Abstract Delta Modeling *

Dave Clarke
IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium
dave.clarke@cs.kuleuven.be

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

Ina Schaefer †
Chalmers University of Technology, Gothenburg, Sweden
schaefer@chalmers.se

Abstract

Delta modeling is an approach to facilitate automated product derivation for software product lines. It is based on a set of deltas specifying modifications that are incrementally applied to a core product. The applicability of deltas depends on feature-dependent conditions. This paper presents *abstract delta modeling*, which explores delta modeling from an abstract, algebraic perspective. Compared to previous work, we take a more flexible approach with respect to conflicts between modifications and introduce the notion of conflict-resolving deltas. We present conditions on the structure of deltas to ensure unambiguous product generation.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Design, Languages, Theory.

Keywords Software Product Lines; Automated Product Derivation; Delta Modeling; Conflict Resolution.

1. Introduction

A *software product line (SPL)* is a set of software systems, called *products*, with well-defined commonalities and variabilities [12, 35]. Software product line engineering aims at developing this set of systems by reuse in order to reduce time to market and to increase product quality. Automated product derivation (or software mass customization [25]) is an approach to generating individual products without the need for manual intervention during application engineering, which can be tedious and error-prone [14].

Currently, product line variability is mostly represented by *feature models* [20, 43]. *Features* are designated product characteristics or increments of product functionality [7]. A product is uniquely identified by a valid *feature configuration*, i.e., a legal combination of features from the feature model. On the feature model level, features are merely labels [13]. In order to be able to automatically derive a product for a particular feature configuration, a correspondence between the features on the feature modeling level and the reusable product line artifacts has to be introduced. Additionally, the product line artifacts have to be organized in such a way that they can be assembled automatically to generate a uniquely determined product.

Feature-oriented programming [7] is a prominent approach for implementing SPLs by composition of feature modules that directly correspond to product features. *Delta modeling* [38–40] extends feature-oriented programming. In the delta modeling approach, a product line is represented by a core product and a set of product deltas. Product deltas specify modifications to the core product to generate further products of the product line. Each delta has an *application condition* specifying for which feature configurations the modifications have to be carried out, connecting features on the feature modeling level with product line artifacts. A product for a feature configuration can be obtained by applying those product deltas with a valid application condition to the core product.

In this paper, we generalize the existing delta modeling approaches [38–40] and present an abstract, algebraic formalization of the delta modeling concepts. The presented abstract delta modeling approach goes beyond existing work with its novel treatment of conflicts between deltas. A conflict between deltas arises if their specified modifications do not commute. This means that applying them in different orders results in different (composed) modifications. In previous work, deltas were either incomparable [38, 40], which required writing additional deltas for every conflicting combination, or they had to be ordered in a very restrictive way [39] to avoid conflicts explicitly. As a main contribution of this paper, we introduce the notion of conflict-resolving deltas that relax these restrictions and make delta modeling of product lines more flexible. A conflict-resolving delta, which is applied after two conflicting deltas, eliminates differences between the (composed) modifications. If for every pair of conflicting deltas a conflict-resolving delta exists, all possible sequences of deltas produce the same modification and generate a uniquely defined product. In order to ensure this result for every valid feature configuration, we provide efficient conditions requiring only the inspection of the product line directly, without having to generate and check all products.

The concepts of abstract delta modeling can be instantiated for different kinds of development artifacts, such as documentation, models or code. We demonstrate the feasibility of the approach by presenting an instantiation of abstract delta modeling for object-oriented implementations of software product lines and extend this with method wrapping. Furthermore, we show that existing formalizations of compositional product line implementations can be seen as instantiations of abstract delta modeling.

The abstract delta modeling formalism consists of a number of ingredients, which are depicted in Figure 1, along with the operations between them. At the top is (a model of) the software product line that is defined by the feature model, the core product, the prod-

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

† This author is supported by the German Science Foundation (DFG).

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.

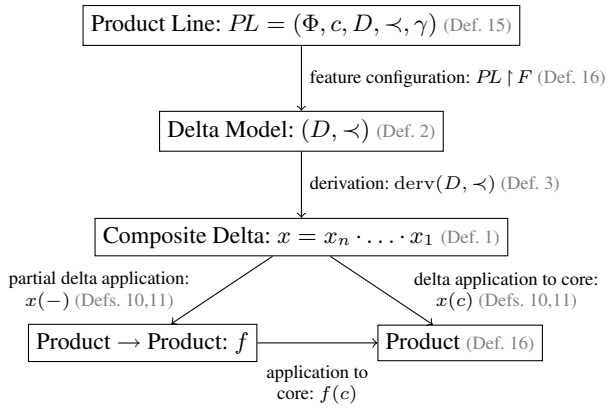

Figure 1. Relationship between artefacts. Relevant definition numbers are indicated in parentheses.

uct deltas specifying modifications of the core product, a partial ordering on the deltas restricting their application and the application conditions for the deltas. By specifying a feature configuration, one can produce a delta model, consisting of an ordered collection of modifications necessary to generate the respective product. The process of derivation applied to a delta model puts the specified modifications in a linear order that is compatible with the partial ordering in order to obtain a valid, ideally unique, composed modification. The partial delta application operation applied to a delta returns a function that takes a product and produces a new product. This function can then be applied to the core product to produce the target product for a given feature configuration.

This paper is organized as follows. Section 2 presents delta models and criteria for their unambiguity. Sections 3 and 4 reintroduce products and incorporate deltas into product lines, transferring the unambiguity properties. Section 5 presents a concrete class of deltas and illustrates our approach using an example. Section 6 compares our approach with existing algebraic approaches from the literature. Finally, Sections 7 and 8 present related work and conclusions. Additional material including full proofs appears in a companion technical report [11].

2. Abstract Delta Modeling

This section presents our approach of abstract delta modeling. We introduce product modifications and their composition as a monoid, called a *deltoid*. Delta models are built on top of this monoid as partially ordered collections of modifications, where the ordering constrains the possible ways such modifications can be applied. It is important that a delta model defines unambiguous modifications as these are used later to obtain distinct products. Thus, we define the notions of conflict, leading to ambiguous modifications, and conflict-resolving deltas eliminating these. We develop conditions to ensure that a delta model is unambiguous.

2.1 Delta Models

In existing compositional approaches for implementing software product lines, such as feature-oriented programming [7] or delta-oriented programming [39], a member product of an SPL is obtained by the application of a number of modifications (deltas) $x_1, \ldots, x_n$ to a core product c, as follows:

$$x_n(\cdots x_1(c)\cdots).$$

In feature-oriented programming the core product is determined by one or more base modules. The modifications are feature modules extending and refining the core product. In delta-oriented programming the core product can be any valid product of the product line.

As both approaches treat the core product as a constant element for all products in the product line, it is useful to focus on the modifications. In this setting, the above modification would be equivalently written as follows, where $\cdot$ refers to the composition of modifications:

$$(x_n \cdot \ldots \cdot x_1)(c).$$

Thus, we will focus exclusively on sequences of modifications such as $x_n \cdot \ldots \cdot x_1$.

It may still be possible to reason about the core product if we choose to see it as a modification x_c applied to the empty product $\mathbf{0}$, i.e. $c = x_c(\mathbf{0})$. Thus:

$$(x_n \cdot \ldots \cdot x_1)(c) = (x_n \cdot \ldots \cdot x_1 \cdot x_c)(\mathbf{0}),$$

so nothing is lost by restricting our attention to modifications.

In abstract delta modeling the main object of interest is a *deltoid*. A deltoid consists of a set of modifications, called *deltas*, along with the operation for composing them sequentially. A deltoid can contain different kinds of deltas for different kinds of development artefacts (e.g., documentation, models or code) and for different levels of abstraction (e.g., when working on component level or working on class level). The concrete nature of the modifications specified in the deltas depends on the capabilities of the underlying modeling or programming languages. Deltas may be functions performing the changes directly or some other structure representing those changes. We abstract away from the internal details of modifications, since many different instantiations are possible.

We define the notions of deltoids and deltas as follows.

Definition 1 (Deltoid)**.** *A* deltoid *is a monoid $(\mathcal{D}, \cdot, \epsilon)$, where $\mathcal{D}$ is a set of modifications (referred to as* deltas*), and the operation $\cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ corresponds to their sequential composition. $y \cdot x$ denotes the modification applying first x and then y. The neutral element ϵ of the monoid corresponds to modifying nothing.*

The operation $\cdot$ is associative but not inherently commutative, as the ordering between two deltas may be significant. We call two deltas $x, y \in \mathcal{D}$ *incompatible* if $y \cdot x \neq x \cdot y$.

A delta model describes the collection of deltas required to build a specific product, along with a strict partial ordering on those deltas restricting the order in which they can be applied. Recall that a *strict partial order* is irreflexive, asymmetric and transitive.

Definition 2 (Delta Model)**.** *A* delta model *is a tuple $(D, \prec)$, where $D \subseteq \mathcal{D}$ is a finite set of deltas and $\prec \subseteq D \times D$ is a strict partial order on D. $x \prec y$ states that x should be applied before, though not necessarily directly before, y.*

The partial order between deltas represents the intuition that a subsequent delta has full knowledge of (and access to) earlier deltas and more authority over modifications to the product. This is realized by applying the deltas in a linear extension of the partial order, as shown in the following definition. A *derivation* is a sequential composition of all deltas in a model to generate a desired product.

Definition 3 (Derivation)**.** *Given a delta model $P = (D, \prec)$, its* derivations *are defined to be*

$$\mathrm{derv}(P) \stackrel{\text{def}}{=} \left\{ x_n \cdot \ldots \cdot x_1 \mid \begin{array}{l} x_1, \ldots, x_n \text{ is a linear extension} \\ \text{of } \prec \text{ where } \{x_1, \ldots, x_n\} = D \end{array} \right\}.$$

Note that $\mathrm{derv}(P)$ may potentially generate more than one distinct derivation as incompatible deltas may be applied in different orderings. However, it is desirable that all possible derivations of a delta model have the same effect, as this corresponds to deriving a unique product. This motivates the following definition.

Definition 4 (Unique Derivation)**.** *A delta model $P = (D, \prec)$ is said to have a* unique derivation *iff $x_n \cdot \ldots \cdot x_1 = x'_n \cdot \ldots \cdot x'_1$ for all pairs of linear extensions $(x_1, \ldots, x_n)$ and $(x'_1, \ldots, x'_n)$ of $\prec$. Or, equivalently, iff $|\operatorname{derv}(P)| = 1$.*

2.2 Unambiguity of Delta Models

The property that a delta model has a unique derivation can be checked by brute force. This means generating all possible derivations (in the worst case, $n!$ for n deltas), and then checking that they all correspond. In order to allow for a more efficient way to establish this property, we introduce *unambiguous delta models* which rely on the notion of conflicting deltas and conflict resolving deltas.

Two deltas are in conflict if they are incompatible and no ordering is placed upon them. Intuitively, the two conflicting deltas are independently modifying the same part of the product in different ways, meaning that multiple distinct derivations may be possible.

Definition 5 (Conflict)**.** *Given a delta model $P = (D, \prec)$, $x, y \in D$ are said to be* in conflict *iff the following condition holds:*

$$x \lightning y \stackrel{\text{def}}{=} y \cdot x \neq x \cdot y \wedge x \not\prec y \wedge y \not\prec x.$$

One way to ensure a unique derivation is to avoid conflicts by always enforcing an ordering between incompatible deltas [39]. However, features with conflicting implementations are often independent in concept. Kästner et al. [24] call the issue of how to model such situations the *optional feature problem*. Imposing an ordering on the deltas of conceptually orthogonal features is often inappropriate. Some (unrelated) functionality may be inadvertently and silently overwritten. Furthermore, sometimes neither of the original choices in functionality is exactly what we need, and some combination of them should be used.

The alternative is to allow conflicts but to provide additional, subsequently applied, deltas to resolve them.

Notation 1. *If $D \subseteq \mathcal{D}$, then D^* and D^+ denote the sequences and non-empty sequences of compositions of deltas from D.*

Definition 6 (Conflict-Resolving Delta)**.** *Given a delta model $P = (D, \prec)$ and $x, y \in D$ which are in conflict, we say that a delta $z \in D$* resolves *their conflict iff the following property holds:*

$$(x, y) \lhd z \stackrel{\text{def}}{=} \begin{array}{l} x \prec z \wedge y \prec z \wedge \\ \quad \forall d \in D^* : z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y. \end{array}$$

An unambiguous delta model is now a delta model containing a conflict-resolving delta for every conflicting pair of deltas. Conflict-resolving deltas take the role of *derivative modules* [24] or *lifters* [36]. They contain only the functionality necessary when several interacting features are selected together.

Definition 7 (Unambiguous Delta Model)**.** *Given a delta model $(D, \prec)$, we say that the model is* unambiguous *iff*

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

If a delta model is unambiguous, we can show that it has a unique derivation. In order to prove this, we need some intermediate results. Lemma 1 states that in an unambiguous delta model, any two deltas in a derivation are either ordered or commutative.

Lemma 1. *Given an unambiguous delta model $P = (D, \prec)$ and $d_2 \cdot y \cdot x \cdot d_1 \in \operatorname{derv}(P)$, where $x, y \in D$ and $d_1, d_2 \in D^*$. Then either $x \prec y$ or $d_2 \cdot y \cdot x \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$.*

Lemma 2 states that removing a minimal element with respect to the partial order preserves unambiguity of delta models.

Lemma 2. *If $P = (D, \prec)$ is an unambiguous delta model and w is minimal in $\prec$, then $(D \setminus \{w\}, \prec')$, where $\prec'$ is $\prec$ restricted to $D \setminus \{w\}$, is also an unambiguous delta model.*

Lemma 3 formulates that a minimal element in the partial order can be moved to the front of any derivation from an unambiguous delta model without changing the meaning of that derivation.

Lemma 3. *Given an unambiguous delta model $P = (D, \prec)$. Let $x_n \cdot \ldots \cdot x_1 \in \operatorname{derv}(P)$, where $\{x_1, \ldots, x_n\} = D$, with x_i minimal in $\prec$. Then $x_n \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i+1} \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_i$.*

The following theorem states that every unambiguous delta model has a unique derivation. This reduces the effort of checking that all possible derivations of a delta model have the same effect to checking that all conflicts between pairs of deltas are eliminated by conflict resolving deltas. The proof is by induction over the size of the delta model.

Theorem 1. *An unambiguous delta model has a unique derivation.*

2.3 Consistent Conflict Resolution

Although the notion of unambiguous delta model alleviates the task of establishing that a delta model has a unique derivation, unambiguity is still quite complex to check. The reason is that the definition of a conflict resolving delta (Definition 6) quantifies over all elements of D^*. Hence, in order to check that a delta is indeed a conflict resolver, all these sequences of deltas have to be inspected. However, for interesting classes of deltoids, a simpler criterion exists to make checking ambiguity more feasible. The *consistent conflict resolution* property states that if a delta z resolves an (x, y)-conflict when it is applied directly after x and y, it also resolves the conflict after the application of any sequence of intermediate deltas.

Definition 8 (Consistent Conflict Resolution)**.** *A deltoid $(\mathcal{D}, \cdot, \epsilon)$ is said to exhibit* consistent conflict resolution *iff the following condition holds:*

$$\begin{array}{l} \forall x, y, z \in \mathcal{D} : \\ \quad z \cdot y \cdot x = z \cdot x \cdot y \Rightarrow \forall d \in \mathcal{D} : z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y. \end{array}$$

If a deltoid $(\mathcal{D}, \cdot, \epsilon)$ exhibits consistent conflict resolution, then a delta model $(D, \prec)$ with $D \subseteq \mathcal{D}$ is also said to exhibit the property.

Note that the consistent conflict resolution property is checked at the level of the underlying deltoid, rather than for any specific delta model. Hence, it has to be established only once for a given deltoid and then holds for all delta models based on that deltoid.

To establish the unambiguity of a delta model exhibiting consistent conflict resolution, it is sufficient to check that for each pair of conflicting deltas x and y there exists a conflict-resolving delta z, such that $x \prec z \wedge y \prec z \wedge z \cdot y \cdot x = z \cdot x \cdot y$. We need not quantify over all possible intermediate sequences of deltas. Consequently, unambiguity of delta models can be established much more efficiently. This is formalized in the next theorem.

Theorem 2. *Given delta model $P = (D, \prec)$ exhibiting consistent conflict resolution, for all deltas $x, y, z \in D$, it is true that $x \prec z \wedge y \prec z \wedge z \cdot y \cdot x = z \cdot x \cdot y \implies (x, y) \lhd z$.*

3. Reintroducing Products

Thus far, only modifications have been considered, without considering the products that we modify. Products can be reintroduced, by defining the notion of application of a delta to a product. Firstly, we select a set of products.

Definition 9 (Products)**.** *Let $\mathcal{P}$ denote a set of possible* products*.*

Applying a delta to a product results in another product. This is captured by the notion of delta application.

Definition 10 (Delta Application)**.** Delta application *is an operation $-(-) : \mathcal{D} \times \mathcal{P} \to \mathcal{P}$. 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.*

This definition implies that one generates a product from a sequence of deltas by first composing the deltas and then applying the result to the core product. A much stronger version of delta application is possible, borrowing the notion of monoid action.

Definition 11 (Delta Action)**.** *A delta application operation* $-(-) : \mathcal{D} \times \mathcal{P} \to \mathcal{P}$ *is called a* delta action *if it satisfies the conditions* $(y \cdot x)(p) = y(x(p))$ *and* $\epsilon(p) = p$*, for all* $x, y \in \mathcal{D}$ *and* $p \in \mathcal{P}$*.*

This generalises the case when $\cdot$ is function composition and $-(-)$ is function application.

4. Product Lines

Using the introduced concepts of delta models, products and delta application we can now abstractly define product lines, thus providing a link from feature configurations on the feature modeling level to product representations. We will extend the concept of unambiguity to the level of product lines and provide an efficient condition to check unambiguity.

4.1 Defining Product Lines

Product line variability is predominantly captured by features where a feature captures a designated product characteristic or an increment to product functionality. At the level of the feature model features are merely labels without inherent semantic meaning. A product can be characterized by the set of features it provides.

Definition 12 (Features)**.** *Let* $\mathcal{F}$ *denote a universal set of* features*.*

The set of products in a product line can be represented by a feature model. Many formal descriptions [18, 20, 43] agree that a feature model determines a set of valid feature configurations.

Definition 13 (Feature Model)**.** *A* feature model $\Phi \subseteq \mathscr{P}(\mathcal{F})$ *is a set of sets of features from* $\mathcal{F}$*. Each* $F \in \Phi$ *is a set of features corresponding to a valid* feature configuration*.*

In order to bridge the gap between features and product line artifacts, we introduce application conditions for deltas. An application condition attached to a delta determines for which feature configuration the delta has to be applied.

Definition 14 (Application Function and Condition)**.** *Let* $D \subseteq \mathcal{D}$ *be a set of deltas. An* application function $\gamma : D \to \mathscr{P}(\mathscr{P}(\mathcal{F}))$ *gives the feature configurations each delta* $x \in D$ *is applicable to. Thus,* $F \in \gamma(x)$ *denotes that delta* x *is applicable for feature configuration* F*.* $\gamma(x)$ *is called the* application condition *for* x*.*

A product line is defined by its feature model, characterizing all member products by a set of valid feature configurations, the core product, the associated delta model, containing the modifications used to obtain further products, and the application function, associating features and deltas. None of these elements can be inferred from the other elements.

Definition 15 (Product Line)**.** *A* product line *is a tuple* $(\Phi, c, D, \prec, \gamma)$*, 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*.*

If feature configuration F is valid according to Φ, its corresponding product is defined by the delta model containing only the deltas applicable to F. Selecting such deltas gives a delta model whose derivations applied to the core product correspond to the desired product for feature configuration F.

Definition 16 (Selected Delta Model)**.** *Given a product line* $PL = (\Phi, c, D, \prec, \gamma)$*, a* selected delta model *for feature configuration* $F \in \Phi$*, denoted* $PL \restriction F$*, is the 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 now define the set of products generated from a product line.

Definition 17 (Generated Products)**.** *Given a product line* $PL = (\Phi, c, D, \prec, \gamma)$*, the set of generated products for feature configuration* $F \in \Phi$ *is defined as follows:*

$$\mathrm{prod}(PL, F) \stackrel{\text{def}}{=} \{\, x(c) \mid x \in \mathrm{derv}(PL \restriction F) \,\}.$$

4.2 Unambiguity of Product Lines

As argued in Section 2, unambiguity of delta models is a desired property because it ensures unique derivation and, consequently, a unique generated product. We now lift unambiguity to the product line level. A product line is unambiguous if every selected delta model is unambiguous. This means that every valid feature configuration yields a uniquely defined product, which is an important condition for the applicability of automated product derivation.

Definition 18 (Unambiguous Product Line)**.** *A product line* $PL = (\Phi, c, D, \prec, \gamma)$ *is* unambiguous *iff*

$$\forall F \in \Phi : PL \restriction F \text{ is an unambiguous delta model.}$$

The unambiguity of a product line can be checked by generating the selected delta models of all valid feature configurations and checking unambiguity by the criteria proposed in Section 2. However, as the set of feature configurations is often exponential in the number of features, that naive approach would be rather expensive. Instead, we propose the notion of a globally unambiguous product line that implies product line unambiguity.

We first introduce a shorthand notation for the set of feature configurations for which two deltas x and y are applicable.

Notation 2. *Given a product-line* $(\Phi, c, D, \prec, \gamma)$*, the set of valid feature configurations to which the deltas* $x, y \in D$ *apply is denoted:*

$$\mathcal{V}^{x,y} \stackrel{\text{def}}{=} \Phi \cap \gamma(x) \cap \gamma(y).$$

A product line is globally unambiguous if for any two conflicting deltas x and y applied together for a set of feature configurations, there is a conflict-resolving delta z applicable in at least the same set of feature configurations. Thus for any selected delta model in which x and y appear together, the conflict is resolved by the same delta z. Global unambiguity of a product line can be checked by inspecting the product line only once and does not require all selected delta models to be generated.

Definition 19 (Globally Unambiguous Product Line)**.** *A product line* $(\Phi, c, D, \prec, \gamma)$ *is called* globally unambiguous *if and only if*

$$\forall x, y \in D : \mathcal{V}^{x,y} = \varnothing \\ \vee\ x \lightning y \Rightarrow \exists z \in D : (\mathcal{V}^{x,y} \subseteq \gamma(z) \wedge (x, y) \lhd z).$$

The following theorem states that any globally unambiguous product line is also an unambiguous product line. Hence, it suffices to check global unambiguity by inspecting the product line once to ensure that all products that can be generated from the product line are uniquely determined.

Theorem 3. *A globally unambiguous product line is unambiguous.*

A product line can be unambiguous, but not globally unambiguous if conflicts between two deltas x and y are resolved by different conflict-resolving deltas z for different feature configurations. For example, take a product line PL in which the only conflicting deltas x and y are applied together for feature configurations F and F'. For feature configuration F only delta z resolves the conflict, $(x, y) \lhd_{PL \restriction F} z$, and for feature configuration F' only delta z' resolves the conflict, $(x, y) \lhd_{PL \restriction F'} z'$, but $z \neq z'$. Hence, the product line is unambiguous, because the conflict is resolved in all selected delta models, but not globally unambiguous, because the conflict resolving delta is not the same in each one. Here the $\lhd$ operator is annotated with the delta model for which it applies.

5. A Deltoid for Object-Oriented Programs

We now present a concrete deltoid for object-oriented programs to demonstrate our approach. In this section, deltas manipulate object-oriented programs on a coarse-grained level. That is, a delta can add, remove or modify classes. Modifications of classes include addition, removal and replacement of fields and methods.

Notation 3. *Let $f : X \rightharpoonup Y$ denote that f is a partial function from X to Y. If $f(x)$ is undefined for $x \in X$, we write $f(x) = \bot$, where $\bot \notin Y$.*

Notation 4. *Given a set X where $- \notin X$, define the notation:*

$$X^- \stackrel{\text{def}}{=} X \cup \{-\}.$$

5.1 Software Products

For simplicity, we abstract from a concrete programming language as well as from the concrete implementation of methods, and focus only on the structural aspects of object-oriented programs. First, we introduce the notion of identifiers for classes, methods and fields.

Definition 20 (Identifiers)**.** *Define a global set of* identifiers $\mathcal{I}$*, used for classes, methods and fields.*

Further, we fix an abstract set of method and field definitions.

Definition 21 (Method and Field Definitions)**.** *Define a global set of* method and field definitions $\mathcal{M}$.

A class is defined as a partial mapping from identifiers to method and field definitions.

Definition 22 (Class Definitions)**.** *The collection of* class definitions *is the set of partial functions $\Psi = \mathcal{I} \rightharpoonup \mathcal{M}$. Such a class definition $\psi \in \Psi$ maps some identifiers to their definition. Unmapped identifiers are not defined in the class.*

As an example, consider the following class definition. Only the explicitly mentioned identifiers are considered to be defined. As we abstract from concrete method implementations, we use capital letters to refer to method implementations, where different letters represent distinct implementations.

$$\left\{ \begin{array}{l} \texttt{f} \mapsto \texttt{f(): void \{ } A \texttt{ \},} \\ \texttt{g} \mapsto \texttt{g(): bool \{ } B \texttt{ \},} \\ \texttt{i} \mapsto \texttt{i: int} \end{array} \right\}.$$

A program is a set of classes, mapping identifiers to class definitions.

Definition 23 (Programs)**.** *Equate the set of products $\mathcal{P}$ with the set of* programs *in an object-oriented language: $\mathcal{P} = \mathcal{I} \rightharpoonup \Psi$.*

As an example, consider the following program definition:

$$\left\{ \begin{array}{l} \texttt{C} \mapsto \left\{ \begin{array}{l} \texttt{f} \mapsto \texttt{f(): void \{ } A \texttt{ \},} \\ \texttt{g} \mapsto \texttt{g(): bool \{ } B \texttt{ \},} \\ \texttt{i} \mapsto \texttt{i: int} \end{array} \right\}, \\ \texttt{D} \mapsto \left\{ \begin{array}{l} \texttt{h} \mapsto \texttt{h(x: int): int \{ } C \texttt{ \},} \\ \texttt{b} \mapsto \texttt{b: bool} \end{array} \right\} \end{array} \right\}.$$

5.2 Software Deltas

Software deltas modify a program by adding, modifying and removing classes. A class modification includes adding, replacing and removing methods and fields, or replacing the class completely.

To ensure that composition of deltas produces a closed form, we distinguish between updating a class and replacing it. A class *replacement* completely replaces an existing class. A class *update* modifies the original class at the method/field level. Modifying a class that does not exist is treated as adding a new class. The definition of a software delta captures this set of program modifications.

Definition 24 (Software Deltas)**.** *The set of* software deltas *is defined as $\mathcal{D} = \mathcal{I} \rightharpoonup (\{\mathsf{r}\} \times (\mathcal{I} \rightharpoonup \mathcal{M}) \cup \{\mathsf{u}\} \times (\mathcal{I} \rightharpoonup \mathcal{M}^-))^-$. Each delta $d \in D$ is a partial function representing class modifications.* r *and* u *represent 'replace' and 'update', respectively. Mapping an identifier to $-$ indicates removal from the product. $\epsilon = \varnothing$ is the empty delta, modifying nothing.*

An example software delta is:

$$\left\{ \begin{array}{l} \texttt{C} \mapsto \mathsf{u} \left\{ \begin{array}{l} \texttt{f} \mapsto -, \\ \texttt{z} \mapsto \texttt{z(): void \{ } D \texttt{ \},} \\ \texttt{i} \mapsto \texttt{i: float} \end{array} \right\}, \\ \texttt{D} \mapsto - \end{array} \right\}.$$

In contrast to previous work [39], the removal of an element in this concrete deltoid does not require that the element is already present, nor does addition require its absence. These simplifications ensure that every derivation of deltas is well-defined.

Now we introduce some notation required in the next few definitions. The first notation is used to combine two partial functions into another partial function by some binary operation on their codomain.

Notation 5. *Use the following notation to lift an operator $\bigcirc$ on two partial functions to the values in their codomain. For $i \in \mathcal{I}$:*

$$(a \mathbin{\overline{\bigcirc}} b)(i) \stackrel{\text{def}}{=} a(i) \bigcirc b(i).$$

The following notation excludes method and field removals from a class update. Since class replacements should not contain removals, this notation is needed when a class update is sequentially composed with a class replacement.

Notation 6. *Given class update $f : \mathcal{I} \rightharpoonup \mathcal{M}^-$, define f^* as f, but without any method or field removals:*

$$f^*(i) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \bot & \text{if } f(i) = - \\ f(i) & \text{otherwise} \end{array} \right.$$

Now define sequential composition of software deltas.

Definition 25 (Sequential Composition of Software Deltas)**.** *The* sequential composition of software deltas *$\cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is defined as*

$$y \cdot x \stackrel{\text{def}}{=} y \mathbin{\overline{\oplus_C}} x,$$

where the operator $\oplus_C$, working on the level of class modifications, with $e, f : \mathcal{I} \rightharpoonup \mathcal{M}^-$ and $g, h : \mathcal{I} \rightharpoonup \mathcal{M}$, is

$\oplus_C$	$\bot$	$-$	$\mathsf{u}\ f$	$\mathsf{r}\ h$
$\bot$	$\bot$	$-$	$\mathsf{u}\ f$	$\mathsf{r}\ h$
$-$	$-$	$-$	$-$	$-$
$\mathsf{u}\ e$	$\mathsf{u}\ e$	$\mathsf{r}\ e^*$	$\mathsf{u}\ (e \mathbin{\overline{\oplus_M}} f)$	$\mathsf{r}\ (e \mathbin{\overline{\oplus_M}} h)^*$
$\mathsf{r}\ g$	$\mathsf{r}\ g$	$\mathsf{r}\ g$	$\mathsf{r}\ g$	$\mathsf{r}\ g$

and $\oplus_M$, working on the level of method and field definitions, with $m, n \in \mathcal{M}$, is

$\oplus_M$	$\bot$	$-$	n
$\bot$	$\bot$	$-$	n
$-$	$-$	$-$	$-$
m	m	m	m .

The options for combining methods are limited, but Section 5.4 will redefine $\oplus_M$ to allow method wrapping. The definition of software delta composition gives concrete meaning to the notion of incompatibility. Two deltas are incompatible if they map the same identifier to two different definitions.

Lemma 4. *Software deltas are a deltoid.*

Lemma 5. *Software deltas exhibit consistent conflict resolution.*

Finally, we define software delta application to apply a software delta to a program.

Definition 26 (Software Delta Application)**.** *Given delta $y \in \mathcal{D}$ and product $p \in \mathcal{P}$,* software delta application *is an operation $-(-) : \mathcal{D} \times \mathcal{P} \to \mathcal{P}$ defined as follows:*

$$y(p) \stackrel{\text{def}}{=} y \mathbin{\overline{\otimes_C}} p,$$

where the operators $\otimes_C$, with $f : \mathcal{I} \rightharpoonup \mathcal{M}^-$ and $g, h : \mathcal{I} \rightharpoonup \mathcal{M}$, and $\otimes_M$, with $m, n \in \mathcal{M}$, are defined as

$\otimes_C$	$\bot$	h
$\bot$	$\bot$	h
$-$	$\bot$	$\bot$
u f	f^*	$f \mathbin{\overline{\otimes_M}} h$
r g	g	g

$\otimes_M$	$\bot$	n
$\bot$	$\bot$	n
$-$	$\bot$	$\bot$
m	m	m .

Lemma 6. *Software delta application is a delta action.*

5.3 Example Product Line

We now show an example product line based on software deltas. It is a product line of editor widgets to be used for integrated development environments and other text-editing applications.

5.3.1 The Core Program

The product line is based on core program c containing one class:

```
{ Editor ↦
  { model       ↦ model: Model,
    draw        ↦ draw(): void,
    getModel    ↦ getModel(): Model,
    font        ↦ font(c: int): Font { A },
    onMouseOver ↦ onMouseOver(c: int): void { B } } }.
```

The `model` field and the `draw` and `getModel` methods implement the basic functionality of the widget and are never modified. The `font` method specifies the proper font for each character in the editor's text area. In the core product it may return a monospaced black font with no decoration. The `onMouseOver` method is an event handler for when the mouse cursor hovers over specific characters of the content. Both methods will be modified by deltas.

5.3.2 The Feature Model

The editor product line is based on the features $\{Editor, SH, ERR, TT\}$. $Editor$ is the mandatory base feature of the product line, implemented by the core program. SH stands for syntax highlighting of programming language constructs. It modifies the `font` method. ERR implements error detection. It underlines errors in code and shows relevant information in a tooltip when the mouse hovers over the error. This feature modifies the `font` and `onMouseOver` methods. TT allows the editor to show generic information in tooltips. It modifies the `onMouseOver` method. The feature model Φ of the editor product line allows any combination the three optional features (Ed abbreviates $Editor$):

$$\Phi = \left\{ \begin{array}{l} \{Ed\}, \{Ed, SH\}, \{Ed, ERR\}, \{Ed, TT\}, \\ \{Ed, SH, ERR\}, \{Ed, SH, TT\}, \\ \{Ed, ERR, TT\}, \{Ed, SH, ERR, TT\} \end{array} \right\}.$$

There are two potential conflicts. SH and ERR both modify `font`. Similarly, ERR and TT both modify `onMouseOver`.

5.3.3 Delta Model and Application Conditions

The base code for all three optional features of the editor product line can be developed in isolation. One delta is created for each, implementing that feature as a modification to the `Editor` class without considering potential conflicts. These deltas work as expected if their respective feature is the only one included in the product. They are depicted in the top row of Figure 2 as $d_1, d_2, d_3 \in D$.

Because some feature configurations include interacting features, conflict resolving deltas for the two potential conflicts in our model are designed ($d_4, d_5 \in D$ in Figure 2). Delta d_4 deals with the interaction between SH and ERR, combining the coloring of SH with the underlining of ERR. Similarly, delta d_5 handles the interaction between ERR and TT.

Figure 2 also shows the application conditions $\gamma(d_i)$ for each delta $d_i \in D$ in the form of propositional logic formulae, where the propositions are features. The application conditions of the conflict resolving deltas ensure that they are applied if and only if their two conflicting deltas are applied.

5.3.4 Global Unambiguity

The editor product line is globally unambiguous. As the underlying deltoid exhibits consistent conflict resolution (cf. Lemma 5), this can easily be verified. There are only two pairs of deltas in conflict: $d_1 \lightning d_2$ and $d_2 \lightning d_3$. For both conflicts, there is a conflict resolving delta: $(d_1, d_2) \lhd d_4$ and $(d_2, d_3) \lhd d_5$. By the choice of γ, the appropriate conflict-resolving delta is present in each feature configuration in which conflicting deltas appear.

5.3.5 Generating a Product

To illustrate the process of product generation (cf. Definition 17), we now derive the product for a given feature configuration $F = \{Editor, SH, ERR\} \in \Phi$. We first generate the selected delta model $PL \restriction F = (D', \prec')$, where $D' = \{d_1, d_2, d_4\}$ and $\prec' = \{(d_1, d_4), (d_2, d_4)\}$. Since PL is globally unambiguous, it is sufficient to select one derivation of the delta model to generate the uniquely defined product. We use $x = d_4 \cdot d_2 \cdot d_1$. Applying Definition 25, x becomes:

```
{ Editor ↦
  u { font        ↦ font(c: int): Font { G },
    { onMouseOver ↦ onMouseOver(c: int): void { E } } } }
```

Applying x to c (Definition 26) results in the product $x(c)$:

```
{ Editor ↦
  { model       ↦ model: Model,
    draw        ↦ draw(): void,
    getModel    ↦ getModel(): Model,
    font        ↦ font(c: int): Font { G },
    onMouseOver ↦ onMouseOver(c: int): void { E } } }.
```

5.4 A Deltoid for Aspect Oriented Programming

Arguably, the essence of AOP is quantification (pointcuts) and wrapping (around advice). A rudimentary semantic interpretation of quantification is simply the set of all possible matching joinpoints with the same advice. Based on this simplification, we adapt the previous concrete deltoid (Sections 5.1 and 5.2) to include method wrapping. We do so by modifying method bodies $\mathcal{M}$ in classes and deltas to have the following (abstract) grammar:

$$\begin{array}{lll} \mathcal{M} \ni m & ::= & b \mid w[m] \qquad b \in \mathcal{B} \\ \mathcal{W} \ni w[\ ] & ::= & e[\ ] \mid w[w[\ ]] \qquad e \in \mathcal{E}. \end{array}$$

where $\mathcal{B}$ is a set of basic method bodies and $\mathcal{E}$ is a set of primitive wrapping methods (around advice). The notation $w[\ ]$ denotes a wrapping method with a hole in it, where the hole corresponds to where the call to the original method is made, and $w[m]$ denotes that body m is wrapped by w. Methods with a hole in them do not appear in products.

Given these ingredients, only the definitions of $\oplus_M$ and $\otimes_M$ from Definitions 25 and 26 need to change (m, n have no hole):

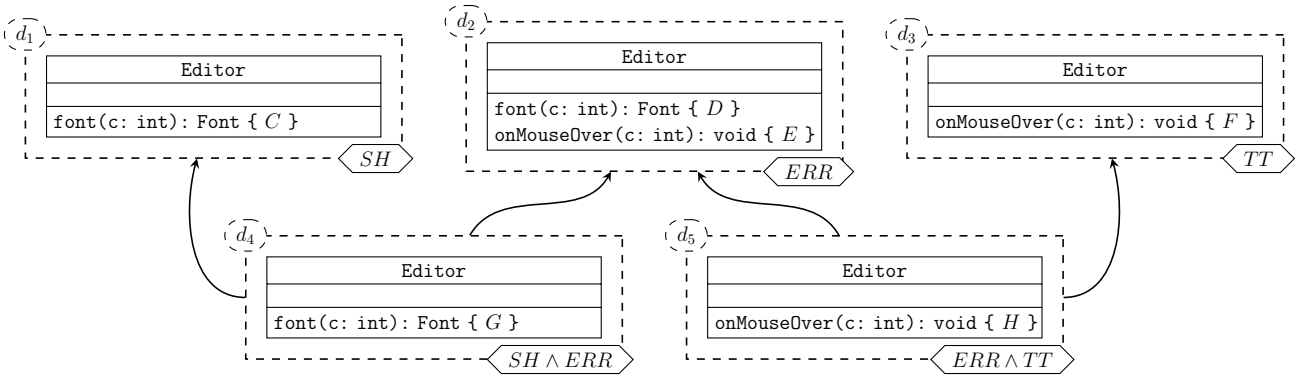

Figure 2. A visual representation of the delta model $(D, \prec)$ from the editor product line. The dashed boxes represent the deltas $d_1, \ldots, d_5 \in D$. The ordering $\prec$ is represented by the arrows. Deltas are decorated with their application condition $\gamma(d_i)$.

$\oplus_M$	$\bot$	$-$	n	$v[\ ]$
$\bot$	$\bot$	$-$	n	$v[\ ]$
$-$	$-$	$-$	$-$	$-$
m	m	m	m	m
$w[\ ]$	$w[\ ]$	$-$	$w[n]$	$w[v[\ ]]$

$\otimes_M$	$\bot$	n
$\bot$	$\bot$	n
$-$	$\bot$	$\bot$
m	m	m
$w[\ ]$	$\bot$	$w[n]$.

6. Other Algebraic Approaches

Other algebraic approaches describing the underlying structure of software product lines exist [4, 8]. These formalise the mechanisms underlying AHEAD [7], GenVoca [6], and FeatureHouse [2]. The key difference is that our approach considers the collection of modifications for an entire product line, rather than a single product at a time. The second difference is that those approaches generally arrange deltas into modifications and introductions, whereas we assume a unified collection of deltas. Here we compare our approach with two recent proposals, namely, Apel et al's Quark model [4] and Batory and Smith's Finite Map Spaces [8]. From an algebraic perspective, these two proposals are quite similar, so we consider them together. By encoding these frameworks, we demonstrate that our formalism is sufficient to express these using simpler notions, as well as providing an alternative foundation for tools based on these formalisms. We also consider the formalization of patch theory underlying the Darcs version control system [19], which is algebraically similar.

6.1 Quarks and Finite Map Spaces

Both Apel et al. [4] and Batory and Smith [8] base the description of a product line on the following ingredients (our notation):

- *introductions*: a commutative idempotent monoid $(I, +, 0)$, where $+ : I \times I \to I$.
- *modifications*: a monoid $(M, \bullet, 1)$, where $\bullet : M \times M \to M$.
- an operation $\odot : M \times I \to I$ applying modifications to introductions, typically satisfying
  - M is an monoid action over I: $1 \odot i = i$ and $(m \bullet n) \odot i = m \odot (n \odot i)$,
  - Distributivity: $m \odot (i + j) = m \odot i + m \odot j$, and
  - $m \odot 0 = 0$.

Introductions play a dual role. They correspond to (elements of) products, as well as acting as one kind of delta; modifications are the other kind. That is, an introduction $i \in I$ in a delta corresponds to introducing a new element into a product and a modification $m \in M$ corresponds to an operation modifying an existing element.

Introductions and modifications are combined to form quarks Q, which correspond to our deltas. Different notions of quark and quark composition ($\blacklozenge : Q \times Q \to Q$) are listed below. These capture combinations of the following: *local composition* which applies modifications to elements (introductions) already in the product; *global composition* which applies to all elements of the final product; and *modifiers of modifiers* which modify modifications rather than elements of the product. Modifiers of modifiers consist of functions $h : M \to M$. Batory and Smith introduce the *syntactic* function R^h to recursively apply all higher-order modifications, as follows, $R^h(m) = h(m)$, for $m \in M$; $R^h(i) = i$ for $i \in I$ and $R^h(m \bullet m') = R^h(m) \bullet R^h(m')$ and so on. We also include the operation $\mathsf{image} : Q \to I$ used (sometimes implicitly) to extract the final product from a quark.

local quark composition (Apel et al.)

- $Q = I \times M$ — an introduction and a local modification
- $\langle i_2, l_2 \rangle \blacklozenge \langle i_1, l_1 \rangle = \langle i_2 + (l_2 \odot i_1), l_2 \bullet l_1 \rangle$
- $\mathsf{image}(\langle i, l \rangle) = i$

global quark composition (Apel et al.)

- $Q = I \times M$ — an introduction and a global modification
- $\langle i_2, g_2 \rangle \blacklozenge \langle i_1, g_1 \rangle = \langle (g_2 \bullet g_1) \odot (i_2 + i_1), g_2 \bullet g_1 \rangle$
- $\mathsf{image}(\langle i, g \rangle) = i$

full quark composition (Apel et al.)

- $Q = M \times I \times M$ — a global modification, an introduction, and a local modification
- $\langle g_2, i_2, l_2 \rangle \blacklozenge \langle g_1, i_1, l_1 \rangle = \langle g_2 \bullet g_1, (g_2 \bullet g_1) \odot (i_2 + (l_2 \odot i_1)), l_2 \bullet l_1 \rangle$
- $\mathsf{image}(\langle g, i, l \rangle) = i$

full quark composition (Batory & Smith)

- $Q = M \times I \times M$ — a global modification, an introduction, and a local modification
- $\langle g_2, i_2, l_2 \rangle \blacklozenge \langle g_1, i_1, l_1 \rangle = \langle g_2 \bullet g_1, i_2 + (l_2 \odot i_1), l_2 \bullet l_1 \rangle$
- $\mathsf{image}(\langle g, i, l \rangle) = g \odot i$

modifiers of modifiers (Batory & Smith)

- $Q = (M \to M) \times M \times I \times M$ — a modifier of modifiers, a global modification, an introduction, and a local modification

- $\langle h_2, g_2, i_2, l_2\rangle \blacklozenge \langle h_1, g_1, i_1, l_1\rangle = \langle h_2 \circ h_1, g_2 \bullet g_1, i_2 + (l_2 \odot i_1), l_2 \bullet l_1\rangle$
- $\mathsf{image}(\langle h, g, i, l\rangle) = R^h(g \odot i)$

In many cases, quark composition $\blacklozenge$ forms a monoid, with the appropriate tuple of units as the unit for $\blacklozenge$. Delta application $-(-) : Q \times I \to I$ (Definition 10) can be defined, for example, as $q(p) = \mathsf{image}(q \blacklozenge \langle p, 1\rangle)$, where $q \in Q$ is a quark and $p \in P$ is the core product. Note that the term $\langle p, 1\rangle$ needs to be adapted depending on the notion of quark being used.

In the absence of other axioms, not all of the quarks above form a deltoid, nor is delta application is always a delta action. Global quark composition and full quark composition (Apel et al.) are not associative, and delta application forms an action only for local quark composition. In addition, Apel et al.s' global quark composition and full quark composition produces results such as the following (for global quark composition)

$$(\langle i_3, g_3\rangle \blacklozenge \langle i_2, g_2\rangle) \blacklozenge \langle i_1, g_1\rangle =$$
$$\langle((q_3 \bullet g_2) \bullet g_1) \odot (((g_3 \bullet g_2) \odot (i_3 + i_2)) + i_1), (q_3 \bullet g_2) \bullet g_1\rangle.$$

which applies modifications g_3 and g_2 multiple times. To get this composition to behave, strong idempotence criteria are proposed, but these exclude modifications such as method wrapping.

We now describe how to encode local quark composition, full quark composition, and modifiers of modifiers more directly in our setting, by making introductions a kind of modification and by eliminating quarks (where possible). By ignoring the distinction between modifications and introductions, we can focus on deltas alone, and work in a simpler algebraic setting.

6.1.1 Encoding Local Quark Composition

Before proceeding, we note that $\langle 0, 1\rangle$ is the unit of $\blacklozenge$ for local quark composition, and that apart from the monoid laws for $\blacklozenge$ it also satisfies: $\langle i_1, m_1\rangle = \langle i_2, m_2\rangle$ if and only if $i_1 = i_2$ and $m_1 = m_2$.

The following definition introduces deltoid M_I consisting of deltas that are modifications $m \in M$ and introductions $i \in I$. We show that this is equivalent to $Q = I \times M$ with $\blacklozenge$ corresponding to local quark composition.

Definition 27. *Given $((M, \bullet, 1), (I, +, 0), \odot)$ as above. Define a monoid $M_I = ((M \cup I)^*, \cdot, \epsilon)$, where $\cdot$ is concatenation with unit the empty sequence ϵ, subject to the following equations ($m, n \in M$, $i, j \in I$, and $\mu, \nu, \eta \in M_I$):*

1. $\epsilon \cdot \mu = \mu = \mu \cdot \epsilon$
2. $\mu \cdot (\nu \cdot \eta) = (\mu \cdot \nu) \cdot \eta$
3. $m \cdot i = (m \odot i) \cdot m$
4. $i \cdot j = i + j = j + i = j \cdot i$
5. $i \cdot i = i + i = i$
6. $m \cdot n = m \bullet n$
7. $\epsilon = 0 = 1$.

Definition 27 forms a deltoid by taking sequences of modifications and introductions, modulo certain equations. The equations interpret various combinations of elements of M_I in terms of the original collection of operations. The most interesting is 3, which applies a modification m to an introduction i, via $m \odot i$, and shuffles m later in the sequence to apply to subsequent introductions. Note that equations 1 and 2 are redundant and follow from the fact that $\cdot$ is concatenation and $1 = \epsilon$ its unit.

Delta action is defined inductively over the elements of M_I, applying each element of M_I to I via the appropriate function from the original monoids.

Definition 28. *The delta action $-(-) : M_I \times I \to I$ for M_I is*

$$\begin{aligned} \epsilon(p) &= p \\ m(p) &= m \odot p \\ i(p) &= i + p \\ (\mu \cdot \nu)(p) &= \mu(\nu(p)). \end{aligned}$$

where $m \in M$, $i \in I$, $\mu, \nu \in M_I$ and $p \in I$.

The following is a monoid homomorphism from quarks to M_I.

Definition 29. *Define $[\![-]\!] : Q \to M_I$ as*

$$[\![\langle i, m\rangle]\!] = i \cdot m.$$

The mapping from M_I to quarks defined in the following is also a monoid homomorphism.

Definition 30. *Define $\langle\!\langle - \rangle\!\rangle : M_I \to Q$ as*

$$\begin{aligned} \langle\!\langle \epsilon \rangle\!\rangle &= \langle 0, 1\rangle \\ \langle\!\langle m \rangle\!\rangle &= \langle 0, m\rangle \\ \langle\!\langle i \rangle\!\rangle &= \langle i, 1\rangle \\ \langle\!\langle \mu \cdot \nu \rangle\!\rangle &= \langle\!\langle \mu \rangle\!\rangle \blacklozenge \langle\!\langle \nu \rangle\!\rangle. \end{aligned}$$

Quarks with local quark composition are isomorphic to M_I which is stated in Theorem 4, supporting that making the distinction between introductions and modifications is unnecessary.

Theorem 4. *For all $q, q' \in Q$ and $\mu, \nu \in M_I$, we have*

1. $\langle\!\langle [\![q]\!] \rangle\!\rangle = q$,
2. $[\![\langle\!\langle \mu \rangle\!\rangle]\!] = \mu$,
3. *if $q = q'$, then $[\![q]\!] = [\![q']\!]$, and*
4. *if $\mu = \nu$, then $\langle\!\langle \mu \rangle\!\rangle = \langle\!\langle \nu \rangle\!\rangle$.*

Theorem 5 shows that not only are quarks and M_I isomorphic, their notions of delta action behave the same.

Theorem 5. *For all $q \in Q$ and all $p \in I$,*

$$\mathsf{image}(q \blacklozenge \langle p, 1\rangle) = [\![q]\!](p)$$

and for all $\mu \in M_I$ and all $i \in I$,

$$\mathsf{image}(\langle\!\langle \mu \rangle\!\rangle \blacklozenge \langle p, 1\rangle) = \mu(p).$$

6.1.2 Encoding Batory and Smith's Full Quark Composition

Encoding full quark composition is straightforward. To do so, we adapt the encoding above to use quarks $Q = M \times M_I$, where quark composition is $\langle m, \mu\rangle \blacklozenge \langle n, \nu\rangle = \langle m \bullet n, \mu \cdot \nu\rangle$ and define delta application $-(-) : Q \times I \to I$ to be $\langle m, \mu\rangle(p) = m \odot (\mu(p))$, relying on delta application for M_I.

In the absence of other assumptions, this notion of delta application is not an action, as for example:

$$\begin{aligned} \langle m, \mu\rangle \blacklozenge \langle n, \nu\rangle(p) &= \langle m \bullet n, \mu \cdot \nu\rangle(p) \\ &= (m \bullet n) \odot (\mu \cdot \nu(p)) \\ &= (m \bullet n) \odot (\mu(\nu(p))) \end{aligned}$$

whereas

$$\langle m, \mu\rangle(\langle n, \mu\rangle(p)) = m \odot (\mu(n \odot (\nu(p))))$$

If we instantiate μ and ν with m' and n' we have in the first case:

$$\begin{aligned} (m \bullet n) \odot m'(n'(p)) &= (m \bullet n) \odot (m' \odot (n \odot p)) \\ &= (m \bullet n \bullet m' \bullet n') \odot p \end{aligned}$$

and in the second case

$$\begin{aligned} m \odot m'(n \odot n'(p)) &= m \odot (m' \odot (n \odot (n' \odot p))) \\ &= (m \bullet m' \bullet n \bullet n') \odot p \end{aligned}$$

which are equal in general only if $\bullet$ is commutative.

6.1.3 Encoding Batory and Smith's Modifiers of Modifiers

Encoding modifiers of modifiers is also straightforward. We assume that such modifiers, $h : M \to M$, are endomorphisms

on the monoid of modifications (this is already implicit in Batory and Smith's R^h function): that is, $h(1) = 1$ and $h(m_2 \bullet m_1) = h(m_2) \bullet h(m_1)$, for all $m_1, m_2 \in M$.

We can extend the previous example to apply higher-order modifiers to the global modifications as follows:

- quarks: $Q = (M \to M) \times M \times M_I$ — a modifier of modifiers, a global modification, and a delta
- composition: $\langle h_2, g_2, \mu_2 \rangle \blacklozenge \langle h_1, g_1, \mu_1 \rangle = \langle h_2, g_2, \mu_2 \cdot \mu_1 \rangle$, and
- delta application is $\langle h, g, \mu \rangle(p) = h(g) \odot (\mu(p))$.

To modify this definition so that h applies also to the local modifications requires lifting $h : M \to M$ to $h_I : M_I \to M_I$, defined by the following equation:

$$h(m \odot i) \quad = \quad h(m) \odot i.$$

where $h(i) = h(1 \cdot i) = h(1 \odot i) = h(1) \odot i = 1 \odot i = i$. In this case the delta application becomes

$$\langle h, g, \mu \rangle(p) = h(g) \odot (h(\mu)(p)).$$

Again delta application is not an action, for the same reason as for full quark composition.

Apel et al. [4] give the signatures of an entire hierarchy of modifiers of modifiers, but provide no further details.

6.2 Darcs and Patch Theory

The version control system Darcs is formalised in terms of *patch theory* [19]. The underlying formalism has some similarities with our work. Most notable is that 'patches' are modeled using a semigroup with inverses. This structure is a monoid at heart, with additional properties (such as inverses) that do not entirely make sense in our setting. The most significant similarity is that they deal with *conflictors* (entities for resolving conflicts), which are similar to our conflict resolving deltas. Conflictors have a more complex set of properties than our conflict resolving deltas due to the added structure of their core setting. Patch theory should nonetheless offer inspiration to guide future research.

7. Related Work

In general, approaches to facilitating automated product generation for software product lines can be classified in two main directions [23]. Firstly, annotative approaches, such as conditional compilation, frames [45] or COLORED FEATHERWEIGHT JAVA (CFJ) [21], mark a model of the complete product line with respect to product features and remove marked product parts to obtain a product for a particular feature configuration.

Secondly, compositional approaches, such as delta modeling [38–40], associate product fragments to product features, which are assembled to implement a particular feature configuration. A prominent example of this approach is AHEAD [7], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques [17, 31, 44]. Apel et al. [37] apply model superposition to compose model fragments. Perrouin et al. [34] obtain a product model by model composition and subsequently refinement by model transformation. In Haugen et al. [16], a set of models is represented by a base model with associated variability and resolution models determining how modeling elements of the base model have to be replaced for a particular product model.

On the programming language level, several program modularization techniques [27], such as aspects [22], framed aspects [28], mixins [41], hyperslices [42] or traits [9, 15], are used to implement features in a compositional fashion. In addition, the modularity concepts of recent languages, such as SCALA [32] or NEWSPEAK [10], can be used to represent product features. CeasarJ [30] and Aspectual Feature Modules [3] are proposed as a combination of feature modules and aspects to modularize crosscutting concerns.

The notion of program deltas was introduced by Lopez-Herrejon et al. [27] to describe the modifications of object-oriented programs. Schaefer et al. [40] introduced the concept of delta modeling as a means to develop product line artifacts suitable for automated product derivation and implemented with frame technology [45]. In subsequent work [38], delta modeling is extended to a seamless model-based development approach for SPLs where an initial product line representation is stepwise refined until an implementation can be generated. The conceptual ideas of delta modeling have also been instantiated on the programming language level in an extension of Java with core and delta modules allowing the automatic generation of Java-based product implementations [39].

Originally, the delta model of a product line consisted of a single core and a set of incomparable product deltas [38, 40]. Conflicts between deltas applicable for the same feature configuration were prohibited. In order to express all possible products, an additional delta covering the combination of the potentially conflicting deltas had to be specified leading to product fragments. Subsequently, a partial ordering between deltas was introduced [39]. However, it was required that all conflicts were manually resolved by specifying an appropriate ordering. In contrast, in this paper, a more flexible notion of conflicts and conflict resolution is proposed that allows intermediate conflicts between deltas as long as they are eliminated later in a derivation by a conflict-resolving delta. The notion of conflict-resolving deltas is similar to lifters [36] or derivatives [26] in feature-oriented programming which are used to facilitate the correct interaction between different feature modules.

The definition of a conflict as a lack of commutativity between modifications is also discussed in the context of program refactoring [29]. The underlying formalisation uses graph transformation systems and critical pair analysis. Oldevik et al. [33] define a conflict in a sequence of model transformations to occur if two transformations do not commute. A similar notion of conflict related to non-commutativity is observed by Apel et. al. [5] when two aspects advise shared join points.

8. Conclusion

Delta modeling is an approach to facilitating automated product derivation for software product lines. In this paper, we generalized the conceptual ideas of delta modeling in an abstract, algebraic setting. The main contribution of this work is the novel treatment of conflicts between deltas by explicit conflict-resolving deltas. In order to ensure that for every valid feature configuration a unique product is generated, a conflict-resolving delta has to exist for every pair of conflicting deltas in the model. We presented efficient conditions that allow checking the unambiguity of a product line without requiring to generate all products.

For future work, we will be using the ideas of abstract delta modeling for the implementation of variability within the HATS ABS language [1]. In addition, we are planning to extend abstract delta modeling with a concept of hierarchy so that a delta can itself be a delta model. This will give rise to a more modular development technique for product lines based on nested delta models. Finally, variants of abstract delta modeling, such as basing the framework on partial monoids with a partial composition operation, will be investigated.

References

[1] Highly Adaptable and Trustworthy Software using Formal Methods (HATS), March 2009. `http://www.hats-project.eu`.

[2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *ICSE*, pages 221–231, 2009.

[3] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Software Eng.*, 34(2):162–180, 2008.

[4] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming (SCP)*, 2010. To appear.

[5] S. Apel, C. Kästner, and D.S. Batory. Program refactoring using functional aspects. In *GPCE*, pages 161–170, 2008.

[6] D.S. Batory and S.W. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

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

[8] D.S. Batory and D. Smith. Finite map spaces and quarks: Algebras of program structure. Technical Report TR-07-66, University of Texas at Austin, Dept. of Computer Sciences, 2007.

[9] 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.

[10] G. Bracha. Executable Grammars in Newspeak. *ENTCS*, 193:3–18, 2007.

[11] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. Technical Report CW592, Dept. Computer Sciences, Katholieke Universiteit Leuven, August 2010.

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

[13] 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.

[14] S. Deelstra, M. Sinnema, and J. Bosch. Product Derivation in Software Product Families: A Case Study. *Journal of Systems and Software*, 74(2):173–194, 2005.

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

[16] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *SPLC*, 2008.

[17] F. Heidenreich and C. Wende. Bridging the Gap Between Features and Models. In *Aspect-Oriented Product Line Engineering (AOPLE'07)*, 2007.

[18] P. Heymans, P.Y. Schobbens, J.C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *Software, IET*, 2(3):281–302, 2008.

[19] J. Jacobson. A formalization of Darcs patch theory using inverse semigroups. Technical Report CAM report 09-83, UCLA, 2009.

[20] 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.

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

[22] C. Kästner, S. Apel, and D.S. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232. IEEE, 2007.

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

[24] C. Kästner, S. Apel, S.S. ur Rahman, M. Rosenmüller, D.S. 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.

[25] C.W. Krueger. New Methods in Software Product Line Development. In *SPLC*, pages 95–102, 2006.

[26] J. Liu, D.S. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE*, pages 112–121, 2006.

[27] R.E. Lopez-Herrejon, D.S. Batory, and W.R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[28] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for AOP. In *ICSR*, volume 3107 of *LNCS*, pages 127–140. Springer, 2004.

[29] T. Mens, G. Taentzer, and O. Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electr. Notes Theor. Comput. Sci.*, 127(3):113–128, 2005.

[30] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136. ACM, 2004.

[31] N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *SPLC*, 2008.

[32] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.

[33] J. Oldevik, Ø. Haugen, and B. Møller-Pedersen. Confluence in domain-independent product line transformations. In *FASE*, pages 34–48, 2009.

[34] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling Automation and Flexibility in Product Derivation. In *SPLC*, 2008.

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

[36] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.

[37] S. Trujillo S. Apel, F. Janda and C. Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT)*, 2009.

[38] 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.

[39] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proceedings, 14th International Software Product Line Conference*, Lecture Notes in Computer Science, Jeju, South Korea, 2010. Springer.

[40] 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.

[41] Y. Smaragdakis and D.S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[42] P. Tarr, H. Ossher, W. Harrison, and S.M Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

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

[44] M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC*, pages 233–242, 2007.

[45] H. Zhang and S. Jarzabek. An XVCL-based Approach to Software Product Line Development. In *Software Engineering and Knowledge Engineering*, pages 267–275, 2003.