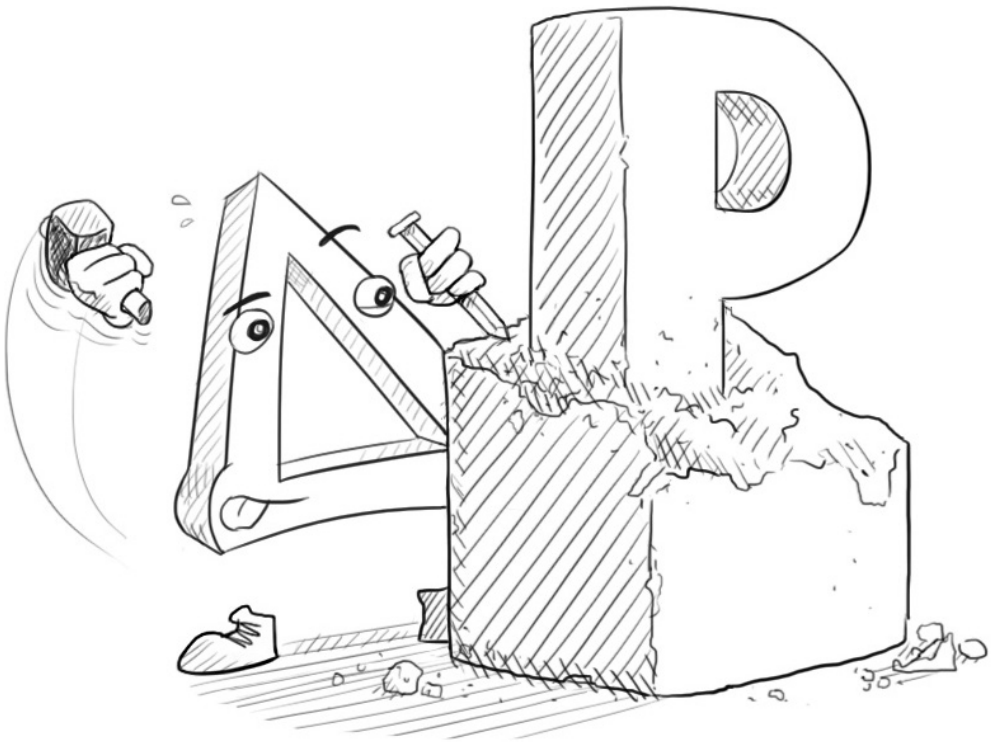
2

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 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:

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

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:

$$cl = \left\{ \begin{array}{l} \text{“}\texttt{m\_name}\text{”} \mapsto \text{“}\texttt{String}\text{”}, \\ \text{“}\texttt{run}\text{”} \quad \mapsto \text{“}\texttt{String[] -> void}\text{”}, \\ \qquad\qquad \left(\begin{array}{l} \text{“}\texttt{m\_name = args[1]}\text{”}, \\ \text{“}\texttt{output("Hello " + m\_name)}\text{”} \end{array} \right) \end{array} \right\}$$

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

```
class {
    m_name: String;
    run(args: String[]) : void {
        m_name = args[1];
        output("Hello " + m_name);
    }
}
```

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 $pkg\colon \mathcal{ID} \rightharpoonup \mathcal{CL}$, 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":

```
package DeltaEditor {
   class Editor {
      m_model: Model;

      init(m : Model) : void {
         m_model = m;
      };

      model() : Model { return m_model; };

      font(c : int) : Font {
         Font result = new Font();
         result.setColor(Color.BLACK);
         result.setBold(false);
         result.setUnderlined(false);
         return result;
      };

      onMouseOver(c : int) : void { };
   };
};
```

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 refered 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 $d_1, ..., d_n$ to a minimal core product c, each changing a specific aspect of it:

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

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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$ with a set of products $\mathcal{P}$, a set of deltas $\mathcal{D}$ and a unary *delta evaluation* operator $[\![\text{-}]\!]\colon \mathcal{D} \to \mathrm{Pow}(\mathcal{P} \times \mathcal{P})$. If $d \in \mathcal{D}$ is a delta, then $[\![\, d \,]\!] \subseteq \mathcal{P} \times \mathcal{P}$ is a binary relation over the set of products, sometimes called a *semantic delta*.

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

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 $\text{-}(\text{-})\colon \mathcal{D} \times \mathcal{P} \to \mathcal{D}$. 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 $\text{-}(\text{-})$ operator of the earlier work:

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

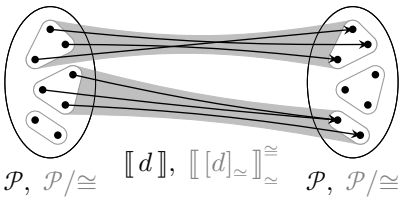


Figure 2.1: A delta d and its equivalence class (in gray) acting on a set of products $\mathcal{P}$ and its quotient set (in gray).

a. $(d, p) \in \text{pre}(\text{apply}) \implies p\ [\![d]\!]\ \text{apply}(d, p)$ ⌟

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 $[\![\text{-}]\!]$ notation, as witnessed in Definition 2.11. In general, we keep to the following convention:

$$[\![\ \langle \text{something syntactic} \rangle\ ]\!] \quad = \quad \langle \text{something semantic} \rangle$$

We introduce several extensions of the $[\![\text{-}]\!]$ 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 $[\![D]\!] \stackrel{\text{def}}{=} \{ [\![d]\!] \mid d \in D \}$. ⌟

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 $[\![\text{-}]\!]$ 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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, the corresponding *quotient deltoid* by equivalence relations $\cong\ \subseteq \mathcal{P} \times \mathcal{P}$ and $\simeq\ \subseteq \mathcal{D} \times \mathcal{D}$, denoted $(\mathcal{P}/\cong, \mathcal{D}/\simeq, [\![\text{-}]\!]^{\cong}_{\simeq})$, exists iff a delta evaluation operator $[\![\text{-}]\!]^{\cong}_{\simeq} : (\mathcal{D}/\simeq) \to \text{Pow}((\mathcal{P}/\cong) \times (\mathcal{P}/\cong))$ exists such that for all deltas $d \in \mathcal{D}$ and products $p, q \in \mathcal{P}$:

$$p\ [\![d]\!]\ q \quad \iff \quad [p]_\cong\ \ [\![[d]_\simeq]\!]^{\cong}_{\simeq}\ \ [q]_\cong$$ ⌟

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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$ may be used iff for all products $p_1, p_2, q_1, q_2 \in \mathcal{P}$ and all deltas $d_1, d_2 \in \mathcal{D}$:

$$p_1 \cong p_2\ \wedge\ q_1 \cong q_2\ \wedge\ d_1 \simeq d_2 \quad \implies \quad (\, p_1\ [\![d_1]\!]\ q_1 \ \Leftrightarrow\ p_2\ [\![d_2]\!]\ q_2 \,)$$ □

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

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:

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

mapping each identifier to a class-level operation:

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

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:

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

mapping each identifier to a package-level operation:

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

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. ⌟

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* $Dt_{\text{pkg}} \stackrel{\text{def}}{=} (\mathcal{PKG}, \mathcal{D}_{\text{pkg}}, [\![\text{-}]\!])$ comprises $\mathcal{PKG}$ from Definition 2.5 as its product set and $\mathcal{D}_{\text{pkg}}$ 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 $[\![\text{-}]\!]\colon \mathcal{ID} \times \mathcal{OP}_{\text{cl}} \to \text{Pow}(\mathcal{CL} \times \mathcal{CL})$ satisfying the following inference rules. For all identifiers $id \in \mathcal{ID}$, classes $cl \in \mathcal{CL}$ and methods or fields $mf \in \mathcal{Mtd} \cup \mathcal{Fld}$:

$$\frac{id \notin \text{pre}(cl)}{cl \quad [\![\, id \;\mapsto\; \textbf{add}\; mf \,]\!] \quad cl[id \mapsto mf]} \text{ method/field addition}$$

$$\frac{id \in \text{pre}(cl)}{cl \quad [\![\, id \;\mapsto\; \textbf{rem} \,]\!] \quad cl[id \mapsto \bot]} \text{ method/field removal}$$

$$\frac{id \in \text{pre}(cl)}{cl \quad [\![\, id \;\mapsto\; \textbf{rep}\; mf \,]\!] \quad cl[id \mapsto mf]} \text{ method/field replacement}$$

$$\frac{id \notin \text{pre}(cl)}{cl \quad [\![\, id \;\mapsto\; \textbf{frb} \,]\!] \quad cl} \text{ method/field forbiddance}$$

$$\frac{}{cl \quad [\![\, id \;\mapsto\; \bot \,]\!] \quad cl} \text{ no operation}$$

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 $[\![\, id \mapsto \textbf{err} \,]\!] = \varnothing$. 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 $[\![\text{-}]\!]\colon \mathcal{D}_{\text{cl}} \to \text{Pow}(\mathcal{CL} \times \mathcal{CL})$ satisfying the following inference rule. For all classes $cl \in \mathcal{CL}$ and class deltas $d_{\text{cl}} \in \mathcal{D}_{\text{cl}}$:

$$\frac{\forall id \in \mathcal{ID}\colon \quad cl(id) \; [\![\, id \mapsto d_{\text{cl}}(id) \,]\!] \; cl'(id)}{cl \quad [\![\, d_{\text{cl}} \,]\!] \quad cl'} \text{ software class delta application}$$

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 $[\![\text{-}]\!]\colon \mathcal{ID} \times \mathcal{OP}_{\text{pkg}} \to \text{Pow}(\mathcal{PKG} \times \mathcal{PKG})$ satisfying the following inference rules. For all identifiers $id \in \mathcal{ID}$, packages $pkg \in \mathcal{PKG}$, classes $cl \in \mathcal{CL}$ and class deltas $d_{\text{cl}} \in \mathcal{D}_{\text{cl}}$:

$$\frac{id \notin \text{pre}(pkg)}{pkg \quad [\![\, id \;\mapsto\; \texttt{add}\; cl \,]\!] \quad pkg[id \mapsto cl]} \text{ class addition}$$

$$\frac{id \in \text{pre}(pkg)}{pkg \quad [\![\, id \;\mapsto\; \texttt{rem} \,]\!] \quad pkg[id \mapsto \bot]} \text{ class removal}$$

$$\frac{id \in \text{pre}(pkg)}{pkg \quad [\![\, id \;\mapsto\; \texttt{rep}\; cl \,]\!] \quad pkg[id \mapsto cl]} \text{ class replacement}$$

$$\frac{id \notin \text{pre}(pkg)}{pkg \quad [\![\, id \;\mapsto\; \texttt{frb} \,]\!] \quad pkg} \text{ class forbiddance}$$

$$\frac{id \in \text{pre}(pkg) \qquad pkg(id) \;\; [\![d_{\text{cl}}]\!] \;\; cl}{pkg \quad [\![\, id \;\mapsto\; \texttt{mod}\; d_{\text{cl}} \,]\!] \quad pkg[id \mapsto cl]} \text{ class modification}$$

$$\frac{}{pkg \quad [\![\, id \;\mapsto\; \bot \,]\!] \quad pkg} \text{ no operation}$$

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 $[\![\text{-}]\!]\colon \mathcal{D}_{\text{pkg}} \to \text{Pow}(\mathcal{PKG} \times \mathcal{PKG})$ satisfying the following inference rule. For all packages $pkg \in \mathcal{PKG}$ and package deltas $d_{\text{pkg}} \in \mathcal{D}_{\text{pkg}}$:

$$\frac{\forall id \in \mathcal{ID}\colon\ \ pkg(id)\ [\![\, id \mapsto d_{\text{pkg}}(id) \,]\!]\ pkg'(id)}{pkg\quad [\![\, d_{\text{pkg}} \,]\!]\quad pkg'}\ \text{software package delta application}$$

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

▷ **2.19. Definition (Software Delta Application):** *Software delta application* is an effective procedure, apply: $\mathcal{D}_{\text{pkg}} \times \mathcal{PKG} \rightharpoonup \mathcal{PKG}$, 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]. ⌟

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

“`DeltaEditor` core” $[\![$ “SH implementation” $]\!]$ “`DeltaEditor` with SH” □

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 $\mathsf{Err} \subseteq \mathcal{D}_{\text{pkg}} \cup \mathcal{OP}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}} \cup \mathcal{OP}_{\text{cl}}$ 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 $d_{\text{pkg}} \in \mathcal{D}_{\text{pkg}}$:

$$\begin{array}{lll} \mathsf{Err}(\, d_{\text{pkg}} \,) & \text{if} & \exists id \in \text{pre}(d_{\text{pkg}})\colon\ \mathsf{Err}(\, d_{\text{pkg}}(id) \,) \\ \mathsf{Err}(\, \texttt{\textbf{modify}}\ d_{\text{cl}} \,) & \text{if} & \mathsf{Err}(\, d_{\text{cl}} \,) \\ \mathsf{Err}(\, d_{\text{cl}} \,) & \text{if} & \exists id \in \text{pre}(d_{\text{cl}})\colon\ \mathsf{Err}(\, d_{\text{cl}}(id) \,) \\ \mathsf{Err}(\, \texttt{\textbf{error}} \,) & & \end{array}$$

⌟

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:

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

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:

$$p_1 \cong p_2 \ \wedge\ q_1 \cong q_2 \ \wedge\ d_1 \simeq d_2 \quad \Longrightarrow \quad (\, p_1 \,[\![d_1]\!]\, q_1 \ \Leftrightarrow\ p_2 \,[\![d_2]\!]\, q_2 \,)$$

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

$$d_1 \simeq d_2 \quad \Longrightarrow \quad (\, p \,[\![d_1]\!]\, q \ \Leftrightarrow\ p \,[\![d_2]\!]\, q \,)$$

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. □

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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, a delta $d \in \mathcal{D}$ is said to *accept* a product $p \in \mathcal{P}$ iff $p \in \mathrm{pre}[\![d]\!]$. ⌟

▶ **2.25. Definition (Fully Defined Delta):** Given a deltoid $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, a delta $d \in \mathcal{D}$ is said to be *fully defined* iff it accepts all products, i.e., iff $\mathrm{pre}[\![d]\!] = \mathcal{P}$. A delta that is not fully defined is *partially defined*. A delta that accepts no products at all is called *undefined* or *invalid*. ⌟

▶ **2.26. Definition (Deterministic Delta):** Given a deltoid $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, a delta $d \in \mathcal{D}$ is said to be *deterministic* iff $[\![d]\!]$ is uniquely defined (Definition 1.13). Otherwise it is called *non-deterministic*. ⌟

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 **`remove`**s an identifier that is not present in the product, or **`add`** an identifier that *is* present.

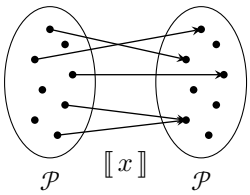


Figure 2.2: A partially defined, deterministic delta x.

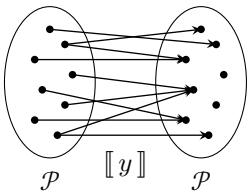


Figure 2.3: A fully defined, non-deterministic delta y.

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 $\varnothing$ (the empty map) are partially defined. □

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 $Dt = (\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, 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 $\mathcal{S}_{Dt}$. ⌟

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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, 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 $\vDash \subseteq \mathcal{D} \times \mathcal{S}$:

$$d \vDash s \iff \underbrace{\forall p \in \text{pre}(s)}_{\text{a}}: \underbrace{p \in \text{pre}\,[\![d]\!]}_{\text{b}} \Rightarrow \underbrace{[\![d]\!](p) \subseteq s(p)}_{\text{c}}$$

$$d \vDash_{\text{tot}} s \iff \underbrace{\forall p \in \text{pre}(s)}_{\text{a}}: \underbrace{p \in \text{pre}\,[\![d]\!]}_{\text{b}} \wedge \underbrace{[\![d]\!](p) \subseteq s(p)}_{\text{c}}$$

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

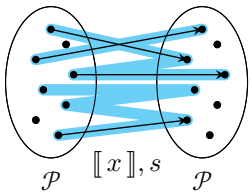


Figure 2.4: A delta x partially correct w.r.t. a specification s.

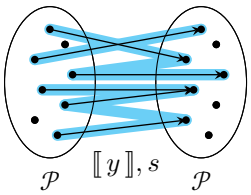


Figure 2.5: A delta y totally correct w.r.t. a specification s.

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:

$$\begin{aligned} d \vDash\ & s \quad \overset{\text{def}}{\iff} \quad [\![d]\!] \subseteq s \\ d \vDash_{\text{tot}}\ & s \quad \overset{\text{def}}{\iff} \quad [\![d]\!] \subseteq s \ \wedge\ \text{pre}\,[\![d]\!] = \text{pre}(s) \end{aligned}$$

□

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 $Dt = (\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, 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 $\Mapsto, \Mapsto_{\text{tot}} \colon \text{Pow}(\mathcal{P}) \times \text{Pow}(\mathcal{P}) \to \text{Pow}(\mathcal{D})$, defined as follows:

$$\begin{aligned} P \Mapsto\ & P' \quad \overset{\text{def}}{=} \quad \{\, d \in \mathcal{D} \mid \forall p \in P\colon\ [\![d]\!](p) \subseteq P' \,\} \\ P \Mapsto_{\text{tot}}\ & P' \quad \overset{\text{def}}{=} \quad \{\, d \in \mathcal{D} \mid \forall p \in P\colon\ \varnothing \subset [\![d]\!](p) \subseteq P' \,\} \end{aligned}$$

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

We now show the connection between derivations and specifications:

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

$$\begin{aligned} d \vDash\ & P \times P' \quad \iff \quad d \in (P \Mapsto\ P') \\ d \vDash_{\text{tot}}\ & P \times P' \quad \iff \quad d \in (P \Mapsto_{\text{tot}} P') \end{aligned}$$

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

$$
\begin{array}{lll}
d \models_{\text{tot}} P \times P' & \iff & \text{(Def. 2.29)} \\
\forall p \in \text{pre}(P \times P')\text{: } p \in \text{pre}[\![d]\!] \wedge [\![d]\!](p) \subseteq (P \times P')(p) & \iff & \text{(Not. 1.12)} \\
\forall p \in P\text{: } \quad p \in \text{pre}[\![d]\!] \wedge [\![d]\!](p) \subseteq P' & \iff & \text{(Not. 1.12)} \\
\forall p \in P\text{: } \quad \varnothing \subset [\![d]\!](p) \subseteq P' & \iff & \text{(Not. 1.1)} \\
d \in \{ x \in \mathcal{D} \mid \forall p \in P\text{: } \varnothing \subset [\![x]\!](p) \subseteq P' \} & \iff & \text{(Def. 2.31)} \\
d \in (P \Mapsto_{\text{tot}} P') &&
\end{array}
$$

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

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 $Dt = (\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, *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 other[1]:

$$
\begin{array}{lll}
x \sqsupseteq y & \overset{\text{def}}{\iff} & [\![x]\!] \subseteq [\![y]\!] \\
x \sqsupseteq_{\text{tot}} y & \overset{\text{def}}{\iff} & [\![x]\!] \subseteq [\![y]\!] \wedge \text{pre}[\![x]\!] = \text{pre}[\![y]\!]
\end{array}
$$

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

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:

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

□

[1] The 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:

$$\begin{array}{lllll} x \sqsupseteq y & \iff & \forall p \in \operatorname{pre} [\![y]\!]: & [\![x]\!](p) \subseteq [\![y]\!](p) \\ x \sqsupseteq_{\text{tot}} y & \iff & \forall p \in \operatorname{pre} [\![y]\!]: & \varnothing \subset [\![x]\!](p) \subseteq [\![y]\!](p) \end{array}$$ □

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 $Dt = (\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$ we define *semantic delta equivalence* $\equiv \subseteq \mathcal{D} \times \mathcal{D}$ as follows for all $x, y \in \mathcal{D}$:

$$x \equiv y \quad \overset{\text{def.}}{\iff} \quad x \sqsupseteq y \ \wedge \ x \sqsubseteq y$$

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

The following is an equivalent formulation:

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

$$x \equiv y \quad \iff \quad [\![x]\!] = [\![y]\!]$$

Proof: $$\begin{array}{lllllll} x \equiv y & \iff & x \sqsupseteq y & \wedge & x \sqsubseteq y & \iff & \\ & & [\![x]\!] \subseteq [\![y]\!] & \wedge & [\![x]\!] \supseteq [\![y]\!] & \iff & [\![x]\!] = [\![y]\!] \end{array}$$ □

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 $\mathrm{id}_{\mathcal{D}} \subset \simeq \subseteq \equiv$, 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 $\mathrm{id}_{\mathcal{P}}$ 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 $(S, \sqcup, \sqcap, {}^{-}, \bot, \top, \cdot, \varepsilon, {}^{\smile})$, with carrier set S, disjunction operator $\sqcup\colon S \times S \to S$, conjunction operator $\sqcap\colon S \times S \to S$, negation operator ${}^{-}\colon S \to S$, an empty element $\bot \in S$, a full element $\top \in S$, a composition operator $\cdot\colon S \times S \to S$, a neutral element $\varepsilon \in S$ and a converse operator ${}^{\smile}\colon S \to S$. 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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$ should respect the following interpretations. For any two deltas $x, y \in \mathcal{D}$:

$$\begin{aligned}
[\![x \sqcup y]\!] &= [\![x]\!] \cup [\![y]\!] & [\![\bot]\!] &= \varnothing \\
[\![x \sqcap y]\!] &= [\![x]\!] \cap [\![y]\!] & [\![\top]\!] &= \mathcal{P} \times \mathcal{P} \\
[\![y \cdot x]\!] &= [\![y]\!] \circ [\![x]\!] & [\![\varepsilon]\!] &= \mathrm{id}_{\mathcal{P}} \\
[\![x^{\smile}]\!] &= [\![x]\!]^{-1} & [\![x^{-}]\!] &= [\![x]\!]^{\complement}
\end{aligned}$$

⌟

In the case where deltas are semantic deltas (and delta evaluation $[\![\text{-}]\!] = \mathrm{id}_{\mathcal{P}\times\mathcal{P}}$ is simply the identity function), they form what is known as a *proper relation algebra* [101, 127, 128], with the signature $(\mathcal{D}, \cup, \cap, {}^{\complement}, \varnothing, \mathcal{P}\times\mathcal{P}, \circ, \mathrm{id}_{\mathcal{P}}, {}^{-1})$. Delta evaluation $[\![\text{-}]\!]$ 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 $\bot$ represents an invalid delta (Definition 2.25). Those are the ones with the most obviously

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

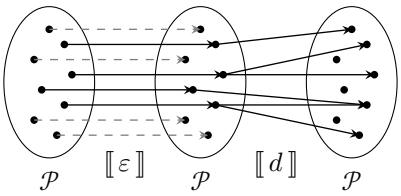


Figure 2.6: A diagrammatic representation of delta composition $d \cdot \varepsilon$. The dashed arrows are part of ε, but not of the full composition.

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., $(\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, \sqcap, \bot, [\![\text{-}]\!])$ (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 $[\![d_n]\!](\cdots[\![d_1]\!](c)\cdots)$, we can see it as the application of a single composed delta:

$$[\![d_n \cdot \ldots \cdot d_1]\!](c)$$

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

$$[\![\, y \cdot x \,]\!]\,(p) \quad = \quad [\![\, y \,]\!]\,(\,[\![\, x \,]\!]\,(p)\,),$$

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:

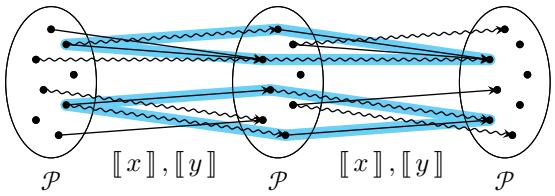


Figure 2.7: A diagrammatic representation of commuting deltas x and y. Highlighting has been added to clarify this commutativity.

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

$$D^* \quad \stackrel{\text{def}}{=} \quad \{\, x_n \cdot \ldots \cdot x_1 \mid x_1, \ldots, x_n \in D \,\} \qquad \text{and}$$
$$D^+ \quad \stackrel{\text{def}}{=} \quad \{\, x_n \cdot \ldots \cdot x_1 \mid x_1, \ldots, x_n \in D \ \wedge\ n > 0 \,\}$$

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

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 $p \notin \text{pre}\,⟦\, d \,⟧$, 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

$$⟦\, \texttt{remove D;} \,⟧(\, \texttt{class C \{\}} \,) \quad = \quad \varnothing.$$

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:

$$(\, \texttt{add class C \{\}} \,) \cdot (\, \texttt{add class C \{\}} \,) \quad \equiv \quad \perp$$

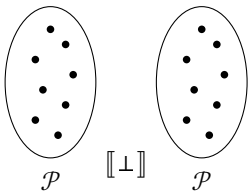


Figure 2.8: The empty delta $\bot$.

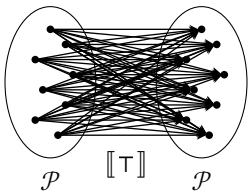


Figure 2.9: The full delta $\top$.

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 $\bot$ 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* $^-$ 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 $[\![d^-]\!](p)$ and $[\![\top]\!]$ could be changed just by extending the set of potential products $\mathcal{P}$, without changing the definitions of d, p or $[\![\text{-}]\!]$.

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 $x, y \in \mathcal{D}$:

$$x \lesssim y \iff y = x \sqcup y$$
$$x \lesssim_{\text{tot}} y \iff y = x \sqcup y \ \wedge\ \top \cdot x = \top \cdot y$$ ⌟

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.

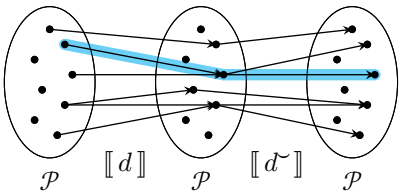


Figure 2.10: A diagrammatic representation of the composition $d^\smile \cdot d$. The highlighted path demonstrates that $d^\smile \cdot d \not\sqsupseteq \varepsilon$.

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 $x, y \in \mathcal{D}$:

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

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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$ well-structured under a lattice order $\lesssim$, we have:

$$\begin{array}{lcl} \lesssim & \subseteq & \sqsupseteq \\ \lesssim_{\text{tot}} & \subseteq & \sqsupseteq_{\text{tot}} \\ \simeq & \subseteq & \equiv \end{array}$$ □

2.6.3 Relation Algebra

Last is the full relation algebra, which adds the *converse* operator $^\smile$ (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 $d^\smile$ 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:

$$d^\smile \cdot d \quad \sqsupseteq \quad \varepsilon$$

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: $(\mathcal{D}_{\text{pkg}}, \sqcap, \bot, \cdot, \varepsilon)$.

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:

$$\begin{aligned} \bot &\stackrel{\text{def}}{=} \{\text{“id”} \mapsto \textbf{err}\} \\ \varepsilon &\stackrel{\text{def}}{=} \varnothing \end{aligned}$$ ⌟

The empty delta needs to have the property that $[\![\bot]\!] = \varnothing$, 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:

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

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

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

with the following refinement rules on the operation level:

$$\frac{}{\textbf{frb} \ \lesssim \ \bot} \text{ by forbiddance} \qquad \frac{x \ \lesssim \ y}{\textbf{mod}\, x \ \lesssim \ \textbf{mod}\, y} \text{ by delegation}$$ ⌟

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 $x, y \in \mathcal{D}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}}$:

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

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

▷ **2.48. Definition (Software Delta Composition):** We define *software delta composition* operator $\cdot\cdot : \mathcal{D}_{\text{pkg}} \times \mathcal{D}_{\text{pkg}} \to \mathcal{D}_{\text{pkg}}$ 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 $e \in \mathsf{Err}$ and all deltas $d \in \mathcal{D}_{\text{pkg}}$:

$$e \cdot d \quad \stackrel{\text{def}}{=} \quad \bot$$
$$d \cdot e \quad \stackrel{\text{def}}{=} \quad \bot$$

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 $op_2 \in \left(\mathcal{OP}_{\text{pkg}} \cup \mathcal{OP}_{\text{cl}} \right)$. The top row is a case distinction on a delta operation $op_1 \in \left(\mathcal{OP}_{\text{pkg}} \cup \mathcal{OP}_{\text{cl}} \right)$. The inner cells form op_{21} so that:

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

	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** $\mathrm{a}(d_2, p_1)$	**rep** $\mathrm{a}(d_2, p_1)$	**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 $x, y \in \left(\mathcal{D}_{\text{pkg}} \cup \mathcal{D}_{\text{cl}} \right)$ and all identifiers $id \in \mathcal{ID}$:

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

Combining the above, we have the software delta algebra:

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

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 assistant[3]:

▷ **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:** $\bot$ is an absorbing element for software delta consensus $\sqcap$. □

▷ **2.55. Lemma:** $\bot$ 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 $Dt = (\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$:

$$Dt \in \mathsf{Full} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \text{img}\,[\![\text{-}]\!] = \mathcal{P} \times \mathcal{P}$$

(See Notation 2.13 and Definition 2.28.) ⌟

[3] Some 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 $Dt = (\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$:

$$Dt \in \mathsf{Max} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \forall p, p' \in \mathcal{P}\colon \left(\{\, p \,\} \mapsto_{\text{tot}} \{\, p' \,\}\right) \neq \varnothing$$

(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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, a product $0 \in \mathcal{P}$ is an *initial product*, indicated by the predicate $\mathsf{init} \subseteq \mathcal{P}$, iff:

$$\mathsf{init}(0) \quad \overset{\text{def}}{\Longleftrightarrow} \quad \forall p \in \mathcal{P}\colon \exists d \in \mathcal{D}\colon [\![\, d \,]\!](0) = \{\, p \,\}$$ ⌟

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 $Dt_{\text{sc}} = (\mathcal{P}_{\text{sc}}, \mathcal{D}_{\text{sc}}, \cdot, \varepsilon, [\![\text{-}]\!])$, 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:

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

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:

$$\mathcal{P}_{\text{sc}} \quad \overset{\text{def}}{=} \quad \left\{\; B \subseteq \mathbb{R}^3 \;\middle|\; \exists l \in \mathbb{R}\colon \forall (X, Y, Z) \in B\colon\; -l \;<\; X, Y, Z \;<\; l \;\right\}$$

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

[4] For 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. □

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

$$[\![\, d_n \cdot ... \cdot d_1 \cdot d_c \,]\!]\,(0),$$

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

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 $(\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$, so then:

$$\begin{array}{lllll}
\forall p, p' \in \mathcal{P}\colon & (\,\{p\} \mapsto_{\text{tot}} \{p'\}\,) & \neq \varnothing & \iff & \text{(Def. 2.31)} \\
\forall p, p' \in \mathcal{P}\colon & \exists d \in \mathcal{D}\colon \varnothing \subset [\![d]\!](p) & \subseteq \{p'\} & \iff & \text{(Def. 1.1)} \\
\forall p, p' \in \mathcal{P}\colon & \exists d \in \mathcal{D}\colon \quad [\![d]\!](p) & = \{p'\} & &
\end{array}$$

□

2.7.2 Classification Based on Deltoid Refinement

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

▶ **2.63. Definition (Deltoid Refinement):** A deltoid $Dt_1 = (\,\mathcal{P}_1, \mathcal{D}_1, [\![\text{-}]\!]\,)$ is said to *refine* another deltoid $Dt_2 = (\,\mathcal{P}_2, \mathcal{D}_2, (\!|\text{-}|\!)\,)$, denoted

$$Dt_1 \quad \sqsupseteq \quad Dt_2$$

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

$$p \;\; [\![\, d \,]\!] \;\; q \quad \Longrightarrow \quad \alpha(p) \;\; (\!|\, \beta(d) \,|\!) \;\; \alpha(q)$$

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

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:

$$Dt \;\in\; \mathsf{Rel} \quad \overset{\text{def.}}{\Longleftrightarrow} \quad Dt \;\sqsupseteq\; (\mathcal{P}, \mathcal{P} \times \mathcal{P}, \mathrm{id}_{\mathcal{P}\times\mathcal{P}})$$ ⌟

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	⊔	⊓	¯	⊥	⊤	·	ε	˘
PFun		⊓		⊥		·	ε	˘
Fun						·	ε	˘

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:

$$Dt \in \mathsf{PFun} \quad \overset{\text{def.}}{\Longleftrightarrow} \quad Dt \sqsupseteq (\mathcal{P}, \mathcal{P} \rightharpoonup \mathcal{P}, \mathrm{id}_{\mathcal{P} \rightharpoonup \mathcal{P}}) \qquad \lrcorner$$

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:

$$Dt \in \mathsf{Fun} \quad \overset{\text{def.}}{\Longleftrightarrow} \quad Dt \sqsupseteq (\mathcal{P}, \mathcal{P} \to \mathcal{P}, \mathrm{id}_{\mathcal{P} \to \mathcal{P}}) \qquad \lrcorner$$

We close off with some observations about these classes:

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

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

$$\mathsf{Fun} \subset \mathsf{PFun} \subset \mathsf{Rel} \qquad \square$$

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

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

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

$$Dt_{\mathrm{sc}} \in \mathsf{Fun} \qquad \square$$

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 $\bot$ or $\sqcap$. If it has at least one product, $\bot$ 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*, $\oplus\colon I \times I \to I$ 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 $i_1, i_2 \in I$:

$$i_1 \sim i_2 \quad \overset{\text{def.}}{\Longleftrightarrow} \quad i_1 \oplus i_2 \oplus i_1 = i_1$$ ⌟

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 $(M, \circledast, \mathbf{1})$, where M is a set of *modifications*, $\circledast\colon M \times M \to M$ 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: $(m_3 \circledast m_2) \circledast m_1 \;=\; m_3 \circledast (m_2 \circledast m_1)$

b. identity element $\mathbf{1}$: $\mathbf{1} \circledast m_1 \;=\; m_1 \;=\; m_1 \circledast \mathbf{1}$ ⌟

2.74. Definition (Modification Application): Given introduction monoid $(I, \oplus, \mathbf{0})$ and modification monoid $(M, \circledast, \mathbf{1})$, *modification application* is a binary operator $\odot\colon M \times I \to I$ 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 $\circledast$: $(m_2 \circledast m_1) \odot i_1 \;=\; m_2 \odot (m_1 \odot i_1)$ ⌟

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 $\blacklozenge\colon Q \times Q \to Q$ 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 $\mathbf{1}_{\mathrm{Q}} \in Q$ and $\mathrm{image}\colon Q \to I$ 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 $(M, \circledast, \mathbf{1})$ and a modification application operator $\odot$, *full quarks* form a magma[5] $(Q_{\mathrm{f}}, \blacklozenge, \mathbf{1}_{\mathrm{Q}}, \mathrm{image})$ where $Q_{\mathrm{f}} \stackrel{\text{def}}{=} M \times I \times M$ 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 $\mathbf{1}_{\mathrm{Q}} \stackrel{\text{def}}{=} \langle\, \mathbf{1}, \mathbf{0}, \mathbf{1} \,\rangle$ is the triple of respective identity elements, the projection function $\mathrm{image}\colon Q_{\mathrm{f}} \to I$ is defined as follows for all introductions $i \in I$ and all modifications $g, l \in M$:

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

and quark composition $\blacklozenge\colon Q_{\mathrm{f}} \times Q_{\mathrm{f}} \to Q_{\mathrm{f}}$ is defined as follows for all introductions $i_1, i_2 \in I$ and all modifications $g_1, g_2, l_1, l_2 \in M$:

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

⌟

2.76. Definition (Local Quarks): *Local quarks* $(Q_{\mathrm{l}}, \blacklozenge, \mathbf{1}_{\mathrm{Q}})$ are a restriction on full quarks, disallowing global modifications. The set $Q_{\mathrm{l}} \stackrel{\text{def}}{=} \{\, \mathbf{1} \,\} \times I \times M$ 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 $l_1, l_2 \in M$:

$$\langle\, i_2, l_2 \,\rangle \blacklozenge \langle\, i_1, l_1 \,\rangle \quad = \quad \langle\, i_2 \oplus (l_2 \odot i_1),\ l_2 \circledast l_1 \,\rangle$$ □

2.78. Definition (Global Quarks): *Global quarks* $(Q_{\mathrm{g}}, \blacklozenge, \mathbf{1}_{\mathrm{Q}})$ are a restriction on full quarks, disallowing local modifications. The set $Q_{\mathrm{g}} \stackrel{\text{def}}{=} M \times I \times \{\, \mathbf{1} \,\}$ 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 $g_1, g_2 \in M$:

$$\langle\, i_2, g_2 \,\rangle \blacklozenge \langle\, i_1, g_1 \,\rangle \quad = \quad \langle\, (g_2 \circledast g_1) \odot (i_2 \oplus i_1),\ g_2 \circledast g_1 \,\rangle$$ □

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

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

2.81. Lemma: Simple quark composition ♦ —derived from Definitions 2.75 and 2.80— is the same as introduction sum ⊕ (Definition 2.69). □

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):

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

To make global quark composition behave they propose to make modification composition ⊛ (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 persue 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* $Dt_Q = (\mathcal{P}, \mathcal{D}, [\![\text{-}]\!])$ 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 $[\![\text{-}]\!]\colon \mathcal{D} \to \text{Pow}(\mathcal{P} \times \mathcal{P})$ is defined as follows for all deltas $\langle g, i, l\rangle \in \mathcal{D}$ and all products $p \in \mathcal{P}$:

$$[\![\langle g, i, l\rangle]\!](p) \quad \stackrel{\text{def}}{=} \quad \{\, \text{image}(\langle g, i, l\rangle \blacklozenge \langle \mathbf{1}, p, \mathbf{1}\rangle)\,\}$$ ⌟

Finally, we classify quarks in our own context:

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

Proof: By Definition 2.82, every semantic delta $[\![\, d \,]\!] \in \left[\!\!\left[\, \mathcal{D}_{Dt_Q} \,\right]\!\!\right]$ is uniquely and fully defined (Definitions 1.13 and 1.16), so the deltoid homomorphism we need for the proof is simply $[\![\text{-}]\!]\colon \mathcal{D} \to (\mathcal{P} \to \mathcal{P})$, its output interpreted as a function (rather than a relation). □

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 $\mathbf{1}_{\mathrm{Q}}$), but don't support associative composition. □

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 at 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 AspectJ™ [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.