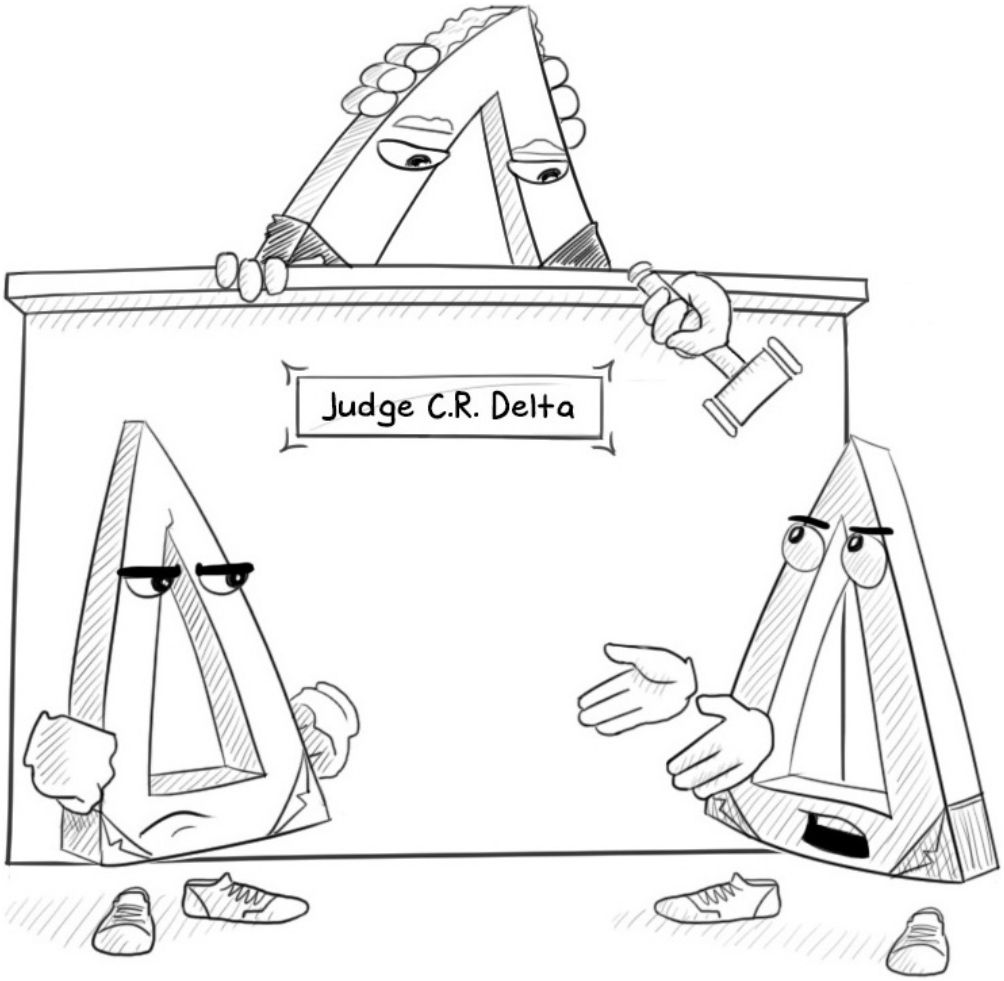
3

Delta Models

On Interaction, Conflict and Conflict Resolution



3.1 Introduction

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

3.1.1 Syntax Highlighting and Error Checking

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

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

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

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

3.1.2 AHEAD

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

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

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

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

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

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

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

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

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

3.1.3 Pre-ADM Delta Oriented Programming

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

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

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

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

3.1.4 The ADM Solution

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

If another conflict ever arises in the future, this can be automatically detected. Furthermore, software deltas can be equipped with a syntax to reference a specific method implementation by name: ⟨*delta*⟩`@`⟨*method*⟩, so no code duplication is required to resolve the conflict. A possible implementation would be:

▷ **3.1. Example:** Software delta $d_{SH \wedge EC}$, the "d_{SH} ↯ d_{EC} conflict resolver":

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

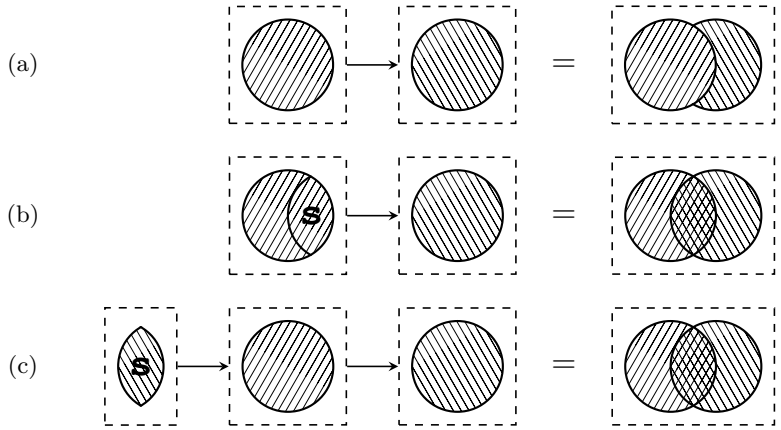


Figure 3.1: Three ways of organizing the *EC* and *SH* features into feature modules with AHEAD, in a Venn-diagram representation. The intersection area represents the `font(int)` method. We need the *SH* ∧ *EC* hybrid version of `font(int)`, so (a) is wrong. A **`super`** call can reuse code from the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

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

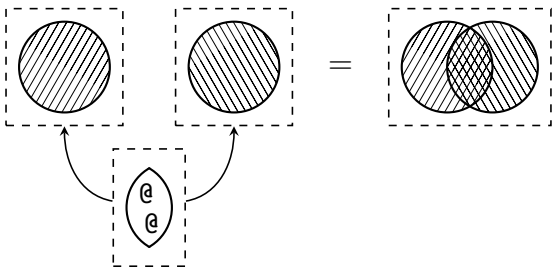


Figure 3.3: The recommended way of organizing the *EC* and *SH* features into deltas with ADM. The flexibility of a partial order allows modularity, separation of concerns and avoids overspecification. Internally referencing specific delta implementations from □ with the `@` operator eliminates code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

3.1.5 Chapter Structure

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

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

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

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

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

3.2 The Delta Model

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

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

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

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

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

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

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

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

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

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

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

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

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

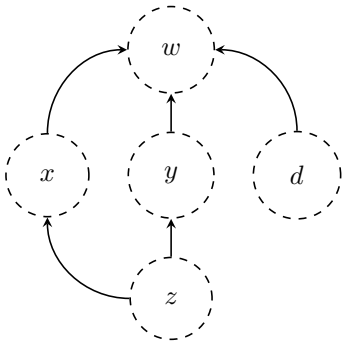


Figure 3.4: A delta diagram of an abstract delta model $dm = (D, \prec)$ with $D = \{\, d, w, x, y, z \,\}$ and order $w \prec x \prec z$, $w \prec y \prec z$ and $w \prec d$.

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

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

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

$$\mathsf{UD}(dm) \quad \overset{\text{def}}{\Longleftrightarrow} \quad |\,\mathrm{derv}(dm)\,| \;=\; 1 \qquad \lrcorner$$

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

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

$$[\![\, dm \,]\!] \quad \overset{\text{def}}{=} \quad [\![\, d \,]\!] \qquad \lrcorner$$

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

▶ **3.6. Lemma:** If a delta model $dm = (D, \prec)$ has a unique derivation and the deltas in D are all deterministic, $[\![\, dm \,]\!]$ is uniquely defined (Def. 1.13). □

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

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

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

3.3 A Conflict Resolution Model

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

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

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

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

3.3.1 Conflicting Deltas

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

▶ **3.7. Definition (Delta Conflict):** Given a delta model $dm = (D, \prec)$, we introduce the *conflict relation* $\lightning \subseteq D \times D$. Deltas $x, y \in D$ are said to be *in conflict* iff the following condition holds:

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

If the delta model is not clear from context, we attach a subscript as in $\lightning_{dm}$.⌟

The idea, given a concrete deltoid, is to develop a decision procedure for delta commutativity[1] so that conflicts can be automatically detected and developers warned: "two deltas that you deemed mutually independent have conflicting implementations". If, for example, deltas y and d from Figure 3.4 stop commuting at some point during development, this would not go unnoticed.

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

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

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

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

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

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

3.3.2 Modifying Conflicting Deltas

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

▶ **3.8. Action (Modifying the Conflicting Deltas):** Given a delta model $(D, \prec)$ and a conflict $x \lightning y$.

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

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

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

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

3.3.3 Linearizing Conflicting Deltas

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

▶ **3.9. Action (Linearizing the Conflicting Deltas):** Given a delta model $(D, \prec)$ and a conflict $x \lightning y$.

$$\prec \quad \rightsquigarrow \quad \prec'$$

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

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

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

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

3.3.4 Introducing a Conflict Resolving Delta

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

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

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

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

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

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

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

▶ **3.11. Action (Introducing a Conflict Resolving Delta):** Given a delta model $(D, \prec)$ and a conflict $x \lightning y$.

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

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

This is what we did for the $d_{SH} \lightning d_{EC}$ conflict. The conflict resolving delta is $d_{SH \wedge EC}$. It replaces the `font(int)` method with a proper combination of the two conflicting versions (Example 3.1).

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

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

3.3.5 Unambiguity

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

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

$$\mathsf{UA}(dm) \quad \overset{\text{def}}{\Longleftrightarrow} \quad \forall x, y \in D\colon \; x \lightning y \;\Rightarrow\; \exists z \in D\colon (x, y) \lhd z \qquad \lrcorner$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof: We proceed by induction on i:

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

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

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

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

We can now prove our main theorem:

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

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

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

3.3.6 Consistent Conflict Resolution

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

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

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

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

$$\mathsf{CCR} \quad \stackrel{\text{def}}{=} \quad \left\{ (\mathcal{D}, \cdot) \;\middle|\; \begin{array}{rl} \forall x, y, z \in \mathcal{D}: & z \cdot y \cdot x \;=\; z \cdot x \cdot y \;\neq\; \bot \;\Rightarrow \\ \forall d \in \mathcal{D}: & z \cdot d \cdot y \cdot x \;=\; z \cdot d \cdot x \cdot y \end{array} \right\}$$

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

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

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

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

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

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

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

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

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

▷ **3.20. Theorem:** The software delta algebra (Definition 2.49) exhibits consistent conflict resolution. □

3.4 A Fine Grained Software Deltoid

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

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

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

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

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

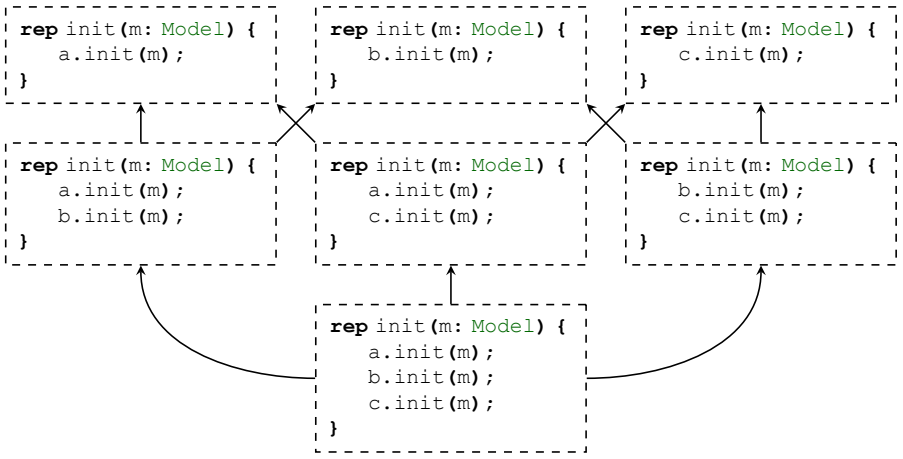


Figure 3.6: A delta model showing the initialization problem on a small scale.

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

3.4.1 Fine-grained Software Deltas

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

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

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

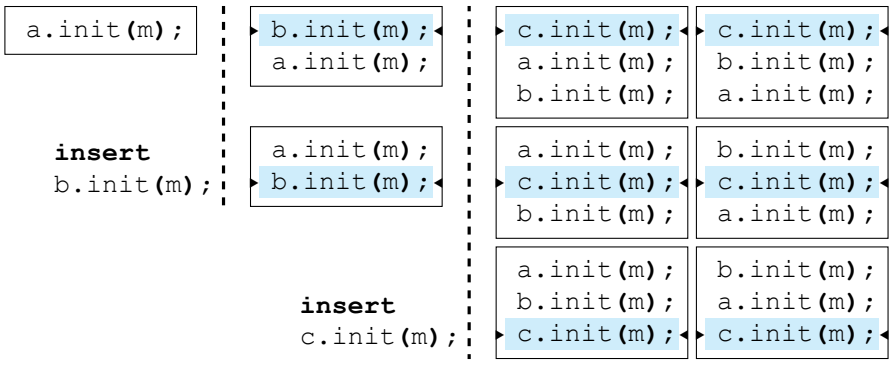


Figure 3.7: The effect of subsequent **insert** operations on a method body. The number of possible output products increases with every insertion. Reversing the order between the two insertions would not affect the final result.

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

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

Statement deltas are sequences of statement-level operations:

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

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

$$\mathcal{OP}_{\text{st+}} \quad \stackrel{\text{def}}{=} \quad \begin{pmatrix} \{\,\textbf{pre}\,\} \times \mathcal{ST} & \cup \\ \{\,\textbf{app}\,\} \times \mathcal{ST} & \cup \\ \{\,\textbf{ins}\,\} \times \mathcal{ST} & \end{pmatrix}$$

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

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

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

And here is how method-level operations are evaluated:

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

a. Statement Level Operations

First, the meaning of the three statement-level operations is as follows, for all method types $tp \in \mathcal{TP}$, all statement sequences $\overline{st}, \overline{st_1}, \overline{st_2} \in \mathcal{ST}^*$ and all statements $st' \in \mathcal{ST}$ (recall Definition 2.3, page 31):

$$\frac{}{(tp, \overline{st}) \quad [\![\,\textbf{pre}\ st'\,]\!] \quad (\,tp,\ st' \frown \overline{st}\,)} \text{ statement prepension}$$

$$\frac{}{(tp, \overline{st}) \quad [\![\,\textbf{app}\ st'\,]\!] \quad (\,tp,\ \overline{st} \frown st'\,)} \text{ statement appension}$$

$$\frac{}{(tp, \overline{st_1} \frown \overline{st_2}) \quad [\![\,\textbf{ins}\ st'\,]\!] \quad (\,tp,\ \overline{st_1} \frown st' \frown \overline{st_2}\,)} \text{ statement insertion}$$

The **pre** and **app** operations are deterministic, placing the new statement squarely at the beginning or end of the method. But the **ins** operation is non-deterministic. It can produce a method with the new statement at the very beginning or end, or at any position in between, depending on how the original sequence of statements is apportioned between $\overline{st_1}$ and $\overline{st_2}$. Also, note that none of them change the type of the method. Software deltas have no way of doing so without replacing the entire method. (Though, as mentioned before, this is an intentional simplification.)

b. Statement Deltas

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

$$\frac{mtd \quad [\![\,\overline{op'}\,]\!] \circ [\![\,op\,]\!] \quad mtd'}{mtd \quad [\![\,op \frown \overline{op'}\,]\!] \quad mtd'} \text{ statement delta application (non-empty)}$$

$$\frac{}{mtd \quad [\![\,()\,]\!] \quad mtd} \text{ statement delta application (empty)}$$

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

c. Class Level Operations

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

$$\frac{cl(id) \;\in\; \mathcal{M}td \qquad cl(id) \;\; [\![\, d_{\text{st+}} \,]\!] \;\; mtd}{cl \;\; [\![\, id \;\mapsto\; \textbf{mod}\; d_{\text{st+}} \,]\!] \;\; cl[id \mapsto mtd]} \text{ method modification}$$ ⌟

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

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

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

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

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

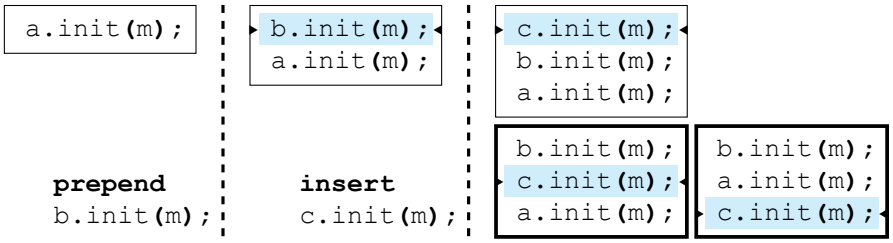
$$\frac{}{(\textbf{ins}\; st_1) \frown (\textbf{ins}\; st_2) \frown \overline{op} \quad \lesssim \quad (\textbf{ins}\; st_2) \frown (\textbf{ins}\; st_1) \frown \overline{op}} \text{ by commutative } \textbf{ins}$$

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

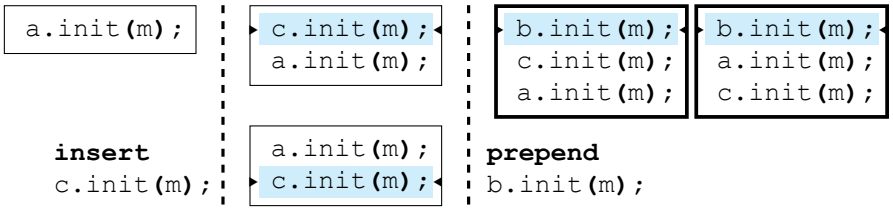
$$\frac{}{(\textbf{ins}\; st_1) \frown (\textbf{pre}\; st_2) \frown \overline{op} \quad \lesssim \quad (\textbf{pre}\; st_2) \frown (\textbf{ins}\; st_1) \frown \overline{op}} \text{ by late prepension}$$

$$\frac{}{(\textbf{ins}\; st_1) \frown (\textbf{app}\; st_2) \frown \overline{op} \quad \lesssim \quad (\textbf{app}\; st_2) \frown (\textbf{ins}\; st_1) \frown \overline{op}} \text{ by late appension}$$

The idea is that *after* a **pre** or **app** (Figure 3.8a), there are more potential places to **ins**ert a new statement than *before* (Figure 3.8b). So a delta is more refined the earlier its **insert** operations occur. ⌟

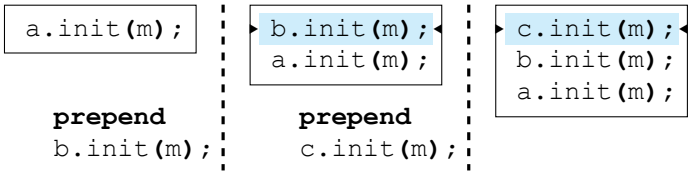


(a) A **prepend** followed by an **insert**.

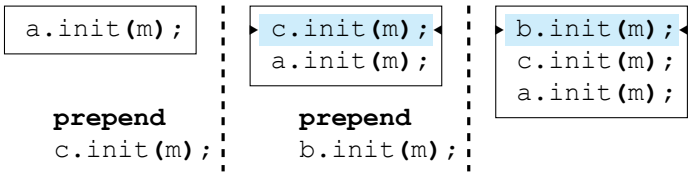


(b) An **insert** followed by a **prepend**.

Figure 3.8: The interaction between a **prepend** and an **insert**. The thick borders in the third column indicate the result of their *consensus*.



(a) **prepend**ing first `b`, then `c`



(b) `prepend`ing first `c`, then `b`

Figure 3.9: The interaction between two **prepend** operations. As the two are incompatible (and deterministic), they have an empty consensus.

3.5 Ambiguous Delta Models

In Section 3.3 we explored techniques for ensuring a unique derivation in a delta model. However, there are legitimate cases for the use of delta models with multiple derivations. To that end, this section proposes two semantics that are more flexible than the sole derivation semantics of Definition 3.5.

In previous work [1–3], we gave delta models *disjunctive semantics*, in which a non-deterministic choice is made between available derivations. We describe this semantics in Section 3.5.1. An alternative semantics is proposed in Section 3.5.2: *conjunctive semantics*, which we believe to be a more natural interpretation.

3.5.1 Disjunctive Semantics

In disjunctive semantics, a delta model is seen as providing a source of non-determinism, just like deltas themselves can. This would lead to the following delta model semantics, which applies regardless of the number of derivations:

▶ **3.25. Definition (Disjunctive Semantics):** The *disjunctive semantics* of a delta model $dm \in \mathcal{DM}$ with $\text{derv}(dm) = \{ d_1, ..., d_n \}$ are defined as follows:

$$[\![dm]\!]_{\cup} \quad \stackrel{\text{def}}{=} \quad [\![d_1]\!] \cup \cdots \cup [\![d_n]\!]$$

Just take the union of the semantic deltas, i.e., a union of the product relations (see Definitions 1.2 and 3.3 and Notation 2.13). ⌟

This semantics was used in the original ADM papers [1–3], though not by the same distinctive name, as alternative semantics were not considered at the time.

Disjunctive semantics is the most flexible of the three delta model semantics we present in this chapter, and the easiest to employ, as it simply requires that any applicable derivation be applied. No special constructions or proofs of unambiguity are necessary. For that reason, this semantics is best used to tolerate ambiguity during development until stricter standards can be established.

3.5.2 Conjunctive Semantics

While disjunctive semantics certainly has its uses, it does not, perhaps, properly correspond to a developer's likely intentions. As a means for introducing nondeterminism, delta models are quite limited. Nondeterminism is much more flexibly introduced by deltas themselves (Sections 2.4 and 3.4). So would it not be better for delta model semantics to instead play a supporting rôle?

Conjunctive semantics is the dual of disjunctive semantics, and has a delta model perform a modification that all derivations agree upon:

▶ **3.26. Definition (Conjunctive Semantics):** The *conjunctive semantics* of a delta model $dm \in \mathcal{DM}$ with $\text{derv}(dm) = \{ d_1, ..., d_n \}$ are defined as follows:

$$[\![dm]\!]_{\cap} \quad \stackrel{\text{def}}{=} \quad [\![d_1]\!] \cap \cdots \cap [\![d_n]\!]$$

So, take the intersection of the product relations (see Definitions 1.2 and 3.3 and Notation 2.13). ⌟

Note, at this point, that both disjunctive and conjunctive semantics are compatible with the sole derivation semantics of Definition 3.5 in the case that a unique derivation exists:

3.27. Lemma: For all delta models $dm \in \mathcal{DM}$ we have:

$$\mathsf{UD}(dm) \quad \Longrightarrow \quad [\![\, dm \,]\!]_{\cup} = [\![\, dm \,]\!]_{\cap} = [\![\, dm \,]\!] \qquad \square$$

Conjunctive semantics is less tolerant to mistakes but offers stronger guarantees. Applying a delta model to a product under conjunctive semantics always results in an end product that might also have resulted from the application of any individual derivation. In other words, a developer may consider a specific derivation $d \in \mathrm{derv}(dm)$ and work under the assumption that d is the derivation that will be chosen to generate the final product, without this leading to contradiction:

3.28. Lemma: For any delta model $dm \in \mathcal{DM}$, any derivation $d \in \mathrm{derv}(dm)$ and any specification $s \in \mathcal{S}$, we have:

$$d \vDash s \quad \Longrightarrow \quad [\![\, dm \,]\!]_{\cap} \subseteq s \qquad \square$$

The same thing is not true for disjunctive semantics.

That being said, this semantics takes more effort to implement. Disjunctive semantics requires only that single derivations are tried until one is found that is applicable to the product at hand (recall that deltas may be partially defined). Conjunctive semantics, on the other hand, permits no such approach. There is no general procedure for generating all possible outputs of a nondeterministic delta in order to produce the required intersection; in fact, such a set may well be infinite. So implementing delta model application under conjunctive semantics requires a greater understanding of the domain. In particular, it requires an implementation of the consensus operator (Definition 1.34):

▶ **3.29. Lemma:** For any deltoid $(\mathcal{P}, \mathcal{D}, \sqcap, \cdot, \varepsilon, [\![\text{-}]\!])$ and delta model $dm \in \mathcal{DM}_{\mathcal{D}}$, we can characterize conjunctive semantics as follows:

$$[\![\, dm \,]\!]_{\cap} \quad = \quad \Big[\!\Big[\bigsqcap \mathrm{derv}(dm) \Big]\!\Big]$$

Proof: This is easily derived from Definitions 2.38 and 3.26. $\square$

So if we have an effective procedure for delta consensus of a specific deltoid, delta models based on that deltoid can exhibit conjunctive semantics. This can be worth the effort. A conflict model based on delta commutativity, as introduced in Section 3.3, is useful, but can be too strict in the presence of more sophisticated interaction. The concept of conjunctive semantics allows separate developers to express intentions that would otherwise be flagged as a conflict, but can now be reconciled without the need for manual onflict resolution.

For example, imagine two unordered software deltas modifying the same method. One of them **`insert`**s a statement "`c.init(m)`". The other one, having stricter requirements, **`prepend`**s a statement "`b.init(m)`" (Figure 3.8). Composing them in two different orders results in two different derivations, but intuitively there should be no conflict. As long as "`b.init(m)`" becomes the

first statement of the method, and "`c.init(m)`" is inserted anywhere else, the intentions of both developers are satisfied. And indeed, this is the result of the consensus between Figure 3.8a and Figure 3.8b, as it should be.

In contrast, two deltas each wanting their own statement to appear first in the method is recognized as a legitimate conflict by the empty consensus between Figure 3.9a and Figure 3.9b.

3.6 Nested Delta Models

This section explores the possibility of deltas that *are* delta models. We then take a particular look at the implications of *nested delta models*. There are a number of reasons we might want delta models to act as deltas inside other delta models, chief among them being the isolated/atomic application of a collection of deltas within a delta model, making sure that any delta outside that collection is applied before or after the entire collection — but not in between.

Let us first establish some terminology:

▶ **3.30. Definition:** A *nesting delta model* is a delta model that contains another delta model. A *simple delta* is a delta that is not a delta model. A *flat delta model* is a delta model that contains only simple deltas. A *nested delta model* is a delta model contained within another delta model. ⌟

Nesting delta models have several uses in the area of modularization:

- Recall from Definitions 3.7 and 3.10 that a conflict is uniquely identified by two conflicting deltas, and that each requires a single delta to resolve it. But a conflict may have several causes. For example, two software deltas may disagree on the implementation of more than one method. Resolution for each of those methods could be modularized as a delta inside a conflict resolving delta model.
- They may also be used for refactoring a single delta into two deltas. Nesting the two together avoids the inadvertent introduction of new conflicts, because the two deltas would still be treated as one (Figure 3.10).
- It may be generally beneficial to structure variability as a hierarchy, i.e., to implement a modification in terms of smaller modifications, in the best traditions of computer programming. Nesting delta models can do this.

As an example of using a nested delta model for refactoring purposes consider Figure 3.10. In this figure, delta d_{SH} from the Editor product line is being refactored into two deltas d^1_{SH} and d^2_{SH}, the first handling the fields and initialization of the feature in the `Editor` class and the second handling the actual functionality of configuring the font. To avoid having to introduce extra ordering into the delta model to preserve the original semantics, the two deltas are placed in a nesting delta model which replaces the original d_{SH}.

3.6.1 Semantics Independent Definitions

Let us first look at a way to syntactically extend a set of deltas to include all delta models that could be built from that set:

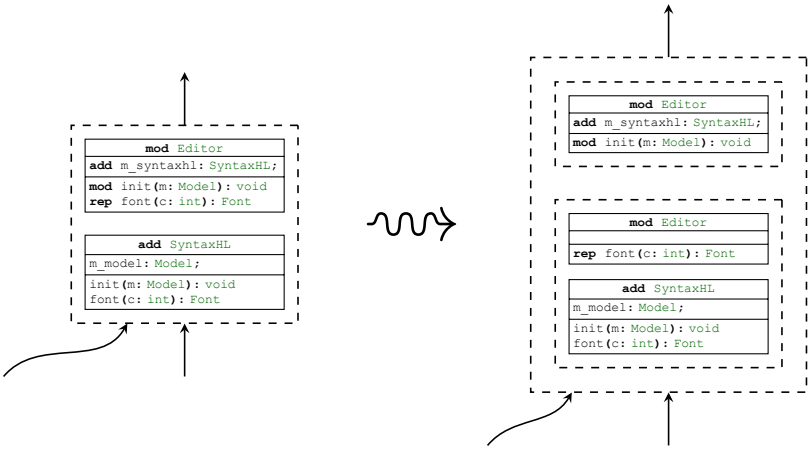


Figure 3.10: Refactoring d_{SH} from Figure 1.3 (page 9) into a nested delta model containing deltas d^1_{SH} and d^2_{SH}.

▶ **3.31. Definition (Delta Model Closure):** Given a delta set $\mathcal{D}$, we define the following family of delta sets for all natural numbers $n \in \mathbb{N}$:

$$\begin{aligned} \mathcal{D}_{\triangle,0} &\stackrel{\text{def}}{=} \mathcal{D} \\ \mathcal{D}_{\triangle,n+1} &\stackrel{\text{def}}{=} \mathcal{D}_{\triangle,n} \cup \mathcal{DM}_{\mathcal{D}_{\triangle,n}} \end{aligned}$$

We then define the *delta model closure* of $\mathcal{D}$ as follows:

$$\mathcal{D}_{\triangle} \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} \mathcal{D}_{\triangle,n}$$

We require that $\mathcal{D}$ did not contain any delta models to begin with. ⌟

This definition allows delta models nested at unbounded —but finite— depth. Note that $\mathcal{D}_{\triangle}$ can be partitioned into the set of simple deltas $\mathcal{D}$ and the set of delta models $\mathcal{DM}_{\mathcal{D}_{\triangle}}$.

We can then define a nesting aware derivation function, first requiring a straightforward extension of the composition operator to sets of deltas:

▶ **3.32. Notation:** We extend $\cdot$ to sets as follows, for all delta sets $D_1, D_2 \subseteq \mathcal{D}$:

$$D_2 \cdot D_1 \stackrel{\text{def}}{=} \{\, d_2 \cdot d_1 \mid d_1 \in D_1 \ \wedge\ d_2 \in D_2 \,\}$$ ⌟

▶ **3.33. Definition (Nesting-aware Derivation):** Define the *nesting-aware derivation* function $\text{derv}_{\triangle}\colon \mathcal{D}_{\triangle} \to \text{Pow}(\mathcal{D})$ as follows, for any simple delta $d \in \mathcal{D}$ and delta model $dm = (D, \prec) \in \mathcal{DM}_{\mathcal{D}_{\triangle}}$:

$$\begin{aligned} \text{derv}_{\triangle}(d) &\stackrel{\text{def}}{=} \{\, d \,\} \\ \text{derv}_{\triangle}(dm) &\stackrel{\text{def}}{=} \bigcup_{\substack{D=\{\,d_1,\ldots,d_n\,\}\ \wedge \\ \forall i,j \in \{\,1,\ldots,n\,\}: \\ d_i \prec d_j \ \Rightarrow\ i<j}} \text{derv}_{\triangle}(d_n) \cdot \ldots \cdot \text{derv}_{\triangle}(d_1) \end{aligned}$$ ⌟

The algebraic interpretation of a delta model closed deltoid is quite simple. It is a matter of setting up one of the following equivalences:

$$dm \simeq \bigsqcup \mathrm{derv}(dm) \quad \text{(under disjunctive semantics)}$$
$$dm \simeq \bigsqcap \mathrm{derv}(dm) \quad \text{(under conjunctive semantics)}$$

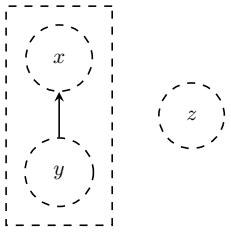
and then to 'flatten' delta models to delta expressions containing only simple deltas. A very similar technique is used in Chapter 6 in order to reduce modalities to simple forms.

3.6.2 Expressiveness of Nested Delta Models

The semantics of nesting cannot be captured with flat delta models. Nesting delta models are syntactically more expressive in the following sense:

▶ **3.34. Theorem:** There are nesting delta models $ndm = (D, \prec) \in \mathcal{DM}_{\mathcal{D}_\triangle}$ that have a derivation set $\mathrm{derv}_\triangle(ndm)$ which is inexpressible with any flat delta model $dm' = (D', \prec')$ containing the same simple deltas.

Proof: Consider the following nesting delta model ndm:



$$\begin{aligned} ndm &= (\,\{\,(\{x,y\},\{\,(x,y)\,\}),z\,\},\varnothing\,) \\ \mathrm{derv}_\triangle(ndm) &= \{\,z\cdot y\cdot x,\ y\cdot x\cdot z\,\} \\ \text{simple deltas:}\quad & x, y, z \end{aligned}$$

To find a flat delta model $dm' = (\{x,y,z\}, \prec')$ s.t. $\mathrm{derv}(dm') = \mathrm{derv}_\triangle(ndm)$, consider all possible strict partial orders $\prec'$ over 3 elements:

$$\begin{aligned}
&\prec' = \varnothing && && \Rightarrow\ |\,\mathrm{derv}(dm')\,| = 6 \\
&\prec' = \{(e,g)\} && \text{s.t. } \{e,g\} \subseteq \{x,y,z\} && \Rightarrow\ |\,\mathrm{derv}(dm')\,| = 3 \\
&\prec' = \{(e,g),(g,h)\} && \text{s.t. } \{e,g,h\} = \{x,y,z\} && \Rightarrow\ |\,\mathrm{derv}(dm')\,| = 1 \\
&\prec' = \{(e,g),(e,h)\} && \text{s.t. } \{e,g,h\} = \{x,y,z\} && \Rightarrow\ \mathrm{derv}(dm') = \{h\cdot g\cdot e,\ g\cdot h\cdot e\} \\
&\prec' = \{(e,h),(g,h)\} && \text{s.t. } \{e,g,h\} = \{x,y,z\} && \Rightarrow\ \mathrm{derv}(dm') = \{h\cdot g\cdot e,\ h\cdot e\cdot g\}
\end{aligned}$$

As ndm has two nesting-aware derivations, only the last two cases are relevant. If ndm were expressible via a flat delta model, there would exist a bijection between $\{\,e,g,h\,\}$ and $\{\,x,y,z\,\}$ such that either $\{\,h\cdot g\cdot e,\ g\cdot h\cdot e\,\} = \{\,z\cdot y\cdot x,\ y\cdot x\cdot z\,\}$ or $\{\,h\cdot g\cdot e,\ h\cdot e\cdot g\,\} = \{\,z\cdot y\cdot x,\ y\cdot x\cdot z\,\}$. But no such bijection exists. Hence, there exists no flat delta model dm' such that $\mathrm{derv}(dm') = \mathrm{derv}_\triangle(ndm)$. □

3.7 Conclusion

In some ways, this chapter describes the most fundamentally novel contribution of ADM: *delta models*, which organize deltas into a *partial application order*. One delta may be dominant over another, or two deltas may be unrelated by the order. This helps developers express their design intentions, and contain the complexity of large system. If two deltas are unrelated, it is still possible that both need access to the same resource, causing a *conflict* if both are applied together, even if each works fine in isolation. To solve this problem and still maintain separation of concerns, *conflict resolving deltas* are introduced.

The chapter then extends the software deltoid to allow *fine-grained* modifications, i.e., manipulating individual statements in methods. This is often neglected in compositional approaches like delta modeling, because unlike classes, methods and fields, statements have no names by which a delta can target their position. *Conjunctive delta model semantics* are introduced to take advantage of fine-grained modifications. The operation of inserting a statement in a non-deterministically chosen location avoids another type of overspecification, representing the intention: "this method should run this statement at some point; I don't care when". This reduces the likelihood that two changes to the same method are seen as a conflict.

Finally, the chapter introduces *nested delta models*, which increase expressiveness of a deltoid and offer a new modularization technique.

3.8 Related Work

We now summarize related work that was not mentioned earlier in this chapter, among which some of the advances since we started our work on ADM.

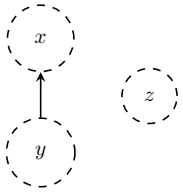
3.8.1 Delta-Oriented Programming

Originally, a delta oriented product line consisted of a single core and a set of incomparable product deltas [160, 163]. Conflicts between deltas applicable for the same feature configuration were prohibited. In order to express all possible products, an additional delta covering the combination of the potentially conflicting deltas had to be specified, leading to code duplication.

Since the work on ADM, Schaefer et al. [162, 164] also introduced a partial order between delta modules. However, it was required that conflicts were removed by changing the deltas or by specifying some linear order (Actions 3.8 and 3.9). They were not allowed to exist and then resolved.

Later work [81, 161, 169] moved away from the partial order in favor of a linearly ordered partition. Delta modules could be freely reordered within a part, but the parts themselves had to be applied in a fixed total order. It is easily proved that this is at least as expressive as Schaefer et al.'s earlier unordered structure as well as the total ordering of AHEAD, as those correspond to the two trivial partitions of the module set. But it is not as expressive as an arbitrary partial order, which can be shown with the following simple example,

inexpressable with a linearly ordered partition:



$$\begin{aligned} dm &= (\{x, y, z\}, \{(x, y)\}) \\ \text{derv}(dm) &= \{z \cdot y \cdot x,\ y \cdot z \cdot x,\ y \cdot x \cdot z\} \end{aligned}$$

3.8.2 Feature Interaction Algebra

Two recent papers by Batory et al. [29, 33] describe a new algebraic treatment of feature interaction. This treatment takes place in the setting of CIDE (Colored IDE) [108], a variability tool based on *code painting* — each feature has an associated color, which is used for the annotation of code. When code is painted in more than one color, it is 'interaction composition code', similar in function to conflict resolving deltas. The algebra introduces a notation # for interaction.

A rather confusing aspect of this work is the fact that # is presented as a *operator*: if `A` and `B` are feature implementations, then `A#B` is also a feature implementation; their resolution. The operator is commutative, associative, and obeys such distributive laws as $\texttt{A}\#(\texttt{B} + \texttt{C}) = (\texttt{A}\#\texttt{B}) + (\texttt{A}\#\texttt{C})$, where + is feature composition. But since the interaction between two features is a design decision, it cannot be 'computed'. And there is almost always more than one way to do it. The operator's exact meaning and purpose are therefore unclear.

Recently, Apel et al. [18] also refer to the # notation, but they use it only as a shorthand for the corresponding coordination code; not as an operator.

3.8.3 Other Related Work

Our distinction between feature interaction and implementational conflicts (Section 3.3) is also described by Kästner et al. [109] in a discussion paper on feature modularity. Mosser et al. [140] —in an otherwise fascinating contribution— missed this distinction in their analysis of our original paper on ADM [1].

Our notion of conflict, based on a lack of commutativity, has already been discussed in a number of publications. Mens et al. [132], for instance, describe this phenomenon in the context of refactoring [71]. Their notion of conflict is very similar to ours, though their solution —based on graph transformation and critical pair analysis— is quite different. Apel et al. [14] and Oldevik et al. [145] observe similar notions of conflict, respectively in aspect oriented programming and model transformations. Apel et al. propose that the aspects involved be refactored following a particular scheme — a measure which falls under our Action 3.8. Oldevik et al. analyze the effect of ordering constraints to resolve conflicts — quite like our Action 3.9.

Interestingly, while Darcs patch theory [97] has quite a different purpose from ADM and the aforementioned literature, they deal with a very similar —and naturally occurring— partially ordered structure: that of branches and commits in a version control system. The most significant similarity is that they deal with *conflictors* (entities for resolving conflicts), which are similar to conflict-resolving deltas. Patch theory could certainly offer inspiration to guide future research (Chapter 9).