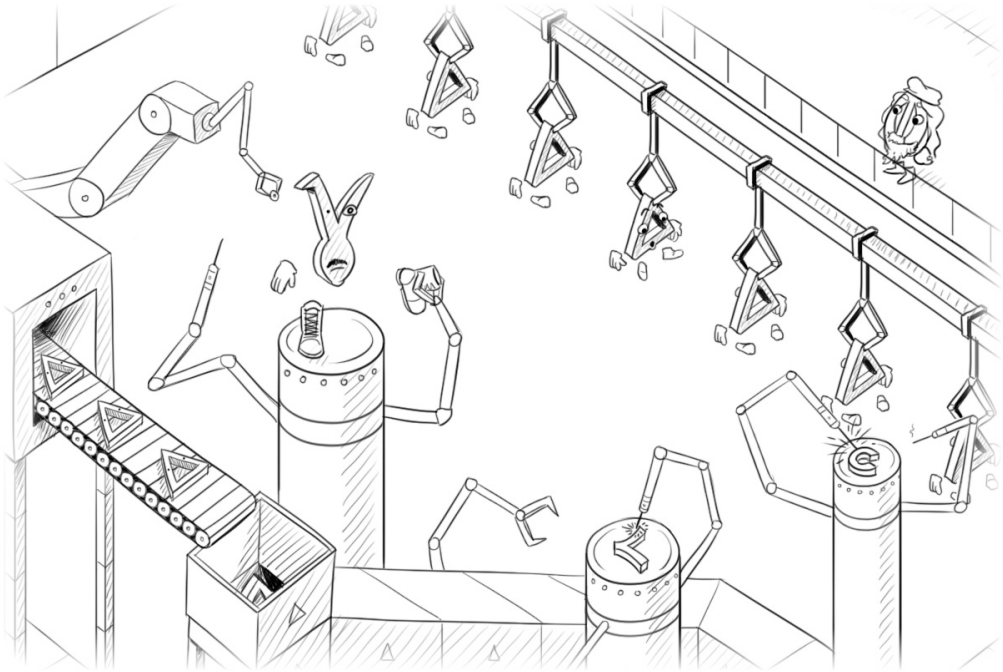
7

Delta Modeling Workflow

On the Development of Delta-based Product Lines



7.1 Introduction

Until now we have seen what is *possible* with Abstract Delta Modeling; what the various formal artefacts look like and what they mean. But it may still not be clear how they should actually be used in practice. If a team of developers started out with only a product line specification, how would they actually build and organize the product line implementation? How should the deltas be ordered and what should be their application conditions and content, for maximal reuse of code and isolated, concurrent development of features?

> ***Goal:*** *Describe how delta-based product lines might be built.*

This chapter proposes a specific development workflow for ADM, dubbed *Delta Modeling Workflow (DMW)*. The structured and flexible nature of ADM lends itself quite naturally to a systematic approach to building product lines. This chapter stays at the same level of abstraction as before, but approaches the topic from the other side. It describes, step-by-step, how to build a product line from scratch. At the moment, it is far from a practical guide, as it requires a fully defined product line specification in advance — an unrealistic requirement in modern engineering practices. But it is the first step in guiding the proper use of the delta modeling concepts introduced in previous chapters.

Of course, there may be many ways to use delta modeling to good effect. Indeed, many tools are eventually put to innovative uses that were initially unintended. Let's just say that ADM lends itself naturally to a certain way of working which happens to exhibit favorable properties. Following it leads to a well-structured product line that automatically exhibits two desirable properties: global unambiguity (Definition 4.14, page 105) and total correctness with regard to the specification (Definition 4.20, page 108). The work is split up into well-defined jobs derived from that specification.

Most importantly, the workflow naturally supports concurrent development. Multiple developers can work on parts of a non-trivial product line implementation at the same time and in isolation without breaking global unambiguity or correctness. An important reason for this is the concept of *delta model locality*: any delta under development need only take into account the existing deltas that occupy *subordinate positions* in the delta model.

Of course, a development workflow, of all things, should be evaluated in practice. Formal proofs, while valuable, are not enough to guarantee a good practical experience. Therefore, the DMW should be evaluated based on its application to an industrial scale system.

> ***Goal:*** *Test the delta modeling workflow on an industrial scale system in order to evaluate its practical applicability.*

To that end, the workflow was used to model the replication system of the *Fredhopper Access Server (FAS)* product line, in one of the industrial scale case studies of the HATS project. This was done using the *Abstract Behavioral Specification (ABS)* language of the HATS project, in which delta modeling is an integrated component.

Figure 7.1: Two different feature diagrams representing the same Definition 4.3 style feature model $\Phi = \{\{f, g\}, \{f, g, h\}\}$.

The chapter is organized as follows: Section 7.2 enriches the specification of product lines to include a subfeature relation. Section 7.3 introduces the fundamental notion of locality, setting the stage for Section 7.4 to describe the workflow itself. Section 7.5 describes the ABS language and provides a succinct DMW description in concrete ABS terms. Following that, Section 7.6 discusses its application to the replication system product line of FAS. Finally, Sections 7.7 and 7.8 offer concluding remarks and discuss related work.

In Appendix A (page 210), the beneficial properties of DMW are proved formally. It includes a formulation of the workflow using operational semantics.

7.2 The Subfeature Relation

> From this point on, assume that a deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \varepsilon, [\![\text{-}]\!])$ and a feature set $\mathcal{F}$ are given. Assume also that the deltoid exhibits consistent conflict resolution (Definition 3.18).

Feature models as formalized in Section 4.2 are not as useful for developers as they could be. When we view a feature model as the set of all possible feature configurations, we disregard the intended hierarchical structure between features. Compared to a traditional feature model [66, 105, 166], the "Φ" representation from Definition 4.3 (page 100) lacks some useful information. For instance, we lose the distinction between the two feature models in Figure 7.1, which would both have $\Phi = \{\{f, g\}, \{f, g, h\}\}$.

Since the feature diagram notation is quite common in product line engineering [166], it is a sensible structure to base the workflow on. To capture the hierarchy represented by feature diagrams, we introduce the following binary relation into the product line specification tuple (Definition 4.19, page 108):

▶ **7.1. Definition (Subfeature):** The binary *subfeature* relation is a strict partial order $\multimap \subseteq \mathcal{F} \times \mathcal{F}$. Write $f \multimap g$ when g is a subfeature of f and f is a *superfeature* of g. ⌟

Note that this definition allows one feature to have multiple direct superfeatures (a 'join' in the feature diagram), something that is not standard in feature modeling, but useful, as you may remember from Section 5.2 (page 120).

The subfeature relation recognizes both mandatory and optional subfeatures. We only use it to formally introduce the feature diagram structure. Information on optionality, grouping, implication, exclusion and more can be derived from the combination of Φ and $\multimap$. For example, if $f \multimap g$ and there exists a feature configuration that includes f but not g, we know that g must be an optional subfeature rather than a mandatory one.

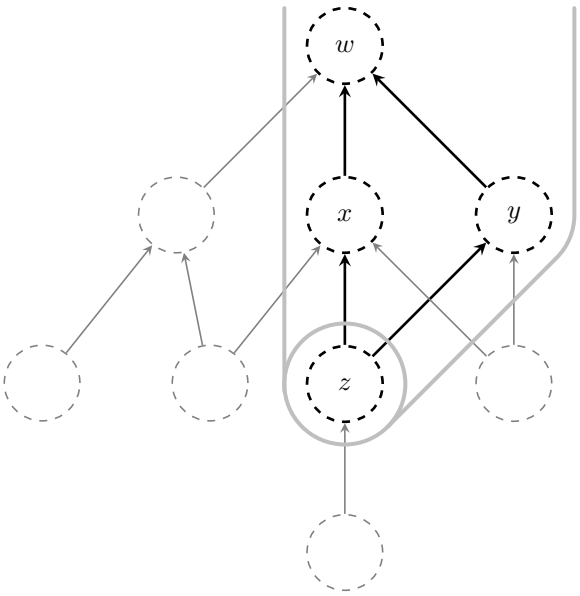


Figure 7.2: The local delta model of z in the context of a larger delta model.

7.3 Locality

When developing a specific delta in the (annotated) delta model of a product line, it would be inconvenient if we had to consider all other deltas to make sure the combined whole works properly — this would defeat the purpose of using delta modeling for modularity and separation of concerns in the first place. Thus the workflow should only require that local constraints are met, and then guarantee beneficial properties for the entire product line implementation by construction. But what does 'local' mean in the context of ADM?

▶ **7.2. Definition (Local Delta Model):** Given a delta model $dm = (D, \prec)$, the *local delta model* of a given delta $d \in D$ is defined as follows:

$$\downarrow d \quad \stackrel{\text{def}}{=} \quad (\, D',\ \prec \cap\ D' \times D' \,)$$

where $D' = \{\, x \in D \mid x \preccurlyeq d \,\}$ (known as the *principal ideal* of d in $\prec$). If the delta model is not clear from context, we attach a subscript as in $\downarrow_{dm}$. ⌟

This concept embodies a basic principle of the DMW: when engaging in the implementation of a new delta z, you already know which position in the delta model it will occupy. During development and maintenance you only have to know about the deltas that are to be applied earlier —those that z has control over (Figure 7.2)— and you need to establish certain correctness properties only over this local delta model.

Conversely, during long term maintenance, when any delta x is changed, it will always be clear which other deltas may now be out-of-date and in need of attention: all deltas $z \succ x$, which have x in their local delta model.

But in order for this to be enough to guarantee global properties, we need to place one restriction regarding the balance between the deltoid and the valuation function. Namely, we need to ensure that deltas that are *not* related

through the application order, and are therefore not in each others local delta model, cannot influence each others semantic effects on the final product; or, if they do, that this can be automatically detected by them being in conflict (Definition 3.7, page 76). We now assume this property of *non-interference* (formally described in Definition A.2, page 211). It is independent of any specific product line, though does depend on the kind of features that need to be implemented. Systems that break this restriction might include deltas that can add advice in aspect-oriented languages [114] that have effects beyond the entities they overwrite, or features with mutually exclusive specifications.

7.4 Workflow Description

The goal of the workflow is to start with a product line specification and implement from this a product line, by implementing all features, resolving all conflicts and implementing all desired feature interaction in an iterative process, maximally exploiting parallelism in the development.

7.4.1 Input

The input to the workflow is a product line specification and subfeature relation. The feature model Φ indicates which feature configurations need to be derivable. The order of the workflow steps is guided by the subfeature relation $\dashrightarrow$. The valuation function V, in a sense, guides the implementation of each individual delta.

7.4.2 Output

The output of the DMW is a product line implementation $(\Phi, c, D, \prec, \gamma)$. The goal is for this implementation to be totally correct with regard to the specification (Definition 4.20, page 108). One of the ways we make this easier is by ensuring global unambiguity. This means we'll be able to work with sole derivation semantics for delta models (Definition 3.5, page 75). Taking advantage of ambiguous semantics (Section 3.5, page 88) is planned as future work.

The feature model Φ of the implementation will be the same as that of the specification. While it is true that a product line implementation is allowed to implement more feature selections than are specified —while maintaining total correctness—, that is not the goal of this workflow.

We make the core product an initial product $c = 0$ (Definition 2.58, page 57) and do everything with deltas, a practice that has been dubbed *pure delta oriented programming* [162], or, in this case, *pure delta modeling*. While this is not required —i.e., it is possible to implement mandatory features in the core product— the choice simplifies the workflow description. Note, however, that optional features should never be implemented in the core product (with the intention of selectively 'removing' them with deltas) as this is incompatible with the workflow and can be said to be less flexible and robust.

The annotated delta model $(D, \prec, \gamma)$ is initialized as empty and built up during the workflow.

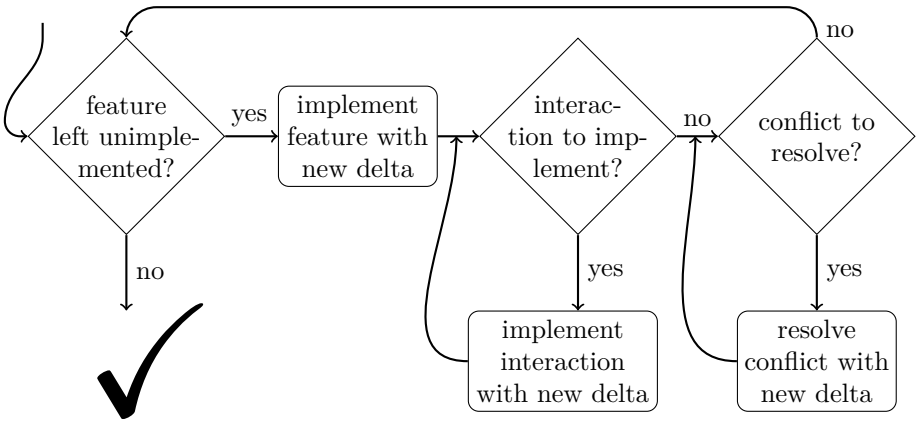


Figure 7.3: An intuitive overview of the development workflow.

7.4.3 A Sequential View

An overview of the workflow process is shown in Figure 7.3. Regard it as the flowgraph that a single developer would follow to implement a product line:

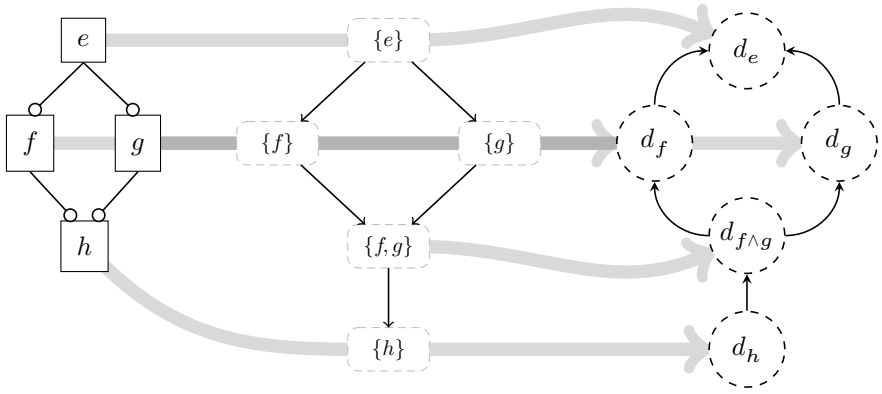
1. Is there a feature f that still needs to be implemented?
   If not, go to step 7.
2. Implement feature f with new *feature implementation delta* d_f.
3. Is there a required interaction between a set of implemented features F with $f \in F$ that still needs to be implemented? If not, skip to step 5.
4. Implement this interaction with a new *feature interaction delta* d_F.
   Then go back to step 3.
5. Is there an unresolved conflict $x \lightning y$ involving any of the deltas introduced in this iteration? If not, go back to step 1.
6. Resolve the conflict with a new *conflict resolving delta* $d_{\{x,y\}}$.
   Then go back to step 5.
7. The product line implementation is fininshed.

This was the workflow description used in the first DMW papers [5, 7]. It gives a good intuition as to what the workflow is all about. But it is not ideal for describing concurrent development by multiple engineers. Therefore, we break up this flowgraph into its constituent steps, and set up a proper dependency model.

7.4.4 Jobs

We now introduce the higher-level concept of *jobs*, each of which involve the development of a specific delta to place into the annotated delta model in progress. We distinguish between two kinds of job, each with a specific purpose:

- *Feature implementation jobs*, identified by a feature set $F \subseteq \mathcal{F}$, are to develop a delta responsible for either the implementation of a single feature —when $F = \{f\}$ for some f— or the interaction between a set of features. Conceptually these two cases are really the same, so it is more elegant to drop the distinction.



(a) feature order (b) job order (c) delta order

Figure 7.4: How the subfeature relation ⇸ informs the job order and how that, in turn, informs the delta application order $\prec$. In this example, features f and g require additional implementation effort, and their mutual subfeature h is implemented last.

- *Conflict resolution jobs*, identified by a finite set of already developed deltas $C \subseteq \mathcal{D}$, implement a delta to resolve the conflict(s) between the deltas in C.

Both the old and new workflow descriptions resemble an algorithm [116], but fail to be one for a simple reason: neither type of job can generally be *automated*. Both feature implementation and conflict resolution require creativity and domain-knowledge. So rather than provide well-defined instructions, a job imposes requirements on the local delta model of the new delta (Definition 7.2). The delta needs to be developed in a way that satisfies those requirements.

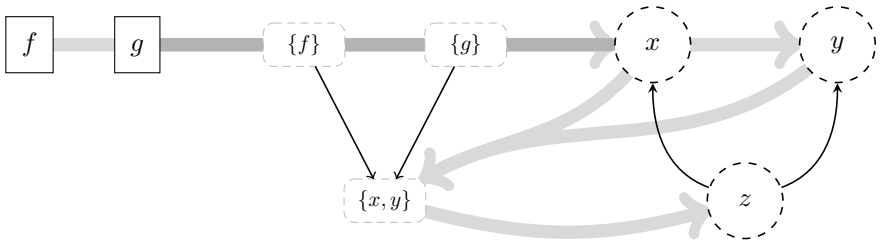
Although there are many jobs that can be performed concurrently, there are some that need to be performed in a specific order. Characterizing this order is one of the main contributions of this chapter. It is a strict partial order, and it is reflected in the following ways (Figure 7.4):

- the ⇸ relation, strict partial order of the feature diagram,
- the order in which the jobs are to be performed, and
- the $\prec$ relation, strict partial order of the delta model under development.

Simply put: the feature order ⇸ informs the job order which, in turn, informs the application order $\prec$.

Let's first discuss how feature implementation jobs are ordered, and forget about conflict resolving jobs for now. The main idea is that individual features f and g are implemented in the strict partial order of the subfeature relation ⇸. That is, if f ⇸ g, then f (the superfeature) will be implemented before g (the subfeature). This is reasonable; as base functionality should naturally be in place before it is extended by subfeatures.

But in the general case, feature implementation jobs are feature *sets*, sometimes larger than one. We extend the subfeature relation ⇸ to sets of features, so we can use it to order all such jobs:



(a) feature order (b) job order (c) delta order

Figure 7.5: How conflicts between existing deltas introduce new jobs.

▶ **7.3. Definition (Feature Combination Order):** We extend subfeature relation $\twoheadrightarrow$ to sets of features. The resulting *feature combination order* is the smallest strict partial order $\twoheadrightarrow \subseteq \mathrm{Pow}(\mathcal{F}) \times \mathrm{Pow}(\mathcal{F})$ such that, for all features $f, g \in \mathcal{F}$ and feature combinations $F, G \subseteq \mathcal{F}$ with $f, g \notin F \cup G$, the following axioms hold:

$$\textbf{a.}\quad F \subset G \implies F \twoheadrightarrow G$$

$$\textbf{b.}\quad f \twoheadrightarrow g \implies F \cup \{f\} \twoheadrightarrow G \cup \{g\}$$

When $F \twoheadrightarrow G$ we say that F is *weaker* and that G is *stronger*. ⌟

▶ **7.4. Example:** Figure 7.4 illustrates how the subfeature relation guides the order in which the job transitions are allowed to take place. For example, the features $e, f, g, h \in \mathcal{F}$ have $e \twoheadrightarrow f,g \twoheadrightarrow h$. So, with regard to feature combinations, we have $\{e\} \twoheadrightarrow \{f\},\{g\} \twoheadrightarrow \{f, g\} \twoheadrightarrow \{\,h\,\}$, as shown in Figure 7.4b. ⌟

Conflict resolving deltas are implemented through conflict jobs $C \subseteq D$. Such jobs are introduced from analysis on the current state of the product line implementation, rather than from analysis of the specification. This is illustrated in Figure 7.5. When existing deltas are in conflict (Figure 7.5c), a conflict job can be introduced (Figure 7.5b) for the implementation of a new delta to resolve the conflict (back to Figure 7.5c).

7.5 The Abstract Behavioral Specification Language

The *Abstract Behavioral Specification (ABS)* language [8, 52, 100] was developed within the FP7 EU project HATS, the project that started and guided my PhD research [80]. This is one of the only languages with delta modeling integrated into its core design (if not the only). As a member of the HATS project, I collaborated on the implementation of delta modeling in ABS and described the delta modeling workflow in terms of ABS constructs.

Section 7.5.1 provides a short background on the ABS language. Section 7.5.2 describes the concrete 'ABS delta modeling workflow manual' [8].

7.5.1 Background

The ABS language is designed for formal modeling and specification of concurrent, component-based systems at a high level, though is perfectly capable of producing executable programs. Particularly, it is targeted at complex software systems that exhibit a high degree of variation, such as software product lines.

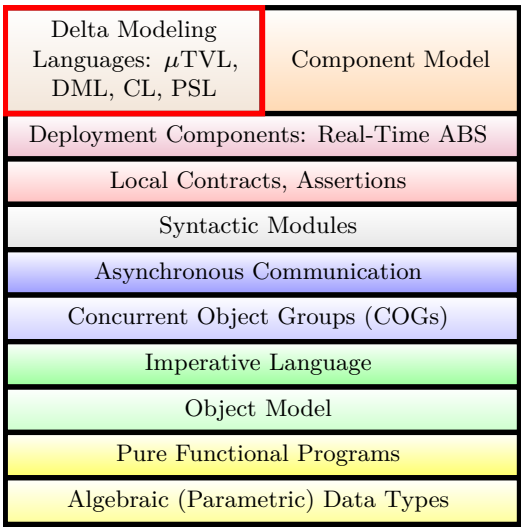


Figure 7.6: Layered Architecture of ABS[1]

Figure 7.6 describes the layered architecture of the ABS language. At its most fundamental levels, it provides pure functional programming constructs, algebraic data types, an object model and imperative language constructs. The main contributions of the HATS project are built on top: concurrency constructs based on the concept of COGs, a language layer for behavioral specification as well as module and component structures.

The top left layer is of interest to us. Delta modeling in ABS consists of four languages: μTVL, DML, CL and PSL.

μTVL is a feature description language based on a subset of TVL [46]. It is used to describe the variability of a product line in terms of (attributed) feature models (Section 4.2).

The delta modeling language DML is used to develop the delta modules containing modifications of a core ABS model. A delta module in ABS is similar to the software deltas of Section 2.3, and can modify classes, methods and fields on a course-grained level. The previous implementation of a method in the derivation can be invoked by using the **`original()`** call (similar to the **`super`** keyword of AHEAD [31]), though it is not possible to invoke specific method verions through the name of the delta.[2] As a bonus, delta modules can be parametrized by specific values as well as the 'feature Booleans' (Section 4.5).

The configuration language CL links μTVL feature models with the DML delta modules that implement the corresponding behavioral modifications, and also specifies the order in which those delta modules should be applied. Therefore, CL specifications fulfil the role of annotated delta models (Definition 4.7). The language provides **`when`** and **`after`** keywords, which work the same way as their equivalents **`if`** and **`after`** in the LaTeX packages of Section 5.2 (page 120).

Finally, the product selection language PSL is used to give names to specific feature configurations, by which the corresponding products of an ABS product line implementation can then be generated. A PSL script contains a feature selection, a set of values for the relevant attributes, and an *initialization block*,

[1] This figure was designed by Reiner Hähnle for the joint Architecture paper [8].
[2] Actually, in Section 7.5.2, we'll pretend that it is.

which is often just a call to an appropriate *main* method, though it may also contain configuration code. After the extraction of the proper selected delta model (Definition 4.8) and its application to the core program, the initialization block is added to the result to be the first code to run.

A more detailed account of these languages may be found in the second report on ABS [52]. The content of this section is based on my work in the third report [8].

7.5.2 DMW for ABS

The following workflow description gives step-by-step instructions for development of an ABS software product line, specifying the proper code templates to use for each of the steps of Figure 7.3.

It often makes sense to put basic code common to all products into the core product directly. In the case of ABS, this means at least the following:

```
1 class Main {  Unit run() {}  } {  new Main();  }
```

We start with a `Main` class with an empty `run` method. We then create a new `Main` instance, implicitly calling the `run` method, which will later be modified by deltas. It is possible to put mandatory features into the core product. But, as mentioned, it is recommended that all features are implemented with deltas, as this makes the product line more robust to evolution, and promotes the separation of concerns.

Also, we begin with a minimal ABS product line configuration: the list of features and the list of desired products. The latter can be empty.

```
1 productline ⟨name⟩ {  features f1, f2, ..., fn;  }
```

In the following workflow description, we'll use a subset of the Editor product line example of Section 1.4, one containing only the *Ed*, *PR*, *SH* and *EC* features. Now we specify each step of the flowchart from Figure 7.3.

Step 1: Feature left unimplemented?

In this stage of the workflow, we choose the next feature to implement. Essentially we walk through the subfeature hierarchy of the feature model in a topological order, i.e., base features first, subfeatures later. If all features have been implemented, we are finished.

For the example, we would have to start with the Editor (`Ed`) feature. Any of the three features on the second level may be chosen next.

Step 2: Implement feature with new delta

Having chosen a feature f, we now write a "feature delta" d_f to implement it:

```
1 delta df {  ...  }
```

The delta may add, remove or modify any classes and methods necessary to realize the functionality of f, while preserving the functionality of all superfeatures. The developer only has to consider the local delta model: the core product and the deltas implementing superfeatures of f. The following four feature deltas implement the four individual features of the Editor product line (some details are left out for the sake of brevity):

```
delta D_Ed {  // Editor Delta
    adds class Model {  …  }
    adds class Font  {  …  }
    adds class Editor (Model m) {
        Model m_model;
        Font m_plain_font;
        {  init(m);  }
        Unit init(Model m) {
            m_model = m;
            m_plain_font = new Font();
        }
        Model model() {  return m_model;  }
        Font font(int c) {  return m_plain_font;  }
        Unit onMouseOver(int c) {  /* nothing */  }
    }
    modifies class Main {
        modifies Unit run() {  new Editor(new Model());  }
}   }
```

```
delta D_Pr {  // Printing Delta
    modifies class Editor {
        adds Printer m_printer;
        modifies Unit init(m: Model) {
            original(m);
            m_printer = new Printer();
        }
        adds Unit print() {  /* print the plain text */  }
}   }
```

```
delta D_SH {  // Syntax Highlighting Delta
    adds class SyntaxHL (Model m) {
        Model m_model;
        {  m_model = m;  }
        Color color(int c) {  …  }
    }
    modifies class Editor {
        adds SyntaxHL m_syntaxhl;
        modifies init(Model m) {
            original(m);
            m_syntaxhl = new SyntaxHL( model() );
        }
        modifies font(int c) {
            Font f = D_Ed.original(c);
            f.setColor( m_syntaxhl.color(c) );
            return f;
}   }   }
```

```
delta D_EC {  // Error Checking Delta
    adds class ErrorCh (Model m) {
        Model m_model;
        {  m_model = m;  }
        Bool errorOn(int c) {  …  }
        String errorText(int c) {  …  }
    }
```

```
8       modifies class Editor {
9           adds ErrorCh m_errorch;
10          modifies init(Model m) {
11              original(m);
12              m_errorch = new ErrorCh( model() );
13          }
14          modifies Font font(int c) {
15              Font f = D_Ed.original(c);
16              f.setUnderlined( getModel().isError(c) );
17              return f;
18          }
19          modifies onMouseOver(int c) {  ...  }
20  }   }
```

Finally, we add the following line to the ABS product line configuration:

```
delta d_f when f after d_s;
```

where d_s is the delta implementing the superfeature of f. If f has no superfeature, the **after** clause may be omitted. Our example requires the following product line configuration:

```
productline PL_Editor {
    features Ed, Pr, SH, EC;
    delta D_Ed when Ed;
    delta D_Pr when Pr after D_Ed;
    delta D_SH when SH after D_Ed;
    delta D_EC when EC after D_Ed;
}
```

Step 3: Interaction to implement?

At the feature modeling and specification level, two features f and g may be independently realizable, but require extra functionality when both are selected. This behavior is not implemented by the feature deltas, so a new delta needs to be created. In our example, this is the case for the features Printing and Syntax Highlighting. When printing, we would like the syntax highlighting colors to be used.

Step 4: Implement interaction with new delta

The new delta $d_{f,g}$ must implement the required interaction without breaking the features f and g or their superfeatures. It may change anything introduced by feature deltas d_f and d_g. When overwriting methods, it may also access the original methods using the syntax d_f**.original()** and d_g**.original()**. In our example:

```
delta D_Pr_SH {  // Pr + SH Interaction Delta
    modifies class Editor {
        modifies Unit print() {
            // print as before, but use
            // colors of D_SH.font(c)
}   }   }
```

Then we add the following to the ABS product line specification:

```
delta d_{f,g} when f && g after d_f, d_g;
```

In our example:

```
delta D_Pr_SH when Pr && SH after D_Pr, D_SH;
```

This may be generalized to interaction between more than two features.

Step 5: Conflict to resolve?

By adding new deltas, we may have introduced an implementation conflict between two deltas d_1 and d_2 that are independent, but modify the same method in a different way. In our example, this is the case for `D_SH` and `D_EC`, as they both modify the `font(int)` method in a different way, and are not ordered in the product line configuration. For each such conflict, we write a delta to resolve it.

Step 6: Resolve conflict with new delta

The conflict resolving delta $d_{1,2}$ must overwrite the methods causing the conflict, while not breaking the features implemented by d_1 or d_2, or their superfeatures. Typically, $d_{1,2}$ invokes d_1`.original()` and d_2`.original()` to combine the functionality of the conflicting deltas. In our example:

```
delta D_SH_EC {  // SH + EC Conflict Resolving Delta
  modifies class Editor {
    modifies Font font(int c) {
      Font result = D_Ed.original(c);
      result.setColor(D_SH.original(c).color());
      result.setU...lined(D_EC.original(c).u...lined());
      return result;
} } }
```

We then add the following to the ABS product line specification:

```
delta d_{1,2} when ( γ(d_1) ) && ( γ(d_2) ) after d_1, d_2;
```

where $\gamma(d)$ is the **when** clause of delta d. In our example:

```
delta D_SH_EC when (SH) && (EC) after D_SH, D_EC;
```

Step 7: Done

This means the product line implementation is finished, and it enjoys total correctness by construction.

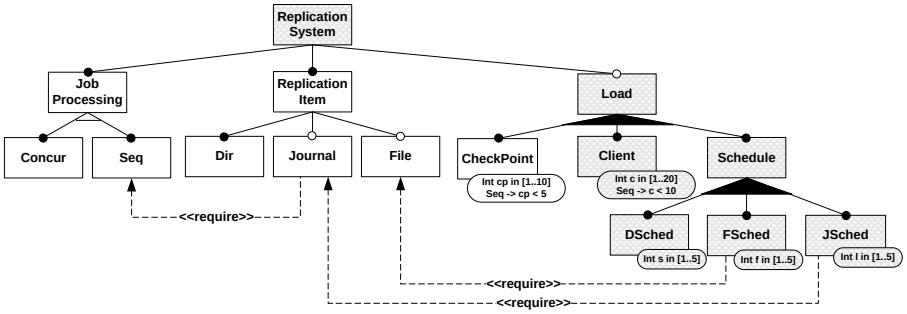


Figure 7.7: Feature diagram of the FAS replication system

7.6 The Fredhopper Access Server

This section discusses the use of the delta modeling workflow for modeling the industrial case study of the Fredhopper Access Server (FAS) product line. FAS, developed by Fredhopper B.V.[3], is a distributed service-oriented software system for Internet search and merchandising. In 2012, we used the workflow to model FAS's *replication system*, which ensures data consistency across a FAS deployment. The FAS product line is modeled using the ABS language (Section 7.5).

First, Section 7.6.1 briefly describes the FAS case study. Then we discuss the results of our modeling efforts in Section 7.6.2. We don't discuss the implementation itself in this thesis, as this would duplicate quite some information. Details can be found in the corresponding paper [7].

7.6.1 FAS Overview

The *Fredhopper Access Server (FAS)* is a component-based and service-oriented distributed software system. It provides search and merchandising services to e-Commerce companies such as large catalogue traders and travel agencies. Without going into too much detail: FAS tries to provide full throughput of data across many different clients by cleverly replicating data across a network of nodes. As part of the *FAS product line*, there are several variants of this *replication system*. We've implemented these variants in ABS using the delta modeling workflow.

We let the product line specification be that of the replication system. The feature model, also shown in Figure 7.7, is expressed in μTVL as follows:

```
1 root RS {
2   group allof {
3     JobProcessing   { ... },
4     ReplicationItem { ... },
5     opt Load {
6       group [1..3] {
7         Client     { Int c in [1..20]; Seq -> c < 10; },
8         CheckPoint { ... },
```

[3] http://www.fredhopper.com

Metrics	Java	ABS
Number of lines of code	~6400	5000
Number of classes	44	40
Number of interfaces	2	43
Number of user-defined functions	*n/a*	80
Number of user-defined data types	*n/a*	17
Number of features	*n/a*	15
Number of deltas	*n/a*	10 (7)
Number of products	*n/a*	12108 (96)

Table 7.8: Metrics on the FAS replication system code

```
9           Schedule {
10            group [1..3] {
11              DSched { Int s in [1..5];                    },
12              FSched { Int f in [1..5]; require: File;     },
13              JSched { Int l in [1..5]; require: Journal; }
14  } } } } } }
```

For reasons of space and to focus on the application of the workflow, rather than the replication system itself, we considered only the modeling of the features `RS`, `Load`, `Client`, `Schedule`, `DSched`, `FSched` and `JSched` as the representative parts of the replication system variability. These are the features that are shaded in Figure 7.7. The other features are also omitted from the μTVL code above.

7.6.2 Results

The existing FAS product line was implemented in Java, and had over 150,000 lines of code. Table 7.8 shows some metrics about the existing implementation and the ABS model of the replication system only. In particular, if parametrized deltas are used to resolve the three-way conflict between `DSched`, `FSched` and `JSched`, the number of deltas reduces from 10 to 7. If feature attributes are ignored, the number of possible feature configurations reduces from 12108 to 96.

We now discuss our experiences while applying the DMW to the implementation of the FAS case study. This case study not only raised discussion points about the pros and cons of the DMW, but also guided the development of DMW while its practical applicability was put to the test.

Correctness Following the DMW we were able to systematically implement all features in the feature model in a top-down fashion to obtain a product line implementation of the replication system. We were also able to systematically implement all necessary feature interaction and resolve implementation conflicts between deltas, since the workflow directed us to consider every situation. So we avoid accidentally forgetting to implement some functionality from a complex feature model.

Collaboration During the case study, the workflow description of the original paper [5] was used, which did not yet focus on concurrent development. We were therefore unsure how to apply DMW in a collaborative development environment. Feedback from this case study has lead to the new job-based description and the formalization found in Appendix A, in which concurrent development is an explicit benefit.

Evolution The original DMW description assumed the core product to be the empty program. In the case study we relaxed this assumption to facilitate product line evolution. In practice it is often the case that a product line will not be implemented from scratch, but will be built on legacy code, which lends itself to be incorporated as the core product. As a result, the formal DMW description no longer requires an empty core.

Overall, the conclusion was that DMW offers a useful guideline for systematically traversing the feature model and implementing its features to arrive at a software product line which is globally unambiguous and correct. And in retrospect, I can add that the feedback gathered during this experience was an invaluable tool in improving the workflow description.

7.7 Conclusion

The formalisation of ADM thus far had been *descriptive*, describing what deltas are, how they work and how they are selected. The other side of the story is *prescriptive*. This chapter describes useful patterns for the implementation of product lines. It delineates a workflow to show insight in how the delta modeling constructs may be used to good effect.

The main contribution is insight in how independent features can be implemented concurrently and in isolation. Important to this is the concept of *locality*. Any delta under development need only take into account the existing deltas that occupy subordinate positions in the delta model — the ones the new delta has control over. In effect, the product line specification is split up and localized. If each delta is developed to satisfy local constraints, the resulting product line will exhibit total correctness. In Appendix A, the workflow is formalized with an operational semantics, and this is proved as a theorem.

Finally, the chapter describes the delta modeling workflow for the *Abstract Behavioral Specification (ABS)* language, which was developed for the HATS project. Then, its application to the *Fredhopper Access Server* is discussed — the results, and the lessons we learned. This industrial scale case study helped validate and improve the workflow.

7.8 Related Work

Software product lines have existed in industry for quite a while, and many useful lessons have been learned from this experience [54, 65, 117, 122, 155]. Reuse in software as a way to improve quality and time to market has been an important theme since the 1960s [65], and software product lines in particular see their origins in the early 1980s, though not yet by that name [65].

However, it has been a predominantly empirical field; it is only recently that formal methods have started being applied, with initiatives such as the HATS project [80]. As such, though various building blocks and algebras of

software composition have been thoroughly formalized over the last decade or so (Sections 2.10, 3.8 and 4.8), We have been unable to find much previous work applying a similar level of formalism to the development process itself, though a number have recently emerged [62, 142]. The work on the delta modeling workflow [5, 7, 8, ¤2] —and therefore this thesis chapter— are, in part, an effort to fill this gap. But they are also a way to tell a part of the delta modeling story that didn't fit in the original ADM papers.

Our formalization of feature models (Definition 4.3) is actually quite close to the practice of *decision modeling*, as, for example, described by Czarnecki et al. [61]. They note that an essential ingredient of feature modeling, in contrast to decision modeling, is the subfeature hierarchy. And indeed, this information seems to express design intentions essential for the development of a properly modular product line. The addition of the subfeature relation in Section 7.2 and its use in guiding the development workflow is in recognition of this.

In a short survey paper, Krueger [117] makes a lot of points relevant to this chapter. For one, he stressed the importance of *mass customization* over application engineering: the practice of working on all product line members at once rather than on one at a time. This is done through the feature-oriented development style of DMW. It is set up so that every product that needs a certain feature gets this feature from the same delta. He also notes the importance of encapsulation among the various implementation artefacts: "[If] any feature can impact any core asset and any core asset many be impacted by any feature, [this] has all of the software engineering comprehension drawbacks as global variables in conventional programming languages." ADM addresses this problem with its partially ordered module structure and conflict resolution model (Chapter 3). The notions of locality and non-interference introduced in Section 7.3 aim to complement these, in order to reduce the combinatorial complexity of software product lines.

Finally, he makes the following point in which DMW still falls short: it is rare that product lines are developed from scratch. Usually, legacy code is already in place, and discarding it to start over is too disruptive to project schedules. Similarly, with the rising popularity of *agile development* methodologies [130], it is now considered unwise to demand a full specification in advance, prefering short cycles of development and adaptation, making that aspect of the workflow description decidedly non-agile. However, the core principles of the workflow do have great potential in that regard. This is discussed in more detail in Chapter 9.