4

Product Lines

On Feature Modeling and Delta Selection

4.1 Introduction

A *product line* is usually defined as a set of systems, called *products*, each of which is characterized by the set of *features* it provides. This allows commonality and variability between these products to be well-defined and amenable to formal analysis. Products were introduced in Chapter 2. Delta models, the tools we use for generating new products, were introduced in Chapter 3. We now introduce the final ingredient to ADM-based product line models: features.

Many different definitions of the term "feature" have been given in literature. Griss [75] defines features as follows:

> "A feature is a product characteristic that users and customers view as important in describing and distinguishing members of [a] product-line."

Classen et al. [53] gathered a number of definitions from literature in order to compare them. The above definition is what they would call *problem-oriented*. They also talk about definitions such as "a logical unit of behavior" by Bosch [45] and "an increment in product functionality" by Batory et al. [27], which are geared more towards implementation.

Griss' definition is especially suitable for us, as our "logical unit of behavior" or "increment in product functionality" is, of course, the delta. We see features more as specifications of product requirements. Deltas and features should not be restricted to a one-to-one relationship; a sentiment first expressed by Schaefer et al. [163].

We are mainly interested in using features as a means of identifying specific products in a product line, allowing us to use features

- to characterize the set of available products through a feature model, which determines the possible *feature configurations*, i.e., the set of feature combinations supported by the product line;
- to formulate *product line implementations* by linking feature symbols to deltas through application conditions, allowing us to select the proper deltas for deriving the implementation of a specific product; and
- to formulate *product line specifications* by associating requirements with each feature.

Product generation is still a time consuming and expensive activity [65]. Ideally, it should be a fully mechanical process, since any manual adjustments after feature selection would need to be repeated whenever the main code-base is changed in some way. This brings us to the main goal of this chapter:

> ***Goal:*** *Develop a technique for organizing a product line code-base in such a way that product generation can be a mechanical process.*

Such generation process is generally known as *automated product derivation* [1, 2, 65, 84, 163] (Figure 1.1, page 5).

When it comes to automated product derivation, *annotative variablitity techniques* are undeniably popular. The idea is to take the full code-base and to annotate the code with feature conditions without otherwise altering its structure. For example, take a look at the following annotative implementation of Syntax Highlighting and Error Checking in the Editor product line:

▷ **4.1. Example:** A Delta Editor product annotated with the *SH* and *EC* features. The **if**-conditions are typically resolved at compile-time:

```
1  package DeltaEditor {
2     class Editor {
3        m_model : Model;
4        if (SH) { m_syntaxhl : SyntaxHL;      }
5        if (EC) { m_errorch  : ErrorChecker; }
6
7        init(m : Model) : void {
8           m_model = m;
9           if (SH) { m_syntaxhl = new SyntaxHL(m);      }
10          if (EC) { m_errorch  = new ErrorChecker(m); }
11       };
12
13       model() : Model { return m_model; };
14
15       font(c : int) : Font {
16          Font result = new Font();
17          if (SH) { result.setColor(m_syntaxhl.font(c));         }
18          if (EC) { result.setUnderlined(m_errorch.errorOn(c)); }
19          return result;
20       };
21
22       onMouseOver(c : int) : void {
23          if (EC) {
24             if (m_errorch.errorOn(c)) {
25                super.showTooltip(m_errorch.errorText(c));
26             }
27          }
28       };
29    };
30
31    if (SH) {
32       class SyntaxHL {
33          m_model : Model;
34
35          init(m : Model) { m_model = m; };
36
37          font(c : int) : Font { /* something complicated */; };
38       };
39    }
40
41    if (EC) {
42       class ErrorChecker {
43          m_model : Model;
44
45          init(m : Model) { m_model = m; };
46
47          errorOn(c : int) : bool { /* some code */; };
48
49          errorText(c : int) : string { /* more code */; };
50       };
51    }
52 };
```

As these annotations are essentially **`if`**-statements, this is a technique all programmers will understand, and one that is relatively effortless to set up. A prominent example of the annotative technique in practice is the Linux kernel, an immense collection of C code annotated by `#ifdef` preprocessor directives, which allow conditional compilation [171].

But as discussed before, there are several disadvantages to this approach. There is a notable lack of modularity and separation of concerns. In Example 4.1, the implementations of *SH* and *EC* are mixed together and spread across the core implementation, making it difficult to get a good overview of the structure. Additionally, there is overspecification because of the linear nature of program code, discussed at length in Chapter 3. In this chapter we build on the concept of delta models in an effort to achieve automated product derivation without neglecting these other goals. We also discuss product line level specifications, allowing us to consider product line *correctness*.

> ***Goal:*** *Develop a formal concept of product line specification, to be used both in verifying product line correctness and in guiding the implementation process.*

This chapter is organized as follows: Section 4.2 reviews *feature modeling*, the discipline of describing product line variability on the high abstraction level of feature symbols. In Section 4.3 we tie features to deltas using *application conditions* and explore a formulation of product line implementations based on delta models. Section 4.4 then introduces product line *specifications*. In Section 4.5 we recognize a possible problem with purely compositional techniques and propose the solution of *parametric deltas* in order to gain some of the benefits of the annotative approach. Finally, Sections 4.7 and 4.8 offer concluding remarks and discuss related work.

4.2 Feature Modeling

Feature-oriented Domain Analysis (FODA) was developed in 1990 in order to study possible *features* of a system to enhance reuse in a particular application domain. *Feature Models* were among its most useful tools, characterized as "the greatest contribution of domain engineering to software engineering" [105], and are still ubiquitous in Software Product Line Engineering today.

Feature models are not concerned with implementation, but with modeling product line commonality and variability on a high level. What kind of features are (should be) available, and what are the relationships between them? The answers to these questions inform both specification and implementation of delta-based product lines. Section 4.2.1 discusses the formal concept of feature. Section 4.2.2 then formalizes feature models.

4.2.1 Features

We first introduce a formal representation for features:

▶ **4.2. Notation (Features):** We denote *features* by the symbols f, g and h. Finite sets of features are denoted by F, G or $\mathcal{F}$. ⌟

These features are just symbols [56], with no inherent meaning. It is what we do with features next that gives them their semantics. They will play several rôles in our formalism, which will be extended in Chapters 6 to 8.

We assume that each feature represents a discrete Boolean value, i.e., something that can be either on or off. This is a simplification. Czarnecki et al. [58, 59], for example, have worked on cardinality-based and attributed feature models. These allow a product to contain a specific feature more than once, or to contain variations of a feature parametrized with arbitrary data-types. We won't discuss them in detail, though in subsequent chapters we'll occasionally dip our toe in the water. Using the simpler Boolean notion will make this chapter easier to follow, without sacrificing generality.

There are two ways in which we give features meaning in this chapter. First, product line implementations (Section 4.3) conditionally select deltas based on a chosen selection of features — linking features to code. Then, product line specifications (Section 4.4) impose requirements on products that claim to implement given features — linking features to code specifications. This will lead to notions of correctness and refinement for product lines.

4.2.2 Feature Models

Features represent the *variability* and *commonality* of the products in a product line: the ways in which they can differ from each other. This is often expressed in terms of a *feature model* [66, 105, 166], which expresses the relations between features and decides which combinations of them are considered valid or conceptually feasible. Such feature combinations are more commonly called *feature configurations*, a term that is appropriate for simple as well as cardinality-based and attributed feature models.

Many formal descriptions [66, 91, 105] agree that, at the very least, a feature model determines a set of valid feature configurations. And for the moment, that is the only aspect of feature models we are interested in, which motivates the following definition:

▶ **4.3. Definition (Feature Model):** Given a set of features $\mathcal{F}$, a *feature model* $\Phi \subseteq \mathrm{Pow}(\mathcal{F})$ is a set of sets of features from $\mathcal{F}$. Each $F \in \Phi$ is a feature set corresponding to a valid *feature configuration*. ⌟

Though this is formally a useful notion, it lacks the intuition provided by more diagrammatic descriptions. The most common representation for feature models is the *feature diagram* [35, 60, 166, 167]. We've already seen one in Figure 1.2 (page 8). Their semantics was described extensively by Czarnecki et al. [60] in the aptly named paper "Feature Diagrams and Logics: There and Back Again". Though they take the concept further, for us it suffices to view a feature diagram as specifying a system of propositional constraints as described in Table 4.1. The corresponding feature model (Definition 4.3) is simply the set of propositional models satisfying those constraints.

▷ **4.4. Example:** Figure 1.2 represents the following propositional constraints:

$$Ed \qquad Ed \Leftarrow Pr \qquad Ed \Leftarrow SH$$

$$Ed \Leftarrow EC \qquad Ed \Leftarrow TI \qquad EC \Leftarrow SA$$

$$Ed \Rightarrow (EC \wedge \neg TI) \vee (\neg EC \wedge TI)$$

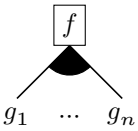
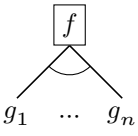
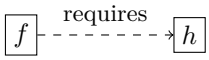
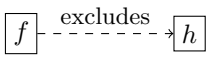
Notation	Meaning
	root feature f is *mandatory* f
	f selects *at least one* branch out of $g_1, ..., g_n$ $f \quad \Leftrightarrow \quad (g_1 \vee \cdots \vee g_n)$
	f selects *exactly one* branch out of $g_1, ..., g_n$ $f \quad \Leftarrow \quad (g_1 \vee \cdots \vee g_n)$ $f \quad \Rightarrow \quad (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \quad \vee$ $\vdots \quad \vee$ $(\neg g_1 \wedge \cdots \wedge \neg g_{n-1} \wedge g_n)$
	h is a *mandatory subfeature* on branch g $g \quad \Leftrightarrow \quad h$
	h is an *optional subfeature* on branch g $g \quad \Leftarrow \quad h$
	feature f *requires* feature h $f \quad \Rightarrow \quad h$
	feature f *excludes* feature h $f \quad \Rightarrow \quad \neg h$

Table 4.1: Compositional semantics for the most common feature diagram notations. More complex feature diagrams are built out of these ingredients by unifying certain features (f, h) and branches (g) without forming directed cycles. A feature can be either a root feature, a mandatory subfeature —● or an optional subfeature —○. A given feature can branch out into subfeatures through any number of groups. Groups can also contain a single branch, in which case exclusive semantics  and inclusive semantics coincide.

This, in turn, yields the following feature model:

$$\Phi_{\text{Editor}} = \left\{ \begin{array}{l} \{ Ed \}, \{ Ed, TI \}, \{ Ed, EC \}, \{ Ed, EC, SA \}, \\ \{ Ed, SH \}, \{ Ed, SH, TI \}, \{ Ed, SH, EC \}, \\ \{ Ed, SH, EC, SA \}, \{ Ed, Pr \}, \{ Ed, Pr, TI \}, \\ \{ Ed, Pr, EC \}, \{ Ed, Pr, EC, SA \}, \\ \{ Ed, Pr, SH \}, \{ Ed, Pr, SH, TI \}, \\ \{ Ed, Pr, SH, EC \}, \{ Ed, Pr, SH, EC, SA \} \end{array} \right\} \text{⌟}$$

We do lose some information in this representation; namely, we can't distinguish a feature from any of its mandatory subfeatures. This distinction is not important for the validity of feature configurations, but it does hold intuitive value for developers, and informs the design of the product line architecture. We can represent this extra information by keeping track of a subfeature relation $\multimap \subseteq \mathcal{F} \times \mathcal{F}$. But we won't have any use for this until Chapter 7.

4.3 Product Line Implementation

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

Our goal now is to set up a model for a product line code base with the ability to generate different products for different feature configurations, i.e., with support for automated product derivation.

4.3.1 Product Line Ingredients

To link feature symbols to the implementation layer, each delta in a product line is equipped with an *application condition*, specifying the feature selections for which it should be applied. We map deltas to their application conditions through an *application function*:

▶ **4.5. Definition (Application Function):** Given a delta set $D \subseteq \mathcal{D}$, an *application function* $\gamma\colon D \to \text{Pow}(\text{Pow}(\mathcal{F}))$ expresses, for each delta $d \in D$, the set of feature selections it is applicable to. Thus, $F \in \gamma(d)$ denotes that delta d is applicable to feature selection F. The set $\gamma(d) \subseteq \text{Pow}(\mathcal{F})$ is called the *application condition* of delta d. ⌟

In Figure 1.3 (page 9), application conditions are displayed as propositional logic formulas to the bottom right of each delta. While sets of feature configurations are convenient for formal reasoning, the original practice of using propositional formulas [163] is better for developers, as they allow deltas to be annotated with an *open world assumption*:

▷ **4.6. Example:** Delta $d_{Pr \wedge SH}$ is annotated with $Pr \wedge SH$, the features with which it is concerned. Its application condition is as follows:

$$\gamma(d_{Pr \wedge SH}) = \left\{ \begin{array}{l} \{ Ed, Pr, SH \}, \{ Ed, Pr, SH, EC, SA \}, \\ \{ Ed, Pr, SH, TI \}, \{ Ed, Pr, SH, EC \} \end{array} \right\} \text{⌟}$$

The advantage of the annotation $Pr \wedge SH$ is that the developer doesn't have to know about the features *EC*, *SA* and *TI*. The propositional annotation continues to be valid even when additional features are added to the feature model; the corresponding set of feature configurations will simply grow with it.

A delta model equipped with an application condition is called *annotated*:

▶ **4.7. Definition (Annotated Delta Model):** An *annotated delta model* is a tuple $adm = (D, \prec, \gamma)$, where $(D, \prec)$ is a delta model and $\gamma\colon D \to 2^{2^{\mathcal{F}}}$ is an application function. The set of all annotated delta models is denoted as $a\mathcal{DM}$. If the delta set or feature set is not clear from context, we attach a subscript as in $a\mathcal{DM}_{\mathcal{D},\mathcal{F}}$. ⌟

From an annotated delta model we can extract the deltas we need using the chosen feature selection, resulting in the *selected delta model*:

▶ **4.8. Definition (Selected Delta Model):** Given annotated delta model $adm = (D, \prec, \gamma)$ the delta model *selected* by feature selection $F \in \mathrm{Pow}(\mathcal{F})$ is defined:

$$adm \restriction F \quad \stackrel{\text{def}}{=} \quad (D_F, \prec_F)$$

where the set $D_F = \{\, d \in D \mid F \in \gamma(d) \,\}$ contains all applicable deltas, and $\prec_F = (\prec \cap\ D_F \times D_F)$ is the partial order, restricted accordingly. ⌟

A selected delta model can then be applied to a product as described in Chapter 3 using either sole-derivation, disjunctive or conjunctive semantics (Definitions 3.5, 3.25 and 3.26) to arrive at a target product.

This is the foundation of a *product line implementation*. Each contains a feature model specifying the implemented variations, an *annotated delta model* containing the modifications necessary to obtain them and a *core product* to apply those deltas to:

▶ **4.9. Definition (Product Line Implementation):** A *product line implementation* is a tuple $PLI = (\Phi, c, D, \prec, \gamma)$, where $\Phi \subseteq \mathrm{Pow}(\mathcal{F})$ is a feature model, $c \in \mathcal{P}$ is the core product, and $adm = (D, \prec, \gamma)$ is an annotated delta model. It is required that the following axioms hold:

a. All application conditions are valid: $\forall d \in D\colon\ \gamma(d) \subseteq \Phi$

b. All selected delta models are applicable: $\forall F \in \Phi\colon\ c \in \mathrm{pre}\,[\![\, adm \restriction F \,]\!]$

The set of all product line implementations is denoted $\mathcal{PLI}$. If the delta-, product- or featureset is not clear from context, we attach a subscript as in $\mathcal{PLI}_{\mathcal{D},\mathcal{P},\mathcal{F}}$. ⌟

So Figures 1.2 and 1.3 (pages 8 and 9) together offer an overview of the whole editor product line implementation, if we take into account that the core product $c = \varnothing$ is just the empty program.

Given a product line implementation, we can generate the end product(s) corresponding to some chosen feature configuration by selecting the correct delta model and applying the result to the core product. We consolidate this in a new notion of semantic evaluation of product line implementations:

▶ **4.10. Definition (Product Line Evaluation):** *product line evaluationProduct line evaluation* is a function $[\![\text{-}]\!]\colon \mathcal{PLI} \to \mathrm{Pow}(\mathrm{Pow}(\mathcal{F}) \times \mathcal{P})$ that associates with a given product line implementation PLI a relation $[\![\, PLI \,]\!] \subseteq \mathrm{Pow}(\mathcal{F}) \times \mathcal{P}$ mapping feature selections $F \subseteq \mathcal{F}$ to the products that may be generated by PLI when given F as input. For all product line implementations $PLI = (\, \Phi, c, adm \,)$:

$$F \;[\![\, PLI \,]\!]\; p \quad \overset{\text{def}}{\iff} \quad F \in \Phi \;\wedge\; c \;[\![\, adm \restriction F \,]\!]\; p$$

The above definition is for sole derivation semantics. Corresponding evaluation functions $[\![\text{-}]\!]_{\cup}$ and $[\![\text{-}]\!]_{\cap}$ for disjunctive and conjunctive semantics (Section 3.5, page 88) are defined analogously. ⌟

An effective procedure $\mathrm{prd}\colon \mathcal{PLI} \times \mathrm{Pow}(\mathcal{F}) \to \mathrm{Pow}(\mathcal{P})$ corresponding to these semantics —responsible for actually producing specific members of the product line— can be mechanically derived from the delta and delta model application procedures (Sections 3.2 and 3.5).

4.3.2 Unambiguity of Product Lines

Recall that even if deltas are deterministic, a delta model can have multiple distinct derivations, so without any further restrictions we are not guaranteed a unique product for a given feature configuration. When employing sole derivation semantics, it is up to the developers to make sure that all selected delta models are unambiguous (Section 3.3), leading to a unique derivation. We now lift conflict resolution and unambiguity to the product line level.

A product line implementation is unambiguous if every selected delta model is unambiguous. Since not all features are necessarily supposed to work together, we only care about the feature configurations from the embedded feature model:

▶ **4.11. Definition (Product Line Unambiguity):** A product line implementation $PLI = (\, \Phi, c, adm \,)$ is *unambiguous* iff:

$$\mathsf{UA}(PLI) \quad \overset{\text{def}}{\iff} \quad \forall F \in \Phi\colon\ \mathsf{UA}(adm \restriction F)$$

Recall the UA predicate for delta models from Definition 3.12 (page 79). ⌟

We can write out and simplify this condition, which first requires the following definition:

▶ **4.12. Definition (Joint Application Condition):** Given a set of deltas D' and an application function γ, the set of feature configurations for which all deltas in $D' \subseteq \mathrm{dom}(\gamma)$ are applicable, known as their *joint application condition*, is denoted as follows:

$$\gamma_{\cap}(D') \quad \overset{\text{def}}{=} \quad \bigcap_{d \in D'} \gamma(d)$$ ⌟

▶ **4.13. Lemma:** Written out and simplified, the unambiguity condition is as follows for a given product line implementation $PLI = (\, \Phi, c, D, \prec, \gamma \,)$

$$\underbrace{\forall x, y \in D\colon\ x \not\leftrightarrow y}_{\text{a}} \;\Rightarrow\; \underbrace{\forall F \in \gamma_{\cap}(\{x, y\})\colon}_{\text{b}}\ \underbrace{\exists z \in D\colon\ F \in \gamma(z) \;\wedge\; (x, y) \triangleleft z}_{\text{c}}$$

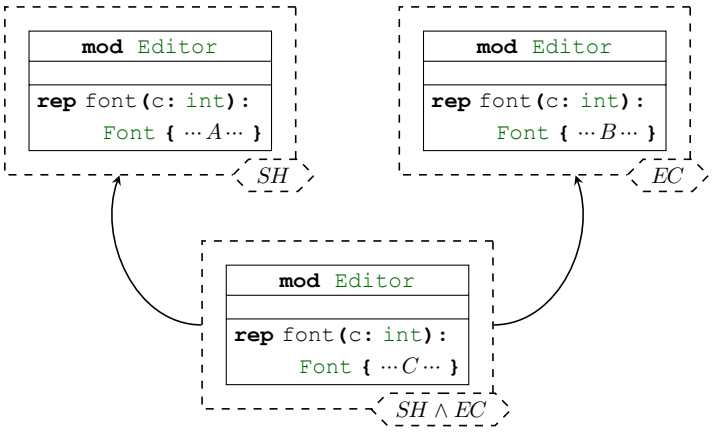


Figure 4.2: We simplify and zoom in on the $d_{SH} \lightning d_{EC}$ conflict of Figure 1.3 (page 9). The method body C can be found in Example 3.1 (page 71).

This means the unambiguity of a product line implementation can be decided by checking whether (a) for all conflicting deltas $x \lightning y$, (b) and all feature configurations for which both x and y are selected, (c) that there is a conflict resolving delta z which is also selected for F.

Proof: $\mathsf{UA}(PLI)$

$$
\begin{aligned}
&\overset{1}{\iff} \forall F \in \Phi\colon\ \mathsf{UA}\big((D, \prec, \gamma) \restriction F\big) \\
&\overset{2}{\iff} \forall F \in \Phi\colon\ \forall x, y \in D_F\colon\ && x \lightning y \ \Rightarrow\ \exists z \in D_F\colon && (x, y) \lhd z \\
&\overset{3}{\iff} \forall x, y \in D\colon\ \forall F \in \gamma_\cap(\{x, y\})\colon\ && x \lightning y \ \Rightarrow\ \exists z \in D_F\colon && (x, y) \lhd z \\
&\overset{4}{\iff} \forall x, y \in D\colon\ \forall F \in \gamma_\cap(\{x, y\})\colon\ && x \lightning y \ \Rightarrow\ \exists z \in D\colon\ F \in \gamma(z) \ \wedge\ && (x, y) \lhd z \\
&\overset{5}{\iff} \forall x, y \in D\colon\ x \lightning y \ \Rightarrow\ \forall F \in \gamma_\cap(\{x, y\})\colon\ && \exists z \in D\colon\ F \in \gamma(z) \ \wedge\ && (x, y) \lhd z
\end{aligned}
$$

Steps 1 and 2 apply Definitions 3.12 and 4.11. Step 3 is valid because both before and after the 'swap', the second universal quantification is restricted so as to end up with two deltas and a feature configuration for which they are applicable. Step 4 pushes the applicability of z inward by similar justification. Finally, in step 5 the conflict condition $x \lightning y$ can be pulled out because it is independent from the choice of F. □

As the set of feature configurations can be exponential in the number of features, this check could be rather expensive. As an alternative, we propose the notion of a *globally unambiguous* product line implementation, a property which implies unambiguity:

▶ **4.14. Definition (Global Unambiguity):** A product line implementation $PLI = (\Phi, c, D, \prec, \gamma)$ is *globally unambiguous* iff the following holds:

$$
\begin{aligned}
\mathsf{GUA}(PLI) \quad \overset{\text{def.}}{\iff} \quad \forall x, y \in D\colon\ & \big(x \lightning y \ \wedge\ \gamma_\cap(\{x, y\}) \neq \varnothing\big) \ \Rightarrow \\
& \exists z \in D\colon\ \gamma_\cap(\{x, y\}) \subseteq \gamma(z) \ \wedge\ (x, y) \lhd z \ \lrcorner
\end{aligned}
$$

So a product line implementation is globally unambiguous iff for every pair of conflicting deltas applied together, there is also one conflict resolving delta applied for at least the same feature configurations.

▷ **4.15. Theorem:** The Editor product line implementation of Section 1.4 is globally unambiguous.

Proof: The only two potential conflicts are $d_{SH} \not\downarrow d_{EC}$ and $d_{EC} \not\downarrow d_{TI}$. The former is resolved by $(d_{SH}, d_{EC}) \triangleleft d_{SH \wedge EC}$, with $\gamma_\cap(\{d_{SH}, d_{EC}\}) = \gamma(d_{SH \wedge EC})$, as illustrated in Figure 4.2. The latter does not need to be resolved, as $\gamma_\cap(\{d_{EC}, d_{TI}\}) = \varnothing$; those two deltas are never selected together. □

Global unambiguity can be checked by inspecting the product line implementation once and does not require any selected delta models to be generated. The following theorem states that any globally unambiguous product line implementation is also unambiguous:

▶ **4.16. Theorem:** A product line implementation that is globally unambiguous is also guaranteed to be unambiguous.

Proof: In the following proof, γ_y^x is used as an abbreviation for $\gamma_\cap(\{x, y\})$:

$$
\begin{array}{lllll}
 & \mathsf{GUA}(PLI) & & & \\
\overset{1}{\Longleftrightarrow} & \forall x, y \in D\colon & (x \not\downarrow y \wedge \gamma_y^x \neq \varnothing) & \Rightarrow & \exists z \in D\colon \gamma_y^x \subseteq \gamma(z) \quad \wedge\ (x, y) \triangleleft z \\
\overset{2}{\Longleftrightarrow} & \forall x, y \in D\colon & (x \not\downarrow y \wedge \gamma_y^x \neq \varnothing) & \Rightarrow & \exists z \in D\colon \forall F \in \gamma_y^x\colon F \in \gamma(z) \ \wedge\ (x, y) \triangleleft z \\
\overset{3}{\Longrightarrow} & \forall x, y \in D\colon & (x \not\downarrow y \wedge \gamma_y^x \neq \varnothing) & \Rightarrow & \forall F \in \gamma_y^x\colon \exists z \in D\colon F \in \gamma(z) \ \wedge\ (x, y) \triangleleft z \\
\overset{4}{\Longleftrightarrow} & \forall x, y \in D\colon & x \not\downarrow y & \Rightarrow & \forall F \in \gamma_y^x\colon \exists z \in D\colon F \in \gamma(z) \ \wedge\ (x, y) \triangleleft z \\
\overset{5}{\Longleftrightarrow} & \mathsf{UA}(PLI) & & &
\end{array}
$$

Steps 1 and 5 apply Definition 4.14 and Lemma 4.13. Step 2 is justified because it is performed in a context where it is known that $\gamma_y^x \neq \varnothing$. In step 4 that condition has become redundant, so it can be removed.

More interesting is the implication of step 3, which clarifies the difference between global unambiguity and general unambiguity. An implementation can be unambiguous without being globally unambiguous if some of its conflicts are resolved by different deltas z for different feature configurations F. □

It seems to be rare that a practical situation calls for different conflict resolvers for different feature configurations. Global unambiguity is easier to establish. In the delta modeling workflow described in Chapter 7, for example, it is guaranteed by construction.

4.4 Product Line Specification

We now define the concept of a *product line specification*: an abstraction of the desired semantics of a product line. Such a specification can be used both to guide the implementation process (further discussed in Chapter 7) and to verify the correctness of a given implementation.

A product line specification has two ingredients. The first is a feature model, as introduced in Section 4.2, which expresses the set of feature configurations that *should* be supported by the product line. The second ingredient is a *valuation function*, an abstract representation of the desired semantics for every feature. Note, in particular, that deltas don't play a rôle, so we can model specifications for any product line implementation technique.

4.4.1 Valuation Functions

A valuation function is a semantic interpretation of the *requirements* imposed on a product when it should support certain features. It maps a feature selection to the set of products deemed to implement those features correctly:

▶ **4.17. Definition (Valuation Function):** A *valuation function* $\mathrm{V}\colon \mathrm{Pow}(\mathcal{F}) \to \mathrm{Pow}(\mathcal{P})$ is a function taking a feature selection and returning the set of products that satisfy the required semantics of that feature selection. The following axiom needs to hold for all valuation functions V:

a. Compositionality: $\forall F, G \subseteq \mathcal{F}\colon \ \mathrm{V}(F \cup G) \ \subseteq \ \mathrm{V}(F) \cap \mathrm{V}(G)$ ⌟

Valuation functions are a concept originally from modal logic , a context in which we shall revisit them in Chapter 6. Axiom 4.17a represents a reasonably intuitive concept: if a product supports some combination of features, it also supports each feature individually — and every combination in between. The reverse is not generally true. A product can support a number of features individually without properly implementing the requirements of their combination.

▷ **4.18. Example:** For example, the Editor product $p = (\, d_{SH} \cdot d_{Pr} \,)\,(c)$ may implement both Printing and Syntax Highlighting, i.e., $p \in \mathrm{V}(\{\, SH \,\}) \cap \mathrm{V}(\{\, Pr \,\})$, but does not implement the combined functionality that we intended; namely, syntax highlighting on the printout: $p \notin \mathrm{V}(\{\, Pr, SH \,\})$. However, the product $q = (\, d_{Pr \wedge SH} \cdot d_{SH} \cdot d_{Pr} \,)\,(c)$ *does* implement the combination: $q \in \mathrm{V}(\{\, Pr, SH \,\})$. This is what we call *desired feature interaction*. ⌟

Even in an abstract setting, we can use the valuation function to express some useful properties. For instance,

$$\mathrm{V}(F \cup G) \ \neq \ \mathrm{V}(F) \ \cap \ \mathrm{V}(G)$$

means that special interaction is desired between the features of F and G. And

$$\forall F \subseteq \mathcal{F}\colon \ \mathrm{V}(F) \ = \ \mathrm{V}\,(\,F \cup G\,)$$

indicates that features in G have no semantics. This sometimes happens when features are used purely to categorize their subfeatures. Chapter 7 includes an example of this in an industrial case study.

We intend a valuation function to be represented *syntactically*, though this is difficult to demonstrate in an abstract setting. The following is an incomplete list of possible representations for the valuation function:

- In an object oriented setting, such as that of our running example, it might contain formal specifications regarding the presence and behavior of packages, classes and methods, to be verified statically. Alas, exploring this option is outside the scope of the thesis.
- In an industrial setting it may be used to support *test driven development* of software product lines. It would be represented as a collection of test cases T annotated with application conditions. The set $\mathrm{V}(F)$ would contain the products that pass all test-cases that are annotated with F.

  Taking this one step further, it would suggest the practice of maintaining test cases as a delta-based product line implementation, in parallel to the main software product line, and with roughly the same shape. It

would be based on a deltoid such as $(\mathrm{Pow}(T), \mathrm{Pow}(T), \cup)$, where products are sets of test-cases that the corresponding software product needs to pass, and deltas can augment them with additional test-cases.

4.4.2 Product Line Specifications

A product line specification contains a feature model and a valuation function:

▶ **4.19. Definition (Product Line Specification):** A *product line specification* is a pair $PLS = (\Phi, \mathrm{V})$ where $\Phi \subseteq \mathrm{Pow}(\mathcal{F})$ is a feature model (Definition 4.3) and $\mathrm{V}\colon \mathrm{Pow}(\mathcal{F}) \to \mathrm{Pow}(\mathcal{P})$ is a valuation function (Definition 4.17). We require that the following axiom holds:

a. No contradictory requirements: $\forall F \in \Phi\colon\ \mathrm{V}(F) \neq \varnothing$

The set of all product line specifications is denoted $\mathcal{PLS}$. If the deltoid or feature set is not clear from context, we attach a subscript as in $\mathcal{PLS}_{Dt,\mathcal{F}}$. ⌟

Semantically speaking, a product line *implementation* can be seen as a (partial) function performing automated product derivation, taking a feature configuration and returning the corresponding product (Figure 1.1). A *specification*, then, can be seen as the pre- and postcondition for that function. The feature model is the precondition; the valuation function is the postcondition.

This allows us to define a formal notion of product line correctness. That is, correctness of a product line implementation, both partial and total, with regard to a product line specification:

▶ **4.20. Definition (Product Line Correctness):** A product line implementation PLI is *partially correct* or *totally correct* w.r.t. a product line specification $PLS = (\Phi_\mathrm{S}, \mathrm{V})$ respectively iff:

$$PLI \models PLS \quad \overset{\text{def}}{\iff} \quad \forall F \in \Phi_\mathrm{S}\colon\ F \in \mathrm{pre}\,[\![PLI]\!] \ \Rightarrow\ [\![PLI]\!](F) \subseteq \mathrm{V}(F)$$

$$PLI \models_{\mathrm{tot}} PLS \quad \overset{\text{def}}{\iff} \quad \underbrace{\forall F \in \Phi_\mathrm{S}\colon}_{\text{a}}\ \underbrace{F \in \mathrm{pre}\,[\![PLI]\!]}_{\text{b}}\ \wedge\ \underbrace{[\![PLI]\!](F) \subseteq \mathrm{V}(F)}_{\text{c}}$$

Correctness for disjunctive and conjunctive semantics are defined analogously (Section 3.5). ⌟

Note its similarity to Definition 2.29 (page 45): (a) for all feature configurations of the specification, (b) if PLI implements that feature configuration, (c) then any product it might generate is valid according to the valuation function.

The following formulation is equivalent, but may offer more insight because it is at a lower level:

4.21. Lemma: For product line implementation $PLI = (\Phi_\mathrm{I}, c, adm)$ and product line specification $PLS = (\Phi_\mathrm{S}, \mathrm{V})$ we have:

$$PLI \models PLS \quad \iff \quad \forall F \in \Phi_\mathrm{S}\colon\ \Big(F \in \Phi_\mathrm{I}\ \Rightarrow\ [\![adm \restriction F]\!](c) \subseteq \mathrm{V}(F)\Big)$$

$$PLI \models_{\mathrm{tot}} PLS \quad \iff \quad \forall F \in \Phi_\mathrm{S}\colon\ \Big(F \in \Phi_\mathrm{I}\ \wedge\ [\![adm \restriction F]\!](c) \subseteq \mathrm{V}(F)\Big)$$

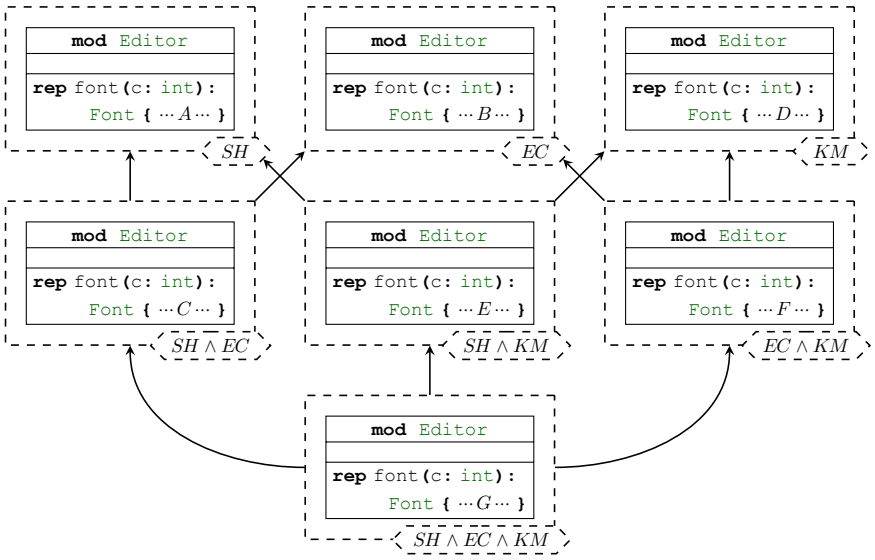


Figure 4.3: An extended version of Figure 4.2, explained in Example 4.22.

Proof: It is a relatively direct translation using Definition 4.10. Just note that

$$F \in \text{pre}\,[\![PLI]\!] \quad \Longleftrightarrow \quad F \in \Phi_{\text{I}}$$

is a valid equivalence because of Axiom 4.9b. □

4.5 Parametric Deltas

Now that the foundation for delta-based product lines is in place, we explore a problem with the approach. Conflict resolving deltas may be flexible, but they are also bulky. Section 4.5.1 considers a situation in the Editor product line where three independent deltas are all in conflict with each other, requiring a number of conflict resolving deltas exponential to the number of features.

To accomodate a more lightweight approach, Section 4.5.2 introduces *parametric deltas*, which can pass a chosen feature configuration on to the language of the underlying deltoid. This can give us some of the convenience of annotative variability techniques inside deltas. Parametric deltas do have their downsides, however. To offer some contrast, Section 4.5.3 presents an alternative solution for the Editor problem, based on fine-grained deltas.

Section 4.6 builds on the potential of parametric deltas and uses them to formalize *nested product lines*: product line implementations using nesting annotated delta models as their base.

4.5.1 Combinatorial Explosion of Conflict Resolvers

A potential problem with "a resolver for every conflict" is that an exponential number of them is required if many deltas mutually conflict:

▷ **4.22. Example:** Let's revisit the conflict in the Editor product line.

The two deltas d_{SH} and d_{EC} are each responsible for implementing a feature: $\gamma(d_{SH}) = \{F \in \Phi \mid SH \in F\}$ and $\gamma(d_{EC}) = \{F \in \Phi \mid EC \in F\}$. They are in conflict: $d_{SH} \lightning d_{EC}$, and we want to maintain global unambiguity. The idea is to develop a conflict resolving delta $d_{SH \wedge EC}$, applicable when both of those features are selected (Definition 4.14): $\gamma(d_{SH \wedge EC}) = \{F \in \Phi \mid SH, EC \in F\}$. So far so good.

Now what if a new feature is implemented: *Keyword Marking* (KM). It is similar to Syntax Highlighting, but responsible for giving keywords a bold typeface. To realize this feature, an additional delta is designed: d_{KM}. The problem is, d_{KM} has to redefine `font(int)` just like its siblings, so it is in conflict with both of them. Moreover, by the new feature model, all three features are independently selectable.

In order to regain global unambiguity and ensure the proper semantics for each feature configuration, we need to create at least two more conflict resolving deltas. Namely, $d_{SH \wedge KM}$ and $d_{EC \wedge KM}$, resolving $d_{SH} \lightning d_{KM}$ and $d_{EC} \lightning d_{KM}$. But it doesn't end there. The three conflict resolving deltas are now in conflict with *each other*. Thankfully, it does end eventually: if any two of the conflict resolving deltas are selected, the third will always be selected too. For example, we have $\gamma(d_{SH \wedge EC}) \cap \gamma(d_{EC \wedge KM}) \subseteq \gamma(d_{SH \wedge KM})$. So to wrap up this *three-way conflict*, one final conflict resolving delta $d_{SH \wedge EC \wedge KM}$ is needed.

Figure 4.3 illustrates that we need four conflict resolving deltas to fully resolve the conflicts arising from three features. The following is method body C, resolving $d_{SH} \lightning d_{EC}$ (recall the `@` notation from page 71):

```
(C) 1 Font result = new Font();
    2 result.setColor     (font@d_SH(c).color());
    3 result.setUnderlined(font@d_EC(c).underlined());
    4 return result;
```

The following is method body E, resolving $d_{SH} \lightning d_{KM}$:

```
(E) 1 Font result = new Font();
    2 result.setColor(font@d_SH (c).color());
    3 result.setBold (font@d_KM(c).bold());
    4 return result;
```

The following is method body F, resolving $d_{EC} \lightning d_{KM}$:

```
(F) 1 Font result = new Font();
    2 result.setUnderlined(font@d_EC (c).underlined());
    3 result.setBold      (font@d_KM(c).bold());
    4 return result;
```

The following is method body G, resolving the three-way conflict:

```
(G) 1 Font result = new Font();
    2 result.setColor     (font@d_SH (c).color());
    3 result.setUnderlined(font@d_EC (c).underlined());
    4 result.setBold      (font@d_KM(c).bold());
    5 return result;                                    ⌟
```

These deltas are not duplicating behavior, as such. They only combine behaviors by referring to the deltas that originally implement them. But they *are* duplicating code; boilerplate code, if you will. In the worst case, n independent features with conflicting implementations require $2^n - 1$ deltas, $2^n - n - 1$ of which are conflict resolvers.

It is possible that each of those feature combinations actually requires a distinct implementation, depending on the product line specification. In that case, this exponential complexity is inherent in the problem and delta models like the one in Figure 4.3 are precisely what we need for full control. But this is obviously not the case for the three-way conflict of the Editor. Indeed, in many practical scenarios such as this one, the glue code follows a uniform pattern which may be much more conveniently expressed in the underlying language.

4.5.2 Parametric Deltas

The way to accomplish the goal as described above is to allow the underlying structure of the delta access to the chosen feature configuration. We first define the abstract notion of such a *parametric delta*. We then instantiate it to software deltas. Finally we show how this solves our three-way conflict problem.

Abstract Parametric Deltas

Intuitively, we want every parametric delta to represent a partial function, accepting a feature selection as parameter and returning a traditional delta. In previous work [5] this was literally the case. But the strict separation of syntax and semantics in this thesis requires a different formulation. A partial function is a semantic concept, and we want to leave the representation of parametric deltas open to be decided for each concrete domain:

▶ **4.23. Definition (Parametric Deltoid):** A *parametric deltoid* is a deltoid $(\mathcal{P}, p\mathcal{D} \times \mathrm{Pow}(\mathcal{F}), [\![\,\text{-}, \text{-}\,]\!])$ with some set $p\mathcal{D}$ of what we call *parametric deltas* and a semantic evaluation function $[\![\,\text{-}, \text{-}\,]\!] : p\mathcal{D} \times \mathrm{Pow}(\mathcal{F}) \to \mathrm{Pow}(\mathcal{P} \times \mathcal{P})$. ⌟

This does not change the basic concept of deltoid (Definition 2.11, page 37); we simply make the set of deltas $\mathcal{D} = p\mathcal{D} \times \mathrm{Pow}(\mathcal{F})$ a set of pairs, each containing a parametric delta and a feature selection (its 'parameter'). The behavior of a semantic delta $[\![\, pd, F \,]\!] \subseteq \mathcal{P} \times \mathcal{P}$ is based on both.

However, when working with a parametric deltoid, delta models and product lines will be based on $p\mathcal{D}$ rather than $\mathcal{D}$. We need to adapt their respective semantic evaluation functions to pass on and supply the feature configuration at selection time:

▶ **4.24. Definition (Parametric Delta Model Evaluation):** Given a parametric deltoid $(\mathcal{P}, p\mathcal{D} \times \mathrm{Pow}(\mathcal{F}), [\![\,\text{-}, \text{-}\,]\!])$, *parametric delta model evaluation* $[\![\,\text{-}, \text{-}\,]\!] : \mathcal{DM}_{p\mathcal{D}} \times \mathrm{Pow}(\mathcal{F}) \to \mathrm{Pow}(\mathcal{P} \times \mathcal{P})$ is defined as follows for all *parametric delta models* $pdm \in \mathcal{DM}_{p\mathcal{D}}$ with a unique derivation $\mathrm{derv}(pdm) = \{\, pd \,\}$, and all feature selections $F \subseteq \mathcal{F}$:

$$[\![pdm, F]\!] \quad \stackrel{\text{def}}{=} \quad [\![pd, F]\!]$$

Corresponding evaluation functions $[\![\,\text{-}, \text{-}\,]\!]_\cup$ and $[\![\,\text{-}, \text{-}\,]\!]_\cap$ for disjunctive and conjunctive semantics are defined analogously. ⌟

▶ **4.25. Definition (Parametric Product Line Evaluation):** Given a parametric deltoid $(\mathcal{P}, p\mathcal{D}\times\mathrm{Pow}(\mathcal{F}), [\![\,\text{-}\,,\,\text{-}\,]\!])$, *parametric product line evaluation* $[\![\,\text{-}\,]\!]\colon \mathcal{PLI}_{p\mathcal{D}} \to \mathrm{Pow}(\mathrm{Pow}(\mathcal{F}) \times \mathcal{P})$ is defined as follows for all *parametric product line implementations* $pPLI = (\Phi, c, padm) \in \mathcal{PLI}_{p\mathcal{D}}$:

$$F \;[\![\, pPLI \,]\!]\; p \quad \overset{\text{def}}{\Longleftrightarrow} \quad F \in \Phi \;\wedge\; c \;[\![\, padm \restriction F, F \,]\!]\; p$$

Corresponding evaluation functions $[\![\,\text{-}\,]\!]_{\cup}$ and $[\![\,\text{-}\,]\!]_{\cap}$ for disjunctive and conjunctive semantics are defined analogously. ⌟

Parametric Software Deltas

Concrete parametric deltas can be realized in any number of ways. For software deltas it would feel most natural to expose the available feature symbols $f \in \mathcal{F}$ as boolean constants in the underlying programming language:

▷ **4.26. Definition (Parametric Software Deltas):** The set of *parametric software deltas* $p\mathcal{D}_{\text{pkg+}}$ is like the set of fine-grained software deltas $\mathcal{D}_{\text{pkg+}}$, but can contain feature symbols $f \in \mathcal{F}$ in any Boolean context. ⌟

Semantic evaluation will substitute truth-values for the feature symbols to get back to the fine-grained software deltoid situation of Definition 3.22:

▷ **4.27. Definition (Parametric Software Delta Evaluation):** Semantic evaluation for parametric software deltas $[\![\,\text{-}\,,\,\text{-}\,]\!]\colon p\mathcal{D}_{\text{pkg+}} \times \mathrm{Pow}(\mathcal{F}) \to \mathrm{Pow}(\mathcal{PKG} \times \mathcal{PKG})$ is defined as follows. For all products $p, q \in \mathcal{PKG}$, deltas $pd \in p\mathcal{D}_{\text{pkg+}}$ and feature configurations $F \subseteq \mathcal{F}$:

$$p \;[\![\, pd, F \,]\!]\; q \quad \overset{\text{def}}{\Longleftrightarrow} \quad p \;(\!|\, d \,|\!)\; q$$

where $(\!|\,\text{-}\,|\!)$ is fine-grained software delta evaluation (Definition 3.22) and $d \in \mathcal{D}_{\text{pkg+}}$ is the software delta obtained by replacing every occurrence of $f \in F$ in pd with `true` and every occurrence of $f \in (\mathcal{F} \setminus F)$ in pd with `false`. ⌟

A Parametric Editor Product Line

The following example, illustrates a possible way to solve the three-way conflict problem of the Editor product line. The four conflict resolving deltas of Example 4.22 are so similar, it would be more sensible to use a single parametric delta:

▷ **4.28. Example:** The parametric solution to the three-way conflict problem in the Editor product line is shown in Figure 4.4. The following is method body H, combining the three features:

```
(H) 1 Font result = new Font();
    2 if (SH ) result.setColor      (font@d_SH (c).color());
    3 if (EC ) result.setUnderlined(font@d_EC (c).underlined());
    4 if (KM) result.setBold       (font@d_KM(c).bold());
    5 return result;                                          ⌟
```

During the process of automated product derivation, the feature symbols in the implementation are replaced with the proper truth values (Definition 4.27):

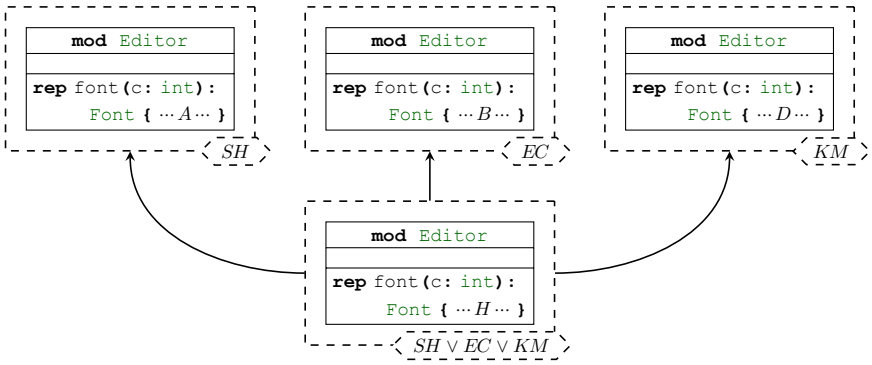


Figure 4.4: A version of Figure 4.3 using a parametric delta. Method body H can be found in Example 4.29.

▷ **4.29. Example:** Given a feature configuration of $F = \{ SH, KM \}$, the method body H from Example 4.28 would become the following:

```
(H′) 1  Font result = new Font();
     2  if (true ) result.setColor      (font@d_SH (c).color());
     3  if (false) result.setUnderlined(font@d_EC (c).underlined());
     4  if (true ) result.setBold       (font@d_KM(c).bold());
     5  return result;
```

We assume that these **if** constructs are resolved at compile-time, so that Line 3 can be discarded. ⌟

Note that the Example 4.28 method body is basically using an *annotative* variability technique. Parametric deltas offer a mix of the annotative and compositional approaches [108]. A powerful combination.

But with great power comes great responsibility [174]. In principle, a whole product line could be encoded in a single parameteric delta, in which the code is annotated with feature conditions to handle all cases. But, as discussed in Section 1.2.3, annotative approaches do not benefit from modularity or separation of concerns, which were among our main goals. Therefore, parametric deltas are recommended only for resolving multi-way conflicts or feature interactions, and then only when this significantly reduces the amount of code or effort required. Parametric deltas are a double-edged sword.

4.5.3 Interlude: A Fine-grained Software Delta Solution

For the three-way conflict problem in the Editor product line, there is actually an alternative solution using only the facilities of fine-grained software deltas:

▷ **4.30. Example:** The three-way conflict can be avoided altogether by using the fine-grained software delta **insert** operation as shown in Figure 4.5. The following is statement delta I, which inserts the behavior of SH:

```
(I)  insert { result.setColor(SHfont.color()); };
```

The following is statement delta J, which inserts the behavior of EC:

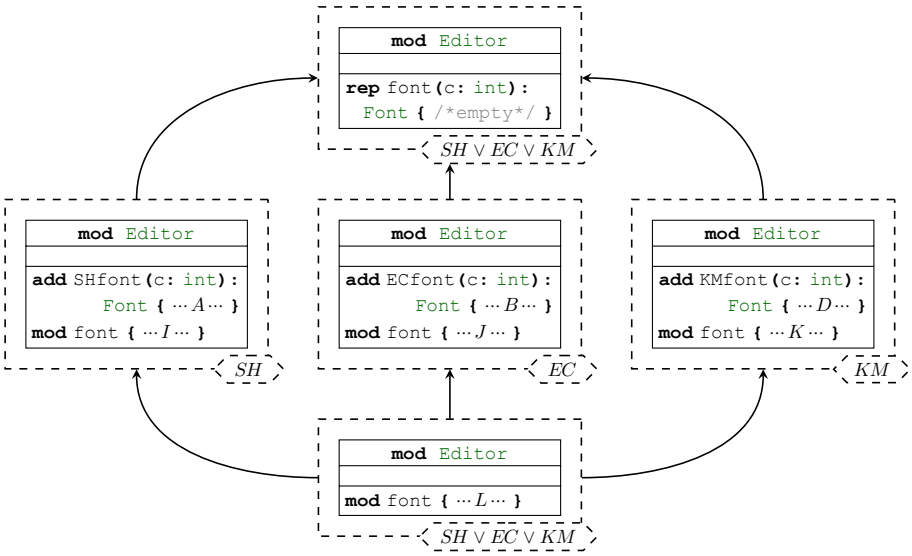


Figure 4.5: An alternative to Example 4.28, explained in Example 4.30.

(J) `insert { result.setUnderlined(ECfont.underlined()); };`

The following is statement delta K, which inserts the behavior of KM:

(K) `insert { result.setBold(KMfont.bold()); };`

The following is statement delta L, which adds the surrounding statements:

(L)
```
1 prepend { Font result = new Font(); };
2 append  { return result; };
```

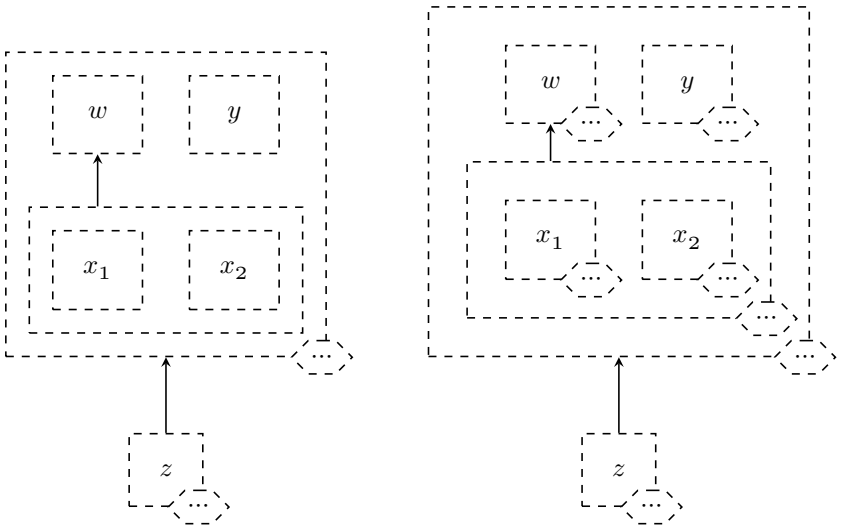
This is more modular, but it is also more bulky, and the nondeterministic nature of **`insert`** needs to be understood well enough to avoid potential pitfalls. Parametric software deltas are based on more generally familiar annotative techniques, and may be preferable in certain situations.

4.6 Nested Product Lines

Section 3.6 discussed nested delta models. At this point the next logical step is to extend them to nested annotated delta models, in order to get *nested product lines*.

There is a way we can achieve something along those lines already, just by applying previously defined concepts. We can equip a product line implementation with an annotated delta model $(D, \prec, \gamma)$ based on a delta set $D \subseteq \mathcal{D}_{\triangle}$ (Definition 3.31). This results in a nesting delta model of which only the outermost deltas are annotated with an application condition, something we might call a *shallow annotation* (Figure 4.6a).

But if we want to achieve a *deep annotation*, parametric deltas provide a way:



(a) shallow annotation (b) deep annotation

Figure 4.6: The difference between a shallow and a deep annotation.

▶ **4.31. Definition (Annotated Delta Model Closure):** Define the following family of parametric delta sets for all natural numbers $n \in \mathbb{N}$:

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

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

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

We require that $\mathcal{D}$ didn't contain any annotated delta models to begin with. ⌟

Finally, we define an appropriate evaluation function, just as we did for parametric software deltas before (Definition 4.27):

▷ **4.32. Definition (Deeply Annotated Delta Evaluation):** Given a deltoid $Dt = (\mathcal{P}, \mathcal{D}, (\!|\,\text{-}\,|\!))$ and parametric deltoid $pDt = (\mathcal{P}, \mathrm{a}\mathcal{D}_{\mathbb{A}} \times \mathrm{Pow}(\mathcal{F}), [\![\,\text{-},\,\text{-}\,]\!])$, we define *deeply annotated delta model evaluation* $[\![\,\text{-},\,\text{-}\,]\!] \colon \mathrm{a}\mathcal{D}_{\mathbb{A}} \times \mathrm{Pow}(\mathcal{F}) \to \mathrm{Pow}(\mathcal{P} \times \mathcal{P})$ as follows for all simple deltas $d \in \mathcal{D}$, annotated delta models $adm \in \mathrm{a}\mathcal{DM}_{\mathrm{a}\mathcal{D}_{\mathbb{A}}}$ and feature configurations $F \subseteq \mathcal{F}$:

$$\begin{aligned} [\![\, adm, F \,]\!] &\stackrel{\text{def}}{=} [\![\, adm \restriction F, F \,]\!] \\ [\![\, d, F \,]\!] &\stackrel{\text{def}}{=} (\!|\, d \,|\!) \end{aligned}$$

Corresponding evaluation functions $[\![\,\text{-},\,\text{-}\,]\!]_{\cup}$ and $[\![\,\text{-},\,\text{-}\,]\!]_{\cap}$ for disjunctive and conjunctive semantics are defined analogously. ⌟

A *nested product line implementation* is then simply a *parametric product line implementation* (Definition 4.25) based on a deeply annotated deltoid (Definition 4.32). A resulting annotated delta diagram would look like Figure 4.6b.

4.7 Conclusion

We've spoken of features before now, but this chapter is where the concept of feature is formally introduced and integrated into ADM. These features are merely labels, but play a prominent rôle throughout the rest of the story. They are traditionally used in a *feature model* as a way to identify the possible products of a *product line*.

In ADM, producing the product corresponding to a specific feature selection is done by preparing a large delta model, and annotating each delta with an *application condition*. Specifications may be prepared for each significant feature combination; feature combination —not feature— because often, two features that are otherwise independent need to satisfy additional requirements when they are selected together.

Apart from providing a characterisation of product line correctness, this chapter lifts a number of concepts from Chapter 3 to the product line level, such as unambiguity and nesting, and introduces a number of other useful concepts, such as *parametrized deltas*.

4.8 Related Work

Much work related to ADM-based product lines has been discussed in earlier chapters. But there are a number of interesting comparisons left to make with regard to feature modeling and product derivation.

4.8.1 Feature Modeling

Feature-Oriented Domain Analysis (FODA) as a way to model the commonality and the variability of a set of systems on a specification level has been in use since the early 1980's [105]. In this thesis, as in its corresponding publications, feature models are used (Section 4.2) to express this variability. But feature modeling is not the only studied approach for characterizing different products in a product line. Czarnecki et al. [61] compare feature modeling with *decision modeling*, which is based on setting values for specific Boolean, numerical and enumerated variables. It is interesting to note that if a feature model is simplified to a set of feature configurations, as we do in Definition 4.3, it becomes very similar to a decision model, e.g., one with a set of Boolean variables. According to Czarnecki et al., most of the differences between the two approaches are historical and the two are converging. The biggest remaining difference is that feature modeling has specific support for expressing commonality as well as variability.

4.8.2 Product Derivation

First, we should note that our definition of 'product' deviates somewhat from existing literature in feature modeling [26, 27, 53, 66], in which a product is usually uniquely defined for a given feature configuration. For us, the term refers to a specific implementation, multiple of which may be potential candidates for a given selection of features (Definition 4.10).

Automated product derivation has been widely recognized to improve quality and reduce time to market for large software systems [57, 65]. Perrouin et al. [150] note a dichotomy in the way variability approaches support product derivation. On the one hand, they say, approaches that support fully automated product derivation lack the flexibility to adapt to the needs of specific customers. On the other hand, approaches that focus on flexibility lack automation. They propose an approach where a feature configuration directs the *composition* of core assets, the result of which is then *transformed* to obtain a target product.

We would suggest that ADM is fully capable of modeling this approach, as composition and transformation are much the same for ADM. The transformation required to implement 'special wishes' for a specific feature configuration can be encoded in a delta. It is then a simple matter to annotate this delta with a specific (i.e., narrow) application condition (Definition 4.5), and make it a maximal element in the application order (Definitions 1.23 and 3.2).

Existing compositional approaches to automated product derivation have been discussed exhaustively in previous chapters. Existing annotative approaches include conditional compilation [171], frames [181] and Colored Featherweight Java (CFJ) [106]. Another noteworthy existing project is CIDE (Colored IDE) [108], a tool which displays annotated code in different colors —each representing a feature— and achieves a kind of visual separation of concerns this way, gaining some of the benefits of compositional techniques.