Abstract delta modeling : software product lines and beyond

Helvensteijn, M.

Citation

Helvensteijn, M. (2014, November 12). *Abstract delta modeling : software product lines and beyond*. *IPA Dissertation Series*. Retrieved from https://hdl.handle.net/1887/29661

Version:	Not Applicable (or Unknown)
License:	Leiden University Non-exclusive license
Downloaded from:	https://hdl.handle.net/1887/29661

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

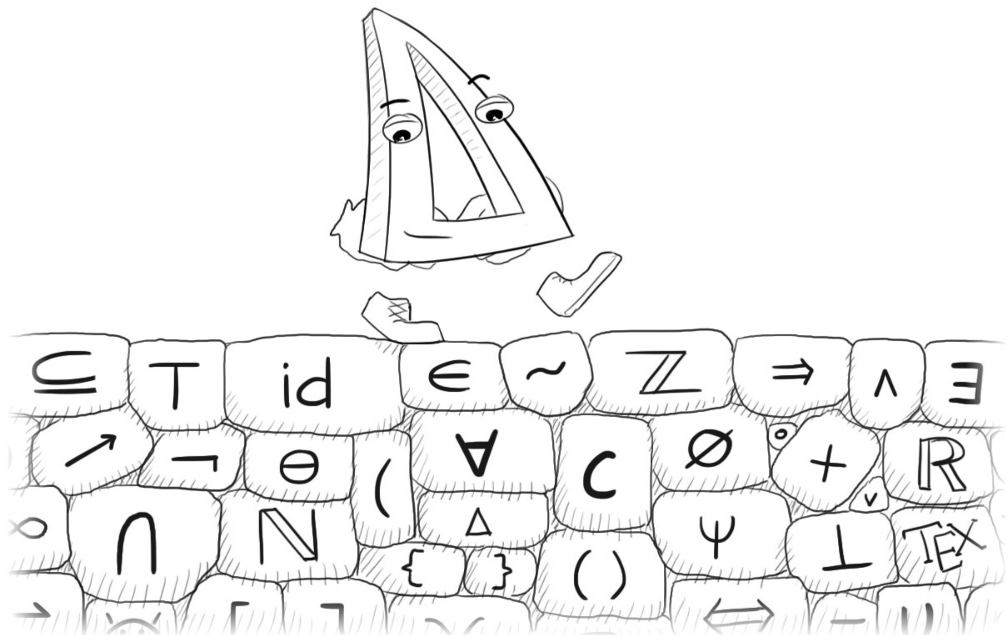
The handle http://hdl.handle.net/1887/29661 holds various files of this Leiden University dissertation

Author: Helvensteijn, Michiel
Title: Abstract delta modeling : software product lines and beyond
Issue Date: 2014-11-12

1

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.

[1] The 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.

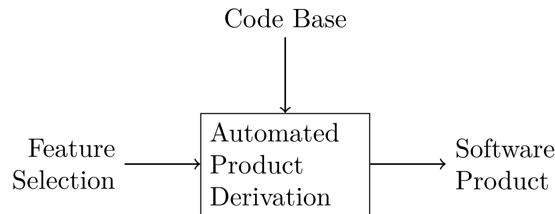


Figure 1.1: The process of Automated Product Derivation.

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 LaTeX. 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.

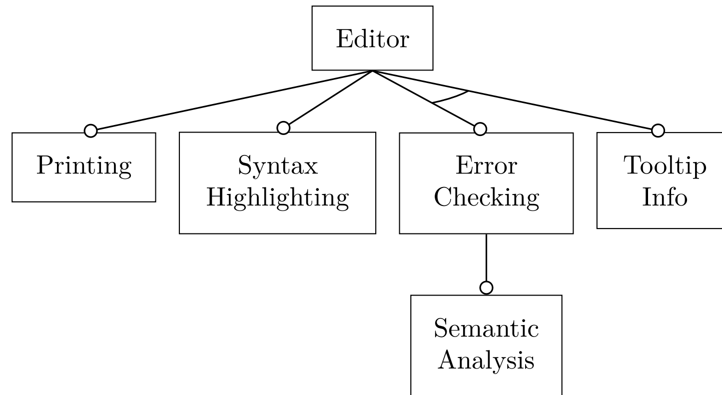


Figure 1.2: The feature diagram of the Editor product line. Each box represents a feature. The base feature (Editor) is mandatory. A connector with an open circle at the end —o indicates an optional subfeature. A subfeature can only be selected if its parent feature is also selected. A curve between two connectors 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.

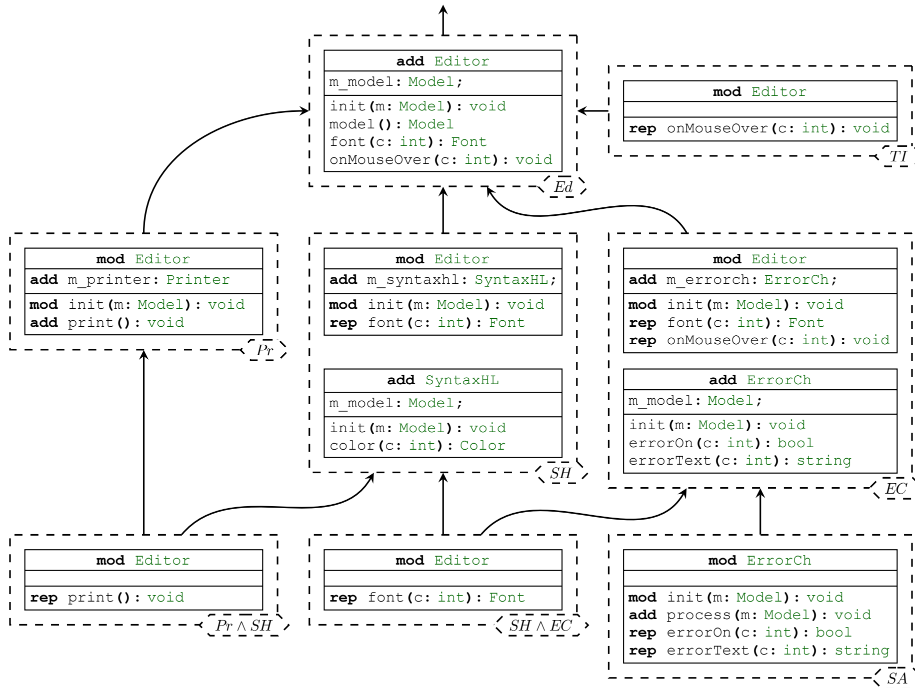


Figure 1.3: A delta diagram of the Editor product line implementation. Each dashed box represents a delta with an overview of the modifications it can apply to the core product in a UML-like notation. Method bodies are omitted for brevity. The arrows ⟶ indicate which deltas are allowed to overwrite or alter the modifications of others. The propositional logic ⟨formula⟩ 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 [⌛1].

[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.

[⧗1] 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.

[⧗2] 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.

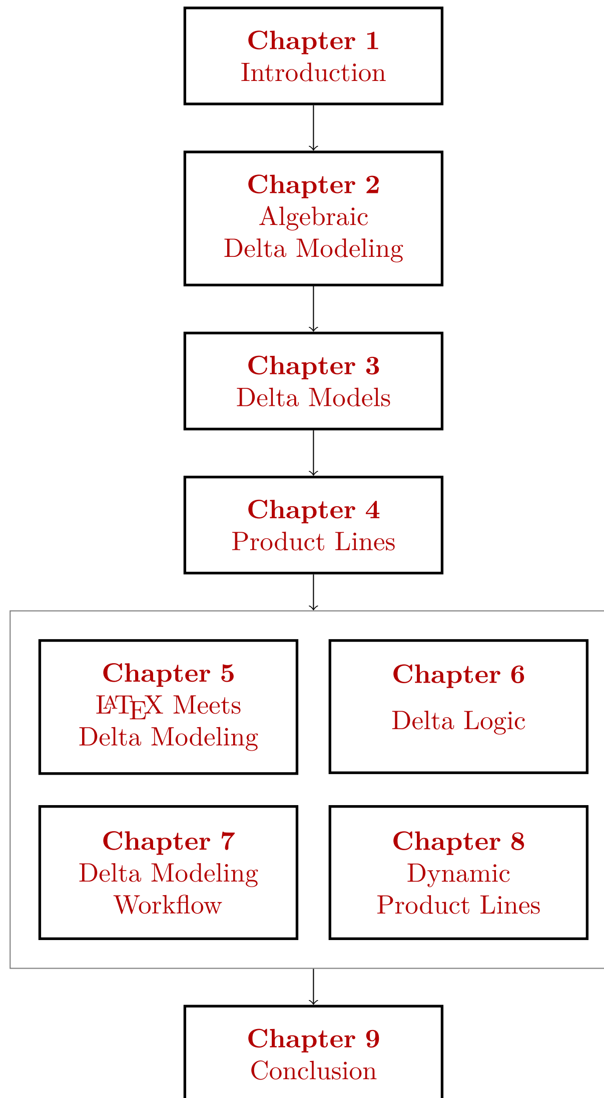


Figure 1.4: The suggested reading order of the thesis. Chapters 5 to 8 can basically be read in any order. Before that, however, each chapter builds upon the theory of its predecessor.

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: LaTeX Meets Delta Modeling

This chapter demonstrates the LaTeX implementation of delta modeling by documenting two new LaTeX-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 LaTeX ecosystem: that of package management. Many document authors suffer from the fact that various LaTeX 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 $e, e_1, \ldots, e_n$, predicate P, relation R and function f:

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

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

$$
\begin{array}{lcl@{\qquad}rcl}
S \subseteq T & \stackrel{\text{def}}{\iff} & \forall e \in S\colon e \in T & S \setminus T & \stackrel{\text{def}}{=} & \{ e \in S \mid e \notin T \} \\
S \subset T & \stackrel{\text{def}}{\iff} & S \subseteq T \wedge S \neq T & S \ominus T & \stackrel{\text{def}}{=} & (S \cup T) \setminus (S \cap T) \\
S \cup T & \stackrel{\text{def}}{=} & \{ e \mid e \in S \vee e \in T \} & \mathrm{Pow}(S) & \stackrel{\text{def}}{=} & \{ S' \mid S' \subseteq S \} \\
S \cap T & \stackrel{\text{def}}{=} & \{ e \mid e \in S \wedge e \in T \} & S^{\complement} & \stackrel{\text{def}}{=} & U \setminus S
\end{array}
$$

A universal set U is assumed to be clear from context when the $S^{\complement}$ 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:

$$
\begin{array}{lcl@{\qquad}rcl}
\bigcup S & \stackrel{\text{def}}{=} & \{ e \mid \exists s \in S\colon e \in s \} & \bigcup_{\mathsf{P}(g)} \mathrm{f}(g) & \stackrel{\text{def}}{=} & \bigcup \{ \mathrm{f}(g) \mid \mathsf{P}(g) \} \\
\bigcap S & \stackrel{\text{def}}{=} & \{ e \mid \forall s \in S\colon e \in s \} & \bigcap_{\mathsf{P}(g)} \mathrm{f}(g) & \stackrel{\text{def}}{=} & \bigcap \{ \mathrm{f}(g) \mid \mathsf{P}(g) \}
\end{array}
$$

where g is a fresh variable name. ⌟

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

$\varnothing$	the empty set
$\mathbb{N}^+$	the positive natural numbers $1, 2, 3, ...$
$\mathbb{N}$	the natural numbers $0, 1, 2, ...$
$\mathbb{Z}$	the integers $..., -2, -1, 0, 1, 2, ...$
$\mathbb{R}$	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:

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

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

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

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

$$S^n \quad \stackrel{\text{def}}{=} \quad \underbrace{S \times \cdots \times S}_{n}$$ ⌟

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:

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

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

$$\overline{e} \frown \overline{g} \quad \stackrel{\text{def}}{=} \quad (e_1, ..., e_n, g_1, ..., g_m)$$

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 $S_1, ..., S_n$ is a subset of their Cartesian product: $\mathsf{R} \subseteq S_1 \times \cdots \times S_n$. A unary relation is also called a *predicate*. ⌟

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

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

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

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

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

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

$$\begin{array}{rcl rcl} e \mathrel{\mathsf{R}} g & \stackrel{\text{def}}{\iff} & (e,g) \in \mathsf{R} & \mathsf{R}(e) & \stackrel{\text{def}}{=} & \{\, g \mid e \mathrel{\mathsf{R}} g \,\} \\ e \mathrel{\not\mathsf{R}} g & \stackrel{\text{def}}{\iff} & \neg(e \mathrel{\mathsf{R}} g) & \mathsf{R}(S') & \stackrel{\text{def}}{=} & \bigcup_{e \in S'} \mathsf{R}(e) \\ e \mathrel{\reflectbox{R}} g & \stackrel{\text{def}}{\iff} & g \mathrel{\mathsf{R}} e & \mathrm{img}(\mathsf{R}) & \stackrel{\text{def}}{=} & \mathsf{R}(S) \\ e \mathrel{\mathsf{R}} g \mathrel{\mathsf{Q}} h & \stackrel{\text{def}}{\iff} & e \mathrel{\mathsf{R}} g \ \wedge\ g \mathrel{\mathsf{Q}} h & \mathrm{pre}(\mathsf{R}) & \stackrel{\text{def}}{=} & \mathsf{R}^{-1}(S) \\ e, g \mathrel{\mathsf{R}} h & \stackrel{\text{def}}{\iff} & e \mathrel{\mathsf{R}} h \ \wedge\ g \mathrel{\mathsf{R}} h & e \mathrel{\not\mathsf{R}} & \stackrel{\text{def}}{\iff} & \mathsf{R}(e) = \varnothing \end{array}$$

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

The notational conventions regarding $\not\mathsf{R}$ and $\reflectbox{$\mathsf{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 $\Leftrightarrow$, are also symmetric in a relational sense (Definition 1.13). ⌟

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

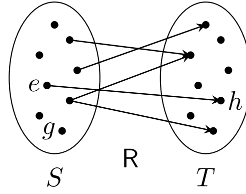


Figure 1.5: The visualization of a specific relation $\mathsf{R} \subseteq S \times T$. The two groups of nodes represent the domain S and codomain T of the relation with $e, g \in S$ and $h \in T$. The arrows represent the relation R itself. They show, for example, that $e \mathrel{\mathsf{R}} h$ and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

uniquely defined:	$\forall e, g, h \in S$:	$e \mathrel{\mathsf{R}} g \wedge e \mathrel{\mathsf{R}} h \Rightarrow g = h$
fully defined:		$\mathrm{pre}(\mathsf{R}) = S$
well defined:		uniquely and fully defined
injective:	$\forall e, g, h \in S$:	$e \mathrel{\mathsf{R}} h \wedge g \mathrel{\mathsf{R}} h \Rightarrow e = g$
surjective:		$\mathrm{img}(\mathsf{R}) = T$
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:	$\forall e \in S$:	$e \mathrel{\mathsf{R}} e$
symmetric:	$\forall e, g \in S$:	$e \mathrel{\mathsf{R}} g \Rightarrow e \mathrel{\text{Я}} g$
transitive:	$\forall e, g, h \in S$:	$e \mathrel{\mathsf{R}} g \wedge g \mathrel{\mathsf{R}} h \Rightarrow e \mathrel{\mathsf{R}} h$
irreflexive:	$\forall e \in S$:	$e \not\mathrel{\mathsf{R}} e$
antisymmetric:	$\forall e, g \in S$:	$e \mathrel{\mathsf{R}} g \wedge e \mathrel{\text{Я}} g \Rightarrow e = g$
asymmetric:	$\forall e, g \in S$:	$e \mathrel{\mathsf{R}} g \Rightarrow e \not\mathrel{\text{Я}} g$
total:	$\forall e, g \in S$:	$e \mathrel{\mathsf{R}} g \vee e \mathrel{\text{Я}} g$
discrete:	$\forall e, g \in S$:	$e \not\mathrel{\mathsf{R}} g \wedge e \not\mathrel{\text{Я}} g$ ⌟

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

$$e \mathrel{\mathsf{R}^+} g \quad \overset{\text{def}}{\Longleftrightarrow} \quad e \mathrel{\mathsf{R}} g \vee \exists h \in S\colon e \mathrel{\mathsf{R}} h \mathrel{\mathsf{R}^+} g$$
$$e \mathrel{\mathsf{R}^*} g \quad \overset{\text{def}}{\Longleftrightarrow} \quad e = g \vee e \mathrel{\mathsf{R}^+} g$$ ⌟

▶ **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:

$$\frac{\langle\textit{premise 1}\rangle \quad \langle\textit{premise 2}\rangle \quad \ldots}{\langle\textit{conclusion}\rangle}$$

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

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*.

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

▶ **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 $\mathrm{f}\colon S \rightharpoonup T$. When $e \notin \mathrm{pre}(\mathrm{f})$ we say that $\mathrm{f}(e)$ is *undefined*, or that $\mathrm{f}(e) = \bot$ (assuming that $\bot \notin T$). Otherwise, the notation is the same as for functions. ⌟

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

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

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

The same notation is defined for partial functions, in which case the update $\mathrm{f}[e \mapsto \bot]$ is also available. ⌟

1.7.6 Transitive Relations

▶ **1.20. Definition (Preorder):** A *preorder* is a transitive and reflexive relation. ⌟

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

▶ **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 $\leq$, $\preccurlyeq$, $\subseteq$, $\sqsubseteq$, ...
- or irreflexive, in which case it is called *strict* and denoted $<$, $\prec$, $\subset$, $\sqsubset$, ...

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

▶ **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 $e \prec g$. ⌟

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

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:

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

▶ **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:

$$S/\!\sim \quad \stackrel{\text{def}}{=} \quad \left\{ [\,e\,]_{\sim} \,\middle|\, e \in S \right\}$$ ⌟

▶ **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. ⌟

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*. ⌟

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

$\star$ is associative:	$e \star (g \star h) \;=\; (e \star g) \star h$
$\star$ is commutative:	$(e \star g) \;=\; (g \star e)$
$\star$ is idempotent:	$e \star e \;=\; e$

With regard to another binary operation $\between\colon S \times S \to S$ we may have:

$\between$ distributes over $\star$:	$e \between (g \star h) \;=\; (e \between g) \star (e \between h)$

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

ϵ is an identity for $\star$:	$\epsilon \star e \;=\; e \;=\; e \star \epsilon$
ϵ absorbs $\star$:	$\epsilon \star e \;=\; \epsilon \;=\; e \star \epsilon$

⌟

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 $(S, \star_1, ..., \star_n)$ —where S is a *carrier set* and $\{\star_1, ..., \star_n\}$ is a set of nullary, unary and binary operators on S. The tuple is also called an *algebraic signature*. ⌟

▶ **1.31. Notation:** We can write S to refer to an algebraic structure $(S, \star_1, ..., \star_n)$ and vice versa, if this does not introduce ambiguity. ⌟

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 ${}^\diamond\colon S \to S$ and binary operators $\star\colon S \times S \to S$, the corresponding *quotient algebra* by equivalence relation $\sim \subseteq S \times S$ would be the algebra with carrier set $S/\sim$ and corresponding operators $[x]_\sim \in (S/\sim)$, ${}^{\diamond_\sim}\colon (S/\sim) \to (S/\sim)$ and $\star_\sim\colon (S/\sim) \times (S/\sim) \to (S/\sim)$, which must satisfy the following for all $e, g \in S$:

$$\begin{aligned} [e]_\sim^{\diamond_\sim} &= [e^\diamond]_\sim \\ [e]_\sim \star_\sim [g]_\sim &= [e \star g]_\sim \end{aligned}$$ ⌟

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

Here follow three common algebraic structures:

▶ **1.33. Definition (Monoid):** A *monoid* is a tuple $(S, \cdot, \varepsilon)$ where $\cdot : S \times S \to S$ 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: $\varepsilon \cdot e = e = e \cdot \varepsilon$ ⌟

▶ **1.34. Definition (Boolean Algebra):** A *Boolean algebra* is a tuple $(S, \sqcup, \sqcap, {}^-, \bot, \top)$ where $\sqcup\colon S \times S \to S$ is a disjunction operator, $\sqcap\colon S \times S \to S$ is a conjunction operator, ${}^-\colon S \to S$ is a negation operator, $\bot \in S$ 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: $e \sqcup \bot = e$

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 $\bot$ with $\top$. ⌟

▶ **1.35. Definition (Relation Algebra):** A *relation algebra* $(S, \sqcup, \sqcap, {}^-, \bot, \top, \cdot, \varepsilon, {}^\smile)$ is a tuple with a monoid $(S, \cdot, \varepsilon)$, a Boolean algebra $(S, \sqcup, \sqcap, {}^-, \bot, \top)$ and a converse operator ${}^\smile : S \to S$. 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.	$\breve{}$ is its own inverse	$e^{\smile\smile} = e$
b.	$\breve{}$ is an anti-involution for $\cdot$	$(e \cdot g)^{\smile} = g^{\smile} \cdot e^{\smile}$
c.	$\cdot$ distributes over $\sqcup$	$e \cdot (g \sqcup h) = (e \cdot g) \sqcup (e \cdot h)$
d.	$\breve{}$ distributes over $\sqcup$	$(e \sqcup g)^{\smile} = e^{\smile} \sqcup g^{\smile}$
e.	Tarski's axiom	$(e^{\smile} \cdot (e \cdot g)^{-}) \sqcup g^{-} = g^{-}$ ⌟

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:

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

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 $\varphi, \psi \in \Psi$:

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

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. ⌟

▶ **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 $R\colon M \rightarrow \text{Pow}(W \times W)$ which maps each modal label $m \in M$ to a corresponding *accessability relation* $\mathsf{R}_m \subseteq W \times W$. ⌟

▶ **1.38. Definition (Valuation Function):** Given a set of worlds W and a set of propositional variables $PROP$, a *valuation function* is a function $V\colon PROP \rightarrow \text{Pow}(W)$ which maps each propositional variable to the set of worlds in which it is true. ⌟

▶ **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 $V\colon PROP \rightarrow \text{Pow}(W)$ is a valuation function. ⌟

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:

$$\begin{array}{lll} w \Vdash \top & \overset{\text{def}}{\Longleftrightarrow} & \text{always} \\ w \Vdash k & \overset{\text{def}}{\Longleftrightarrow} & w \in V(k) \\ w \Vdash \varphi \vee \psi & \overset{\text{def}}{\Longleftrightarrow} & (\, w \Vdash \varphi \,) \ \vee \ (\, w \Vdash \psi \,) \\ w \Vdash \neg\varphi & \overset{\text{def}}{\Longleftrightarrow} & \neg(\, w \Vdash \varphi \,) \\ w \Vdash \langle m \rangle \varphi & \overset{\text{def}}{\Longleftrightarrow} & \exists w' \in W\colon (w, w') \in R(m) \ \wedge \ (\, w' \Vdash \varphi \,) \end{array}$$

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 $\Vdash_{\mathfrak{M}}$. ⌟

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:

$$\begin{array}{rcll} \Vdash_{\mathfrak{M}} \varphi & \overset{\text{def}}{\Longleftrightarrow} & \forall w' \in W\colon & w' \Vdash_{\mathfrak{M}} \varphi \\ w \Vdash_{\mathfrak{F}} \varphi & \overset{\text{def}}{\Longleftrightarrow} & \forall V\colon PROP \to \mathrm{Pow}(W)\colon & w \Vdash_{(\mathfrak{F}, V)} \varphi \\ \Vdash_{\mathfrak{F}} \varphi & \overset{\text{def}}{\Longleftrightarrow} & \forall w' \in W\colon & w' \Vdash_{\mathfrak{F}} \varphi \end{array}$$

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

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 S, we say that $\varphi \in \Psi$ is a *local consequence* or *global consequence* of Γ iff:

$$\begin{array}{lll} \Gamma \Vdash_{\mathsf{S}} \varphi & \overset{\text{def}}{\Longleftrightarrow} & \forall \mathfrak{M} \in \mathsf{S}\colon \forall w \in W\colon (\, \mathfrak{M}, w \Vdash \Gamma \,) \implies (\, \mathfrak{M}, w \Vdash \varphi \,) \\ \Gamma \Vdash^{g}_{\mathsf{S}} \varphi & \overset{\text{def}}{\Longleftrightarrow} & \forall \mathfrak{M} \in \mathsf{S}\colon (\, \forall w \in W\colon \mathfrak{M}, w \Vdash \Gamma \,) \implies (\, \forall w \in W\colon \mathfrak{M}, w \Vdash \varphi \,) \end{array}$$ ⌟

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**: $[m](\varphi \to \psi) \to ([m]\varphi \to [m]\psi) \in \Gamma$
- *the formula* **Dual**: $\langle m \rangle \varphi \leftrightarrow \neg[m]\neg\varphi \in \Gamma$

and is closed by the following properties:

- *modus ponens*: $\varphi,\ \varphi \to \psi \in \Gamma \implies \psi \in \Gamma$
- *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. ⌟

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* $\vdash_\Lambda \subseteq \mathrm{Pow}(\Psi \times \Psi)$ is defined as follows, for all sets of formulas $\Gamma \subseteq \Psi$:

$$\Gamma \vdash_\Lambda \varphi \quad \overset{\text{def}}{\Longleftrightarrow} \quad \exists \psi_1, ..., \psi_n \in \Gamma\colon \; \Big(\bigwedge_{1 \le i \le n} \psi_i \Big) \to \varphi \;\; \in \;\; \Lambda$$

A shorthand notation is defined for provability without premises:

$$\vdash_\Lambda \varphi \quad \overset{\text{def}}{\Longleftrightarrow} \quad \varnothing \vdash_\Lambda \varphi$$ ⌟

▶ **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 S, if for any set of formulas $\Gamma \subseteq \Psi$ and any formula $\varphi \in \Psi$, we have:

$$\Gamma \vdash_\Lambda \psi \quad \Longrightarrow \quad \Gamma \Vdash_{\mathsf{S}} \varphi$$ ⌟

▶ **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 S, if for any set of formulas $\Gamma \subseteq \Psi$ and any formula $\varphi \in \Psi$, we have:

$$\Gamma \Vdash_{\mathsf{S}} \varphi \quad \Longrightarrow \quad \Gamma \vdash_\Lambda \psi$$ ⌟

▶ **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 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:

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

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; $\langle cn \rangle$. ⌟

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

$$\langle\, cn \,\rangle \;\longrightarrow\; \langle\, cn' \,\rangle$$

means that cn' is a possible *next configuration* of cn. We use $\longrightarrow^{+}$ and $\longrightarrow^{*}$ 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 $\langle\, cn \,\rangle \not\longrightarrow$ (Notation 1.12). ⌟

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

$$\langle\, cn \,\rangle \not\longrightarrow^{\infty} \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall cn' \in CS\colon\; \langle\, cn \,\rangle \longrightarrow \langle\, cn' \,\rangle \;\Longrightarrow\; \langle\, cn' \,\rangle \not\longrightarrow^{\infty}$$ ⌟

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:

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

$$\frac{\text{eval}(B,\sigma) = \texttt{false}}{\langle\, \textbf{while}\;\; B \;\;\textbf{do}\;\; st \;\;\textbf{od},\, \sigma \,\rangle \;\longrightarrow\; \langle\, \lambda,\, \sigma \,\rangle}$$ ⌟

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 $(\, S, \Sigma, \Lambda, \mathrm{T}, \mathrm{O} \,)$.

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 $\mathrm{T}\colon S \times \Sigma \rightharpoonup S$ returns the next state and the output function $\mathrm{O}\colon S \times \Sigma \rightharpoonup \Lambda$ returns the corresponding output symbol. The functions O and T are defined for the same inputs, i.e., $\text{pre}(\mathrm{O}) = \text{pre}(\mathrm{T})$. A 'current' state $s \in S$ and input symbol $i \in \Sigma$ together constitute a transition in the machine iff $(s, i) \in \text{pre}(\mathrm{T})$. ⌟

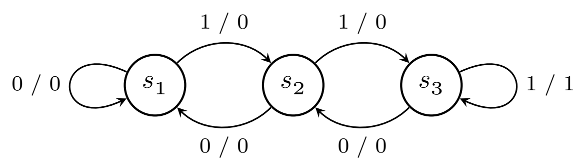


Figure 1.6: An example of a simple Mealy machine with $\Sigma = \Lambda = \{ 0, 1 \}$. Each node represents a state. Each arrow is annotated with i/o: an input and an output symbol.

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.