Math. Struct. in Comp. Science (2015), *vol.* 25, *pp.* 482–527. © Cambridge University Press 2014
doi:10.1017/S0960129512000941 First published online 10 November 2014

Abstract delta modelling

DAVE CLARKE[†], MICHIEL HELVENSTEIJN[‡]
and INA SCHAEFER[§]

[†]*IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium*
Email: dave.clarke@cs.kuleuven.be
[‡]*CWI, Amsterdam and LIACS, Leiden University, The Netherlands*
Email: michiel.helvensteijn@cwi.nl
[§]*University of Braunschweig, Germany*
Email: i.schaefer@tu-braunschweig.de

Received 23 December 2010; revised 9 December 2011

Delta modelling is an approach to facilitate the 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 application conditions over features. This paper presents *abstract delta modelling*, which explores delta modelling from an abstract, algebraic perspective. Compared to the previous work, we take a more flexible approach to conflicts between modifications by introducing the notion of conflict-resolving deltas. Furthermore, we extend our approach to allow the nesting of delta models for increased modularity. We also present conditions on the structure of deltas to ensure unambiguous product generation.

1. Introduction

A *software product line (SPL)* is a set of software systems, called *products*, with well-defined commonalities and variabilities (Clements and Northrop 2001; Pohl *et al.* 2005). SPL engineering aims at developing this set of systems by managed reuse in order to reduce time to market and to increase product quality.

1.1. *Motivation*

Product line variability at the requirements level is predominantly represented by *feature models* (Kang *et al.* 1990; van Deursen and Klint 2002). *Features* are user-visible increments of product functionality (Batory *et al.* 2004). Each product in the product line is identified by a valid *feature configuration*, i.e., a legal combination of features from the feature model. In order to be able to automatically derive a product for a particular feature configuration, some correspondence between the features on the feature modelling level and the reusable product line artefacts has to be introduced. *Delta modelling* (Schaefer 2010; Schaefer *et al.* 2009, 2010) is a modular, yet flexible and expressive modelling approach for expressing product line variability on the artefact level. In the delta modelling approach, a product line is represented by a core product and a set of product deltas. Product deltas specify modifications to the core product required to generate further products of the product line. Each delta has an *application condition*

specifying the feature configurations to which the modifications are applicable, thereby connecting features on the feature modelling level with product line artefacts. A product corresponding to a feature configuration is obtained by applying the deltas with a valid application condition to the core product.

However, the order in which the deltas are applied may influence the resulting composite modification. A *conflict* between deltas arises if their specified modifications do not commute, meaning that if they are applied in different orders they result in different modifications. In previous work, deltas were either considered incomparable (Schaefer 2010), which required writing additional deltas for every conflicting combination, or they had to be ordered in a very restrictive way to avoid conflicts explicitly (Schaefer *et al.* 2010). In this article, we generalize existing delta modelling approaches and present an abstract, algebraic formalization of delta modelling. Abstract delta modelling goes beyond existing work with its novel treatment of conflicts between deltas. As a main contribution of this article, we introduce the notion of conflict-resolving deltas that relaxes previous restrictions and makes delta modelling of product lines even more flexible. A conflict-resolving delta, which is applied after two conflicting deltas, eliminates any conflict that the deltas could introduce. If for every pair of conflicting deltas a conflict-resolving delta exists, all possible sequences of deltas represent the same modification and thus generate a uniquely-defined product. In order to ensure this result for every valid feature configuration, we provide efficient-to-check conditions requiring only the inspection of the product line directly, without having to generate and check all products.

1.2. *Approach*

The abstract delta modelling formalism consists of a number of ingredients, depicted in Figure 1, along with the operations between them; the basic ingredients are a set of possible products, a set of features and a collection of deltas (*delta monoid*). A product line is defined as a feature model denoting the set of possible feature configurations, a core product, the deltas specifying modifications to the core product, a partial order on those deltas restricting their order of application and, lastly, their application conditions. By selecting a particular feature configuration, one can derive a delta model consisting of an ordered collection of the modifications necessary to generate the corresponding product. The process of derivation puts the selected modifications in a linear order compatible with the partial order in order to obtain a valid, ideally composite modification. Finally, the delta application function applies the selected modifications to the core product to produce a product for a given feature configuration. (Delta application can also be partially applied to a modification to produce a function from products to products.)

The concepts of abstract delta modelling can be instantiated for different kinds of development artefacts, such as documentation, models or code. We demonstrate the approach by presenting an instantiation of abstract delta modelling for object-oriented implementations of SPL and extend this with method wrapping, a key operation from aspect-oriented programming (Kiczales *et al.* 1997) and approaches based on step-wise refinement (Batory *et al.* 2004). We demonstrate that existing abstract formalizations of product line composition can be seen as instantiations of abstract delta modelling.

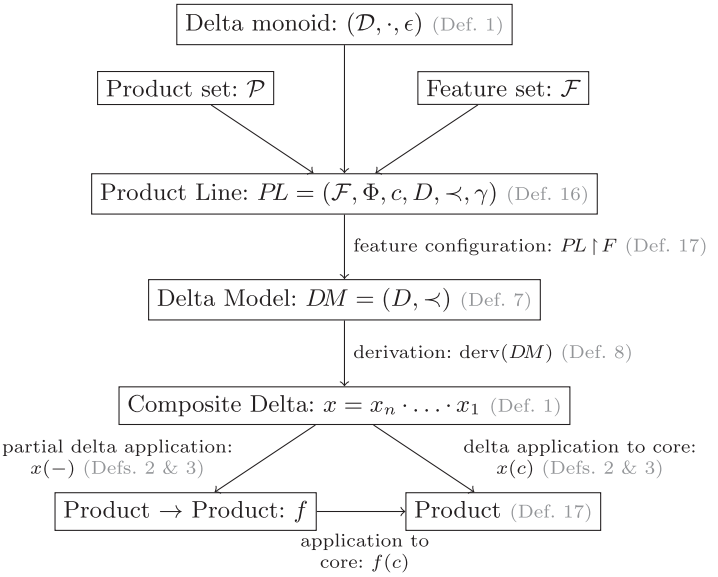


Fig. 1. Relationship between artefacts. Relevant definition numbers are indicated in parentheses.

Furthermore, we extend the formalism with nested delta models as an additional way of imposing structure and increasing modularity. Nested delta models allow a group of deltas to be treated as a single delta. They are processed atomically, avoiding interference with unrelated deltas in the outer model. Thus, nesting can greatly simplify delta models, by allowing modifications to be specified within separate deltas without worrying about interference from other deltas.

This paper is organized as follows. Section 2 gives an overview of delta modelling and describes the Editor product line, which is used as illustrative example in this article. Section 3 presents delta models and criteria for their unambiguity. Section 4 shows how to define product lines over delta models and how to transfer the unambiguity properties to this level. Section 5 presents the concrete class of deltas over object-oriented programs to illustrate the approach. Section 6 introduces nested delta models and explores their expressiveness. Section 7 relates our approach to existing algebraic approaches from the literature. Finally, Sections 8 and 9 present related work and conclusions.

This article extends our previous work on abstract delta modelling (Clarke *et al.* 2010) by providing additional examples, the notion of nested delta model, and a complete formalism including all proofs.

2. Delta modelling overview and running example

In principle, (abstract) delta modelling can be applied to different kinds of development artefacts, such as documentation, models or code. The key objectives for developing product lines using delta modelling are minimizing (code) duplication and unnecessary dependencies between deltas, as well as providing the maximum opportunity for concurrent development of deltas.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

In this article, we illustrate the concepts of delta modelling using object-oriented implementations of SPL, where a delta comprises a set of modifications to an object-oriented program. In this section, we briefly explain object-oriented software deltas. They will be formally introduced in Section 5. A software delta for object-oriented programs can add, remove and modify classes. Class modifications may be subdivided into class updates and class replacements. Updating a class may consist of adding and removing fields and methods, as well as replacing implementations of existing methods. A class replacement is semantically the same as removing an existing class and then adding a new one with the same name.

The base of a SPL developed using delta modelling is a core product. If there is an existing legacy application, this can be chosen as the core product in order to save development effort, following the extractive product line development principles suggested by Krueger (2002). Having a solid core architecture, which is reusable among all products, is a favourable characteristic of a SPL (Pohl *et al.* 2005). Alternatively, if there is no legacy code, the 'empty product' could be chosen as core product and all functionality could be introduced by deltas. This has advantages for the evolution of the product line, as pointed out by Schaefer and Damiani (2010), for example, when mandatory features become optional.

One approach to developing the deltas that realize the different products of a product line is to use the product line's feature model as a template. More specifically, its subfeature relation. We could specify exactly one delta d_f for each feature f, with the requirement that d_f must be applied before d_g, denoted by $d_f \prec d_g$, whenever g is a subfeature of f. In this way, the structure of the delta model mimics the structure of the feature model. Each of these deltas only implements its corresponding feature and can be developed without taking possible conflicts into account.

Implementation conflicts between deltas can certainly occur, however in our case, this happens when two unordered deltas manipulate the same program entity in different ways. For each implementation conflict, a conflict-resolving delta has to be provided to ensure that a unique product implementation can be generated for each feature configuration. The application condition of a conflict-resolving delta is generally the intersection of the application conditions of the conflicting deltas. Furthermore, the conflict-resolving delta has to be later in the application order $\prec$ than both conflicting deltas. In general, deltas placed later in the application order can also be used to combine the functionality of other deltas, for example, to implement desired interactions between features.

2.1. *Editor feature model*

We now introduce a product line of code editors that will be used as an illustrative example throughout this article. First, we describe the features of the Editor product line, and then the software deltas providing the implementation.

The *Editor product line* consists of a set of valid feature configurations corresponding to different editor instantiations. Figure 2 depicts the feature model of the Editor product line. In the diagram, every box represents a feature. A line with an open circle at the end stands for the optional subfeature relation. A horizontal line segment indicates exclusive

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

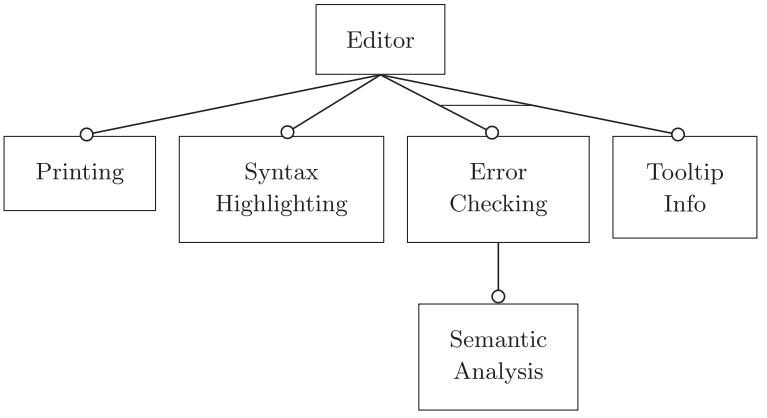


Fig. 2. Feature model of the Editor product line.

choice between subfeatures. A subfeature can only be selected if its parent feature is also selected. The Editor product line has the following features.

— *Editor* (*Ed*) is the only mandatory feature of the product line representing basic editing functionality.
— *Printing* (*Pr*) allows the user to print the content of an editor window.
— *Syntax highlighting* (*SH*) colours code for easier recognition of programming language constructs.
— *Error checking* (*EC*) performs simple grammatical analysis on code and underlines certain errors. Hovering over an error gives extra information in a tooltip.
— The optional subfeature *semantic analysis* (*SA*) of the feature *error checking* performs more sophisticated error analysis of program code.
— *Tooltip info* (*TI*) gives information about code fragments by hovering over them. The features *tooltip info* and *error checking* are mutually exclusive, since both produce different kinds of tooltips.

This product line consists of 16 different editors, as there are 16 possible feature configurations.

2.2. *Software deltas*

Firstly, the Editor product line uses the empty program as the core product – the product line is implemented entirely by deltas.

Figure 3 shows the deltas of the Editor product line using a UML-like notation. The dashed boxes represent software deltas; they are named using a label in their top-left corner and contain class replacements and class updates. Method implementations are denoted using an italic capital letter. For instance, delta d_1 provides the basic functionality of an editor by adding a new class `Editor`. It is declared as a replacement (as opposed to an update) using the annotation (r). The class contains a field and three methods. All other deltas update this class (indicated by the annotation (u)) by adding and modifying methods. For instance, delta d_2 adds the `print` method, and delta d_3 modifies the

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

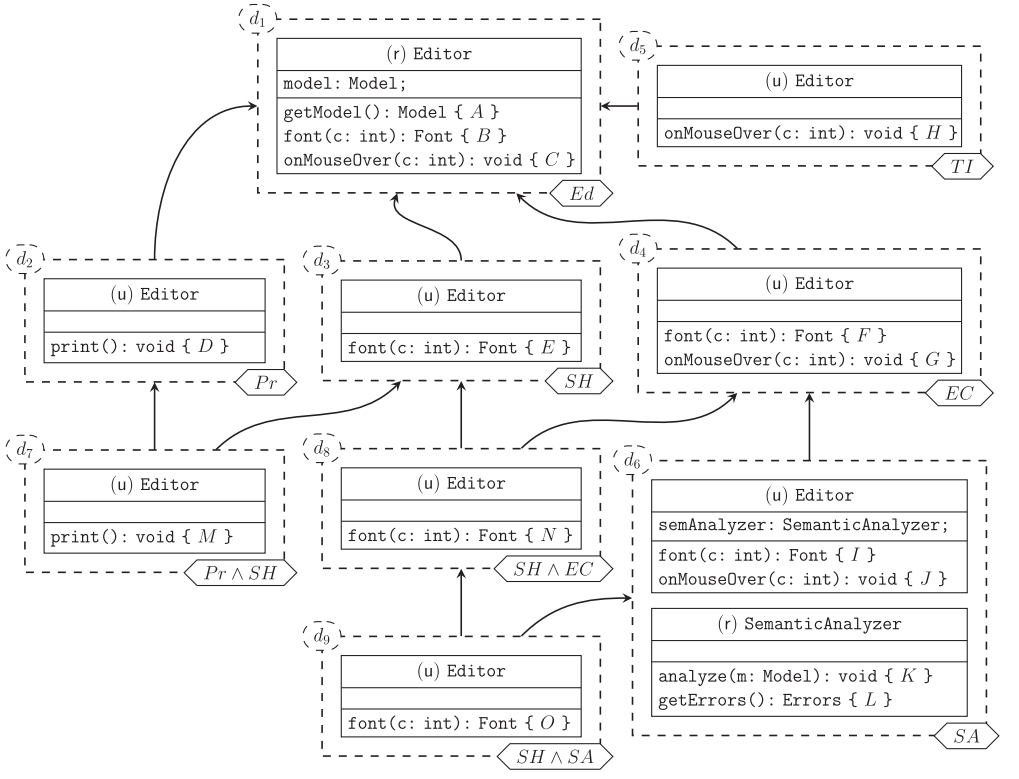


Fig. 3. Graphical representation of the delta model for the Editor product line.

implementation of `font` with a new method body E. Besides updating the `Editor` class, delta d_6 also adds a new class of its own.

The deltas are decorated with their application conditions (bottom right of each box), which are propositional logic formulas linking the delta to the set of feature configurations for which it is applicable. The arrows between the deltas represent the application order – deltas in the order are applied later. By not placing an order between two deltas, the designer indicates that the order in which the deltas are applied should not matter. Delta d_1 is applicable in any valid feature configuration, because it is annotated only with the mandatory Ed feature. Deltas $d_2, \ldots, d_5$ implement the four optional features of the product line, Pr, SH, EC and TI. They are applied whenever their corresponding feature is selected. All of those deltas could be developed concurrently without considering potential conflicts. Delta d_6 implements the subfeature SA of the feature EC. It is applied whenever the SA feature is selected. (It also requires that feature EC is selected, but this is omitted as it is implied by the feature model.)

Deltas d_3 and d_4 are in conflict, because they both redefine the `font` method independently (the two deltas are not ordered). This conflict is resolved by conflict-resolving delta d_8. Delta d_8 is selected only if the two conflicting deltas are also selected and applied later in the order to resolve their conflict and provide the appropriate semantics for the combination of the SH and EC features. The similar conflict between deltas d_3/d_8 and

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

d_6 is resolved by conflict-resolving delta d_9. Note that a conflict-resolving delta for the apparent `onMouseOver` conflict between deltas d_4 and d_5 is not required, because the *EC* and *TI* features are mutually exclusive in the feature model; thus the two deltas are never applied together, and so there is no conflict to resolve.

Delta d_7 appears later in the order than d_2 (printing) and d_3 (syntax highlighting). When both deltas d_2 and d_3 are selected, namely, when $Pr \wedge SH$ holds, delta d_7 implements the desired interaction between these two features, providing printouts containing syntax highlighting, by overwriting the `print` method and using the syntax highlighting information of delta d_3.

3. Abstract delta modelling

This section presents the core ingredients of abstract delta modelling. Product modifications and their composition will form a monoid, called a *delta monoid*. Delta models are built on top of this monoid as partially ordered collections of modifications, where the order constrains the possible ways in which those modifications can be applied. It is important that a delta model defines unambiguous modifications in order to obtain a distinct product. Thus to formalise the notion of an ambiguous delta model and the resolution of the ambiguity, we define the notions of conflict and conflict-resolving deltas. In addition, we develop conditions to ensure that a delta model is unambiguous.

3.1. *Products and deltas*

In existing compositional approaches to implementing SPL, such as feature-oriented programming (Batory *et al.* 2004) and delta-oriented programming (Schaefer *et al.* 2010), a member product of an SPL is obtained by the application of a number of modifications $x_1, \ldots, x_n$ to a core product c, denoted as $x_n(\cdots x_1(c)\cdots)$. In feature-oriented programming, the core product is determined by one or more base modules. The modifications are specified by feature modules that extend and refine the core product. In delta-oriented programming, the core product is typically some valid product of the product line, which deltas subsequently refine. As both approaches treat the core product as a constant element, which generally contains the bulk of the code of the product line, it is useful to focus on the modifications as sequences such as $x_n \cdot \ldots \cdot x_1$, where $\cdot$ is sequential composition. The above product then becomes $(x_n \cdot \ldots \cdot x_1)(c)$. It may still be possible to reason about the core product if we choose to see it as a modification x_c applied to some initial (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 usually nothing is lost by restricting our attention to modifications.

In abstract delta modelling, the main object of interest is the *delta monoid*. A delta monoid is a monoid of modifications $\mathcal{D}$, called *deltas*, that act (on the left) on the set of products $\mathcal{P}$. A delta monoid 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 modelling or programming languages. The example in Section 2 considers

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

modifications to object-oriented programs by adding, removing and modifying classes, methods and fields.

Firstly, we assume a set of *products*, $\mathcal{P}$. The set of possible modifications to products forms a delta monoid, as follows.

Definition 1 (delta monoid). A *delta monoid* is a monoid $(\mathcal{D}, \cdot, \epsilon)$, where $\mathcal{D}$ is a set of product 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 modify nothing.

The operation $\cdot$ is associative, though not inherently commutative, as the order between two deltas may be significant. Two deltas $x, y \in \mathcal{D}$ are said to be *non-commutative* if $y \cdot x \neq x \cdot y$. This notion will later form the basis of our definition of conflict.

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

Definition 2 (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.

Delta application will often satisfy the stronger property of being a monoid action.

Definition 3 (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 $p \in \mathcal{P}$.

Given a delta action, a notion of equivalence can be defined on deltas, namely that for $x, y \in \mathcal{D}$, $x \equiv y$ iff for all $p \in \mathcal{P} : x(p) = y(p)$. A more precise notion of non-commutativity would be that $x \cdot y \not\equiv y \cdot x$. Based upon this notion of equivalence, the quotient delta monoid $(\mathcal{D}/\equiv)$, defined in the usual fashion, could be used in place of $\mathcal{D}$.

Definition 4 (deltoid). A *deltoid* is a 5-tuple $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$, where $\mathcal{P}$ is a product set, $(\mathcal{D}, \cdot, \epsilon)$ is a delta monoid and $-(-)$ is a delta application operator.

A deltoid completely defines a concrete domain and abstraction level for product lines.

3.2. *Notions of expressiveness*

The expressiveness of a particular deltoid can be characterised in terms of the existence of an element in the product set from which all products can be generated – called an *initial product* – and the ability to transform each product in the product set into every other product via the application of deltas – called *maximal expressiveness.*

Definition 5 (initial product). Given a deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$, a product $p \in \mathcal{P}$ is an *initial product* iff $\forall p' \in \mathcal{P} : \exists x \in \mathcal{D} : x(p) = p'$.

Definition 6 (maximal expressivity). A deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$ is said to be *maximally expressive* iff $\forall p, p' \in \mathcal{P} : \exists x \in \mathcal{D} : x(p) = p'$.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Note that for a maximally expressive deltoid, every product is an initial product.

Some formalisms have no notion of initial product. In aspect-oriented programming (Kiczales *et al.* 1997), for example, where advice is woven in at pointcuts which have to be already defined in the existing classes of a base program, advice can only add statements before, after or around existing statements identified by the pointcut. However, advice can neither add new classes nor remove them. A delta monoid that is able to add but not to remove elements will not be maximally expressive. The OOP and AOP deltoids presented in Section 5 are both maximally expressive, as they have these capabilities.

3.3. *Delta models*

A delta model describes the set of deltas required to build a specific product, along with a strict partial order on those deltas, restricting the order in which they may be applied. For example, Figure 3 showed the base delta model for the Editor product line.

Definition 7 (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 the deltas captures the intuition that a subsequent delta has full knowledge of (and access to) earlier deltas and more authority over modifications to the product. The possibility of using a partial order, rather than a total order or unordered set of deltas, is important for expressing certain design intentions regarding the interdependency of deltas, as well as to avoid and resolve conflicts (Section 3.4).

The semantics of a delta model is defined by its derivations. A *derivation* is a delta formed by a sequential composition of all deltas from D, respecting the partial order.

Definition 8 (derivation). Given a delta model $\mathrm{DM} = (D, \prec)$, its *derivations* are defined to be

$$\mathrm{derv}(\mathrm{DM}) \stackrel{\text{def}}{=} \left\{ x_n \cdot \ldots \cdot x_1 \mid x_1, \ldots, x_n \text{ is a linear extension}^{\ddagger} \text{ of } (D, \prec) \right\}.$$

Observe that when D is empty, $\mathrm{derv}\,\mathrm{DM} = \{\epsilon\}$. Also note that $\mathrm{derv}\,\mathrm{DM}$ may potentially generate more than one distinct derivation, as non-commutative deltas may be applied in different orders. However, it is desirable that all derivations of a delta model have the same effect, as this corresponds to generating a unique product. This motivates the following definition.

Definition 9 (unique derivation). A delta model $\mathrm{DM} = (D, \prec)$ 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 $(D, \prec)$, or, equivalently, iff $|\,\mathrm{derv}(\mathrm{DM})| = 1$.

[†] Recall that a *strict partial order* is irreflexive, asymmetric and transitive.
[‡] Recall that a *linear extension* is a total order compatible with the partial order.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

3.4. *Unambiguity of delta models*

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

Two deltas in a delta model are in conflict if they are non-commutative and no order is placed upon them. Generally, two conflicting deltas are independently modifying the same part of a product in different ways, meaning that multiple distinct derivations may be possible. For example, in Figure 3 deltas d_3 and d_4 are in conflict, since they both redefine the `font` method in different ways.

Definition 10 (conflict). Given a delta model $\mathrm{DM} = (D, \prec)$, $x, y \in D$ are said to be *in conflict*, denoted $x \lightning y$, 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 order between non-commutative deltas (Schaefer *et al.* 2010). However, features with conflicting implementations often correspond to independent concepts. Kästner *et al.* (2009) call the issue of modelling such situations the *optional feature problem*. Imposing an order on the deltas of conceptually orthogonal features is often inappropriate, as some (unrelated) functionality may be inadvertently and silently overwritten. Furthermore, sometimes neither of the original choices in functionality is exactly what is required, and instead some combination has to be used.

The alternative is to allow conflicts, but to provide additional, subsequently-applied deltas to resolve them. In Figure 3, the conflict-resolving delta for the conflict between deltas d_3 and d_4 is delta d_8, which combines the functionality introduced by d_3 and d_4.

Definition 11 (conflict-resolving delta). Given a delta model $\mathrm{DM} = (D, \prec)$ 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:

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

in which D^* denotes the sequences of compositions of deltas from D.

Conflict-resolving deltas take the role of *derivative modules* (Kästner *et al.* 2009) or *lifters* (Prehofer 1997). An unambiguous delta model is a delta model containing a conflict-resolving delta for every conflicting pair of deltas.

Definition 12 (unambiguous delta model). Given a delta model $(D, \prec)$, we say that the model is *unambiguous* iff

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

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

If we ignore the application conditions in Figure 3, the underlying delta model of the presented product line is not unambiguous, since d_4 and d_5 are in conflict without a conflict-resolving delta. However, in Section 4 we explain that we never take derivations directly from the underlying model, but first apply a feature selection, by which certain deltas are filtered out. Since the two deltas in question have mutually exclusive application conditions, they are never applied together and every selected delta model (Definition 17) for the Editor product line *will* be unambiguous.

If a delta model is unambiguous, it can be shown to have a unique derivation. In order to prove this, some intermediate results are required. 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 $\mathrm{DM} = (D, \prec)$ and $d_2 \cdot y \cdot x \cdot d_1 \in \mathrm{derv}(\mathrm{DM})$, where $x, y \in D$ and $d_1, d_2 \in D^*$. Then either $x \prec y$ or $d_2 \cdot y \cdot x \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$.

Proof. By case distinction on the unambiguity of DM for deltas x and y:

— Case $y \cdot x = x \cdot y$. It follows directly that $d_2 \cdot y \cdot x \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$.
— Case $x \prec y$. Immediate.
— Case $y \prec x$. Cannot happen, as $d_2 \cdot y \cdot x \cdot d_1$ is a linear extension of $\prec$.
— Case $\exists z \in D : (x, y) \lhd z$. Firstly, from the definition of conflict-resolving delta we have that $x, y \prec z$, hence there exist $d_2', d_2'' \in D^*$ such that $d_2 \cdot y \cdot x \cdot d_1 = d_2' \cdot z \cdot d_2'' \cdot y \cdot x \cdot d_1$. From the remaining condition on z, we have $z \cdot d_2'' \cdot y \cdot x = z \cdot d_2'' \cdot x \cdot y$, from which we deduce $d_2 \cdot y \cdot x \cdot d_1 = d_2' \cdot z \cdot d_2'' \cdot y \cdot x \cdot d_1 = d_2' \cdot z \cdot d_2'' \cdot x \cdot y \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$. □

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

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

Proof. If $(D, \prec)$ is unambiguous, then $\forall x, y \in D : x \not\prec y \Rightarrow \exists z \in D : (x, y) \lhd z$. For this to be true in $D \setminus \{w\}$ we need to show that there are no x and y such that $x \not\prec y$ with w such that $(x, y) \lhd w$. But w could not have been a conflict resolver, as it is minimal, contradicting conditions $x \prec w$ and $y \prec w$ of $(x, y) \lhd w$. □

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

Lemma 3. Given an unambiguous delta model $\mathrm{DM} = (D, \prec)$. Let $x_n \cdot \ldots \cdot x_1 \in \mathrm{derv}(\mathrm{DM})$, where $\{x_1, \ldots, x_n\} = D$, with x_i minimal in $\prec$. Then

$$x_n \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i+1} \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_i.$$

Proof. By induction on i:

— Case $i = 1$. Immediate.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

— Case $i > 1$. As x_i is minimal, $x_{i-1} \not\prec x_i$ holds. Lemma 1 implies that $x_n \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i+1} \cdot x_{i-1} \cdot x_i \cdot x_{i-2} \cdot \ldots \cdot x_1$. Now x_i is in position $i-1$, so induction gives that $x_n \cdot \ldots \cdot x_{i+1} \cdot x_{i-1} \cdot x_i \cdot x_{i-2} \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i+1} \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_i$. □

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 as checking that all conflicts between pairs of deltas are eliminated by conflict-resolving deltas.

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

Proof. Given unambiguous delta model $\mathrm{DM} = (D, \prec)$. Proceed by induction on the size of D:

— Case $|D| = 0$. Immediate as $\mathrm{derv}(\mathrm{DM}) = \{\epsilon\}$.
— Case $|D| = 1$. Immediate as $\mathrm{derv}(\mathrm{DM}) = \{x\}$, where $D = \{x\}$.
— Case $|D| > 1$. For any two $d_1, d_2 \in \mathrm{derv}(\mathrm{DM})$, let $d_1 = d_1' \cdot x$ and $d_2 = d_2' \cdot x \cdot d_2''$, where $x \in D$ and $d_1', d_2', d_2'' \in D^*$. As x is the last element of d_1, it must be minimal in $\prec$. Thus, by Lemma 3, $d_2' \cdot x \cdot d_2'' = d_2' \cdot d_2'' \cdot x$. Now by Lemma 2, $\mathrm{DM}' = (D \setminus \{x\}, \prec')$, where $\prec'$ is $\prec$ restricted to $D \setminus \{x\}$, is an unambiguous delta model, and $d_1',\ d_2' \cdot d_2'' \in \mathrm{derv}(\mathrm{DM}')$. By the induction hypothesis, $d_1' = d_2' \cdot d_2''$ and thus $d_1 = d_1' \cdot x = d_2' \cdot d_2'' \cdot x = d_2' \cdot x \cdot d_2'' = d_2$. Hence, $|\mathrm{derv}(\mathrm{DM})| = 1$. □

3.5. *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 11) 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. Naturally, we could restrict the checks to consider only relevant elements of D^*, but instead in this section, we propose a simpler criterion to make checking ambiguity more feasible for interesting classes of deltoids. The *consistent conflict resolution* property states that if a delta z resolves an (x, y)-conflict when applied directly after x and y, it also resolves the conflict after the application of any sequence of intermediate deltas.

Definition 13 (consistent conflict resolution). A delta monoid $(\mathcal{D}, \cdot, \epsilon)$ is said to exhibit *consistent conflict resolution* iff the following condition holds:

$$\forall x, y, z \in \mathcal{D} : z \cdot y \cdot x = z \cdot x \cdot y \Rightarrow \forall d \in \mathcal{D} : z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y.$$

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

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

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

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

Theorem 2. Given a delta model $\mathrm{DM} = (D, \prec)$ exhibiting consistent conflict resolution. For all deltas $x, y \in D$ which are in conflict and $z \in D$ such that $x \prec z$ and $y \prec z$, if $z \cdot y \cdot x = z \cdot x \cdot y$, then $(x, y) \lhd z$.

Proof. From the definition of consistent conflict resolution, we have the following:

$$\begin{aligned} z \cdot y \cdot x = z \cdot x \cdot y &\Rightarrow \forall d \in \mathcal{D} : z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y \\ &\Rightarrow \forall d \in D^* : z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y. \end{aligned}$$

Combined with the facts $x \prec z$ and $y \prec z$, this is precisely the definition of $(x, y) \lhd z$. □

4. Product lines

Using the introduced concepts of delta models, products and delta application, we can now abstractly define product lines. We extend the concept of unambiguity to the level of product lines and provide an efficient condition to efficiently check it.

4.1. *Defining product lines*

A product line is organised around a set of *features*. Let $\mathcal{F}$ denote a set of features, which are merely labels without any inherent meaning.

At the highest level of abstraction, the set of products in a product line is represented by a feature model. Many formal descriptions (Heymans *et al.* 2008; Kang *et al.* 1990; van Deursen and Klint 2002) agree that a feature model determines a set of valid feature configurations.

Definition 14 (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*.

For example, the Editor product line from Section 2 is described by the feature model depicted in Figure 2. The set of valid feature configurations is the following:

$$\Phi = \left\{ \begin{array}{l} \{Ed, Pr\}, \{Ed, Pr, SH\}, \{Ed, Pr, SH, EC\}, \{Ed, Pr, SH, EC, SA\}, \\ \{Ed, Pr, SH, TI\}, \{Ed, Pr, EC\}, \{Ed, Pr, EC, SA\}, \{Ed, Pr, TI\}, \\ \{Ed, SH\}, \{Ed, SH, EC\}, \{Ed, SH, EC, SA\}, \{Ed, SH, TI\}, \\ \{Ed, EC\}, \{Ed, EC, SA\}, \{Ed, TI\}, \{Ed\} \end{array} \right\}.$$

For the sake of simplicity, our formalism does not consider extended feature models, such as cardinality-based or attributed feature models (Czarnecki and Kim 2005; Czarnecki *et al.* 2004), though we believe that they can be handled in a straightforward fashion.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

In order to bridge the gap between features and product line artefacts, application conditions are introduced. An application condition is associated with a delta and determines the feature configurations to which the delta is applicable.

Definition 15 (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. The set $\gamma(x)$ is called the *application condition* for delta x.

The application conditions of the Figure 3 deltas are shown in the form of propositional logic formulae, in which the atomic formulas are feature names.

A product line is defined by its feature model, describing the set of valid feature configurations, a core product, a base delta model, containing the modifications used to obtain further products, and an application function, associating features and deltas.

Definition 16 (product line). A *product line* is a tuple $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$, where

— $\mathcal{F}$ is a feature set,
— $\Phi \subseteq \mathscr{P}(\mathcal{F})$ 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 $\forall x \in D : \gamma(x) \subseteq \Phi$.

If feature configuration F is valid according to Φ, its corresponding product is defined using the delta model containing only the deltas applicable to F.

Definition 17 (selected delta model). Given a product line $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$, the *selected delta model* for feature configuration $F \in \Phi$, denoted $PL \upharpoonright F$, is the delta model $(D_F, \prec_F)$ where $D_F = \{ d \in D \mid F \in \gamma(d) \}$ is the set of applicable deltas, and $\prec_F$ is $\prec$ restricted to D_F.

The derivation(s) of this delta model will be applied to the core product to generate the product(s) corresponding to feature configuration F. The set of products generated from a product line given a feature configuration is defined as follows.

Definition 18 (generated products). Given a product line $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$, the set of generated products for feature configuration $F \in \Phi$ is defined as:

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

Note that the above definition of 'product' slightly deviates from existing literature, where a product is uniquely defined for a given feature configuration. Our 'product' refers to a specific implementation. There may be ambiguity in the derivation process resulting in more than one implementation per feature configuration. Having a unique implementation for a given feature configuration is a property we strive for in Section 4.2.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

4.2. *Unambiguity of product lines*

As argued in Section 3, unambiguity of delta models is a desired property because it ensures a 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 implementation.

Definition 19 (unambiguous product line). A product line $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$ 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 the unambiguity using the criteria proposed in Section 3. However, as the set of feature configurations is often exponential in the number of features, this naive approach would be rather expensive. Instead, we propose the notion of a globally unambiguous product line which implies product line unambiguity. First, we introduce a shorthand notation for the set of feature configurations for which two deltas x and y are applicable.

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

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

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. 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 20 (globally unambiguous product line). A product line $(\mathcal{F}, \Phi, c, D, \prec, \gamma)$ is called *globally unambiguous* if and only if for all $x, y \in D$ such that $x \lightning y$ and $\mathcal{V}^{x,y} \neq \emptyset$, there exists a $z \in D$ such that $\mathcal{V}^{x,y} \subseteq \gamma(z)$ and $(x, y) \lhd z$.

The following theorem states that any globally unambiguous product line is also an unambiguous product line. In the following proof, we annotate the $\lightning$ and $\lhd$ operators with the delta model for which they apply, e.g., $x \lightning_{\text{DM}} y$ and $(x, y) \lhd_{\text{DM}} z$.

Theorem 3. A globally unambiguous product line is unambiguous.

Proof. Assume $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$ is a globally unambiguous product line. Let $F \in \Phi$ be a valid feature configuration. We show that $\text{DM} = PL \restriction F = (D_F, \prec_F)$ is an unambiguous delta model.

Given arbitrary $x, y \in D_F$ (so also $x, y \in D$), perform a case analysis on the global unambiguity of PL (Definition 20). Observe that $F \in \mathcal{V}^{x,y}$, otherwise x and y would not be in D_F. Now for the cases:

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

— Case $\neg(x \not\frown_{(D,\prec)} y)$. So, also $\neg(x \not\frown_{\mathrm{DM}} y)$.
— Case $\mathcal{V}^{x,y} = \emptyset$. Cannot happen, otherwise $x, y \notin D_F$.
— Case $\exists z \in D : \mathcal{V}^{x,y} \subseteq \gamma(z) \wedge (x, y) \lhd_{(D,\prec)} z$. Note that $z \in D_F$, as $\mathcal{V}^{x,y} \subseteq \gamma(z)$. It follows that $(x, y) \lhd_{\mathrm{DM}} z$ (Definition 17).

So, for all $x, y \in D_F$, either $\neg(x \not\frown_{\mathrm{DM}} y)$ or $\exists z \in D_F : (x, y) \lhd_{\mathrm{DM}} z$. Hence, DM is an unambiguous delta model (Definition 12). □

A product line can be unambiguous without being globally unambiguous if conflicts between two deltas x and y are resolved by different conflict-resolving deltas z for different feature configurations.

5. A deltoid for object-oriented programs

This section presents a concrete deltoid for object-oriented programs to demonstrate our approach. 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. In this section, we abstract away from issues such as well-typedness. For more details on how to compositionally type check product lines, the reader is referred to Schaefer *et al.* (2011).

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

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

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

The deltas presented in this section will be based on partial functions. $\bot$ will denote that the function is *not defined* for a particular element, whereas $-$ denotes the *removal* of the element.

5.1. *Software products*

For simplicity, we abstract from a concrete programming language, as well as from concrete implementations of methods, and focus on the structural aspects of object-oriented programs. First we introduce $\mathcal{I}_c$ and $\mathcal{I}_m$, denoting sets of *identifiers* used for naming classes and methods/fields respectively, and $\mathcal{M}$, a set of *method and field definitions*.

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

Definition 21 (class definitions). The collection of *class definitions* is the set of partial functions $\Psi = \mathcal{I}_m \rightharpoonup \mathcal{M}$. One such 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. Italic capital letters refer to method implementations,

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

where different letters represent distinct implementations.

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

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

Definition 22 (Programs). We define the set of products $\mathcal{P}$ as the set of *programs* in an object-oriented language: $\mathcal{P} = \mathcal{I}_c \rightharpoonup \Psi$.

As an example, consider the following program definition:

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

5.2. *OOP software deltas*

OOP 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 entirely. 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 (i.e. deletes the old one and puts a new one in its place). 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 an OOP software delta captures this set of program modifications. In contrast to previous work (Schaefer *et al.* 2010), the removal of an element in this concrete deltoid does not require that the element is already present, nor does addition require its absence. This ensures that every derivation of deltas is well-defined.

Definition 23 (OOP software deltas). The set of *software deltas* is defined as

$$\mathcal{D} = \mathcal{I}_c \rightharpoonup (\{\mathsf{r}\} \times (\mathcal{I}_m \rightharpoonup \mathcal{M}) \ \cup \ \{\mathsf{u}\} \times (\mathcal{I}_m \rightharpoonup \mathcal{M}^-))^-.$$

r and u represent 'replace' and 'update', respectively. Mapping an identifier to $-$ indicates removal from the product. $\epsilon = \lambda x.\bot$ is the empty delta, modifying nothing.

On the top level, Definition 23 defines a delta as a partial function from identifiers representing class names to class-level operations, which are class removals ($-$), class replacements (tagged with r) or class updates (tagged with u). Class replacements are partial functions from method and field name identifiers to method and field definitions. Class updates are similar, but also allow methods and fields to be removed ($-$).

The following example of an OOP software delta contains three class-level operations: class C is updated by removing field/method f and adding method z and field i; class D

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

is removed and class `E` is replaced with a new class.

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

Applying this delta to the example program defined at the end of Section 5.1 results in the following (formally defined in Definition 25):

$$\left\{\begin{array}{l} \texttt{C} \mapsto \left\{\begin{array}{l}\texttt{z} \mapsto\ \texttt{z(): void \{ } D \texttt{ \},}\\ \texttt{g} \mapsto\ \texttt{g(): bool \{ } B \texttt{ \},}\\ \texttt{i} \mapsto\ \texttt{i: float}\end{array}\right\},\\ \texttt{E} \mapsto\ \left\{\texttt{b} \mapsto\ \texttt{b: bool}\right\}\end{array}\right\}.$$

Now, we introduce some notation required for the next few definitions.

Notation. Given a binary operation $\bigcirc : S \times S' \to S''$ for some sets S, S', S'' that contain $\bot$, and some set of identifiers $\mathcal{I}$, the derived operation $\overline{\bigcirc} : (\mathcal{I} \to S) \times (\mathcal{I} \to S') \to (\mathcal{I} \to S'')$ applies $\bigcirc$ to the codomain of two partial functions. For all $i \in \mathcal{I}$, $a : \mathcal{I} \to S$ and $b : \mathcal{I} \to S'$:

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

When a class update is applied where no class is present or when a class update is composed with a class removal, any method and field removals need to be discarded from the resulting class update, as they no longer make sense. The following notation captures this.

Notation. Given a class update $f : \mathcal{I}_m \rightharpoonup \mathcal{M}^-$, define $f^* : \mathcal{I}_m \rightharpoonup \mathcal{M}$ as f without any method or field removals. For all $i \in \mathcal{I}_m$:

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

Now we define the sequential composition of software deltas $y \cdot x = y \,\overline{\oplus_C}\, x$, which combines class-level modifications to produce composite modifications. This operation depends on operation $\overline{\oplus_M}$, defined below, for combining method definitions.

Definition 24 (sequential composition of OOP software deltas). The *sequential composition of OOP software deltas* $\cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is defined as

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

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

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

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

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 $\oplus_C$ and $\oplus_M$ operators formalize the intuition that their left-hand operand has priority over their right-hand operand, since it represents the delta that is applied later. $-$, $\mathsf{r}\ g$ and m act as left-zero elements in these operations, i.e. they completely overwrite the right-hand operand. $\bot$ is a neutral element. $\mathsf{u}\ e$ represents a class update, which indicates that the left-hand operand only overwrites the right-hand at the method-level, not the class level.

The options for combining methods are limited here, since $\oplus_M$ only supports removing and replacing methods. However, Section 5.3 will redefine $\oplus_M$ to allow method wrapping.

We now illustrate two of the more complicated cases from the tables above. Say we have two class operations, both updates:

$$\mathsf{u}\ h \stackrel{\text{def}}{=} \mathsf{u} \left\{ \begin{array}{lll} \texttt{x} & \mapsto & \texttt{x(): void \{ } A \texttt{ \},} \\ \texttt{y} & \mapsto & -, \\ \texttt{z} & \mapsto & \texttt{z: int} \end{array} \right\} \qquad \mathsf{u}\ e \stackrel{\text{def}}{=} \mathsf{u} \left\{ \begin{array}{lll} \texttt{w} & \mapsto & \texttt{w(): bool \{ } B \texttt{ \},} \\ \texttt{x} & \mapsto & -, \\ \texttt{z} & \mapsto & \texttt{z: float} \end{array} \right\}.$$

The composition of these two deltas, according to Definition 24, is another class update, which applies first $\mathsf{u}\ h$ and then $\mathsf{u}\ e$:

$$\mathsf{u}\ e \oplus_C \mathsf{u}\ h \;=\; \mathsf{u}\ (e \overline{\oplus_M} h) \;=\; \mathsf{u} \left\{ \begin{array}{lll} \texttt{w} & \mapsto & \texttt{w(): bool \{ } B \texttt{ \},} \\ \texttt{x} & \mapsto & -, \\ \texttt{y} & \mapsto & -, \\ \texttt{z} & \mapsto & \texttt{z: float} \end{array} \right\}.$$

Another interesting case appears when a class update is composed with a class replacement. Take the following class replacement:

$$\mathsf{r}\ f \stackrel{\text{def}}{=} \mathsf{r} \left\{ \begin{array}{lll} \texttt{x} & \mapsto & \texttt{x(): void \{ } A \texttt{ \},} \\ \texttt{z} & \mapsto & \texttt{z: int} \end{array} \right\}.$$

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Being a replacement, it cannot contain method or field removals, since they would not make sense. Its composition with `u` e would also be a class replacement:

$$\texttt{u}\ e \oplus_C \texttt{r}\ f \;=\; \texttt{r}\ (e \mathbin{\overline{\oplus_M}} f)^* \;=\; \texttt{r} \left\{ \begin{array}{ll} \texttt{w} \mapsto & \texttt{w(): bool \{ } B \texttt{ \}}, \\ \texttt{z} \mapsto & \texttt{z: float} \end{array} \right\}.$$

Note that the * operator discards (`x` $\mapsto$ $-$) from the result. As we are replacing an entire class, there will be no `x` to remove.

Definition 24 makes the notion of non-commutativity concrete: two OOP software deltas are non-commutative if they map the same identifier to two different definitions, that is, if they modify the same field, method or class in incompatible ways.

OOP software deltas satisfy the following properties. The proofs of Lemmas 4–6 are subsumed by the proofs of analogous properties of AOP software deltas (Lemmas 8–10), as they are a conservative extension of OOP software deltas. These proofs appear in the appendix.

Lemma 4. OOP software deltas are a delta monoid (cf. Definition 1).

Lemma 5. OOP software deltas exhibit consistent conflict resolution (cf. Definition 13).

With this result, it is easy to verify that the Editor product line from Section 2 is globally unambiguous. There are three pairs of deltas in conflict: $d_3 \lightning d_4$, $d_3 \lightning d_6$ and $d_8 \lightning d_6$. The first conflict is resolved by d_8. The second and third conflict have the same resolver d_9. By the choice of γ, the conflict-resolving delta is present in each feature configuration in which the conflicting deltas appear, and if it is applied directly after the conflicting deltas, it makes them commute again. By Lemma 5, these conditions are enough to ensure global unambiguity.

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

Definition 25 (OOP software delta application). Given delta $x \in \mathcal{D}$ and product $p \in \mathcal{P}$, *OOP software delta application* is an operation $-(-) : \mathcal{D} \times \mathcal{P} \to \mathcal{P}$ defined as follows:

$$x(p) \;\stackrel{\text{def}}{=}\; x \mathbin{\overline{\odot_C}} p,$$

where the operators $\odot_C$, with $e : \mathcal{I}_m \rightharpoonup \mathcal{M}^-$ and $g, h : \mathcal{I}_m \rightharpoonup \mathcal{M}$, and $\odot_M$, with $m, n \in \mathcal{M}$, are defined as

$\odot_C$	$\bot$	h
$\bot$	$\bot$	h
$-$	$\bot$	$\bot$
`u` e	e^*	$e \mathbin{\overline{\odot_M}} h$
`r` g	g	g

$\odot_M$	$\bot$	n
$\bot$	$\bot$	n
$-$	$\bot$	$\bot$
m	m	m.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

For example, say we have a product containing class definition h and a delta containing class update $\mathtt{u}\ e$:

$$h \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \mathtt{x} \mapsto & \mathtt{x(): void \{}\ A\ \mathtt{\},} \\ \mathtt{z} \mapsto & \mathtt{z: int} \end{array} \right\} \qquad \mathtt{u}\ e \stackrel{\text{def}}{=} \mathtt{u} \left\{ \begin{array}{ll} \mathtt{w} \mapsto & \mathtt{w(): bool \{}\ B\ \mathtt{\},} \\ \mathtt{x} \mapsto & -, \\ \mathtt{z} \mapsto & \mathtt{z: float} \end{array} \right\}.$$

Applying $\mathtt{u}\ e$ to h results in a new class definition:

$$\mathtt{u}\ e \odot_C h \ = \ e \overline{\odot_M} h \ = \ \left\{ \begin{array}{ll} \mathtt{w} \mapsto & \mathtt{w(): bool \{}\ B\ \mathtt{\},} \\ \mathtt{z} \mapsto & \mathtt{z: float} \end{array} \right\}.$$

Lemma 6. OOP software delta application is a delta action (cf. Definition 3).

The proof appears in the appendix. Finally, OOP software deltas can be applied to construct any product from any other product.

Lemma 7. OOP software deltas are maximally expressive (cf. Definition 6).

Proof. Because any method and class can be removed, all elements not required can be removed using a delta. New elements can be added using an additional delta. The composition of these deltas is the delta required to complete the proof. □

Since Lemma 7 states that all products in $\mathcal{P}$ are initial products, it follows directly that the *empty program*, which we use as the core product in our running example, is an initial product too.

Corollary. The empty program $\lambda x.\bot \in \mathcal{P}$ is an initial product.

Now is a good moment to illustrate product generation (Definition 18) for the Editor product line. We want the product for feature configuration $F = \{Ed, SH, EC, SA\} \in \Phi$. The selected delta model is $PL \restriction F = (D_F, \prec_F)$, with $D_F = \{d_1, d_3, d_4, d_6, d_8, d_9\}$ and

$$\prec_F = \left\{ \begin{array}{l} (d_1,d_3),(d_1,d_4),(d_1,d_6),(d_1,d_8),(d_1,d_9),(d_3,d_8), \\ (d_3,d_9),(d_4,d_6),(d_4,d_8),(d_4,d_9),(d_6,d_9),(d_8,d_9) \end{array} \right\}.$$

Since the Editor product line is globally unambiguous, it is sufficient to select one derivation of the delta model, such as $x = d_9 \cdot d_8 \cdot d_6 \cdot d_4 \cdot d_3 \cdot d_1$. Applying Definition 24, x is defined as follows:

$$\left\{ \begin{array}{l} \mathtt{Editor} \mapsto \\ \quad \mathtt{r} \left\{ \begin{array}{ll} \mathtt{model} & \mapsto \mathtt{model: Model;,} \\ \mathtt{semAnalyzer} & \mapsto \mathtt{semAnalyzer: SemanticAnalyzer;,} \\ \mathtt{getModel} & \mapsto \mathtt{getModel(): Model \{}\ A\ \mathtt{\},} \\ \mathtt{font} & \mapsto \mathtt{font(c: int): Font \{}\ O\ \mathtt{\},} \\ \mathtt{onMouseOver} & \mapsto \mathtt{onMouseOver(c: int): void \{}\ J\ \mathtt{\}} \end{array} \right\}, \\ \mathtt{SemanticAnalyzer} \mapsto \\ \quad \mathtt{r} \left\{ \begin{array}{ll} \mathtt{analyze} & \mapsto \mathtt{analyze(m: Model): void \{}\ K\ \mathtt{\},} \\ \mathtt{getErrors} & \mapsto \mathtt{getErrors(): Errors \{}\ L\ \mathtt{\}} \end{array} \right\} \end{array} \right\}.$$

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Applying x to c (Definition 25) results in a product that has the same form (only without the annotation r), since the core product is the empty program $c = \lambda x.\bot$.

Note that for the sake of simplicity we chose to ignore many features of OOP in this deltoid. Fully featured object-oriented languages would support concepts such as interfaces and inheritance. It is not difficult to extend the deltoid of this section with those concepts. For example, $\mathcal{P}$ could support classes with inheritance accompanied by more sophisticated deltas in $\mathcal{D}$ which can add and remove subclass relationships.

5.3. *A deltoid for aspect-oriented programming*

In AOP (Kiczales *et al.* 1997) and in languages based on feature-oriented programming (Apel *et al.* 2009b; Prehofer 1997), delta-oriented programming (Schaefer *et al.* 2010), context-oriented programming (Costanza and Hirschfeld 2005) and step-wise refinement (Batory *et al.* 2004), it is possible to refine a method implementation in such a way that it uses the previous method implementation. This can be thought of as *method wrapping*, and is realised, for example, by the `original` keyword in delta-oriented programming. The following is a simple delta-oriented programming example illustrating the idea.

```
class A {
    int m() { E }
}
delta D {
    modifies class A {
        modifies int m() { F; int x = original(); G }
    }
}
```

Applying delta `D` to class `A` results in a new implementation of `m`, which effectively corresponds to the old implementation placed where the call to `original()` is made:

```
class A {
    int original_m() { E }
    int m() { F; int x = original_m(); G }
}
```

To model this approach, we adapt the OOP software deltas (Sections 5.1 and 5.2) to include method wrapping by modifying method bodies $\mathcal{M}$ in classes and deltas to have the following (abstract) grammar:

$$\begin{array}{lll} \mathcal{M} \ni wb & ::= b \mid w[m] \mid w[\,] & b \text{ is a normal method body} \\ \mathcal{B} \ni m & ::= b \mid w[m] & \\ \mathcal{W} \ni w[\,] & ::= e[\,] \mid w[w[\,]] & e \text{ is a basic method wrapper} \end{array}.$$

The notation $w[\,]$ denotes a wrapping method with a hole in it, where the hole corresponds to the place where the call to the original method is made, and $w[m]$ denotes that body m is wrapped by w. Methods with a hole do not appear in products.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Given these ingredients, only the definitions of $\oplus_M$ and $\odot_M$ from Definitions 24 and 25 need to change. In the following, m, n denote methods with no hole:

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

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

The example above has the following correspondence with our setting:

— $b =$ `int m() { B }`,
— $w[\ ] =$ `int m() { C; int x = [ ]; D }` and
— $w[b] =$ `int m() { C; int x = B; D }`.

The AOP software deltas enjoy the same properties as the OOP software deltas. Proofs of Lemmas 8–10 appear in the appendix.

Lemma 8. AOP software deltas are a delta monoid (cf. Definition 1).

Lemma 9. AOP software deltas exhibit consistent conflict resolution (cf. Definition 13).

Lemma 10. AOP software delta application is a delta action (cf. Definition 3).

Lemma 11. AOP software deltas are maximally expressive (cf. Definition 6).

Proof. Follows from the fact that AOP software deltas are a conservative extension of OOP software deltas. □

6. Nested delta models

This section extends the notion of abstract delta modelling to incorporate nested delta models. Nested delta models can express the isolated, atomic application of a collection of deltas within a model by allowing a delta model to be used as a delta in another model. Nested deltas are useful when refactoring a delta into two deltas; grouping the two deltas together avoids creating conflicts, because the two deltas would be treated atomically.

As an example of a nested delta model consider Figure 4. In this figure delta d_4 from the Editor product line is refactored into two deltas d_4^1 and d_4^2, the first handling the `font` method and the second handling the `onMouseOver` method. To avoid having to introduced an extra ordering into the delta model to preserve the original semantics, the two deltas obtained from the refactoring are grouped in a nested delta model, which replaces d_4.

Observe from this example that nesting allows greater modularity and structure in delta models, in particular by allowing local refactoring of deltas without worrying about the potential interference from other deltas. This is because the derivation function, defined below, treats nesting deltas atomically. The definition imposes the further requirement that the nested delta models are unambiguous. This enforces in a kind of *local consistency*, avoiding the problem whereby refactoring introduces ambiguities that need to be resolved outside of the nested delta model.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

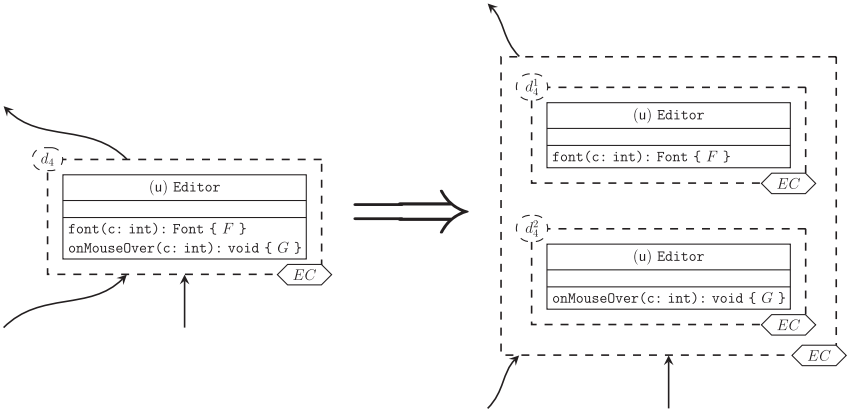


Fig. 4. Refactoring of delta d_4 into a nested delta model with deltas d_4^1 and d_4^2.

Nested delta models, along with nested delta monoids, and new notions of derivation and composition, are given by the following construction.

Definition 26. Given a delta monoid $(\mathcal{D}, \cdot, \epsilon)$, its corresponding *nested delta monoid* is the monoid $(\mathcal{D}_N, \bullet, \epsilon)$ as follows:

$$\begin{aligned}
\mathcal{D}_N^0 &= \mathcal{D} \\
DM_N^{i+1} &= \{(D, \prec) \mid D \subseteq \mathcal{D}_N^i,\ \prec \text{ is a strict partial order on } D \\
&\qquad\qquad \text{and } |\operatorname{derv}((D, \prec))| = 1\} \\
\mathcal{D}_N^{i+1} &= \mathcal{D} \cup DM_N^{i+1} \\
\mathcal{D}_N &= \bigcup_{i \geqslant 0} \mathcal{D}_N^i.
\end{aligned}$$

$$\begin{aligned}
\operatorname{derv}(m) &= \{m\}, && \text{where } m \in \mathcal{D} \\
\operatorname{derv}((D, \prec)) &= \bigcup_{\substack{x_1, \ldots, x_n \text{ is a linear} \\ \text{extension of } \prec}} x_n \bullet \ldots \bullet x_1, && \text{where } (D, \prec) \in \mathcal{D}_N \setminus \mathcal{D}.
\end{aligned}$$

$$N_1 \bullet N_2 = \operatorname{derv}(N_1) \cdot \operatorname{derv}(N_2), \text{ for } N_1, N_2 \in \mathcal{D}_N.$$

Note that the definition of $N_1 \bullet N_2$ abuses notation: technically, $\operatorname{derv}(N_1)$ and $\operatorname{derv}(N_2)$ return singleton sets, but the definition of $N_1 \bullet N_2$ implicitly treats them as the sole elements of that set.

Treating nested delta monoids as delta monoids and nested delta models as delta models, the definitions of *unique derivation* (Definition 9), *conflict* (Definition 10), *conflict-resolving delta* (Definition 11) and *unambiguous delta model* (Definition 12) carry across.

For product lines containing nested deltas, application conditions can apply at all levels of nesting. This definition can readily be formulated inductively, along the lines of Definition 26.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Definition 27 (nested delta model with application conditions). A *nested delta model with application conditions* is a triple $(D, \prec, \gamma)$ where the elements of D are either modifications $m \in \mathcal{D}$ or nested delta models with application conditions, $\prec$ is as before, and $\gamma : D \to \mathscr{P}(\mathscr{P}(F))$, such that if $N = (D', \prec', \gamma') \in D$, then $\gamma(N) = \bigcup_{d \in D'} \gamma'(d)$.

For reasons of simplicity, the condition $\gamma(N) = \bigcup_{d \in D'} \gamma'(d)$ derives the application condition of the whole nested delta model at some level based on those of its inner deltas; if at least one inner delta is applicable, then the nested delta model is applicable as well.

Based on Definition 27, we can now define nested product lines:

Definition 28 (Nested Product Line). A *nested product line* is a tuple $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$, where

— $\mathcal{F}$ is a feature set,
— $\Phi \subseteq \mathscr{P}(\mathcal{F})$ is a feature model,
— $c \in \mathcal{P}$ is the core product and
— $(D, \prec, \gamma)$ is a nested delta model with application conditions.

The nested delta model selected for given a feature configuration is obtained by recursing the nested delta model with application conditions of the nested product line, and selecting the relevant ingredients.

Definition 29 (selected nested delta model). Given a nested product line $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$, a selected nested delta model for a feature configuration $F \in \Phi$, denoted as $PL \restriction F$, is the nested delta model $(D, \prec, \gamma) \restriction F$, where $- \restriction F$ is defined on nested delta models as

$$
\begin{aligned}
m \restriction F &= m \\
(D, \prec, \gamma) \restriction F &= (D_F, \prec_F) \\
&\quad \text{where } D_F = \{\, d \restriction F \mid d \in D, F \in \gamma(d) \,\} \text{ and} \\
&\quad\quad \prec_F \text{ is } \prec \text{ restricted to } D_F.
\end{aligned}
$$

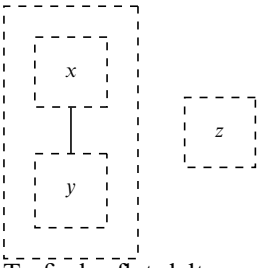
The definition of *generated products* (Definition 18) is as before.

6.1. *Expressiveness*

Nested delta models are more expressive than the old 'flat' delta models in the following sense: there exists a set of deltas and a way of ordering and nesting them such that its set of linear extensions cannot be expressed using the same deltas in a flat model.

Consider the following nested delta model N. We prove that there exists no flat delta model that has the same derivations (ignoring equations and considering just the sequences of deltas from $\mathcal{D}$).

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press



$$N = (\{(\{x, y\}, \{(x, y)\}), z\}, \varnothing)$$
$$\text{derv}(N) = \{z \cdot y \cdot x, y \cdot x \cdot z\}$$

To find a flat delta model $N' = (\{x, y, z\}, \prec')$ such that $\text{derv}(N') = \text{derv}(N)$, consider all possible partial orders $\prec'$ over 3 elements:

$$\begin{array}{lll}
\prec' = \varnothing & & \Longrightarrow |\,\text{derv}(N')| = 6 \\
\prec' = \{(A, B)\} & \text{s.t. } \{A, B\} \subseteq \{x, y, z\} & \Longrightarrow |\,\text{derv}(N')| = 3 \\
\prec' = \{(A, B), (B, C)\} & \text{s.t. } \{A, B, C\} = \{x, y, z\} & \Longrightarrow |\,\text{derv}(N')| = 1 \\
\prec' = \{(A, B), (A, C)\} & \text{s.t. } \{A, B, C\} = \{x, y, z\} & \Longrightarrow \text{derv}(N') = \{C \cdot B \cdot A,\ B \cdot C \cdot A\} \\
\prec' = \{(A, C), (B, C)\} & \text{s.t. } \{A, B, C\} = \{x, y, z\} & \Longrightarrow \text{derv}(N') = \{C \cdot B \cdot A,\ C \cdot A \cdot B\}.
\end{array}$$

As the number of derivations of N is 2, only the last two cases are relevant. If N were expressible via a flat delta model, there would exist a bijection between $\{A, B, C\}$ and $\{x, y, z\}$ such that either $\{C \cdot B \cdot A, B \cdot C \cdot A\} = \{z \cdot y \cdot x, y \cdot x \cdot z\}$ or $\{C \cdot B \cdot A, C \cdot A \cdot B\} = \{z \cdot y \cdot x, y \cdot x \cdot z\}$. However no such bijection exists, hence, there exists no flat delta model N' such that $\text{derv}(N') = \text{derv}(N)$. Since any flat delta model, trivially, is a nested delta model, this shows that nested delta models are strictly more expressive than flat delta models.

7. Related algebraic approaches

Other algebraic approaches describing the underlying structure of SPL have been proposed (Apel *et al.* 2010; Batory and Smith 2007). These formalise the mechanisms underlying AHEAD (Batory *et al.* 2004), GenVoca (Batory and O'Malley 1992), and FeatureHouse (Apel *et al.* 2009b). The first difference with our approach is that we present machinery to consider the collection of modifications for an entire product line, rather than a single product at a time, and thus are able to talk about conflicts and conflict resolution at the level of the product line. The second difference is that those approaches generally arrange 'deltas' into introductions and modifications – introductions correspond to the core ingredients of product, whereas modifications modify existing ingredients, whereas we assume a single, unified collection of deltas. Here, we compare our approach with two recent proposals, namely, the Quark model (Apel *et al.* 2010) and Finite Map Spaces (Batory and Smith 2007). 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.

This section gives an overview of quarks and finite map spaces in Section 7.1 and presents encodings of key elements of these formalisms into our setting in Sections 7.2–7.4.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

7.1. *Quarks and finite map spaces*

Both Apel *et al.* (2010) and Batory and Smith (2007) base the description of a product on the following ingredients (our notation).

— *Introductions*: a commutative idempotent monoid $(I, +, 0)$, where $+ : I \times I \to I$, of which some are 'atomic' and form a basis $\mathfrak{I} \subseteq I$ (in the sense of vector spaces/modules).
— *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, satisfying
  – M is a 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
  – $m \odot 0 = 0$.
— *Products*: elements of I, which are of the form $\sum_{j=1}^{n}(m_j \odot \mathbf{i}_j)$, where each $m_j \in M$ and $\mathbf{i}_j \in \mathfrak{I}$.

Introductions and modifications are combined to form *quarks* Q, which correspond to our deltas. Different notions of quark and quark composition ($\blacklozenge : Q \times Q \to Q$) have been defined – corresponding approximately to our notion of delta monoid – to capture combinations of the following operations:

— *local composition*: apply modifications to elements already in the product;
— *global composition*: apply modifications to all elements of the final product, and thus their application is delayed until after all introductions have been made; and
— *modifiers of modifiers*: modify modifications rather than elements of the product.

In addition to the quark and quark composition, the unit of quark composition and an operation $\mathsf{image} : Q \to I$ used when extracting the final product from a quark need to specified. The image operation also applies globally applicable operations at the last minute, where relevant.

local quark composition (Apel *et al.* 2010)

— $Q = I \times M$ – an introduction and a local modification
— $\langle i_2, l_2 \rangle \blacklozenge \langle i_1, l_1 \rangle = \langle i_2 + (l_2 \odot i_1), l_2 \bullet l_1 \rangle$
— unit is $\langle 0, 1 \rangle$
— $\mathsf{image}(\langle i, l \rangle) = i$

global quark composition (Apel *et al.* 2010)

— $Q = I \times M$ – an introduction and a global modification
— $\langle i_2, g_2 \rangle \blacklozenge \langle i_1, g_1 \rangle = \langle (g_2 \bullet g_1) \odot (i_2 + i_1), g_2 \bullet g_1 \rangle$
— unit is $\langle 0, 1 \rangle$
— $\mathsf{image}(\langle i, g \rangle) = i$

full quark composition (Apel *et al.* 2010)

— $Q = M \times I \times M$ – a global modification, an introduction and a local modification
— $\langle g_2, i_2, l_2 \rangle \blacklozenge \langle g_1, i_1, l_1 \rangle = \langle g_2 \bullet g_1, (g_2 \bullet g_1) \odot (i_2 + (l_2 \odot i_1)), l_2 \bullet l_1 \rangle$
— unit is $\langle 1, 0, 1 \rangle$
— $\mathsf{image}(\langle g, i, l \rangle) = i$

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

full quark composition (Batory and Smith 2007)

— $Q = M \times I \times M$ – a global modification, an introduction and a local modification
— $\langle g_2, i_2, l_2\rangle \blacklozenge \langle g_1, i_1, l_1\rangle = \langle g_2 \bullet g_1, i_2 + (l_2 \odot i_1), l_2 \bullet l_1\rangle$
— unit is $\langle 1, 0, 1\rangle$
— $\mathsf{image}(\langle g, i, l\rangle) = g \odot i$

modifiers of modifiers (Batory and Smith 2007)

— $Q = (M \to M) \times M \times I \times M$ – a modifier of modifiers, a global modification, an introduction and a local modification
— $\langle h_2, g_2, i_2, l_2\rangle \blacklozenge \langle h_1, g_1, i_1, l_1\rangle = \langle h_2 \circ h_1, g_2 \bullet g_1, i_2 + (l_2 \odot i_1), l_2 \bullet l_1\rangle$
— unit is $\langle id, 1, 0, 1\rangle$
— $\mathsf{image}(\langle h, g, i, l\rangle) = R^h(g \odot i)$.

The function R^h used in the definition of modifiers of modifiers applies the modifiers. Given a modifier of modifiers $h : M \to M$, Batory and Smith (2007) introduce a set of rewriting rules defining a function R^h to recursively apply all higher-order modifications. Their definition is equivalent to the following set of equations, where $m, m' \in M$, $i, i' \in I$, and $\mathbf{i} \in \mathfrak{I}$ is a basis element:

$$\begin{aligned} R^h(m) &= h(m) \\ R^h(0) &= 0 \\ R^h(\mathbf{i}) &= \mathbf{i} \\ R^h(i + i') &= R^h(i) + R^h(i') \\ R^h(m \bullet m') &= R^h(m) \bullet R^h(m') \\ R^h(m \odot i) &= R^h(m) \odot R^h(i). \end{aligned}$$

Note that R^h is overloaded to apply to both elements of M and of I.

Our observation is that this definition amounts to saying that $h : M \to M$ acts like monoid homomorphism on the modifications lifted to introductions. Thus:

$$R^h\left(\sum_{j=1}^{n} m_j \odot \mathbf{i}_j\right) = \sum_{j=1}^{n} (h(m_j) \odot \mathbf{i}_j).$$

For local quark composition, Batory and Smith (2007)'s full quark composition, and modifiers of modifiers, the quark composition operation $\blacklozenge$ forms a monoid over the corresponding set of quarks, with the appropriate tuple of units as the unit for $\blacklozenge$. Delta application $-(-) : Q \times I \to I$ (Definition 2) can be defined, for example, as $q(p) = \mathsf{image}(q \blacklozenge \langle p, 1\rangle)$, where $q \in Q$ is a quark and $p \in I$ 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, the other kinds of quarks above do not form a delta monoid; global quark composition and full quark composition (Apel *et al.* 2010) are not even associative. In addition, Apel *et al.*'s global quark composition and full quark composition produce results such as the following (for global quark composition):

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

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

which applies modifications g_3 and g_2 multiple times. To get this composition to behave, strong idempotence criteria are proposed (Apel *et al.* 2010), but these exclude modifications such as method wrapping. In addition, delta application is an action only for local quark composition.

We now describe how to encode local quark composition, full quark composition (Batory and Smith 2007), and modifiers of modifiers (Batory and Smith 2007) more directly in our setting. From our perspective, introductions play a dual role. They correspond to (elements of) products, as well as represent 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. In our encoding, we make introductions a kind of modification and eliminate quarks from the local composition variant. By ignoring the distinction between modifications and introductions, we can focus on deltas alone, and work in a simpler algebraic setting. For full quark composition and modifiers of modifiers, the notion of quark needs to be reintroduced.

7.2. *Encoding local quark composition*

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

The following definition introduces delta monoid M_I consisting of deltas that are sequences of 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 30 (delta monoid M_I). Given a monoid $(M, \bullet, 1)$, a commutative monoid $(I, +, 0)$, and an operation $\odot : M \times I \to I$ satisfying the equations, for all $m, n \in M$ and $i, j \in I$:

1 $1 \odot i = i$
2 $(m \bullet n) \odot i = m \odot (n \odot i)$
3 $m \odot (i + j) = (m \odot i) + (m \odot j)$
4 $m \odot 0 = 0$.

Define delta monoid $M_I = ((M \cup I)^*, \cdot, \epsilon)$ of finite sequences of elements of M and I, 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 30 forms a delta monoid by taking sequences of modifications and introductions, modulo the given 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

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

redundant and follow from the fact that $\cdot$ is concatenation and ϵ its unit, but we include them for completeness.

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 31. The delta action $-(-) : M_I \times I \to I$ for M_I is defined inductively as follows:

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

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

The following lemma captures that our notion of delta action is sensible, in that it preserves the equations in Definition 30. More precisely, given elements $\mu, \nu \in M_I$ that are perhaps syntactically distinct but equal by the equations, then they produce equal results when applied to a product $p \in I$ – recall that we consider elements of I as both introductions and ultimately as products. The proof of Lemma 12 appears in the appendix.

Lemma 12. For all $\mu, \nu \in M_I$ and all $p \in I$, if $\mu = \nu$ by Equations (1)–(7) of Definition 30, then $\mu(p) = \nu(p)$.

We now show the equivalence of quarks and M_I by producing homomorphisms in each direction. The following is a monoid homomorphism from quarks to M_I.

Definition 32. Define $[\![-]\!] : Q \to M_I$ as

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

Lemma 13. The function $[\![-]\!] : Q \to M_I$ is a homomorphism. That is, $[\![\langle 0, 1 \rangle]\!] = \epsilon$ and for all $q, q' \in Q$, $[\![q \blacklozenge q']\!] = [\![q]\!] \cdot [\![q']\!]$.

The proof of Lemma 13 appears in the appendix.

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

Definition 33. 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 (Theorem 4), supporting our claim that making the distinction between introductions and modifications is unnecessary. The proof of the following theorem appears in the appendix.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

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

1 $\langle\langle [\![q]\!] \rangle\rangle = q$, 3 if $q = q'$, then $[\![q]\!] = [\![q']\!]$, and

2 $[\![\langle\langle \mu \rangle\rangle]\!] = \mu$, 4 if $\mu = \nu$ via Equations (1)–(7) of Definition 30, then $\langle\langle \mu \rangle\rangle = \langle\langle \nu \rangle\rangle$.

Finally, Theorem 5 shows that not only are quarks and M_I isomorphic, but their notions of delta action correspond, so they will generate the same products. Again, its proof can be found in the appendix.

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 $i \in I$,

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

7.3. *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 the definition of delta action for M_I (Definition 31).

The full quark $\langle g, i, l \rangle$ is encoded as $\langle g, [\![\langle i, l \rangle]\!] \rangle = \langle g, i \cdot l \rangle$, using Definition 32. The results from the previous section can be extended to establish an isomorphism between the two forms of quark, in the obvious manner.

It is easy to show that the notions of delta application for full quark composition and this encoding coincide. On one hand, for full quark composition

$$\begin{aligned} \mathsf{image}(\langle g, i, l \rangle \blacklozenge \langle 1, p, 1 \rangle) &= \mathsf{image}(\langle g, i + (l \odot p), l \rangle) \\ &= g \odot (i + (l \odot p)). \end{aligned}$$

On the other hand, for our encoding:

$$\begin{aligned} \langle g, [\![i, l]\!] \rangle(p) &= g \odot (([\![i, l]\!])(p)) \\ &= g \odot (i \cdot l)(p) \\ &= g \odot (i(l(p))) \\ &= g \odot (i + (l \odot p)). \end{aligned}$$

However, in the absence of other assumptions, this notion of delta application is not an action – that is $(q \blacklozenge q')(p) = q(q'(p))$ does not hold in general – 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

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

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

If we instantiate μ and ν with m' and n', such that $m \neq m'$ and $n \neq n'$, 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

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

However, $(m \bullet n \bullet m' \bullet n') \odot p$ and $(m \bullet m' \bullet n \bullet n') \odot p$ are equal in general only if $\bullet$ is commutative.

7.4. *Encoding Batory and Smith's modifiers of modifiers*

Encoding modifiers of modifiers is also relatively straightforward. We assume that such modifiers, $h : M \to M$, are endomorphisms on the monoid of modifications, as described above when introducing R^h: 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 encoding to apply higher-order modifiers to global modifications as follows:

— quarks: $Q = (M \to M) \times M \times M_I$ consist of a modifier of modifiers, a global modification, and a delta
— composition: $\langle h_2, g_2, \mu_2 \rangle \blacklozenge \langle h_1, g_1, \mu_1 \rangle = \langle h_2 \circ h_1, g_2 \bullet g_1, \mu_2 \cdot \mu_1 \rangle$, and
— delta application is $\langle h, g, \mu \rangle(p) = h_I(g) \odot h_I(\mu)(h_I(p))$, where $h_I : M_I \to M_I$ is $h : M \to M$ lifted to M_I, defined by the following:
  – $h_I(\epsilon) = \epsilon$
  – $h_I(\mu \cdot \nu) = h_I(\mu) \cdot h_I(\nu)$
  – $h_I(\mathbf{i}) = \mathbf{i}$
  – $h_I(m \odot i) = h(m) \odot h_I(i)$.

Note that h_I is essentially $R^h : I \to I$, defined above, lifted from an overloaded function on M and I to a function on M_I.

Delta application for Batory and Smith (2007) of quark $\langle h, g, i, l \rangle$ to product p is:

$$\begin{aligned}\mathsf{image}(\langle h, g, i, l \rangle \blacklozenge \langle id, 1, p, 1 \rangle) &= \mathsf{image}(\langle h, g, i + (l \odot p), l \rangle) \\ &= R^h(g \odot (i + (l \odot p))) \\ &= R^h(g) \odot (R^h(i) + (R^h(l) \odot R^h(p))).\end{aligned}$$

Our delta application produces the same result:

$$\begin{aligned}(\langle h, g, [\![\langle i, l \rangle]\!] \rangle)(p) &= \langle h, g, i \cdot l \rangle(h_I(p)) \\ &= h_I(g) \odot (h_I(i \cdot l))(h_I(p)) \\ &= h_I(g) \odot (h_I(i) \cdot h_I(l))(h_I(p)) \\ &= h_I(g) \odot h_I(i)(h_I(l)(h_I(p))) \\ &= h_I(g) \odot (h_I(i) + (h_I(l) \odot h_I(p)).\end{aligned}$$

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

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

8. Related work

In general, approaches facilitating automated product generation for SPL can be classified in two main directions (Kästner *et al.* 2008). Firstly, annotative approaches, such as conditional compilation, frames (Zhang and Jarzabek 2003) or Colored Featherweight Java (Kästner and Apel 2008), annotate a model of the complete product line based on product features and remove irrelevant annotated product parts to obtain a product for a particular feature configuration.

Secondly, compositional approaches, such as delta modelling (Schaefer 2010; Schaefer and Damiani 2010; Schaefer *et al.* 2009, 2010), associate product fragments to product features, which are assembled to implement a particular feature configuration. A prominent example of this approach is AHEAD (Batory *et al.* 2004), 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 (Heidenreich and Wende 2007; Noda and Kishi 2008; Völter and Groher 2007). Apel *et al.* (2009a) apply model superposition to compose model fragments. Perrouin *et al.* (2008) obtain a product model by model composition and subsequently refinement by model transformation. In Haugen *et al.* (2008), a set of models is represented by a base model with associated variability and resolution models determining how modelling elements of the base model have to be replaced for a particular product model.

On the programming language level, several program modularization techniques (Lopez-Herrejon *et al.* 2005), such as aspects (Kästner *et al.* 2007), framed aspects (Loughran and Rashid 2004), mixins (Smaragdakis and Batory 2002), hyperslices (Tarr *et al.* 1999) or traits (Bettini *et al.* 2010; Ducasse *et al.* 2006), can be used to implement features in a compositional fashion. In addition, the modularity concepts of recent languages, such as Scala (Odersky 2007) or NewSpeak (Bracha 2007), can be used to represent product features. CeasarJ (Mezini and Ostermann 2004) and aspectual feature modules (Apel *et al.* 2008b) are proposed as a combination of feature modules and aspects to modularize crosscutting concerns.

The notion of program delta was introduced by Lopez-Herrejon *et al.* (2005) to describe the modifications of object-oriented programs. Schaefer *et al.* (2009) introduced delta modelling as a means to develop product line artefacts suitable for automated product derivation and implemented it using frame technology (Zhang and Jarzabek 2003). In subsequent work (Schaefer 2010), delta modelling was 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 modelling 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 (Schaefer *et al.* 2010). In Schaefer and Damiani (2010) and Schaefer *et al.* (2011), a version of delta-oriented programming is proposed where products are generated only from delta modules applied to the empty product.

Originally, the delta model of a product line consisted of a single core and a set of incomparable product deltas (Schaefer 2010; Schaefer *et al.* 2009). Conflicts between

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

deltas applicable for the same feature configuration were prohibited. In order to express all possible products, an additional delta covering the combination of the potentially conflicting deltas had to be specified leading to code duplication. Subsequently, a partial order between deltas was introduced (Schaefer and Damiani 2010; Schaefer *et al.* 2010, 2011). However, it was required that conflicts were avoided by specifying an appropriate order. In contrast, in this paper, a more flexible notion of conflict 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 (Prehofer 1997) or derivatives (Kästner *et al.* 2009; Liu *et al.* 2006) in feature-oriented programming, which are used to facilitate the correct interaction between different feature modules. A delta that is applied for the combination of certain features to resolve a conflict can fill in the role of a lifter or derivative. However, deltas are more expressive than lifters or derivatives, as deltas allow the removals of entities and the specification of complex application conditions to deal with arbitrary combinations of features.

The definition of a conflict as a lack of commutativity between modifications is also discussed in the context of program refactoring (Mens *et al.* 2005). The underlying formalisation uses graph transformation systems and critical pair analysis. Oldevik *et al.* (2009) 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.* (2008a) when two aspects advise shared join points. In order to make non-commutative aspects commute, the aspects have to be refactored following a particular scheme. In contrast, in delta modelling, the conflicting deltas do not have to be changed, only a conflict-resolving delta has to be added.

On a completely different note, the version control system Darcs is formalised in terms of *patch theory* (Jacobson 2009). The underlying formalism has some similarities with our work. Most notable is that 'patches' are modelled 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.

9. Conclusion

Delta modelling is an approach to facilitating automated product derivation for SPL. In this paper, we studied the conceptual ideas of delta modelling in an abstract, algebraic setting. One 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 efficiently computable conditions that allow checking the unambiguity of a product line without requiring that all products

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

be generated. Further, we extended the formalism with nested delta models to provide additional means for imposing structure in a delta model and to increase modularity.

For future work, we will be using the ideas of abstract delta modelling for the implementation of variability within the HATS ABS language (Hähnle 2010). ABS is an abstract executable modelling language for adaptable, object-oriented, distributed systems. By defining delta modification operations for ABS modelling entities, the variability of an ABS model can be specified by an ABS model delta. An ABS model for a particular configuration in space or in time can be generated from a core ABS model by application of ABS model deltas. The abstract, algebraic results presented in this article, in particular regarding consistent conflict resolution, can be immediately transferred to ABS models. Finally, variants of abstract delta modelling, such as basing the framework on partial monoids with a partial composition operation, will be investigated.

Appendix A. Proofs

The first part of this appendix gives proofs of properties for the OOP software deltoid and the AOP software deltoid (Section 5). Because the AOP software deltoid is a conservative extension of the OOP software deltoid, we give only proofs for the former. Relevant proofs for the OOP software deltas can be obtained by ignoring cases involving wrapping. The definitions of $\mathcal{B}$, $\mathcal{M}$, and $\mathcal{W}$ used are those found in Section 5.3. The second part of the appendix gives proofs for properties stated in Section 7.

A.1. *Useful lemmas*

The following lemmas will be useful later on. Their proofs are all straightforward from the definitions.

Lemma 14. If $f \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$ and $g \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$, then $(f \overline{\oplus_M} g)^* = (f \overline{\oplus_M} g^*)^*$.

Lemma 15. If $f \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$ and $g \in (\mathcal{I} \rightharpoonup \mathcal{M})$, then $(f \overline{\oplus_M} g)^* = f \overline{\odot_M} g$

Lemma 16. If $f \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$ and $g \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$, then $(f \overline{\oplus_M} g)^* = f \overline{\odot_M} g^*$.

Lemma 17. If $f \in (\mathcal{I} \rightharpoonup \mathcal{M})$, then $f^* = f$.

A.2. *Proof of Lemma 8 (and hence Lemma 4)*

Proof. We show that $\cdot$ is associative and $\lambda x.\bot$ is its neutral element. We start by working at the level of method modifications, then consider class-level modifications. First, note that if an operator $\bigcirc$ is associative, then operator $\overline{\bigcirc}$ is also associative. For arbitrary $a, b, c \in \mathcal{M} \cup \{-, \bot\}$, we show that $\oplus_M$ is associative, i.e. $(a \oplus_M b) \oplus_M c = a \oplus_M (b \oplus_M c)$, by case distinction on a:

— Case $a = \bot$.

$$(a \oplus_M b) \oplus_M c = (\bot \oplus_M b) \oplus_M c = b \oplus_M c = \bot \oplus_M (b \oplus_M c) = a \oplus_M (b \oplus_M c).$$

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

— Case $a = -$.

$$(a \oplus_M b) \oplus_M c = (- \oplus_M b) \oplus_M c = - \oplus_M c = - = - \oplus_M (b \oplus_M c) = a \oplus_M (b \oplus_M c).$$

— Case $a = m$ for some $m \in \mathcal{B}$.

$$(a \oplus_M b) \oplus_M c = (m \oplus_M b) \oplus_M c = m \oplus_M c = m = m \oplus_M (b \oplus_M c) = a \oplus_M (b \oplus_M c).$$

— Case $a = w[\ ]$ for some $w[\ ] \in \mathcal{W}$. We make a case distinction on b:

– Case $b = \bot$.

$$(a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M \bot) \oplus_M c = w[\ ] \oplus_M c = w[\ ] \oplus_M (\bot \oplus_M c) = a \oplus_M (b \oplus_M c).$$

– Case $b = -$.

$$(a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M -) \oplus_M c = - \oplus_M c = \\ - = w[\ ] \oplus_M - = w[\ ] \oplus_M (- \oplus_M c) = a \oplus_M (b \oplus_M c).$$

– Case $b = m$ for some $m \in \mathcal{B}$.

$$(a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M m) \oplus_M c = w[m] \oplus_M c = \\ w[m] = w[\ ] \oplus_M m = w[\ ] \oplus_M (m \oplus_M c) = a \oplus_M (b \oplus_M c).$$

– Case $b = u[\ ]$ for some $u[\ ] \in \mathcal{W}$. We make a case distinction on c:

• Case $c = \bot$.

$$(a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M \bot = w[u[\ ]] \oplus_M \bot = w[u[\ ]] = \\ w[\ ] \oplus_M u[\ ] = w[\ ] \oplus_M (u[\ ] \oplus_M \bot) = a \oplus_M (b \oplus_M c).$$

• Case $c = -$.

$$(a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M - = w[u[\ ]] \oplus_M - = - = w[\ ] \oplus_M - \\ = w[\ ] \oplus_M (u[\ ] \oplus_M -) = a \oplus_M (b \oplus_M c).$$

• Case $c = m$ for some $m \in \mathcal{B}$.

$$(a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M m = w[u[\ ]] \oplus_M m = \\ w[u[m] = w[\ ] \oplus_M u[m] = w[\ ] \oplus_M (u[\ ] \oplus_M m) = a \oplus_M (b \oplus_M c).$$

• Case $c = v[\ ]$ for some $v[\ ] \in \mathcal{W}$.

$$(a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M v[\ ] = w[u[\ ]] \oplus_M v[\ ] = w[u[v[\ ]]] \\ = w[\ ] \oplus_M u[v[\ ]] = w[\ ] \oplus_M (u[\ ] \oplus_M v[\ ]) = a \oplus_M (b \oplus_M c).$$

Thus $\oplus_M$ is associative. Consequently, so is $\overline{\oplus_M}$.

For arbitrary $a, b, c \in \{\mathsf{r}\} \times (\mathcal{I} \rightharpoonup \mathcal{M}) \cup \{\mathsf{u}\} \times (\mathcal{I} \rightharpoonup \mathcal{M}^-) \cup \{-, \bot\}$, we show that $\oplus_C$ is associative, i.e. $(a \oplus_C b) \oplus_C c = a \oplus_C (b \oplus_C c)$ by case distinction on a:

— Case $a = \bot$.

$$(a \oplus_C b) \oplus_C c = (\bot \oplus_C b) \oplus_C c = b \oplus_C c = \bot \oplus_C (b \oplus_C c) = a \oplus_C (b \oplus_C c).$$

— Case $a = -$.

$$(a \oplus_C b) \oplus_C c = (- \oplus_C b) \oplus_C c = - \oplus_C c = - = - \oplus_C (b \oplus_C c) = a \oplus_C (b \oplus_C c).$$

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

— Case $a = \mathsf{r}\ f$ for some $f \in (\mathcal{I} \rightharpoonup \mathcal{M})$.

$$(a \oplus_C b) \oplus_C c = (\mathsf{r}\ f \oplus_C b) \oplus_C c = \mathsf{r}\ f \oplus_C c = \mathsf{r}\ f = \mathsf{r}\ f \oplus_C (b \oplus_C c) = a \oplus_C (b \oplus_C c).$$

— Case $a = \mathsf{u}\ f$ for some $f \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$. We make a case distinction on b:

– Case $b = \bot$.

$$(a \oplus_C b) \oplus_C c = a \oplus_C c = a \oplus_C (b \oplus_C c).$$

– Case $b = -$. For some $f \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$:

$$\begin{aligned}(a \oplus_C b) \oplus_C c &= (\mathsf{u}\ f \oplus_C -) \oplus_C c = \mathsf{r}\ f^* \oplus_C c = \mathsf{r}\ f^* \\ &= \mathsf{u}\ f \oplus_C - = \mathsf{u}\ f \oplus_C (- \oplus_C c) = a \oplus_C (b \oplus_C c).\end{aligned}$$

– Case $b = \mathsf{r}\ g$ for some $g \in (\mathcal{I} \rightharpoonup \mathcal{M})$.

$$\begin{aligned}(a \oplus_C b) \oplus_C c &= (\mathsf{u}\ f \oplus_C \mathsf{r}\ g) \oplus_C c = \mathsf{r}\ (f\ \overline{\oplus_M}\ g)^* \oplus_C c = \mathsf{r}\ (f\ \overline{\oplus_M}\ g)^* \\ &= \mathsf{u}\ f \oplus_C \mathsf{r}\ g = \mathsf{u}\ f \oplus_C (\mathsf{r}\ g \oplus_C c) = a \oplus_C (b \oplus_C c).\end{aligned}$$

– Case $b = \mathsf{u}\ g$ for some $g \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$. We make a case distinction on c:

• Case $c = \bot$.

$$(a \oplus_C b) \oplus_C c = (a \oplus_C b) \oplus_C \bot = a \oplus_C b = a \oplus_C (b \oplus_C \bot) = a \oplus_C (b \oplus_C c).$$

• Case $c = -$.

$$\begin{aligned}&(a \oplus_C b) \oplus_C c \quad = (\mathsf{u}\ f \oplus_C \mathsf{u}\ g) \oplus_C - = \mathsf{u}\ (f\ \overline{\oplus_M}\ g) \oplus_C - = \\ &\mathsf{r}\ (f\ \overline{\oplus_M}\ g)^* \quad \overset{\star}{=} \mathsf{r}\ (f\ \overline{\oplus_M}\ g^*)^* \quad = \mathsf{u}\ f \oplus_C \mathsf{r}\ g^* \quad = \\ &\mathsf{u}\ f \oplus_C (\mathsf{u}\ g \oplus_C -) = a \oplus_C (b \oplus_C c).\end{aligned}$$

Step $\overset{\star}{=}$ follows from Lemma 14.

• Case $c = \mathsf{r}\ h$ for some $h \in (\mathcal{I} \rightharpoonup \mathcal{M})$.

$$\begin{aligned}&(a \oplus_C b) \oplus_C c \quad = (\mathsf{u}\ f \oplus_C \mathsf{u}\ g) \oplus_C \mathsf{r}\ h = \mathsf{u}\ (f\ \overline{\oplus_M}\ g) \oplus_C \mathsf{r}\ h \ = \\ &\mathsf{r}\ ((f\ \overline{\oplus_M}\ g)\ \overline{\oplus_M}\ h)^* \ = \mathsf{r}\ (f\ \overline{\oplus_M}(g\ \overline{\oplus_M}\ h))^* \ = \mathsf{u}\ f \oplus_C \mathsf{r}\ (g\ \overline{\oplus_M}\ h)^* = \\ &\mathsf{u}\ f \oplus_C (\mathsf{u}\ g \oplus_C \mathsf{r}\ h) = a \oplus_C (b \oplus_C c).\end{aligned}$$

• Case $c = \mathsf{u}\ h$ for some $h \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$.

$$\begin{aligned}&(a \oplus_C b) \oplus_C c \quad = (\mathsf{u}\ f \oplus_C \mathsf{u}\ g) \oplus_C \mathsf{u}\ h = \mathsf{u}\ (f\ \overline{\oplus_M}\ g) \oplus_C \mathsf{u}\ h = \\ &\mathsf{u}\ ((f\ \overline{\oplus_M}\ g)\ \overline{\oplus_M}\ h) \quad = \mathsf{u}\ (f\ \overline{\oplus_M}(g\ \overline{\oplus_M}\ h)) \quad = \mathsf{u}\ f \oplus_C \mathsf{u}\ (g\ \overline{\oplus_M}\ h) = \\ &\mathsf{u}\ f \oplus_C (\mathsf{u}\ g \oplus_C \mathsf{u}\ h) = a \oplus_C (b \oplus_C c).\end{aligned}$$

Thus $\oplus_C$ is associative. Consequently, so are $\overline{\oplus_C}$ and $\cdot$.
Neutrality of $\lambda x.\bot$ in $\cdot$ follows directly from the definition of $\oplus_C$. □

A.3. *Proof of Lemma 9 (and hence Lemma 5)*

For this case, we state and prove a lemma which will prove invaluable later on. Firstly, define (overload) $(-)^* : \mathcal{M} \cup \{\bot, -\} \to \mathcal{M} \cup \{\bot\}$ as $(-)^* = \bot$, $(\bot)^* = \bot$, $(m)^* = m$, for $m \in \mathcal{M}$.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Lemma 18. Given $a, d \in \mathcal{M} \cup \{-, \bot\}$.

1. Given $b \in \mathcal{M} \cup \{-, \bot\}$. If $(a \oplus_M b^*)^* = a^*$, then $(a \oplus_M d \oplus_M b^*)^* = (a \oplus_M d)^*$.
2. Given $b \in \mathcal{M} \cup \{\bot\}$. If $(a \oplus_M b)^* = a^*$, then $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d)^*$.
3. Given $b, c \in \mathcal{M} \cup \{-, \bot\}$. If $a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b$, then $a \oplus_M d \oplus_M b \oplus_M c = a \oplus_M d \oplus_M c \oplus_M b$.
4. Given $b \in \mathcal{M} \cup \{-, \bot\}$ and $c \in \mathcal{M} \cup \{\bot\}$. If $(a \oplus_M b)^* = (a \oplus_M (c \oplus_M b)^*)^*$, then $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M (c \oplus_M b)^*)^*$.
5. Given $b, c \in \mathcal{M} \cup \{\bot\}$. If $(a \oplus_M b)^* = (a \oplus_M c)^*$, then $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M c)^*$.

Proof. **Case 1**.
Assume that $(a \oplus_M b^*)^* = a^*$, where $b \in \mathcal{M} \cup \{-, \bot\}$. By case distinction on a:

— $a = \bot$. Hence $b^* = \bot$, thus $b = \bot$ or $b = -$. In the both cases $(a \oplus_M d \oplus_M b^*)^* = (a \oplus_M d \oplus_M \bot)^* = (a \oplus_M d)^*$.
— $a = -$. Then clearly $(a \oplus_M d \oplus_M b^*)^* = (-)^* = (a \oplus_M d)^*$.
— $a = m$ for some $m \in \mathcal{B}$. Then clearly $(a \oplus_M d \oplus_M b^*)^* = (m)^* = (a \oplus_M d)^*$.
— $a = w[\ ]$ for some $w[\ ] \in \mathcal{W}$. Hence $b = \bot$ or $b = -$. In the both cases, as with $a = \bot$, $(a \oplus_M d \oplus_M b^*)^* = (a \oplus_M d \oplus_M \bot)^* = (a \oplus_M d)^*$.

Case 2.
Assume that $(a \oplus_M b)^* = a^*$, where $b \in \mathcal{M} \cup \{\bot\}$. By case distinction on a:

— $a = \bot$. Hence $b^* = \bot$, thus $b = \bot$. Now $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M \bot)^* = (a \oplus_M d)^*$.
— $a = -$. Then, clearly $(a \oplus_M d \oplus_M b)^* = (-)^* = (a \oplus_M d)^*$.
— $a = m$ for some $m \in \mathcal{B}$. $(a \oplus_M d \oplus_M b)^* = (m)^* = (a \oplus_M d)^*$.
— $a = w[\ ]$ for some $w[\ ] \in \mathcal{W}$. Hence $b = \bot$. Thus, as with $a = \bot$. Now $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M \bot)^* = (a \oplus_M d)^*$.

Case 3.
For arbitrary $a, b, c \in \mathcal{M} \cup \{-, \bot\}$, assume $a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b$. Then for arbitrary $k \in \mathcal{M} \cup \{-, \bot\}$, we show that $a \oplus_M k \oplus_M b \oplus_M c = a \oplus_M k \oplus_M c \oplus_M b$, by case distinction on a:

— Case $a = \bot$.

$$\begin{aligned} a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b &\iff \bot \oplus_M b \oplus_M c = \bot \oplus_M c \oplus_M b \\ &\iff b \oplus_M c = c \oplus_M b. \end{aligned}$$

Hence $a \oplus_M k \oplus_M b \oplus_M c = a \oplus_M k \oplus_M c \oplus_M b$ for all k.
— Case $a = -$.
Observe that $- \oplus_M d = -$ for all d. Thus $a \oplus_M k \oplus_M b \oplus_M c = - \oplus_M k \oplus_M b \oplus_M c = - = - \oplus_M k \oplus_M c \oplus_M b = a \oplus_M k \oplus_M c \oplus_M b$ for all k.
— Case $a = m$ for some $m \in \mathcal{B}$.
Observe that $m \oplus_M d = m$ for all d. Thus $a \oplus_M k \oplus_M b \oplus_M c = m \oplus_M k \oplus_M b \oplus_M c = m = m \oplus_M k \oplus_M c \oplus_M b = a \oplus_M k \oplus_M c \oplus_M b$ for all k.
— Case $a = w[\ ]$ for some $w[\ ] \in \mathcal{W}$.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Consider the following table whose elements are $(a \oplus_M b \oplus_M c, a \oplus_M c \oplus_M b)$ – omitting entries above the diagonal for reasons of symmetry:

$b \backslash c$	$\bot$	$-$	n	$v[\ ]$
$\bot$	$(w[\ ], w[\ ])$			
$-$	$(-,-)$	$(-,-)$		
m	$(w[m], w[m])$	$(w[m], -)$	$(w[m], w[n])$	
$u[\ ]$	$(w[u[\ ]], w[u[\ ]])$	$(-,-)$	$(w[u[m]], w[n])$	$(w[u[v[\ ]]], w[v[u[\ ]]])$

Clearly $a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b$ only when $b = \bot, c = \bot$ *or* $b = c$. It is straightforward to see in these cases that $a \oplus_M k \oplus_M b \oplus_M c = a \oplus_M k \oplus_M c \oplus_M b$ for all k.

Thus $\forall a, b, c \in \mathcal{M} \cup \{-, \bot\} : a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b \Rightarrow \forall k \in \mathcal{M} \cup \{-, \bot\} : a \oplus_M k \oplus_M b \oplus_M c = a \oplus_M k \oplus_M c \oplus_M b$.

Case 4.

Assume that $(a \oplus_M b)^* = (a \oplus_M (c \oplus_M b)^*)^*$, where $b \in \mathcal{M} \cup \{-, \bot\}$ and $c \in \mathcal{M} \cup \{\bot\}$. By case distinction on a:

— $a = \bot$. This means that $b^* = (c \oplus_M b)^*$. Case analysis on c results in the following:
  - $c = \bot$: Hence $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M \bot \oplus_M b)^* = (a \oplus_M d \oplus_M b)^*$, as desired.
  - $c = m, b = m$: Hence $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M m \oplus_M m)^* = (a \oplus_M d \oplus_M m)^* = (a \oplus_M d \oplus_M b)^*$, as desired.
  - $c = -, b = -$. Hence, $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M - \oplus_M -)^* = (a \oplus_M d \oplus_M -)^* = (a \oplus_M d \oplus_M -)^*$, as desired.
  - $c = w[\ ], b = -$:
  - $c = -, b = -$. Hence, $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M w[\ ] \oplus_M -)^* = (a \oplus_M d \oplus_M -)^* = (a \oplus_M d \oplus_M -)^*$, as desired.

— $a = -$. Hence $(a \oplus_M d \oplus_M b)^* = (-)^* = (a \oplus_M d \oplus_M (c \oplus_M b)^*)^*$.

— $a = m$ for some $m \in \mathcal{B}$. Hence $(a \oplus_M d \oplus_M b)^* = (m)^* = (a \oplus_M d \oplus_M (c \oplus_M b)^*)^*$.

— $a = w[\ ]$ for some $w[\ ] \in \mathcal{W}$. Again we can deduce that $b^* = (c \oplus_M b)^*$, and perform the same case analysis as for $a = \bot$. The cases are written sufficiently generally to apply directly here.

Case 5.

Assume that $(a \oplus_M b)^* = (a \oplus_M c)^*$, where $b, c \in \mathcal{M} \cup \{\bot\}$. By case distinction on a:

— $a = \bot$. From this we deduce that $b = c$, and hence $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M c)^*$.

— $a = -$. Hence $(a \oplus_M d \oplus_M b)^* = (-)^* = (a \oplus_M d \oplus_M c)^*$.

— $a = m$ for some $m \in \mathcal{B}$. Hence $(a \oplus_M d \oplus_M b)^* = (m)^* = (a \oplus_M d \oplus_M c)^*$.

— $a = w[\ ]$ for some $w[\ ] \in \mathcal{W}$. From this we deduce that $b = c$, and hence $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M c)^*$.

□

Now we begin the main proof. Let $\mathcal{Y} = (\{\mathsf{r}\} \times (\mathcal{I} \rightharpoonup \mathcal{M})) \cup (\{\mathsf{u}\} \times (\mathcal{I} \rightharpoonup \mathcal{M}^-)) \cup \{-, \bot\}$.

It is clear that for $x, y, z \in \mathcal{D}$, $z \cdot y \cdot x = z \cdot x \cdot y$ implies $z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y$ if and only if for all $i \in \mathcal{I} : z(i) \oplus_C y(i) \oplus_C x(i) = z(i) \oplus_C x(i) \oplus_C y(i)$ implies $z(i) \oplus_C d(i) \oplus_C y(i) \oplus_C x(i) = z(i) \oplus_C d(i) \oplus_C x(i) \oplus_C y(i)$ $(\star\star\star)$. Note that each $x(i), y(i), z(i), d(i) \in \mathcal{Y}$.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

So we must show that for $p, q, r \in \mathcal{Y}$ that if $p \oplus_C q \oplus_C r = p \oplus_C r \oplus_C q$, then for all $o \in \mathcal{Y}$ $p \oplus_C o \oplus_C q \oplus_C r = p \oplus_C o \oplus_C r \oplus_C q$.

Assume $p \oplus_C q \oplus_C r = p \oplus_C r \oplus_C q$. Let $o \in \mathcal{Y}$.

Firstly by considering the different cases for o, we see that when $o = \bot$, we have that $p \oplus_C o \oplus_C q \oplus_C r = p \oplus_C q \oplus_C r$, from which the result follows, and when $o = -$ or $o = \mathsf{r}\ f$, we have that for all x $o \oplus_C x = o$, from which the desired result again quickly follows.

Assume that $o = \mathsf{u}\ d$. We proceed by case analysis on p.

— Case $p = \bot$.
From the definition of $\oplus_C$, it is clear that $\bot$ is the unit of $\oplus_C$, thus $\bot \oplus_C q \oplus_C r = \bot \oplus_C r \oplus_C q$ implies that $q \oplus_C r = r \oplus_C q$, from which the result follows immediately.

— Case $p = -$.
From the definition of $\oplus_C$, it is clear that for all q we have $- \oplus_C q = -$, from which the result follows immediately.

— Case $p = \mathsf{r}\ h$ for some $h \in (\mathcal{I} \rightharpoonup \mathcal{M})$.
From the definition of $\oplus_C$, it is clear that for all q we have $\mathsf{r}\ h \oplus_C q = \mathsf{r}\ h$, from which the result follows immediately.

— Case $p = \mathsf{u}\ h$ for some $h \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$.
Firstly observe, as before, if $q = \bot$ or $r = \bot$, then $q \oplus_C r = r \oplus_C q$, from which the desired result follows.
The following table contains $(p \oplus_C q \oplus_C r, p \oplus_C r \oplus_C q)$ for the remaining combinations, again removing symmetry.

$q \backslash r$	$-$	$\mathsf{u}\ f$	$\mathsf{r}\ f$
$-$	$(\mathsf{r}\ h^*, \mathsf{r}\ h^*)$		
$\mathsf{u}\ e$	$(\mathsf{r}\ (h \overline{\oplus_M} e^*)^*, \mathsf{r}\ h^*)$	$(\mathsf{u}\ (h \overline{\oplus_M} (e \overline{\oplus_M} f)), \mathsf{u}\ (h \overline{\oplus_M} (f \overline{\oplus_M} e)))$	
$\mathsf{r}\ e$	$(\mathsf{r}\ (h \overline{\oplus_M} e)^*, \mathsf{r}\ h^*)$	$(\mathsf{r}\ (h \overline{\oplus_M} e)^*, \mathsf{r}\ (h \overline{\oplus_M} (f \overline{\oplus_M} e)^*)^*)$	$(\mathsf{r}\ (h \overline{\oplus_M} e)^*, \mathsf{r}\ (h \overline{\oplus_M} f)^*)$

Note that we can determine the values for $p \oplus_C o \oplus_C q \oplus_C r$ and $p \oplus_C o \oplus_C r \oplus_C q$) by replacing h by $h \overline{\oplus_M} d$ in each case.

Now we perform a case analysis based on these entries:

— Case $q = r = -$. Easy.

— Case $q = \mathsf{u}\ e$, $r = -$. From the table we deduce $(h \overline{\oplus_M} e^*)^* = h^*$. The desired result, namely $((h \overline{\oplus_M} d) \overline{\oplus_M} e^*)^* = (h \overline{\oplus_M} d)^*$, follows from Lemma 18(1), lifted from $\oplus_M$ to $\overline{\oplus_M}$.

— Case $q = \mathsf{r}\ e$, $r = -$. From the table we deduce $(h \overline{\oplus_M} e)^* = h^*$. The desired result, namely $((h \overline{\oplus_M} d) \overline{\oplus_M} e)^* = (h \overline{\oplus_M} d)^*$ follows from Lemma 18(2), lifted from $\oplus_M$ to $\overline{\oplus_M}$.

— Case $q = \mathsf{u}\ e$, $r = \mathsf{u}\ f$. From the table we deduce $h \overline{\oplus_M} (e \overline{\oplus_M} f) = h \overline{\oplus_M} (f \overline{\oplus_M} e)$. The desired result, namely $(h \overline{\oplus_M} d) \overline{\oplus_M} (e \overline{\oplus_M} f) = (h \overline{\oplus_M} d) \overline{\oplus_M} (f \overline{\oplus_M} e)$, follows from Lemma 18(3), lifted from $\oplus_M$ to $\overline{\oplus_M}$.

— Case $q = \mathsf{r}\ e$, $r = \mathsf{u}\ f$. From the table we deduce $(h \overline{\oplus_M} e)^* = (h \overline{\oplus_M} (f \overline{\oplus_M} e)^*)^*$. The desired result, namely $((h \overline{\oplus_M} d) \overline{\oplus_M} e)^* = ((h \overline{\oplus_M} d) \overline{\oplus_M} (f \overline{\oplus_M} e)^*)^*$, follows from Lemma 18(4), lifted from $\oplus_M$ to $\overline{\oplus_M}$.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

— Case $q = \mathsf{r}\ e$, $r = \mathsf{r}\ f$. From the table we deduce $(h\ \overline{\oplus_M}\ e)^* = (h\ \overline{\oplus_M}\ f)^*$. The desired result, namely $((h\ \overline{\oplus_M}\ d)\ \overline{\oplus_M}\ e)^* = ((h\ \overline{\oplus_M}\ d)\ \overline{\oplus_M}\ f)^*$, follows from Lemma 18(5), lifted from $\oplus_M$ to $\overline{\oplus_M}$.

Thus we have that for all $p, q, r \in \mathcal{Y}$, that $p \oplus_C q \oplus_C r = p \oplus_C r \oplus_C q$ implies that $\forall o \in \mathcal{Y} : p \oplus_C o \oplus_C q \oplus_C r = p \oplus_C o \oplus_C r \oplus_C q$.

Hence from $(\star\star\star)$, we have the desired result.

A.4. *Proof of Lemma 10 (and hence Lemma 6)*

We start by working at the level of method modifications, then consider class-level modifications.

For arbitrary $a, b \in \mathcal{M} \cup \{-, \bot\}$ and arbitrary $c \in \mathcal{M} \cup \{\bot\}$, we show that $(a \oplus_M b) \odot_M c = a \odot_M (b \odot_M c)$ by case distinction on a:

— Case $a = \bot$.

$$(a \oplus_M b) \odot_M c = (\bot \oplus_M b) \odot_M c = b \odot_M c = \bot \odot_M (b \odot_M c) = a \odot_M (b \odot_M c).$$

— Case $a = -$.

$$(a \oplus_M b) \odot_M c = (- \oplus_M b) \odot_M c = - \odot_M c = \bot = - \odot_M (b \odot_M c) = a \odot_M (b \odot_M c).$$

— Case $a = m$ for some $m \in \mathcal{B}$.

$$(a \oplus_M b) \odot_M c = (m \oplus_M b) \odot_M c = m \odot_M c = m = m \odot_M (b \odot_M c) = a \odot_M (b \odot_M c).$$

— Case $a = w[\ ]$ for some $w[\ ] \in \mathcal{W}$. We make a case distinction on b.

– Case $b = \bot$.

$$(a \oplus_M b) \odot_M c = (w[\ ] \oplus_M \bot) \odot_M c = w[\ ] \odot_M c = w[\ ] \odot_M (\bot \odot_M c) = a \odot_M (b \odot_M c).$$

– Case $b = -$.

$$\begin{aligned}(a \oplus_M b) \odot_M c &= (w[\ ] \oplus_M -) \odot_M c = - \odot_M c = \bot \\ &= w[\ ] \odot_M \bot = w[\ ] \odot_M (- \odot_M c) = a \odot_M (b \odot_M c).\end{aligned}$$

– Case $b = m$ for some $m \in \mathcal{B}$.

$$\begin{aligned}(a \oplus_M b) \odot_M c &= (w[\ ] \oplus_M m) \odot_M c = w[m] \odot_M c = w[m] = w[\ ] \odot_M m \\ &= w[\ ] \odot_M (m \odot_M c) = a \odot_M (b \odot_M c).\end{aligned}$$

– Case $b = u[\ ]$ for some $u[\ ] \in \mathcal{W}$. We make a case distinction on c.

• Case $c = \bot$.

$$\begin{aligned}(a \oplus_M b) \odot_M c &= (w[\ ] \oplus_M u[\ ]) \odot_M \bot = w[u[\ ]] \odot_M \bot = \bot \\ &= w[\ ] \odot_M \bot = w[\ ] \odot_M (u[\ ] \odot_M \bot) = a \odot_M (b \odot_M c).\end{aligned}$$

• Case $c = m$ for some $m \in \mathcal{B}$.

$$\begin{aligned}(a \oplus_M b) \odot_M c &= (w[\ ] \oplus_M u[\ ]) \odot_M m = w[u[\ ]] \odot_M m = w[u[m]] \\ &= w[\ ] \odot_M w[m] = w[\ ] \odot_M (u[\ ] \odot_M m) = a \odot_M (b \odot_M c).\end{aligned}$$

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Thus $\forall a, b \in \mathcal{M} \cup \{-, \bot\} : \forall c \in \mathcal{M} \cup \{\bot\} : (a \oplus_M b) \odot_M c = a \odot_M (b \odot_M c)$. Consequently, also $\forall f, g \in (\mathcal{I} \rightharpoonup \mathcal{M}^-) : \forall h \in (\mathcal{I} \rightharpoonup \mathcal{M}) : (f \overline{\oplus_M} g) \overline{\odot_M} h = f \overline{\odot_M}(g \overline{\odot_M} h)$ holds.

Now, we move to class-level modifications. For arbitrary $a, b \in \{\mathsf{r}\} \times (\mathcal{I} \rightharpoonup \mathcal{M}) \cup \{\mathsf{u}\} \times (\mathcal{I} \rightharpoonup \mathcal{M}^-) \cup \{-, \bot\}$ and arbitrary $c \in (\mathcal{I} \rightharpoonup \mathcal{M}) \cup \{\bot\}$, we show that $(a \oplus_C b) \odot_C c = a \odot_C (b \odot_C c)$ by case distinction on a:

— Case $a = \bot$.

$$(a \oplus_C b) \odot_C c = (\bot \oplus_C b) \odot_C c = b \odot_C c = \bot \odot_C (b \odot_C c) = a \odot_C (b \odot_C c).$$

— Case $a = -$.

$$(a \oplus_C b) \odot_C c = (- \oplus_C b) \odot_C c = - \odot_C c = \bot = - \odot_C (b \odot_C c) = a \odot_C (b \odot_C c).$$

— Case $a = \mathsf{r}\ f$ for some $f \in (\mathcal{I} \rightharpoonup \mathcal{M})$.

$$(a \oplus_C b) \odot_C c = (\mathsf{r}\ f \oplus_C b) \odot_C c = \mathsf{r}\ f \odot_C c = f = \mathsf{r}\ f \odot_C (b \odot_C c) = a \odot_C (b \odot_C c).$$

— Case $a = \mathsf{u}\ f$ for some $f \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$. We make a case distinction on b:

– Case $b = \bot$.

$$(a \oplus_C b) \odot_C c = (a \oplus_C \bot) \odot_C c = a \odot_C c = a \odot_C (\bot \odot_C c) = a \odot_C (b \odot_C c).$$

– Case $b = -$.

$$\begin{aligned}(a \oplus_C b) \odot_C c &= (\mathsf{u}\ f \oplus_C -) \odot_C c = \mathsf{r}\ f^* \odot_C c = f^* = \mathsf{u}\ f \odot_C \bot \\ &= \mathsf{u}\ f \odot_C (- \odot_C c) = a \odot_C (b \odot_C c).\end{aligned}$$

– Case $b = \mathsf{r}\ g$ for some $g \in (\mathcal{I} \rightharpoonup \mathcal{M})$.

$$\begin{aligned}(a \oplus_C b) \odot_C c &= (\mathsf{u}\ f \oplus_C \mathsf{r}\ g) \odot_C c = \mathsf{r}\ (f \overline{\oplus_M} g)^* \odot_C c \overset{\star}{=} (f \overline{\oplus_M} g)^* = f \overline{\odot_M} g \\ &= \mathsf{u}\ f \odot_C g = \mathsf{u}\ f \odot_C (\mathsf{r}\ g \odot_C c) = a \odot_C (b \odot_C c).\end{aligned}$$

Step $\overset{\star}{=}$ follows from Lemma 15.

– Case $b = \mathsf{u}\ g$ for some $g \in (\mathcal{I} \rightharpoonup \mathcal{M}^-)$. We make a case distinction on c:

• Case $c = \bot$.

$$\begin{aligned}(a \oplus_C b) \odot_C c &= (\mathsf{u}\ f \oplus_C \mathsf{u}\ g) \odot_C \bot = \mathsf{u}\ (f \overline{\oplus_M} g) \odot_C \bot = (f \overline{\oplus_M} g)^* \\ &\overset{\star}{=} f \overline{\oplus_M} g^* = \mathsf{u}\ f \odot_C g^* = \mathsf{u}\ f \odot_C (\mathsf{u}\ g \odot_C \bot) = a \odot_C (b \odot_C c).\end{aligned}$$

Step $\overset{\star}{=}$ follows from Lemma 16.

• Case $c = h$ for some $h \in (\mathcal{I} \rightharpoonup \mathcal{M})$.

$$\begin{aligned}(a \oplus_C b) \odot_C c &= (\mathsf{u}\ f \oplus_C \mathsf{u}\ g) \odot_C h = \mathsf{u}\ (f \overline{\oplus_M} g) \odot_C h = (f \overline{\oplus_M} g) \overline{\odot_M} h \\ &= f \overline{\odot_M}(g \overline{\odot_M} h) = f \overline{\odot_M}(\mathsf{u}\ g \odot_C h) = \mathsf{u}\ f \odot_C (\mathsf{u}\ g \odot_C h) = a \odot_C (b \odot_C c).\end{aligned}$$

Thus $(y \overline{\oplus_C} x) \overline{\odot_C} p = y \overline{\odot_C}(x \overline{\odot_C} p)$ also holds for all $x, y \in \mathcal{D}$ and $p \in \mathcal{P}$. Or in standard notation: $(y \cdot x)(p) = y(x(p))$.

A.5. *Proof of Lemma 12*

Proof. We need to show that this holds for each axiom in Definition 30.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

1. $(\epsilon \cdot \mu)(p) = \epsilon(\mu(p)) = \mu(p) = \mu(\epsilon(p)) = (\mu \cdot \epsilon)(p)$.
2. $(\mu \cdot (\nu \cdot \eta))(p) = \mu((\nu \cdot \eta)(p)) = \mu(\nu(\eta(p))) = (\mu \cdot \nu)(\eta(p)) = ((\mu \cdot \nu) \cdot \eta)(p)$.
3. $(m \cdot i)(p) = m(i(p)) = m \odot (i + p) = (m \odot i) + (m \odot p) = (m \odot i) + m(p) = (m \odot i)(m(p)) = ((m \odot i) \cdot m)(p)$.
4. $(i \cdot j)(p) = i(j(p)) = i + (j + p) = j + (i + p) = (j \cdot i)(p)$. This is also equal to $(i + j)(p)$ and $(j + i)(p)$, as, for example $i + (j + p) = (i + j) + p = (i + j)(p)$.
5. $(i \cdot i)(p) = i(i(p)) = i + (i + p) = (i + i) + p = (i + i)(p)$. Also, $(i + i) + p = i + p = i(p)$.
6. $(m \cdot n)(p) = m(n(p)) = m \odot (n \odot p) = (m \bullet n) \odot p = (m \bullet n)(p)$.
7. $1(p) = 1 \odot p = p = \epsilon(p) = p = 0 + p = 0(p)$. □

A.6. *Proof of Lemma 13*

Proof. Firstly, $[\![\langle 0, 1\rangle]\!] = 0 \cdot 1 = \epsilon \cdot \epsilon = \epsilon$. Secondly, let $q = \langle i, m\rangle$ and $q' = \langle i', m'\rangle$. On one hand, $[\![\langle i, m\rangle \blacklozenge \langle i', m'\rangle]\!] = [\![\langle i + m \odot i', m \bullet m'\rangle]\!] = (i + m \odot i') \cdot (m \bullet m')$. On the other hand, $[\![\langle i, m\rangle]\!] \cdot [\![\langle i', m'\rangle]\!] = i \cdot m \cdot i' \cdot m' = i \cdot (m \odot i') \cdot (m \cdot m') = (i + m \odot i') \cdot (m \bullet m')$. □

A.7. *Proof of Theorem 4*

Proof.

1. $\langle\!\langle [\![\langle i, l\rangle]\!] \rangle\!\rangle = \langle\!\langle l \cdot i \rangle\!\rangle = \langle 0, l\rangle \blacklozenge \langle i, 1\rangle = \langle i + (1 \odot 0), 1 \bullet l\rangle = \langle i + 0, 1 \bullet l\rangle = \langle i, l\rangle$.
2. By induction on μ.
   - — Case i. $[\![\langle\!\langle i \rangle\!\rangle]\!] = [\![\langle i, 1\rangle]\!] = i \cdot 1 = i \cdot \epsilon = i$.
   - — Case m. $[\![\langle\!\langle m \rangle\!\rangle]\!] = [\![\langle 0, m\rangle]\!] = 0 \cdot m = \epsilon \cdot m = m$.
   - — Case $\mu \cdot \nu$: $[\![\langle\!\langle \mu \bullet \nu \rangle\!\rangle]\!] = [\![\langle\!\langle \mu \rangle\!\rangle \blacklozenge \langle\!\langle \nu \rangle\!\rangle]\!] = [\![\langle\!\langle \mu \rangle\!\rangle]\!] \cdot [\![\langle\!\langle \nu \rangle\!\rangle]\!] = \mu \cdot \nu$. using Lemma 13 and induction hypothesis.

1. Associativity is preserved as $\bullet$ is associative. That $\langle 0, 1\rangle$ is the unit of $\blacklozenge$ is preserved by $[\![-]\!]$ follows from the fact that $0 = 1 = \epsilon$ and ϵ is the unit of $\cdot$. Finally, if $\langle i_1, m_1\rangle = \langle i_2, m_2\rangle$, then $i_1 = i_2$ and $m_1 = m_2$. Now $[\![\langle i_1, m_1\rangle]\!] = m_1 \cdot i_1 = m_2 \cdot i_2 = [\![\langle i_2, m_2\rangle]\!]$.
2. We need to show that the axioms in Definition 30 are preserved by $\langle\!\langle - \rangle\!\rangle$.

   (1) Follows because $\langle\!\langle \epsilon \rangle\!\rangle = \langle 0, 1\rangle$ is the unit of $\blacklozenge$.
   (2) Follows because $\blacklozenge$ is associative.
   (3) $\langle\!\langle m \cdot i \rangle\!\rangle = \langle 0, m\rangle \blacklozenge \langle i, 1\rangle = \langle m \odot i, m\rangle$ which is the same as $\langle\!\langle (m \odot i) \cdot m \rangle\!\rangle = \langle m \odot i, 1\rangle \blacklozenge \langle 0, m\rangle = \langle m \odot i, m\rangle$.
   (4) $\langle\!\langle i \cdot j \rangle\!\rangle = \langle i, 1\rangle \blacklozenge \langle j, 1\rangle = \langle i + j, 1\rangle = \langle\!\langle i + j \rangle\!\rangle$, which is clearly also equal to $\langle\!\langle j + i \rangle\!\rangle$ and $\langle\!\langle j \cdot i \rangle\!\rangle$.
   (5) $\langle\!\langle i \cdot i \rangle\!\rangle = \langle i, 1\rangle \blacklozenge \langle i, 1\rangle = \langle i + i, 1\rangle = \langle i, 1\rangle = \langle\!\langle i \rangle\!\rangle$.
   (6) $\langle\!\langle m \cdot n \rangle\!\rangle = \langle 0, m\rangle \blacklozenge \langle 0, n\rangle = \langle 0 + (m \odot 0), m \bullet n\rangle = \langle 0, m \bullet n\rangle = \langle\!\langle m \bullet n \rangle\!\rangle$.
   (7) By definition we have $\langle\!\langle \epsilon \rangle\!\rangle = \langle 0, 1\rangle$, $\langle\!\langle 0 \rangle\!\rangle = \langle 0, 1\rangle$, and $\langle\!\langle 1 \rangle\!\rangle = \langle 0, 1\rangle$. □

A.8. *Proof of Theorem 5*

Proof. Firstly, $\mathsf{image}(\langle i, m\rangle \blacklozenge \langle p, 1\rangle) = \mathsf{image}(\langle i + (m \odot p), m\rangle) = i + (m \odot p)$ and $[\![\langle i, m\rangle]\!](p) = (i \cdot m)(p) = i(m(p)) = i + (m \odot p)$.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Secondly, we prove by induction using the stronger hypothesis that for all $\mu \in M_I$ and all $i \in I$, there exists an m such that $\langle\langle \mu \rangle\rangle \blacklozenge \langle p, 1 \rangle = \langle \mu(p), m \rangle$. Proceed by induction on form of μ:

— Case i. $\langle\langle i \rangle\rangle \blacklozenge \langle p, 1 \rangle = \langle i, 1 \rangle \blacklozenge \langle p, 1 \rangle = \langle i + p, 1 \rangle$. Now $i(p) = i + p$, as desired.
— Case m. $\langle\langle m \rangle\rangle \blacklozenge \langle p, 1 \rangle = \langle 0, m \rangle \blacklozenge \langle p, 1 \rangle = \langle m \odot p, m \rangle$. Now $m(p) = m \odot p$, as desired.
— Case $\mu \cdot \nu$. $\langle\langle \mu \cdot \nu \rangle\rangle \blacklozenge \langle p, 1 \rangle = \langle\langle \mu \rangle\rangle \blacklozenge \langle\langle \nu \rangle\rangle \blacklozenge \langle p, 1 \rangle$. By the induction hypothesis, there exists an m such that $\langle\langle \nu \rangle\rangle \blacklozenge \langle p, 1 \rangle = \langle \nu(p), m \rangle$. Similarly, applying the induction hypothesis to $\mu \in M_I$ and $\langle \nu(p), m \rangle \in P$, we obtain that there exists an n such that $\langle\langle \mu \rangle\rangle \blacklozenge \langle \nu(p), m \rangle = \langle \mu(\nu(p)), n \rangle$. By Definition 30 this equals $\langle (\mu \cdot \nu)(p), n \rangle$, and we are done. □

References

Apel, S., Janda, F., Trujillo, S. and Kästner, C. (2009a) Model superimposition in software product lines. In: *International Conference on Model Transformation (ICMT)* 4–19.

Apel, S., Kästner, C. and Batory, D.S. (2008a) Program refactoring using functional aspects. In: *GPCE* 161–170.

Apel, S., Kästner, C. and Lengauer, C. (2009b) FeatureHouse: Language-independent, automated software composition. In: *ICSE* 221–231.

Apel, S., Leich, T. and Saake, G. (2008b) Aspectual feature modules. *IEEE Transactions on Software Engineering* **34** (2) 162–180.

Apel, S., Lengauer, C., Möller, B. and Kästner, C. (2010) An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming (SCP)* **75** (11) 1022–1047.

Batory, D. and O'Malley, S. (1992) The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology* **1** (4) 355–398.

Batory, D., Sarvela, J. and Rauschmayer, A. (2004) Scaling step-wise refinement. *IEEE Transactions on Software Engineering* **30** (6) 355–371.

Batory, D. and Smith, D. (2007) Finite map spaces and quarks: algebras of program structure. *TechnicalReport* TR-07-66, University of Texas at Austin, Department of Computer Sciences.

Bettini, L., Damiani, F. and Schaefer, I. (2010) Implementing software product lines using traits. In: *Proceedings of Object-Oriented Programming Languages and Systems (OOPS)*, Track of ACM SAC 2096–2102.

Bracha, G. (2007) Executable grammars in Newspeak. *Electronic Notes in Theoretical Computer Science* **193** 3–18.

Clarke, D., Helvensteijn, M. and Schaefer, I. (2010) Abstract delta modelling. In: *Proceedings of GPCE*, ACM 13–22.

Clements, P. and Northrop, L. (2001) *Software Product Lines: Practices and Patterns*, Addison Wesley Longman.

Costanza, P. and Hirschfeld, R. (2005) Language constructs for context-oriented programming: an overview of ContextL. In: *DLS*, ACM Press 1–10.

Czarnecki, K., Helson, S. and Eisenecker, U. (2004) Staged configuration using feature models. In: R. Nord (ed.) Proceedings of 3rd International Software Product Line Conference (SPLC 2004). *Lecture Notes in Computer Science* **3154** 266–283.

Czarnecki, K. and Kim, C. (2005) Cardinality-based feature modelling and constraints: a progress report. In: *International Workshop on Software Factories at OOPSLA'05*, San Diego, USA: ACM Press 1–9.

https://doi.org/10.1017/S0960129512000941 Published online by Cambridge University Press

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R. and Black, A. (2006) Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems* **28** (2) 331–388.

Hähnle, R. (2010) HATS: highly adaptable and trustworthy software using formal methods. In: *ISoLA (2)* 3–8.

Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G. and Svendsen, A. (2008) Adding standardized variability to domain specific languages. In: *SPLC* 139–148.

Heidenreich, F. and Wende, C. (2007) Bridging the gap between features and models. In: *Aspect-Oriented Product Line Engineering (AOPLE'07)*.

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

Jacobson, J. (2009) A formalization of Darcs patch theory using inverse semigroups. *TechnicalReport* CAM report 09-83, UCLA.

Kang, K. C., Cohen, S., Hess, J., Nowak, W. and Peterson, S. (1990) Feature-oriented domain analysis (FODA) feasibility study. *TechnicalReport* CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute.

Kästner, C. and Apel, S. (2008) Type-checking software product lines - A formal approach. In: *ASE*, IEEE 258–267.

Kästner, C., Apel, S. and Batory, D. (2007) A case study implementing features using AspectJ. In: *SPLC*, IEEE 223–232.

Kästner, C., Apel, S. and Kuhlemann, M. (2008) Granularity in software product lines. In: *ICSE* 311–320.

Kästner, C., Apel, S., ur Rahman, S., Rosenmüller, M., Batory, D. and Saake, G. (2009) On the impact of the optional feature problem: Analysis and case studies. In: *Proceedings of International Software Product Line Conference (SPLC)*. 181–190.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. and Irwin, J. (1997) Aspect-oriented programming. In: ECOOP. *Springer Lecture Notes in Computer Science* **1241** 220–242.

Krueger, C. (2002) Eliminating the adoption barrier. *IEEE Software* **19** (4) 29–31.

Liu, J., Batory, D. S. and Lengauer, C. (2006) Feature oriented refactoring of legacy applications. In: *ICSE* 112–121.

Lopez-Herrejon, R., Batory, D. and Cook, W. (2005) Evaluating support for features in advanced modularization technologies. In: ECOOP. *Springer Lecture Notes in Computer Science* **3586** 169–194.

Loughran, N. and Rashid, A. (2004) Framed aspects: supporting variability and configurability for AOP. In: ICSR. *Springer Lecture Notes in Computer Science* **3107** 127–140.

Mens, T., Taentzer, G. and Runge, O. (2005) Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science* **127** (3) 113–128.

Mezini, M. and Ostermann, K. (2004) Variability management with feature-oriented programming and aspects. In: *SIGSOFT FSE*, ACM 127–136.

Noda, N. and Kishi, T. (2008) Aspect-oriented modeling for variability management. In: *SPLC* 213–222.

Odersky, M. (2007) The Scala Language Specification, version 2.4. *TechnicalReport*, Programming Methods Laboratory, EPFL.

Oldevik, J., Haugen, Ø. and Møller-Pedersen, B. (2009) Confluence in domain-independent product line transformations. In: *FASE* 34–48.

Perrouin, G., Klein, J., Guelfi, N. and Jézéquel, J.-M. (2008) Reconciling automation and flexibility in product derivation. In: *SPLC* 339–348.

Pohl, K., Böckle, G. and van der Linden, F. (2005) *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, Heidelberg.

Prehofer, C. (1997) Feature-oriented programming: A fresh look at objects. In: *ECOOP*, *Springer Lecture Notes in Computer Science* **1241** 419–443.

Schaefer, I. (2010) Variability modelling for model-driven development of software product lines. In: *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)* 85–92.

Schaefer, I., Bettini, L., Bono, V., Damiani, F. and Tanzarella, N. (2010) Delta-oriented programming of software product lines. In: SPLC. *Springer Lecture Notes in Computer Science* **6287** 77–91.

Schaefer, I., Bettini, L. and Damiani, F. (2011) Compositional type-checking for delta-oriented programming. In: *International Conference on Aspect-oriented Software Development (AOSD'11)* 43–56.

Schaefer, I. and Damiani, F. (2010) Pure delta-oriented programming. In: *FOSD (2010)* 49–56.

Schaefer, I., Worret, A. and Poetzsch-Heffter, A. (2009) A model-based framework for automated product derivation. In: *Proceedings of workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009)* 14–21.

Smaragdakis, Y. and Batory, D. (2002) Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology* **11** (2) 215–255.

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

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

Völter, M. and Groher, I. (2007) Product line implementation using aspect-oriented and model-driven software development. In: *SPLC* 233–242.

Zhang, H. and Jarzabek, S. (2003) An XVCL-based approach to software product line development. In: *Software Engineering and Knowledge Engineering* 267–275.