Delta Modeling Workflow*

Michiel Helvensteijn
CWI, Amsterdam, The Netherlands
Leiden University, The Netherlands
michiel.helvensteijn@cwi.nl

ABSTRACT

In previous work we show how abstract delta modeling can be used to model product lines. The formalism assigns a functional meaning to features from a feature model and provides a novel mechanism for resolving implementation conflicts without code duplication or overspecification. But in the vast expressive space of delta modeling, it may not be clear to a developer how to create a product line from scratch. The formalism was descriptive rather than prescriptive. To that end, we propose a development workflow based directly on Abstract Delta Modeling. We show preservation of global unambiguity and completeness in the product lines resulting from this workflow. We also show that the workflow naturally supports concurrent development.

1. INTRODUCTION

A *software product line (SPL)* (or *software family*) is a set of software systems, called *software products*, with well-defined commonality and variability [5, 15]. In software product line engineering, SPLs are developed by structured reuse in order to reduce time to market and to increase product quality. *Automated product derivation* generates individual products from the product line artifacts by a mechanical process which requires no human intervention by virtue of a sufficiently expressive code base.

Different software products are distinguished from each other by which *features* they provide. Which feature combinations (or *feature configurations*) are supported in an SPL is expressed by *feature models* [11, 21]. Features can be described as designated product characteristics or increments of product functionality [1]. A product is uniquely identified by a valid feature configuration. On the feature model level, features are merely labels [6]. In order to mechanically derive a product for a particular feature configuration, the code base has to be designed with a clear link between features and code. It is important that all possible products can be generated from a trivial composition of this code.

A recently proposed method for organizing a codebase in such a way is *delta modeling* [16, 19, 17, 18], wherein the code-base is split up into *deltas*, which can modify a *core product*. Deltas would be annotated with an *application condition*, stating for which feature configurations a given delta should be applied. At the time, application conditions to annotate a delta with more than one feature were novel compared to other known approaches, in which there was a one-to-one relation between code modules and features [1]. Later work [17] introduced a partial order on deltas as a limited means of avoiding *conflicts* between deltas which do not commute.

Clarke et al. [3] abstract away from software and generalize the concepts of delta model and product line in an abstract algebraic setting, known as *Abstract Delta Modeling (ADM)*. In this work *conflict resolving deltas* were introduced to make full use of the partial order between deltas. Rather than avoiding conflicts, conflict resolving deltas can be applied after a conflicting pair of deltas to 'equalize' the two possible orderings between them, i.e. make them commute again. The absense of unresolved conflicts leads to a single unambiguous product for each feature configuration. The paper also introduced efficient conditions for checking the unambiguity of a product line as a whole.

But whereas [3] shows what is *possible* with abstract delta modeling, it has not yet been described what is *recommended*. If a team of developers started out with only a feature model, how would they actually build and organize the product line? How should the deltas be ordered and what should their application conditions and content be if they want to maximize reuse of code and isolated, concurrent development of features?

We now propose 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 paper stays at the same level of abstraction as [3] but approaches the topic from the other side. It gives a step-by-step guide to building a product line from scratch (Figure 1). Following this workflow will lead to a well-structured product line that automatically exhibits two desirable properties. The first of these properties is the global unambiguity of the product line as described in [3], i.e. for every feature configuration, there is only one product (which requires all conflicts to be resolved). The other property is product line completeness: the property that every generated product satisfies the spec-

*This research is funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`)

VaMoS '12 January 25-27, 2012 Leipzig, Germany

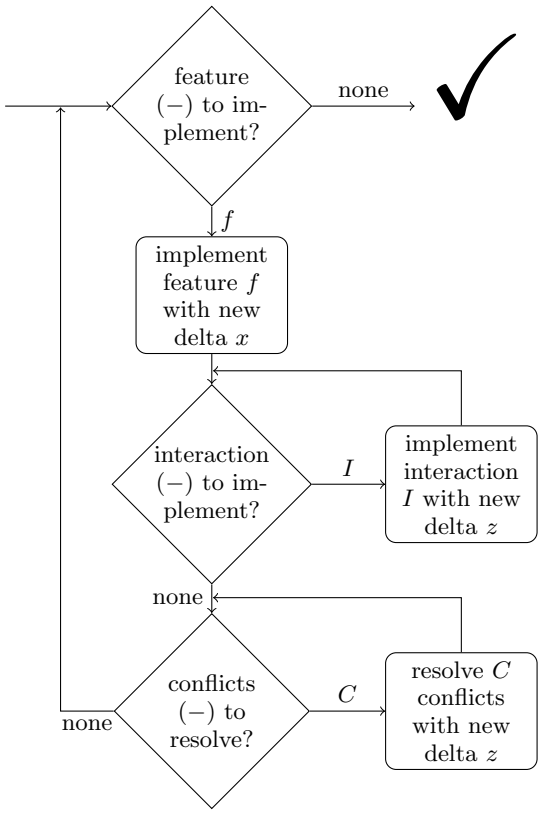

Figure 1: Overview of the development workflow

ifications of the features that it is supposed to implement. This includes specifications for desired *feature interaction*. Take, for instance, the example given in [14] of a database management system. If both 'transactions' and 'logging' are included, it is required that the transactions are also logged, functionality that goes beyond the sum of both individual features. We also show that the workflow always terminates and that it supports concurrent development, i.e. that multiple developers can work on parts of non-trivial product lines at the same time and in isolation without breaking global unambiguity or completeness.

This paper is organized as follows. Section 2 summarizes the relevant theory from Abstract Delta Modeling [3]. Section 3 extends that theory by enriching product line specifications and separating them from product line implementations. They respectively form the input and output of the development workflow we describe in Section 4, which is the main focus of this paper. In Section 5 we analyze the workflow and show the beneficial properties it exhibits. Section 6 describes a shortcoming of the workflow, and proposes the solution of *parametrized deltas*. Section 7 shows both standard and parametrized deltas in a brief but concrete example. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2. PRELIMINARIES

To make this paper self-contained, we now repeat the relevant theory from ADM. For more detailed information, we refer the reader to [3]. Readers familiar with the theory can skip this section.

2.1 Products and Deltas

Firstly, we assume a set of products $\mathcal{P}$, which includes possible core products, intermediate products and end-products.

Secondly, we have a set of deltas $\mathcal{D}$, each describing a product modification. $\mathcal{D}$ forms a monoid, together with sequential composition operator $\cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ and neutral element ϵ. Deltas $x, y \in \mathcal{D}$ are called *non-commutative* if $y \cdot x \neq x \cdot y$.

Applying a delta to a product results in another product. Delta application is a function $-(-) : \mathcal{D} \times \mathcal{P} \to \mathcal{P}$. If $d \in \mathcal{D}$ and $p \in \mathcal{P}$, then $d(p) \in \mathcal{P}$ is the product resulting from applying delta d to product p.

2.2 Delta Models

A *delta model* is a tuple $(D, \prec)$, where $D \subseteq \mathcal{D}$ is a finite set of deltas and $\prec \subseteq D \times D$ is a strict partial order on D. $x \prec y$ states that x must be applied before y, though not necessarily directly before. It represents the intuition that a delta applied later has full access to earlier deltas and more authority over modifications to the product.

A *derivation* is a delta formed by the sequential composition of all deltas from a delta model. Given a delta model $DM = (D, \prec)$, its *derivations* are defined to be

$$\left\{ x_n \cdot \ldots \cdot x_1 \mid \begin{array}{c} x_1, \ldots, x_n \text{ is a linear extension} \\ \text{of } \prec, \text{ where } \{x_1, \ldots, x_n\} = D \end{array} \right\}$$

Note that it is possible for a delta model to generate more than one distinct derivation since non-commutative deltas may be applied in different orders. We do strive for a unique derivation, however, as this corresponds to deriving a unique product. To allow for an efficient way to establish this property, we rely on notions of conflicting deltas and conflict-resolving deltas.

2.3 Conflicts

Two deltas are in *conflict*, denoted $x \lightning y$, if they are non-commutative and not ordered by $\prec$. The changes they describe are incompatible, and neither can override the other.

It is not a problem for a delta model to have conflicts, if they are later *resolved* by a third delta. Given deltas $x, y \in D$ which are in conflict, we say that a delta z resolves their conflict, denoted $(x, y) \lhd z$, iff

$$x \prec z \ \wedge \ y \prec z \ \wedge \forall d \in D^* : z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y.$$

where D^* contains all finite sequential compositions of deltas from D.

A delta model is *unambiguous* if it contains a conflict-resolving delta for every conflicting pair of deltas:

$$\forall x, y \in D : x \lightning y \Rightarrow \exists z \in D : (x, y) \lhd z.$$

If a delta model is unambiguous, it has a unique derivation.

2.4 Product Lines

We introduce a finite set of *features* $\mathcal{F}$ relevant to a specific product line. These features have no inherent semantic meaning. The set of products in a product line can be represented by a feature model $\Phi \subseteq \mathscr{P}(\mathcal{F})$, where each $F \in \Phi$ corresponds to a valid *feature configuration*.

To bridge the gap between features and product line artifacts (such as code), we introduce *application conditions* for deltas. An application condition attached to a delta determines for which feature configurations the delta has to be applied. An *application function* $\gamma : D \to \mathscr{P}(\mathscr{P}(\mathcal{F}))$ gives

the feature configurations each delta $x \in D$ is applicable to. $\gamma(x)$ is the application condition for delta x.

A *product line* is a tuple $PL = (\Phi, c, D, \prec, \gamma)$, where Φ is a feature model, $c \in \mathcal{P}$ is the core product, $(D, \prec)$ is a delta model and γ is an application function with domain D such that $\forall x \in D : \gamma(x) \subseteq \Phi$. So the product line describes all possible products, and how to generate them.

Given a feature configuration $F \in \Phi$, we can extract the corresponding products $\text{prod}(PL, F)$ out of a product line. To do this, we first extract its *selected delta model* $(D', \prec')$ where $D' = \{ d \in D \mid F \in \gamma(d) \}$ is the set of applicable deltas, and $\prec'$ is $\prec$ restricted to D'. We can then apply its derivations (as there may be more than one) to core product c. The result is a set of *generated products*.

Of course, we'd like a product line in which every selected delta model is unambiguous, since this means that every feature configuration leads to a unique product. An efficient way of checking this on the level of the product line itself is the following: For any two conflicting deltas x and y that can be applied together, there must be a conflict-resolving delta z applicable in at least the same set of feature configurations. This makes a product line *globally unambiguous*. The following is a more precise description of this property:

$$\forall x, y \in D : \\ \mathcal{V}^{x,y} = \varnothing \ \vee \ x \not\prec y \Rightarrow \exists z \in D : (\mathcal{V}^{x,y} \subseteq \gamma(z) \wedge (x, y) \lhd z)$$

where

$$\mathcal{V}^{x,y} \ = \ \gamma(x) \cap \gamma(y)$$

A globally unambiguous product line has only unambiguous selected delta models. This is one of the important properties we want to maintain with the workflow (Section 4).

3. PRODUCT LINE SPECIFICATION

In this section we enrich the specification of product lines and separate it from the implementation.

The specification is what the developers start off with when they design the product line. The workflow we describe assumes it as input. A product line is now represented as $(\Phi, c, D, \prec, \gamma)$. In this representation, the feature model Φ constitutes the specification. The rest is implementation.

3.1 Structural Feature Model

Feature models as we have represented them so far are not particularly useful for developers. When we view a feature model as the set of all possible feature configurations, we disregard the intended hierarchical relations between features. So if we start from a traditional feature model [11, 20, 21], we lose some useful information in the Φ representation. For instance, we lose the distinction between the two feature models in Figure 2, which would both have $\Phi = \{\{f, g\}, \{f, g, h\}\}$.

Since the feature diagram notation from Figure 2 is quite common in product line engineering [20], it is a much more convenient structure to base our workflow on.

So we introduce a *structural feature model* Ψ. It is a 5-tuple $(B, —\bullet, —\circ, \oplus, \blacktriangleright)$, where $B \subseteq \mathcal{F}$ is a set of mandatory base-features, $—\bullet \subset \mathcal{F} \times \mathcal{F}$ is the mandatory subfeature relation, $—\circ \subset \mathcal{F} \times \mathcal{F}$ is the optional subfeature relation, $\oplus \subset \mathcal{F} \times \mathcal{F}$ indicates those features that may not appear together in one feature configuration and $\blacktriangleright \subset \mathcal{F} \times \mathcal{F}$ indicates which features require which others. The symmetric closures of $—\bullet$, $—\circ$, $\oplus$ and $\blacktriangleright$ must be disjoint.

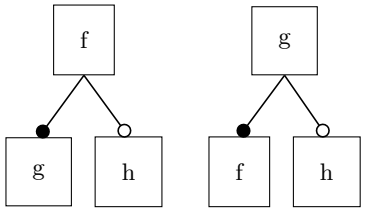

Figure 2: Two different feature diagrams representing the same feature model $\Phi = \{\{f, g\}, \{f, g, h\}\}$.

This new representation is able to distinguish the two feature models shown in Figure 2: $(\{f\}, \{(f, g)\}, \{(f, h)\}, \varnothing, \varnothing)$ and $(\{g\}, \{(g, f)\}, \{(g, h)\}, \varnothing, \varnothing)$.

It is a simple matter to recover the original feature model Φ from $\Psi = (B, —\bullet, —\circ, \oplus, \blacktriangleright)$ as follows, where $f, g \in \mathcal{F}$:

$$\Phi = \left\{ F \subseteq \mathcal{F} \;\middle|\; \begin{array}{l} \forall\ f \in B : f \in F \\ \forall\, f —\bullet g : f \in F \iff g \in F \\ \forall\, f —\circ g : f \in F \impliedby g \in F \\ \forall\ \ f \oplus g : f \notin F \vee g \notin F \\ \forall\ \ f \blacktriangleright g : f \in F \implies g \in F \end{array} \right\}$$

We will continue to use Φ under these semantics.

You may notice that $—\circ$ and $\blacktriangleright$ have nearly the same semantics. However, since they reflect different intentions, we treat them differently in the workflow.

3.2 Feature Specifications

In reality, a feature is more than a label. It represents some functionality we want a product to have. In industry, there may be informal descriptions or use-cases describing what a feature is supposed to do. In a more academic setting, there may even be formal specifications. In any case, we assume that there is some way to determine whether or not a given product actually implements a certain feature. Since we are working on an abstract level, we cannot go into detail on what this determination may entail. We cover it with the following *feature satisfaction relation*. For $p \in \mathcal{P}$ and $F \subseteq \mathcal{F}$:

$$p \vDash F$$

indicates that product p satisfies the specifications of (the combination of) the features in F. When we want to express that product p satisfies the single feature f, we write $p \vDash \{f\}$. We take as an axiom the fact that for $p \in \mathcal{P}$ and $F, G \subseteq \mathcal{F}$:

$$p \vDash F \cup G \implies p \vDash F \wedge p \vDash G$$

Recall that sometimes we want a combination of features in a product to expose extra functionality. Functionality that would not be present if any strict subset of those features was included. So the reverse of the axiom is *not* always true. If $p \vDash F \wedge p \vDash G$, the combined functionality between F and G may not yet be implemented in p. When the features between F and G should offer no combined functionality, the reverse of the axiom holds for that specific case.

The specification of a product line now consists of:

$$(\Psi, \vDash)$$

And its implementation consists of:

$$(c, D, \prec, \gamma)$$

Our workflow takes the former as input and produces the latter as output.

4. DEVELOPMENT WORKFLOW

The goal of this workflow is to start with a complete product line specification and implement from this a complete 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. We now describe the process illustrated by Figure 1 in detail.

Before we begin, we define a few useful notations. First, the meaning of *delta ideal*, the set of all deltas smaller than or equal to a given delta in the partial order, which will be useful to describe local properties during the workflow (as *local* in a delta model refers to a delta and all deltas it 'knows'). Given a delta model $(D, \prec)$ and delta $x \in D$, we define the delta ideal as follows:

$$D_{\preceq x} \stackrel{\text{def}}{=} \{\, y \in D \mid y \prec x \vee y = x \,\}$$

The product generated by applying the sole derivation of $(D_{\preceq x}, \prec')$ (assuming there is only one; where $\prec'$ is $\prec$ restricted to $D_{\preceq x}$) to core product c is denoted $\text{prod}(x)$. The non-reflexive transitive closure of $\multimap\!\bullet \cup \multimap\!\circ$ is denoted $\multimap\!\circ^*$.

In the workflow, we start with a product line specification $(\Psi, \vDash)$ as input and introduce an empty 'current product line implementation' $PL = (c, D, \prec, \gamma)$ where $D = \prec = \gamma = \varnothing$.

c is taken to be some *empty product* $\mathbf{0}$. This is not absolutely required, and it is possible to implement mandatory features in the core product. However, 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. In any case, assuming c to be the empty product simplifies the following workflow description.

We will add elements to D, $\prec$ and γ as we implement features from Ψ. For each feature f we implement, we have to implement any desired interaction in which f is involved and then resolve any conflicts introduced by implementing f and its interactions.

We also maintain a set of locks which starts empty and is updated whenever a developer chooses to start writing an implementation. It can be subdivided into feature implementation locks $L_f \subseteq \mathcal{F}$, interaction implementation locks $L_I \subseteq \mathscr{P}(\mathcal{F})$ and conflict resolution locks $L_C \subseteq \mathcal{D}$. These 'locks' are never released. They simply indicate that some developer following the workflow has made the commitment to implement a specific part of the product line, so no one else should try to implement the same part.

We now explain and formalize all steps in the workflow. The following subsection headers refer to the nodes of the Figure 1 flowchart. Note that all diamonds in the flowchart are about deciding which part of the product line to implement next (Sections 4.1, 4.3 and 4.5) and the rectangles are about the actual implementation (Sections 4.2, 4.4 and 4.6).

4.1 Feature $(-)$ to implement?

In this stage, we choose the next feature to implement. The choice is made by following the partial order introduced by $\multimap\!\circ^*$. Basically, given a feature diagram, we work on it in a topological order from top to bottom, implementing first the base features and then their subfeatures. This is because deltas implementing subfeatures often need to make assumptions about – and changes to – the implementation of the base features.

So, choose an $f \in \mathcal{F}$ such that:

- all dependencies have been implemented:
  $\forall g \multimap\!\circ^* f : \exists d \in D : \gamma(d) = \{\, G \in \Phi \mid g \in G \,\}$
- this feature has not been locked yet: $f \notin L_f$

Upon choosing f, L_f has to be updated to include it. Performing the above tests and then updating L_f is assumed to be an atomic operation. This is so different developers can develop different features concurrently and in isolation. Many features can be worked on simultaniously, so long as they are independent (Section 5.4).

If there are no more features to implement ($L_f = \mathcal{F}$), and for each previously implemented feature, the steps were properly followed, there is no more work to be done.

4.2 Implement feature f with new delta x

In this step, having chosen a feature f, we need to develop a new delta $x \in D$ to implement it. This delta has to be applied only when f is selected. So, $\gamma(x) = \{\, F \in \Phi \mid f \in F \,\}$. Its place in the partial order $\prec$ should mirror f's place in the feature diagram. So, it should be greater than the delta that implements its superfeature and incomparable to all other deltas currently present. In fact, the deltas that implement features, linked by the transitive reduction of $\prec$, will form a graph that is isomorphic with the feature diagram.

The delta has to be designed such that the following local guarantees hold:

- Delta x should implement feature f, so: $\text{prod}(x) \vDash \{f\}$
- Delta x should not break existing features, so:
  $\forall w \prec x : \text{prod}(w) \vDash \{g\} \implies \text{prod}(x) \vDash \{g\}$

In working on x, it is not necessary to consider possible conflicts, since there will be an opportunity to resolve them later in the workflow. (It will help, of course, if code is written in a modular way, which lends itself better to conflict resolution in the future. This depends on the concrete domain of the deltas and products.) Note that x may assume and use the implementations of superfeatures of f, but also those of required features g such that $f \blacktriangleright g$. This may be a motivation to implement those g before f, but it is not a requirement of the workflow, since x cannot change anything from the g implementation.

4.3 Interaction $(-)$ to implement?

At this point, we need to know if by introducing feature f, there are now sets of features that require extra work to make them interact properly. In other words, we now select the smallest feature set $I \subseteq L_f$ with $f \in I$ that is a subset of some valid feature configuration in Φ such that:

- the features in I should interact:
  $\exists p \in \mathcal{P} : p \not\vDash I \wedge \exists I_1, I_2 : I = I_1 \cup I_2 \wedge p \vDash I_1 \wedge p \vDash I_2$
- this interaction has not been locked yet: $I \notin L_I$

If there is no such I, we proceed to Section 4.5. After choosing a suitable I, we record it in L_I. As before, performing the above tests and then updating L_I is assumed to be an atomic operation, so different interactions I can be implemented concurrently and in isolation (Section 5.4).

We will implement interaction I in the next step. Note that during several iterations of implementing feature interaction, we may deal with many overlapping feature-sets.

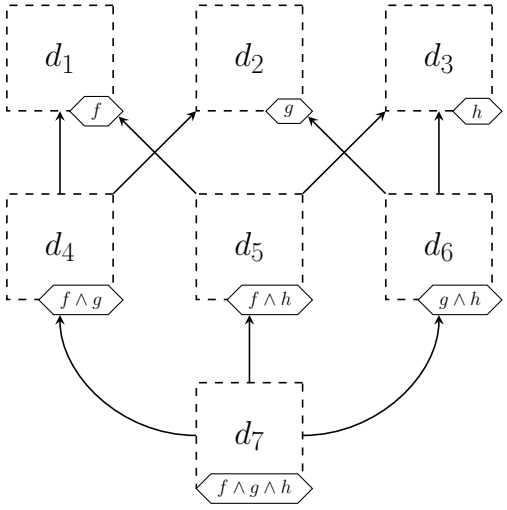

Figure 3: Example product line with a three-way interaction implementation or conflict resolution. The dashed boxes are deltas. The partial order $\prec$ is represented by the arrows and each delta $x \in D$ is decorated with a propositional logic formula representing its application condition $\gamma(x)$.

4.4 Implement interaction I with new delta z

Given a set I of features whose interaction we need to implement, we develop a new delta $z \in D$ to do just that. z has to be applied exactly when the features from I are selected. So, $\gamma(z) = \{ F \in \Phi \mid I \subseteq F \}$. It should be greater in the partial order $\prec$ than the deltas that implement its individual features, as well as the deltas that implement interactions of strict I subsets.

The delta z has to be designed such that the following local guarantees hold:

- z should implement the interactions between the features in I, so: $\text{prod}(z) \models I$
- Delta z should not break existing features, so: $\forall w \prec z : \text{prod}(w) \models G \implies \text{prod}(z) \models G$

After applying this step four times to implement the interaction between hypothetical features f, g and h, the resulting product line may look something like Figure 3.

4.5 Conflicts $(-)$ to resolve?

After implementing our feature and any desired interaction related to it, we now look for any conflicts $y_1 \lightning y_2$ we might have introduced in this iteration. We have to consider conflicts involving delta x, all deltas z we used for implementing desired interaction and all deltas z we used for resolving earlier conflicts this iteration.

Formally, a conflict occurs between two deltas. However, when a there is a set of deltas with many (related) conflicts, we will also want to introduce conflict-resolving deltas for larger sets with a non-empty *joint application condition*, so we eventually cover all combinations.

So we now find the sets $C \subseteq D$ with the widest (largest) joint application condition $(\bigcap_{d \in C} \gamma(d)) \subseteq \Phi$ such that:

- all deltas in C are in unresolved conflict: $|C| \geq 2 \wedge \forall y_1 \in C : \exists y_2 \in C : y_1 = y_2 \vee y_1 \lightning y_2 \wedge \neg \exists z \in D : (y_1, y_2) \lhd z$
- this conflict set has not been locked yet: $C \notin L_C$

From those, we select the C with the largest number of deltas. We choose the widest joined application condition so we eventually handle all relevant cases. We then choose the largest set, because this is required for termination of the workflow (Section 5.1). Having made the choice, we record C in L_C. We again assume that performing the above tests and then updating L_C is an atomic operation, so conflict-resolving deltas for different conflict sets C can be implemented concurrently and in isolation (Section 5.4).

If no conflicts are left to resolve, we can start a new iteration of the main loop.

4.6 Resolve C conflicts with new delta z

Given a set of deltas C with conflicts we need to resolve, we develop a new delta $z \in D$ to do it. Its application condition reflects the joint application condition of C. So, $\gamma(z) = \bigcap_{d \in C} \gamma(d)$. It should be greater in the partial order $\prec$ than all deltas in C.

The delta z has to be designed such that the following local guarantees hold:

- z resolves all conflicts: $\forall y_1, y_2 \in C : (y_1, y_2) \lhd z$
- z does not break existing features, so: $\forall w \prec z : \text{prod}(w) \models G \implies \text{prod}(z) \models G$

After applying this step four times to resolve the conflicts between three hypothetical deltas d_1, d_2 and d_3, the resulting product line may look something like Figure 3. In this way, interaction implementing deltas and conflict resolving deltas are quite similar. For a concrete example, see Section 7.

5. ANALYSIS

We now analyze the workflow above. It is inherent in the field of engineering that we cannot give small, algorithmic steps for this creative process, and it requires the developers to 'design delta x such that ...'. But we gave formal constraints so we can still prove several important properties about the process. The developers have to satisfy only local constraints in order to benefit from global properties.

5.1 Termination

First, we show that the workflow eventually terminates. There are only three loops that may be sources of divergence. They are visible in the flow-graph of Figure 1.

The first inner loop, which implements all necessary feature interaction, will terminate when there are no more selectable feature-sets which should interact, but for which the interaction has not been locked. Since there are only a finite number of feature-sets, and after choosing such an interaction set, it is immediately locked, this loop will terminate.

The second inner loop, which resolves all remaining conflicts, will terminate when there are no more conflict sets left unlocked. Every conflict set that is detected is locked right away and resolved in the next step of the workflow. But doing so adds a new delta z that may itself be in conflict again. However, each new z will have increasingly restrictive application conditions $\lambda(z) = \bigcap_{d \in C} \lambda(d)$, and be greater than the

conflicting deltas in the partial order. This being the case, and because the set C is always chosen as large as possible given a widest joined application condition, there will eventually be a z that introduces no new conflicts. In the most extreme example, we could end up with a delta z greater than all other deltas in the partial order so there can not be a delta left in conflict with it.

The outer loop visits all features in the feature model once. Since the feature model is finite, this loop must terminate as well.

5.2 Global Unambiguity

Next, we are interested in the following property over any generated product line *PL*:

$$\forall F \in \Phi : |\operatorname{prod}(PL, F)| = 1$$

It specifies that for every feature configuration, we can generate a unique product. As proved in [3], this property is guaranteed if *PL* is globally unambiguous. We now show by induction that product lines generated using the workflow of Section 4 are always globally unambiguous.

The workflow starts with the empty product line implementation $(\mathbf{0}, \varnothing, \varnothing, \varnothing)$. This product line is trivially unambiguous, since D is empty.

Then we assume that we have a globally unambiguous product line implementation $(c, D, \prec, \gamma)$ at the start of a new iteration. A new delta is introduced to implement the new feature, then a (possibly empty) set of deltas implementing desired interaction, then a (possibly empty) set resolving conflicts. The second inner loop of the workflow terminates only when there are no longer any unresolved conflicts involving any of those deltas, since a new delta z is always created to resolve any detected conflict set. Section 5.1 shows that this loop always terminates. By the induction hypothesis, all conflicts not involving those deltas were already resolved before this iteration. So each iteration of the main loop results again in a globally unambiguous product line. And then so does the application of the whole workflow.

5.3 Complete Product Line

A product line is *complete* if every valid feature configuration has a corresponding product that actually implements the required features:

$$\forall F \in \Phi : \exists p \in \operatorname{prod}(PL, F) : p \vDash F$$

We want to show that this property holds for any product line generated with the workflow of Section 4.

5.3.1 Non-interference

Let us first consider a simple setting with the following two properties:

- No two features are supposed to interact: $\forall p \in \mathcal{P} : \forall F, G \subseteq \mathcal{F} : (p \vDash F \wedge p \vDash G) \implies (g \vDash F \cup G)$
- All deltas implementing these features will be commutative: $\forall x, y \in D : x \cdot y = y \cdot x$

Here, we already run into a problem. The workflow step of Section 4.2 enforces the local constraint that the new delta does not break any superfeatures. It says nothing, however, about not breaking any *other* features. If we want to achieve product line completeness by construction, developers would need to give more than just local guarantees.

This motivates the restriction of *non-interference* on the product set $\mathcal{P}$, monoid $(\mathcal{D}, \cdot, \epsilon)$ and feature satisfaction relation $\vDash$. The restriction holds if two commutative deltas cannot break each others features:

$$\forall p \in \mathcal{P} : \forall x, y, z \in \mathcal{D} : \forall F \subseteq \mathcal{F} : \\ (x \cdot y = y \cdot x) \wedge ((z \cdot x)(p) \vDash F) \implies ((z \cdot y \cdot x)(p) \vDash F)$$

Systems that may break this restriction may include deltas that can add advice in aspect oriented languages or features whose specifications are mutually exclusive. We assume non-interferent systems from here on.

5.3.2 Completeness for Non-interferent Systems

We can now show that in our simple setting described above, any resulting product line is complete. Each delta is developed such that it implements its own feature and does not break superfeatures. By non-interference, they also do not break features from the other deltas, as all deltas commute. No feature interaction is required, so this is sufficient.

Now we look at a setting without the first of the above properties. That is, features may now require extra implementation to interact. In that case, the main loop may implement all individual features properly, but completeness requires that the product implement all interaction as well. If there are such interactions to be implemented, they will be detected and implemented by the first inner loop of the workflow. These implementations are also sure not to break any superfeatures. Without conflicts, this is sufficient.

Lastly we look at a setting in which conflicts may also appear. If two deltas are not ordered and do not commute, they may break each others features, even in non-interferent systems. However, each conflict is resolved by a conflict resolving delta that makes the two deltas commute, so non-interference applies again. And the conflict resolving deltas are sure not to break any superfeatures.

5.4 Concurrent Development

Finally we explain why this workflow is suitable for concurrent development. Mainly it is made possible by the workflow's directive to develop each feature independently of the features unrelated to it, without having to consider possible conflicts between the implementations. The developers have the opportunity to work on features uninterrupted, and then to collaboratively develop a conflict resolving delta to reconcile any conflicting implementations.

At each choice in the workflow (Sections 4.1, 4.3 and 4.5), the process can 'fork' with multiple people implementing different parts of the product line at the same time. The locks in L_f, L_I and L_C prevent the same work from being done more than once. Deadlocks cannot occur, since no one has to wait for locks to be released. The locks are meant to notify others that a specific piece of work has already started. Even if necessary implementations are skipped because they are already locked, all work is sure to get done because the holder of the lock will still be directed to implement all dependent functionality.

Note that the entities walking the flowgraph concurrently need not be specific people or teams. A 'thread' of work may be handed over to different developers at any time. The reason for even speaking in terms of the flowgraph is to make sure that no required implementation is forgotten, e.g. that no conflict is left unresolved or interaction left unimplemented.

For example: take two teams, both in their own 'thread' in the flowgraph, both implementing a different feature. If these features should interact, both teams are directed by the workflow to implement that interaction if the lock is not yet taken. Only one of the threads can take the lock. But it makes sense for both teams to work together on this implementation, and it does not matter which thread is used.

6. PARAMETRIZED DELTAS

In the inner loops of the workflow, interaction implementation deltas and conflict resolving deltas are created, sometimes to be applied layer upon layer, with increasingly narrow application conditions. This can result in delta structures such as the one in Figure 3, with one delta for every possible combination of features. In the worst case, that is an exponential number: $2^n - 1$ deltas for n features.

It is theoretically possible in the specification of a product line that each of those cases requires a distinct solution. In that case, this complexity is inherent in the problem and structures like the one in Figure 3 are precisely what we need for full control. However, in many practical scenarios, the implementations reconciling these different combinations follow a very similar pattern which may be much more conveniently expressed in the underlying (programming) language of the deltas $\mathcal{D}$ and products $\mathcal{P}$.

To accomodate those scenarios, we propose *parametrized deltas*. We now slightly redefine product line implementations. They are represented by triple $(c, D^\Phi, \prec)$. c is the core product as before. $D^\Phi \subset \Phi \rightharpoonup \mathcal{D}$ is a set of partial functions (called parametrized deltas), each mapping the feature configurations it is applicable for to specific deltas. $\prec \subseteq D^\Phi \times D^\Phi$ is a partial order on the parametrized deltas, which is intuitively the same as before.

The application function $\gamma : D^\Phi \to \mathscr{P}(\mathscr{P}(\mathcal{F}))$ is now redundant (since $\gamma(d) = \mathrm{dom}(d)$ for all $d \in D^\Phi$) so we leave it out of the product line implementation tuple. Though we will continue to use the γ notation for the sake of consistency.

Given a specific product line implementation $(c, D^\Phi, \prec)$, the selected delta model for feature configuration $F \in \Phi$ is $(D', \prec')$ where $D' = \{ d(F) \mid d \in D^\Phi \wedge F \in \gamma(d) \}$ and $d_1(F) \prec' d_2(F)$ whenever $d_1 \prec d_2$ with $F \in \gamma(d_1) \cap \gamma(d_2)$.

If a 'non-parametrized' delta x is required, it can still be simulated by a parametrized delta d which maps every feature configuration to that delta: $\forall F \in \gamma(d) : d(F) = x$. So parametrized deltas are strictly more expressive.

We now alter the four relevant steps of the workflow to use parametrized deltas instead of regular deltas. But we first need the following convenient notation: For $F \in \Phi$, take F's selected delta model $(D', \prec')$ of product line $(c, D^\Phi_{\preceq x}, \prec)$. Its sole derivation (assuming there is only one) applied to core product c is denoted $\mathrm{prod}(x, F)$.

6.1 Interaction $(-)$ to implement?

Let us start with the selection procedure in Section 4.3. Whereas we first had to find a single smallest set I of features whose interaction we wanted to implement, we can now find a set of sets $I^\star \subseteq \mathscr{P}(\mathcal{F})$, choosing each element by the same criteria as in Section 4.3, plus this one:

- For all I in $I^\star$, and all $I' \subset I$: if the features in I' should interact (as in Section 4.3), then $I' \in I^\star \cup L_I$.

For the sake of concurrency, we need to record all $I \in I^\star$ in L_I in one atomic operation.

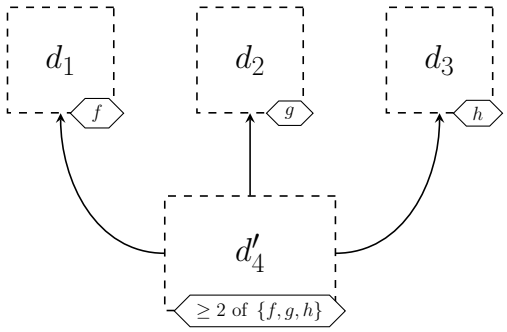

Figure 4: Example product line with a three-way interaction implementation or conflict resolution implemented with a parametrized delta d'_4.

6.2 Implement interaction I with new delta z

In the Section 4.4 stage of the workflow, we can now develop a single parametrized delta z to implement all interaction cases in $I^\star$. z's application condition should be wide enough to encompass all different cases we want to handle, so $\gamma(z) = \bigcup_{I \in I^\star} \{ F \in \Phi \mid I \subseteq F \}$. It should be greater in the partial order $\prec$ than any existing parametrized deltas that implement its individual features, as well as any existing parametrized deltas that implement interactions of strict subsets of some $I \in I^\star$.

The parametrized delta z has to be designed such that the following local guarantees hold:

- For all $F \in \gamma(z)$, if there is an interaction set $I \in I^\star$ such that $I \subseteq F$, then z should implement the interactions between the features in I, so: $\mathrm{prod}(z, F) \vDash I$
- z should not break existing features, so:
  $\forall F \in \gamma(z) : \forall w \prec z :$
  $\mathrm{prod}(w, F) \vDash G \Longrightarrow \mathrm{prod}(z, F) \vDash G$

Compared to the situation of Figure 3, applying this technique to hypothetical interacting features f, g and h can result in the product line from Figure 4, reducing the number of required deltas from an exponential to a constant number. In a concrete settings we assume that the underlying programming language has access to the chosen feature configuration. For instance, by exposing the feature names as boolean constants.

6.3 Conflict $(-)$ to resolve?

Instead of selecting a conflict set C with the widest joined application condition, we can now immediately find the largest set $C \subseteq D$ by the same criteria as in Section 4.5.

For the sake of concurrency, we need to record all subsets $C' \subseteq C$ with $|C| \geq 2$ in L_C in one atomic operation.

6.4 Resolve C conflicts with new delta z

Developing parametrized delta z to resolve the conflicts for all relevant cases is quite similar to implementing a paramatrized interaction delta (Section 4.4). Its application condition should be wide enough to encompass all conflict subsets of size 2 or larger, so:

$$\gamma(z) = \{ F \mid \exists y_1, y_2 \in C : y_1 \neq y_2 \wedge F \in \gamma(y_1) \cap \gamma(y_2) \}.$$

It should be greater in $\prec$ than all deltas in C. It should be designed such that the following local guarantees hold:

- z resolves all conflicts for every feature configuration it is applicable for: $\forall y_1, y_2 \in C : \forall F \in \gamma(y_1) \cap \gamma(y_2) \cap \gamma(z) : (y_1(F), y_2(F)) \lhd z(F)$
- Delta z should not break existing features, so: $\forall F \in \gamma(z) : \forall w \prec z :$ $\text{prod}(w, F) \vDash G \Longrightarrow \text{prod}(z, F) \vDash G$

In the same way as for Section 6.2, applying this technique to hypothetical conflicting deltas d_1, d_2 and d_3 can result in the product line from Figure 4. Continue to Section 7 for a concrete example.

7. EXAMPLE

In this section, we briefly illustrate the conflict resolution process with a concrete example, to give the reader an idea of how it works in practice. The example is part of the Editor product line [3]. It implements a code-editor widget with some optional features:

Syntax Highlighting (SH) can change the text-color.

Error Checking (EC) can underline certain errors.

Keyword Marking (KM) can give keywords a bold font.

Let's assume that the editor is implemented in an object oriented programming language and that the only way for deltas to alter the font is to overwrite the general '`font()`' method, which returns the font for a specific part of the text. So all three feature-implementing deltas will be in conflict with each other. Figure 3 illustrates this situation exactly for $f = SH$, $g = EC$ and $h = KM$.

Assume that we already have deltas d_1, d_2, d_3 which implement these features by modifying '`font()`', and in any conflict resolving delta, we can use the syntax d_i`.original()` to call '`font()`' as implemented by d_i.

Following the workflow, the conflicts may be resolved as in Figure 3 with the following four deltas $(d_4, \ldots, d_7)$. Delta d_4:

```
Font f = new Font();
f.color      = d1.original().color;
f.underlined = d2.original().underlined;
return f;
```

Delta d_5:

```
Font f = new Font();
f.color = d1.original().color;
f.bold  = d3.original().bold;
return f;
```

Delta d_6:

```
Font f = new Font();
f.underlined = d2.original().underlined;
f.bold       = d3.original().bold;
return f;
```

Delta d_7:

```
Font f = new Font();
f.color      = d1.original().color;
f.underlined = d2.original().underlined;
f.bold       = d3.original().bold;
return f;
```

Deltas d_4, d_5 and d_6 handle the cases in which two features are selected. Delta d_7 handles the case in which all three features are selected. It seems clear that we can do better. There is no duplication of behavior, as such, since we call the original methods of d_1, d_2 and d_3. But there is duplication of code; every line is used at least three times.

Parametrized deltas can be the solution in this case. Assume that the names of our features are available in the code as constant boolean values. These are the parameters of the following parametrized delta $d4'$, which can resolve all cases, placed as in Figure 4:

```
Font f = new Font();
if (SH) f.color      = d1.original().color;
if (EC) f.underlined = d2.original().underlined;
if (KM) f.bold       = d3.original().bold;
return f;
```

Now the question can arise: when exactly should we use parametrized deltas? And when we do, how many conflicts do we want to resolve at the same time?

In general, this is a judgement call. We would follow the original workflow until we appeared to be resolving a lot of conflicts (implementing a lot of interactions) in a similar way, layer upon layer. This is usually an indication that parametrized deltas can reduce the complexity of the delta model. Often, this can be anticipated, and parametrized deltas can be used straight away.

How far should we take this strategy of expressing variability in-code? Theoretically, a whole product line could be encoded as a single parameterized delta, in which the code is annotated with feature conditions to handle all cases. However, such an annotative approach would not benefit from the flexibility and structure that delta modeling can provide. With parametrized deltas, we offer a mix of annotative and compositional approaches [13]. We recommend using parametrized deltas only where they significantly reduce the amount of code or effort, as it is a tradeoff.

8. RELATED WORK

The Delta Modeling Workflow (DMW) is an extension of the Abstract Delta Modeling (ADM) formalism by Clarke et al. [3]. This paper works in the same abstract algebraic setting, and extends the earlier work.

ADM is not the first attempt to model variability of product lines [1, 2, 7, 12, 19], but it is the first that inherently lends itself to a systematic workflow for developing product lines from scratch that support automated generation of all member products with minimal code duplication and explicit handling of interaction and conflicts.

During the early phases of writing this paper, we started a collaborative effort in applying DMW to an industrial case study. The Fredhopper Access Server (FAS) is a software product line modeled using the Abstract Behavioral Specification (ABS) language [4, 10] designed within the HATS project [8]. A paper was written documenting this effort [9] and evaluating DMW for practical purposes. It applies DMW to a concrete domain, in contrast to this paper, which remains abstract. Feedback from modeling the FAS case study has improved later versions of this paper.

9. CONCLUSIONS AND FUTURE WORK

We have given step-by-step instructions on building a product line from scratch, yet remained in an abstract setting. The instructions in this paper may be instantiated to concrete programming or modeling languages. It can then be used to develop software families with a minimum of code duplication, ensuring unique software product generation and offering concrete guidance towards full satisfaction of feature specifications.

For future work, we plan to give full formal proofs of the claims in Section 5. As mentioned in Section 8, DMW has now been evaluated in an industrial setting. Even so, the feature satisfaction relation warrants further study. We plan to describe feature specifications as use-cases in software models, and to explore their impact on the workflow. Another interesting avenue of research would be a measure of code duplication applied to product lines produced by DMW, compared to those produced by other methods. Lastly, this paper assumes that the product line specification remains fixed during execution of the workflow. This restriction may be relaxed in future work.

Acknowledgements

I would like to thank my colleagues Radu Muschevici and Peter Wong, with whom I applied this workflow to an industrial case study, improving the formalism in the process. I would also like to thank the anonymous referees, whose suggestions helped make this a better paper.

10. REFERENCES

[1] D.S. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.

[2] L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *Proc. of Object-Oriented Programming Languages and Systems (OOPS), Track of ACM SAC*, 2010.

[3] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[4] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. volume 6957, 2011.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[6] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Conf. on Generative Programming and Component Engineering(GPCE)*, 2005.

[7] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2), 2006.

[8] R. Hähnle. HATS: Highly Adaptable and Trustworthy Software Using Formal Methods. In *ISoLA (2)*, pages 3–8, 2010.

[9] M. Helvensteijn, R. Muschevici, and P.Y.H. Wong. Delta Modeling in Practice, a Fredhopper Case Study. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.

[10] E. Broch Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2011.

[11] K.C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute, 1990.

[12] C. Kästner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *ASE*, pages 258–267. IEEE, 2008.

[13] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320, 2008.

[14] C. Kästner, S. Apel, S.S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proc. Int'l Software Product Line Conference (SPLC). SEI*, 2009.

[15] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.

[16] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

[17] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010.

[18] I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *Workshop on Feature Oriented Software Development 2010*.

[19] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009)*, 2009.

[20] P. Schobbens, P. Heymans, and J. Trigaux. Feature diagrams: A survey and a formal semantics. *Requirements Engineering, IEEE International Conference on*, 0:139–148, 2006.

[21] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.