. 7–20. ISBN: 978-3-540-28936-4, 978-3-540-32064-7.
- [27] D. Batory, D. Benavides, and A. Ruiz-Cortés. “Automated Analyses of Feature Models: Challenges Ahead”. In: *Communications of the ACM* 49 (12 Dec. 2006), pp. 45–47. ISSN: 0001-0782.
- [28] D. Batory and B. J. Geraci. “Composition Validation and Subjectivity in GenVoca Generators”. In: *IEEE Transactions on Software Engineering* 23 (2 1997), pp. 67–82. ISSN: 0098-5589.
- [29] D. Batory, P. Höfner, and J. Kim. “Feature Interactions, Products, and Composition”. In: 10th ACM international conference on Generative Programming and Component Engineering. Vol. 47. GPCE '11. ACM, 2011, pp. 13–22. ISBN: 978-1-4503-0689-8.
- [30] D. Batory and S. O’Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. In: *ACM Transactions on Software Engineering Methodology* 1 (4 Oct. 1992), pp. 355–398. ISSN: 1049-331X.
- [31] D. Batory, J. N. Sarvela, and A. Rauschmayer. “Scaling Step-Wise Refinement”. In: *IEEE Transactions on Software Engineering* 30 (6 2004), pp. 355–371. ISSN: 0098-5589.
- [32] D. Batory and D. Smith. *Finite Map Spaces and Quarks: Algebras of Program Structure*. TR-07-66. Computer Science Department, University of Texas at Austin, 2007.
- [33] D. Batory et al. *Features, Modularity, and Variation Points*. TR-2147. University of Texas at Austin, June 6, 2013.
- [34] G. Bellin et al. “A Term Assignment for Dual Intuitionistic Logic”. In: LICS’05-IMLA’05 Workshop. 2005.
- [35] D. Benavides, S. Segura, and A. Ruiz-Cortés. “Automated Analysis of Feature Models 20 Years Later: a Literature Review”. In: *Information Systems* 35 (6 Sept. 2010), pp. 615–636. ISSN: 0306-4379.

- [36] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. 1st. Texts in Theoretical Computer Science. Springer, June 24, 2004. 472 pp. ISBN: 978-3-662-07964-5, 978-3-540-20854-9, 978-3-642-05880-6.
- [37] L. Bettini, F. Damiani, and I. Schaefer. “Implementing Software Product Lines using Traits”. In: 2010 ACM Symposium on Applied Computing. ACM, 2010, pp. 2096–2102. ISBN: 978-1-60558-639-7.
- [38] G. Biegel and V. Cahill. “A Framework for Developing Mobile, Context-aware Applications”. In: *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004*. Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004. 2004, pp. 361–365.
- [39] P. Blackburn. “Nominal Tense Logic”. In: *Notre Dame Journal of Formal Logic* 34 (1 1993), pp. 56–83.
- [40] P. Blackburn. “Representation, Reasoning, and Relational Structures: a Hybrid Logic Manifesto”. In: *Logic Journal of IGPL* 8 (2000), pp. 339–365.
- [41] P. Blackburn, J. F. A. K. v. Benthem, and F. Wolter, eds. *Handbook of Modal Logic*. Vol. 3. Elsevier, Nov. 3, 2006. 1262 pp. ISBN: 9780080466668.
- [42] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001. 574 pp. ISBN: 9780521527149.
- [43] P. Blackburn and J. Seligman. “Hybrid Languages”. In: *Journal of Logic, Language and Information* 4 (3 Sept. 1, 1995), pp. 251–272. ISSN: 0925-8531, 1572-9583.
- [44] M. M. Bonsangue and J. N. Kok. “The Weakest Precondition Calculus: Recursion and Duality”. In: *Formal Aspects of Computing* 6 (1 Nov. 1, 1994), pp. 788–800. ISSN: 0934-5043, 1433-299X.
- [45] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN: 0-201-67494-7.
- [46] Q. Boucher et al. “Introducing TVL, a Text-based Feature Modelling Language”. In: Fourth International Workshop on Variability Modelling of Software-intensive Systems. Linz, Austria, 2010, pp. 27–29.
- [47] G. Bracha and W. Cook. “Mixin-Based Inheritance”. In: *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications. OOPSLA/E-COOP '90*. New York, NY, USA: ACM, 1990, pp. 303–311. ISBN: 0-89791-411-2.
- [48] G. Canfora et al. “In memory of Manny Lehman, ‘Father of Software Evolution’”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 23 (3 2011), pp. 137–144. ISSN: 1532-0618.
- [49] D. Carlisle. *xii.tex*. Dec. 11, 1998. URL: <http://www.ctan.org/pkg/xii>.

- [50] B. H. C. Cheng et al. “Software Engineering for Self-Adaptive Systems: A Research Roadmap”. In: *Software Engineering for Self-Adaptive Systems*. Ed. by B. H. C. Cheng et al. Lecture Notes in Computer Science 5525. Springer Berlin Heidelberg, 2009, pp. 1–26. ISBN: 978-3-642-02160-2, 978-3-642-02161-9.
- [51] R. Chitchyan, J. Fabry, and L. Bergmans. “Aspects, Dependencies, and Interactions”. In: *Object-Oriented Technology. ECOOP 2006 Workshop Reader*. Ed. by M. Südholt and C. Consel. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 26–39. ISBN: 978-3-540-71772-0, 978-3-540-71774-4.
- [52] D. Clarke et al. “Variability Modelling in the ABS Language”. In: 9th international symposium on formal methods for components and objects (FMCO). Ed. by B. K. Aichernig, F. S. d. Boer, and M. M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science. Graz, Austria: Springer Berlin Heidelberg, 2012, pp. 204–224. ISBN: 978-3-642-25270-9, 978-3-642-25271-6.
- [53] A. Classen, P. Heymans, and P.-Y. Schobbens. “What’s in a Feature: A Requirements Engineering Perspective”. In: *Fundamental Approaches to Software Engineering*. Ed. by J. L. Fiadeiro and P. Inverardi. Lecture Notes in Computer Science 4961. Springer Berlin Heidelberg, 2008, pp. 16–30. ISBN: 978-3-540-78742-6, 978-3-540-78743-3.
- [54] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Boston, 2002.
- [55] Colin Pickard. *product line*. In: July 25, 2011.
- [56] K. Czarnecki and M. Antkiewicz. “Mapping Features to Models: A Template Approach Based on Superimposed Variants”. In: 4th International Conference on Generative Programming and Component Engineering. Ed. by R. Glück and M. Lowry. Vol. 3676. Lecture Notes in Computer Science. Tallinn, Estonia: Springer Berlin Heidelberg, 2005, pp. 422–437. ISBN: 978-3-540-29138-1, 978-3-540-31977-1.
- [57] K. Czarnecki and U. W. Eisenecker. “Generative Programming”. In: Addison-Wesley, 2000, pp. 2–19.
- [58] K. Czarnecki, S. Helsen, and U. Eisenecker. “Staged Configuration Using Feature Models”. In: 3rd International Software Product Line Conference. Ed. by R. L. Nord. Vol. 3154. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 266–283. ISBN: 978-3-540-22918-6, 978-3-540-28630-1.
- [59] K. Czarnecki and C. H. P. Kim. “Cardinality-Based Feature Modeling and Constraints: A Progress Report”. In: International Workshop on Software Factories at OOPSLA’05. 2005, pp. 1–9.
- [60] K. Czarnecki and A. Wasowski. “Feature Diagrams and Logics: There and Back Again”. In: *Proceedings of the 11th International Software Product Line Conference*. 2007, pp. 23–34.

- [61] K. Czarnecki et al. “Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches”. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. 6th International Workshop on Variability Modeling of Software-Intensive Systems. ACM, 2012, pp. 173–182. ISBN: 978-1-4503-1058-1.
- [62] F. Damiani, C. Gladisch, and S. Tyszberowicz. “Refinement-based Testing of Delta-oriented Product Lines”. In: *The 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. PPPJ '13*. New York, NY, USA: ACM, 2013, pp. 135–140. ISBN: 978-1-4503-2111-2.
- [63] F. Damiani, L. Padovani, and I. Schaefer. “A Formal Foundation for Dynamic Delta-oriented Software Product Lines”. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering. GPCE '12*. New York, NY, USA: ACM, 2012, pp. 1–10. ISBN: 978-1-4503-1129-8.
- [64] F. Damiani and I. Schaefer. “Dynamic Delta-Oriented Programming”. In: *15th International Software Product Line Conference*. Vol. 2. Munich, Germany: ACM, Aug. 22, 2011, p. 34. ISBN: 978-1-4503-0789-5.
- [65] S. Deelstra, M. Sinnema, and J. Bosch. “Product Derivation in Software Product Families: A Case Study”. In: *Journal of Systems and Software* 74 (2 2005), pp. 173–194. ISSN: 0164-1212.
- [66] A. van Deursen and P. Klint. “Domain-Specific Language Design Requires Feature Descriptions”. In: *Journal of Computing and Information Technology* 10 (1 2002), pp. 1–17.
- [67] S. Ducasse et al. “Traits: A Mechanism for Fine-Grained Reuse”. In: *ACM Trans. Program. Lang. Syst.* 28 (2 Mar. 2006), pp. 331–388. ISSN: 0164-0925.
- [68] Eclipse Foundation. *AspectJ*. 2001. URL: <http://eclipse.org/aspectj/>.
- [69] M. J. Fischer and R. E. Ladner. “Propositional Dynamic Logic of Regular Programs”. In: *Journal of Computer and System Sciences* 18 (2 Apr. 1979), pp. 194–211. ISSN: 0022-0000.
- [70] Florent Chervet. *The interfaces package for LaTeX*. 2010. URL: <http://www.ctan.org/pkg/interfaces>.
- [71] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. 474 pp. ISBN: 9780201485677.
- [72] G. Gargov and V. Goranko. “Modal Logic with Names”. In: *Journal of Philosophical Logic* 22 (6 Dec. 1, 1993), pp. 607–636. ISSN: 0022-3611, 1573-0433.
- [73] C. Ghezzi and A. M. Sharifloo. “Dealing with Non-Functional Requirements for Adaptive Systems via Dynamic Software Product-Lines”. In: *Software Engineering for Self-Adaptive Systems II*. Ed. by R. d. Lemos et al. Vol. 7475. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 191–213. ISBN: 978-3-642-35812-8, 978-3-642-35813-5.

- [74] d. G. Giuseppe. “Eliminating ”Converse” from Converse PDL”. In: *Journal of Logic, Language and Information* 5 (2 Apr. 1, 1996), pp. 193–208. ISSN: 0925-8531, 1572-9583.
- [75] M. L. Griss. “Implementing Product-Line Features by Composing Aspects”. In: the First Conference on Software Product Lines. Ed. by P. Donohoe. The Springer International Series in Engineering and Computer Science. Springer, 2000, pp. 271–288. ISBN: 978-1-4613-6949-3, 978-1-4615-4339-8.
- [76] A. Haber et al. “Engineering Delta Modeling Languages”. In: *Proceedings of the 17th International Software Product Line Conference*. SPLC ’13. ACM, 2013, pp. 22–31. ISBN: 978-1-4503-1968-3.
- [77] A. Haber et al. “Evolving Delta-Oriented Software Product Line Architectures”. In: *Large-Scale Complex IT Systems. Development, Operation and Management*. Ed. by R. Calinescu and D. Garlan. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 183–208. ISBN: 978-3-642-34058-1, 978-3-642-34059-8.
- [78] A. Haber et al. “First-Class Variability Modeling in Matlab/Simulink”. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS ’13. ACM, 2013. ISBN: 978-1-4503-1541-8.
- [79] A. Haber et al. “Towards a Family-based Analysis of Applicability Conditions in Architectural Delta Models”. In: VARY International Workshop affiliated with ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems. 2011, pp. 43–52.
- [80] R. Hähnle. “HATS: Highly Adaptable and Trustworthy Software Using Formal Methods”. In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by T. Margaria and B. Steffen. Vol. 6416. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 3–8. ISBN: 978-3-642-16560-3, 978-3-642-16561-0.
- [81] R. Hähnle and I. Schaefer. “A Liskov Principle for Delta-Oriented Programming”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by T. Margaria and B. Steffen. Vol. 7609. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 32–46. ISBN: 978-3-642-34025-3, 978-3-642-34026-0.
- [82] S. Hallsteinsen et al. “Dynamic Software Product Lines”. In: *Computer* 41 (4 Apr. 2008), pp. 93–95. ISSN: 0018-9162.
- [83] P. R. Halmos. *Naive Set Theory*. Springer, 1960. 120 pp. ISBN: 9780387900926.
- [84] F. Heidenreich and C. Wende. “Bridging the Gap Between Features and Models”. In: Aspect-Oriented Product Line Engineering (AOPLE’07). 2007.
- [85] M. Helvensteijn. *Delta Profiles*. 2013. URL: <http://code.google.com/p/delta-profiles>.
- [86] M. Helvensteijn. *The lt3graph package for LaTeX3*. 2013. URL: <http://www.ctan.org/pkg/lt3graph>.

- [87] M. Helvensteijn. *The withargs package for LaTeX*. 2013. URL: <http://www.ctan.org/pkg/withargs>.
- [88] M. Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. Wiley (Chichester England and New York), 1990.
- [89] R. E. Herrejon. “Understanding Feature Modularity”. University of Texas at Austin, 2006.
- [90] R. E. Herrejon, D. Batory, and W. Cook. “Evaluating Support for Features in Advanced Modularization Technologies”. In: ECOOP 2005 - Object-Oriented Programming. Ed. by A. P. Black. Vol. 3586. Lecture Notes in Computer Science. Springer, July 25, 2005, pp. 169–194. ISBN: 978-3-540-27992-1, 978-3-540-31725-8.
- [91] P. Heymans et al. “Evaluating Formal Properties of Feature Diagram Languages”. In: *IET Software* 2 (3 2008), pp. 281–302. ISSN: 1751-8806.
- [92] A. Heyting. *Intuitionism: An Introduction*. Vol. 41. Access Online via Elsevier, 1966.
- [93] C. Hoare. “A Calculus of Total Correctness for Communicating Processes”. In: *Science of Computer Programming* 1 (1–2 Oct. 1981), pp. 49–72. ISSN: 0167-6423.
- [94] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12 (10 Oct. 1969), pp. 576–580. ISSN: 0001-0782.
- [95] D. R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. New York: Basic Books, 1999. ISBN: 0394756827 9780394756820 0465026567 9780465026562.
- [96] W. L. Hürsch and C. V. Lopes. *Separation of Concerns*. College of Computer Science, Northeastern University, 1995.
- [97] J. Jacobson. *A Formalization of Darcs Patch Theory using Inverse Semigroups*. CAM report 09-83. UCLA, 2009.
- [98] N. Jacobson. *Lectures in Abstract Algebra*. Vol. 1. van Nostrand New York, 1951.
- [99] JetBrains. *IntelliJ IDEA*. Version 13. 2014. URL: <http://www.jetbrains.com/idea/>.
- [100] E. B. Johnsen et al. “ABS: A Core Language for Abstract Behavioral Specification”. In: *Formal Methods for Components and Objects*. 9th International Symposium on Formal Methods for Components and Objects. Ed. by B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue. Vol. 6957. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 142–164. ISBN: 978-3-642-25270-9, 978-3-642-25271-6.
- [101] B. Jónnson and A. Tarski. “Boolean Algebras with Operators, Part II”. In: *American Journal of Mathematics* 74 (1 1952), pp. 127–162. ISSN: 00029327.
- [102] B. Jónsson and A. Tarski. “Representation Problems for Relation Algebras”. In: *Bull. Amer. Math. Soc* 54 (80 1948), p. 1192.

- [103] C. Julien and G.-C. Roman. “Egocentric Context-aware Programming in Ad Hoc Mobile Environments”. In: *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. SIGSOFT ’02/FSE-10. New York, NY, USA: ACM, 2002, pp. 21–30. ISBN: 1-58113-514-9.
- [104] K. C. Kang, J. Lee, and P. Donohoe. “Feature-Oriented Product Line Engineering”. In: *IEEE Software* 5 (4 2002), pp. 58–65. ISSN: 0740-7459.
- [105] K. C. Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. SEI-90-TR-21. Carnegie Mellon University, 1990.
- [106] C. Kästner and S. Apel. “Type-Checking Software Product Lines - A Formal Approach”. In: 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2008, pp. 258–267.
- [107] C. Kästner, S. Apel, and D. Batory. “A Case Study Implementing Features Using AspectJ”. In: 11th International Software Product Line Conference. IEEE, Sept. 10, 2007, pp. 223–232.
- [108] C. Kästner, S. Apel, and M. Kuhlemann. “Granularity in Software Product Lines”. In: 30th international conference on Software Engineering. ACM, 2008, pp. 311–320. ISBN: 978-1-60558-079-1.
- [109] C. Kästner, S. Apel, and K. Ostermann. “The Road to Feature Modularity?”. In: 15th International Software Product Line Conference. SPLC ’11. ACM, 2011, 5:1–5:8. ISBN: 978-1-4503-0789-5.
- [110] C. Kästner et al. “FeatureIDE: A Tool Framework for Feature-Oriented Software Development”. In: IEEE 31st International Conference on Software Engineering, 2009, pp. 611–614.
- [111] C. Kästner et al. “On the Impact of the Optional Feature Problem: Analysis and Case Studies”. In: The 13th International Software Product Line Conference. 2009, pp. 181–190.
- [112] G. Kiczales and M. Mezini. “Separation of Concerns with Procedures, Annotations, Advice and Pointcuts”. In: ECOOP 2005 - Object-Oriented Programming. Ed. by A. P. Black. Vol. 3586. Lecture Notes in Computer Science. Springer, 2005, pp. 195–213. ISBN: 978-3-540-27992-1, 978-3-540-31725-8.
- [113] G. Kiczales et al. “An Overview of AspectJ”. In: *ECOOP 2001 — Object-Oriented Programming*. ECOOP 2001 — Object-Oriented Programming. Ed. by J. L. Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 327–354. ISBN: 978-3-540-42206-8, 978-3-540-45337-6.
- [114] G. Kiczales et al. “Aspect-Oriented Programming”. In: ECOOP’97 — Object-Oriented Programming. Ed. by M. Akşit and S. Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 220–242. ISBN: 978-3-540-63089-0, 978-3-540-69127-3.
- [115] D. E. Knuth. *The TeXbook*. In collab. with D. Bibby. Vol. 1993. Addison-Wesley Reading, MA, USA, 1986.
- [116] D. E. Knuth. *The Art of Computer Programming*. 3rd ed. Vol. Volume 1 (Fundamental Algorithms). Reading, Mass.: Addison-Wesley, July 7, 1997. 672 pp. ISBN: 978-0201896831.

- [117] C. W. Krueger. “New Methods in Software Product Line Development”. In: 10th International Software Product Line Conference. 2006, pp. 95–102.
- [118] L. Lamport. *LATEX: A Document Preparation System*. Pearson Education, Sept. 1, 1994. 292 pp. ISBN: 9788177584141.
- [119] M. M. Lehman. “Laws of Software Evolution Revisited”. In: *Software Process Technology*. Ed. by C. Montangero. Lecture Notes in Computer Science 1149. Springer, 1996, pp. 108–124. ISBN: 978-3-540-61771-6, 978-3-540-70676-2.
- [120] M. Lienhardt and D. Clarke. “Row Types for Delta-Oriented Programming”. In: 6th International Workshop on Variability Modeling of Software-Intensive Systems. ACM, 2012, pp. 121–128. ISBN: 978-1-4503-1058-1.
- [121] M. Lienhardt and D. Clarke. “Conflict Detection in Delta-Oriented Programming”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change*. 5th international conference on Leveraging Applications of Formal Methods, Verification and Validation. Ed. by T. Margaria and B. Steffen. Vol. 7609. Lecture Notes in Computer Science. Heraklion, Crete, Greece: Springer Berlin Heidelberg, Oct. 15, 2012, pp. 178–192. ISBN: 978-3-642-34025-3, 978-3-642-34026-0.
- [122] F. v. d. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. Berlin: Springer, 2010. 333 pp. ISBN: 9783642090615 3642090613 9783540714378 3540714375.
- [123] S. Lipschutz and M. Lipson. *Schaum’s Outline of Discrete Mathematics*. Revised Third Edition. New York; London: Schaum, 2007. ISBN: 9780071615860 0071615865.
- [124] J. Liu, D. S. Batory, and S. Nedunuri. “Modeling Interactions in Feature Oriented Software Designs”. In: *FIW (2005)*, pp. 178–197.
- [125] J. Liu, D. Batory, and C. Lengauer. “Feature Oriented Refactoring of Legacy Applications”. In: 28th international conference on Software Engineering. ICSE ’06. ACM, 2006, pp. 112–121. ISBN: 1-59593-375-1.
- [126] N. Loughran and A. Rashid. “Framed Aspects: Supporting Variability and Configurability for AOP”. In: *Software Reuse: Methods, Techniques, and Tools*. 8th International ICSR Conference. Ed. by J. Bosch and C. Krueger. Vol. 3107. Lecture Notes in Computer Science. Madrid, Spain: Springer, July 5, 2004, pp. 127–140. ISBN: 978-3-540-22335-1, 978-3-540-27799-6.
- [127] R. C. Lyndon. “The Representation of Relational Algebras”. In: *Annals of Mathematics* 51 (3 May 1, 1950), pp. 707–729. ISSN: 0003-486X.
- [128] R. Maddux. “Some Sufficient Conditions for the Representability of Relation Algebras”. In: *Algebra Universalis* 8 (1 1978), pp. 162–172.
- [129] A. Marot and R. Wuyts. “Composing Aspects with Aspects”. In: 9th International Conference on Aspect-Oriented Software Development. AOSD ’10. ACM, 2010, pp. 157–168. ISBN: 978-1-60558-958-9.

- [130] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445.
- [131] G. Mealy. “A Method for Synthesizing Sequential Circuits”. In: *Bell System Technical Journal* 34 (5 1955), pp. 1045–1079.
- [132] T. Mens, G. Taentzer, and O. Runge. “Detecting Structural Refactoring Conflicts Using Critical Pair Analysis”. In: *Electronic Notes in Theoretical Computer Science* 127 (3 Apr. 11, 2005), pp. 113–128. ISSN: 1571-0661.
- [133] M. Mezini and K. Ostermann. “Variability Management with Feature-Oriented Programming and Aspects”. In: *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*. SIGSOFT ’04/FSE-12. ACM, 2004, pp. 127–136. ISBN: 1-58113-855-5.
- [134] Microsoft. *Windows Phone*. 2010. URL: <http://www.microsoft.com/windowsphone>.
- [135] A. Mikhajlova and E. Sekerinski. “Class Refinement and Interface Refinement in Object-Oriented Programs”. In: *FME’97: Industrial Applications and Strengthened Foundations of Formal Methods*. 4th International Symposium of Formal Methods Europe. Ed. by J. Fitzgerald, C. B. Jones, and P. Lucas. Vol. 1313. Lecture Notes in Computer Science. Graz, Austria: Springer Berlin Heidelberg, Sept. 15, 1997, pp. 82–101. ISBN: 978-3-540-63533-8, 978-3-540-69593-6.
- [136] R. Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer Verlag, 1980. ISBN: 3-540-10235-3.
- [137] F. Mittelbach and C. Rowley. “The LATEX3 project”. In: *MAPS 93* (1993), pp. 95–99.
- [138] *Monkey patch*. In: *Wikipedia, the free encyclopedia*. In collab. with Wikipedia. 2014.
- [139] B. Moore et al. *Eclipse Development*. IBM Corporation, International Technical Support Organization, 2004.
- [140] S. Mosser et al. “Using Domain Features to Handle Feature Interactions”. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS ’12. ACM, 2012, pp. 101–110. ISBN: 978-1-4503-1058-1.
- [141] R. Muschevici, D. Clarke, and J. Proença. “Executable Modelling of Dynamic Software Product Lines in the ABS Language”. In: *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*. FOSD ’13. New York, NY, USA: ACM, 2013, pp. 17–24. ISBN: 978-1-4503-2168-6.
- [142] E. Y. Nakagawa, M. Becker, and J. C. Maldonado. “Towards a Process to Design Product Line Architectures Based on Reference Architectures”. In: *Proceedings of the 17th International Software Product Line Conference*. SPLC ’13. New York, NY, USA: ACM, 2013, pp. 157–161. ISBN: 978-1-4503-1968-3.

- [143] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. “Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL (T)”. In: *Journal of the ACM* 53 (6 Nov. 2006), pp. 937–977. ISSN: 0004-5411.
- [144] N. Noda and T. Kishi. “Aspect-Oriented Modeling for Variability Management”. In: 12th International Software Product Line Conference. Sept. 8, 2008, pp. 213–222.
- [145] J. Oldevik, Ø. Haugen, and B. Møller-Pedersen. “Confluence in Domain-Independent Product Line Transformations”. In: 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. Ed. by M. Chechik and M. Wirsing. Vol. 5503. Lecture Notes in Computer Science. York, UK: Springer, Mar. 22, 2009, pp. 34–48. ISBN: 978-3-642-00592-3, 978-3-642-00593-0.
- [146] P. Oreizy et al. “An Architecture-based Approach to Self-adaptive Software”. In: *Intelligent Systems and Their Applications, IEEE* 14 (3 1999), pp. 54–62. ISSN: 1094-7167.
- [147] H. Ossher and P. Tarr. “Multi-Dimensional Separation of Concerns and the Hyperspace Approach”. In: *Software Architectures and Component Technology*. Ed. by M. Aksit. The Springer International Series in Engineering and Computer Science 648. Springer US, 2002, pp. 293–323. ISBN: 978-1-4613-5286-0, 978-1-4615-0883-0.
- [148] T. Panas, J. Andersson, and U. Aßmann. “The Editing Aspect of Aspects”. In: *Software Engineering and Applications*. ACTA Press, 2002.
- [149] S. Passy and T. Tinchev. “An Essay in Combinatory Dynamic Logic”. In: *Information and Computation* 93 (2 Aug. 1991), pp. 263–332. ISSN: 0890-5401.
- [150] G. Perrouin et al. “Reconciling Automation and Flexibility in Product Derivation”. In: 12th International Software Product Line Conference. 2008, pp. 339–348.
- [151] B. C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, 2005. 600 pp. ISBN: 9780262162289.
- [152] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 656 pp. ISBN: 9780262162098.
- [153] G. D. Plotkin. *A Structural Approach to Operational Semantics*. DAIMI FN-19. Aarhus University, 1981.
- [154] G. D. Plotkin. “The Origins of Structural Operational Semantics”. In: *The Journal of Logic and Algebraic Programming* 60–61 (July 2004), pp. 3–15. ISSN: 1567-8326.
- [155] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [156] C. Prehofer. “Feature-Oriented Programming: A Fresh Look at Objects”. In: *ECOOP’97 — Object-Oriented Programming*. Ed. by M. Aksit and S. Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer, 1997, pp. 419–443. ISBN: 978-3-540-63089-0, 978-3-540-69127-3.

- [157] *product line*. In: red. by Douglas Harper.
- [158] Rob Kyff. “The Whole Ball Of Facts About Wax”. In: *Hartford Courant* (Apr. 17, 2002).
- [159] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., 1999. ISBN: 0-201-30998-X.
- [160] I. Schaefer. “Variability Modelling for Model-Driven Development of Software Product Lines”. In: 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010). 2010, pp. 85–92.
- [161] I. Schaefer, L. Bettini, and F. Damiani. “Compositional Type-Checking for Delta-oriented Programming”. In: 10th international conference on Aspect-oriented software development. New York, NY, USA: ACM, 2011, pp. 43–56. ISBN: 978-1-4503-0605-8.
- [162] I. Schaefer and F. Damiani. “Pure Delta-Oriented Programming”. In: 2nd International Workshop on Feature-Oriented Software Development. FOSD ’10. ACM, 2010, pp. 49–56. ISBN: 978-1-4503-0208-1.
- [163] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. “A Model-Based Framework for Automated Product Derivation”. In: 1st International Workshop on Model-Driven Approaches in Software Product Line Engineering. Aug. 24, 2009, pp. 14–21.
- [164] I. Schaefer et al. “Delta-Oriented Programming of Software Product Lines”. In: *Software Product Lines: Going Beyond*. 14th International Software Product Line Conference. Ed. by J. Bosch and J. Lee. Vol. 6287. Lecture Notes in Computer Science. Jeju Island, South Korea: Springer, Sept. 13, 2010, pp. 77–91. ISBN: 978-3-642-15578-9, 978-3-642-15579-6.
- [165] N. Schärli et al. “Traits: Composable Units of Behaviour”. In: *ECOOP 2003 – Object-Oriented Programming*. Ed. by L. Cardelli. Vol. 2743. Lecture Notes in Computer Science. Springer, 2003, pp. 248–274. ISBN: 978-3-540-40531-3, 978-3-540-45070-2.
- [166] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. “Feature Diagrams: A Survey and a Formal Semantics”. In: 14th IEEE International Requirements Engineering Conference. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 11, 2006, pp. 139–148.
- [167] P.-Y. Schobbens et al. “Generic Semantics of Feature Diagrams”. In: *Computer Networks* 51 (2 Feb. 7, 2007), pp. 456–479. ISSN: 1389-1286.
- [168] R. Schöpf and R. Fairbairns. *CTAN - Comprehensive TeX Archive Network*. 1992. URL: <http://www.ctan.org>.
- [169] S. Schulze, O. Richers, and I. Schaefer. “Refactoring Delta-Oriented Software Product Lines”. In: *Proceedings of the 12th annual international conference on Aspect-oriented software development*. 2013, pp. 73–84.

- [170] L. Shen et al. “Towards Feature-Oriented Variability Reconfiguration in Dynamic Software Product Lines”. In: *Top Productivity through Software Reuse*. Ed. by K. Schmid. Vol. 6727. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 52–68. ISBN: 978-3-642-21346-5, 978-3-642-21347-2.
- [171] J. Sincero and W. Schröder-Preikschat. “The Linux Kernel Configurator as a Feature Modeling Tool”. In: Software Product Line Conference. 2008, pp. 257–260.
- [172] Y. Smaragdakis and D. Batory. “Mixin layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”. In: *ACM Transactions on Software Engineering and Methodology*. Transactions of Software Engineering Methodology 11 (2 Apr. 2002), pp. 215–255. ISSN: 1049-331X.
- [173] Software Systems Generator Research Group. *AHEAD Tool Suite*. 2002. URL: <http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/>.
- [174] Stan Lee. “Amazing Fantasy #15”. Aug. 10, 1962.
- [175] A. Tarski. “On the Calculus of Relations”. In: *The Journal of Symbolic Logic* 6 (3 Sept. 1941), pp. 73–89. ISSN: 0022-4812, 1943-5886.
- [176] The LaTeX Team. *The expl3 wrapper package for experimental LaTeX3*. 1990. URL: <http://www.ctan.org/pkg/expl3>.
- [177] V. A. Yankov (originator). *Brouwer Lattice*. In: *Encyclopedia of Mathematics*. 2011.
- [178] M. Völter and I. Groher. “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development”. In: 11th International Software Product Line Conference. IEEE Computer Society, 2007, pp. 233–242. ISBN: 0-7695-2888-0.
- [179] S. Warner. *Modern algebra*. New York: Dover Publications, 1990. 818 pp. ISBN: 0486663418.
- [180] D. Weyns et al. “Claims and Supporting Evidence for Self-Adaptive Systems: A Literature Study”. In: 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). 2012, pp. 89–98.
- [181] H. Zhang and S. Jarzabek. “An XVCL-based Approach to Software Product Line Development”. In: 15th International Conference on Software Engineering and Knowledge Engineering. 2003, pp. 267–275.

Titles in the IPA Dissertation Series since 2008

- W. Pieters.** *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01
- A.L. de Groot.** *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02
- M. Bruntink.** *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03
- A.M. Marin.** *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04
- N.C.W.M. Braspenning.** *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05
- M. Bravenboer.** *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06
- M. Torabi Dashti.** *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07
- I.S.M. de Jong.** *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08
- I. Hasuo.** *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09
- L.G.W.A. Cleophas.** *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10
- I.S. Zapreev.** *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11
- M. Farshi.** *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12
- G. Gulesir.** *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

- F.D. Garcia.** *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14
- P. E. A. Dürr.** *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15
- E.M. Bortnik.** *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16
- R.H. Mak.** *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17
- M. van der Horst.** *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18
- C.M. Gray.** *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19
- J.R. Calamé.** *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20
- E. Mumford.** *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21
- E.H. de Graaf.** *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22
- R. Brijder.** *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23
- A. Koprowski.** *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24
- U. Khadim.** *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25
- J. Markovski.** *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26
- H. Kastenbergh.** *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27
- I.R. Buhan.** *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28
- R.S. Marin-Perianu.** *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29
- M.H.G. Verhoef.** *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01
- M. de Mol.** *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02
- M. Lormans.** *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

- M.P.W.J. van Osch.** *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04
- H. Sozer.** *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05
- M.J. van Weerdenburg.** *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06
- H.H. Hansen.** *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07
- A. Mesbah.** *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08
- A.L. Rodriguez Yakushev.** *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9
- K.R. Olmos Joffré.** *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10
- J.A.G.M. van den Berg.** *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11
- M.G. Khatib.** *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12
- S.G.M. Cornelissen.** *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13
- D. Bolzoni.** *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14
- H.L. Jonker.** *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15
- M.R. Czenko.** *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16
- T. Chen.** *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17
- C. Kaliszyk.** *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18
- R.S.S. O'Connor.** *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19
- B. Ploeger.** *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20
- T. Han.** *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21
- R. Li.** *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to*

Medical Image Analysis. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäuser. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

- J. Proença.** *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05
- A. Morali.** *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06
- M. van der Bijl.** *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07
- C. Krause.** *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08
- M.E. Andrés.** *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09
- M. Atif.** *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10
- P.J.A. van Tilburg.** *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11
- Z. Protic.** *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12
- S. Georgievska.** *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13
- S. Malakuti.** *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14
- M. Raffelsieper.** *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15
- C.P. Tsirogiannis.** *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16
- Y.-J. Moon.** *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17
- R. Middelkoop.** *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18
- M.F. van Amstel.** *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19
- A.N. Tamalet.** *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20
- H.J.S. Basten.** *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21
- M. Izadi.** *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22
- L.C.L. Kats.** *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23
- S. Kemper.** *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

- J. Wang.** *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25
- A. Khosravi.** *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01
- A. Middelkoop.** *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02
- Z. Hemel.** *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03
- T. Dimkov.** *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04
- S. Sedghi.** *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05
- F. Heidarian Dehkordi.** *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06
- K. Verbeek.** *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07
- D.E. Nadales Agut.** *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08
- H. Rahmani.** *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09
- S.D. Vermolen.** *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10
- L.J.P. Engelen.** *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11
- F.P.M. Stappers.** *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12
- W. Heijstek.** *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13
- C. Kop.** *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14
- A. Osaiweran.** *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15
- W. Kuijper.** *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16
- H. Beohar.** *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01
- G. Igna.** *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02
- E. Zambon.** *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

- B. Lijnse.** *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04
- G.T. de Koning Gans.** *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05
- M.S. Greiler.** *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06
- L.E. Mamane.** *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07
- M.M.H.P. van den Heuvel.** *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08
- J. Businge.** *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09
- S. van der Burg.** *A Reference Architecture for Distributed Software Deployment.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10
- J.J.A. Keiren.** *Advanced Reduction Techniques for Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2013-11
- D.H.P. Gerrits.** *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points.* Faculty of Mathematics and Computer Science, TU/e. 2013-12
- M. Timmer.** *Efficient Modelling, Generation and Analysis of Markov Automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13
- M.J.M. Roeloffzen.** *Kinetic Data Structures in the Black-Box Model.* Faculty of Mathematics and Computer Science, TU/e. 2013-14
- L. Lensink.** *Applying Formal Methods in Software Development.* Faculty of Science, Mathematics and Computer Science, RU. 2013-15
- C. Tankink.** *Documentation and Formal Mathematics — Web Technology meets Proof Assistants.* Faculty of Science, Mathematics and Computer Science, RU. 2013-16
- C. de Gouw.** *Combining Monitoring with Run-time Assertion Checking.* Faculty of Mathematics and Natural Sciences, UL. 2013-17
- J. van den Bos.** *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics.* Faculty of Science, UvA. 2014-01
- D. Hadziosmanovic.** *The Process Matters: Cyber Security in Industrial Control Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02
- A.J.P. Jeckmans.** *Cryptographically-Enhanced Privacy for Recommender Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03
- C.-P. Bezemer.** *Performance Optimization of Multi-Tenant Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04
- T.M. Ngo.** *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes.* Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems.* Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities.* Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories.* Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond.* Faculty of Mathematics and Natural Sciences, UL. 2014-14