Variability Modelling in the ABS Language*

Dave Clarke[1], Radu Muschevici[1], José Proença[1],
Ina Schaefer[2], and Rudolf Schlatte[3]

[1] IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium
[2] University of Braunschweig, Germany
[3] University of Oslo, Norway

Abstract. The HATS project aims at developing a model-centric methodology for the design, implementation and verification of highly configurable systems, such as software product lines, centred around the Abstract Behavioural Specification (ABS) modelling Language. This article describes the variability modelling features of the ABS Modelling framework. It consists of four languages, namely, μTVL for describing feature models at a high level of abstraction, the Delta Modelling Language DML for describing variability of the 'code' base in terms of delta modules, the Product Line Configuration Language CL for linking feature models and delta modules together and the Product Selection Language PSL for describing a specific product to extract from a product line. Both formal semantics and examples of each language are presented.

1 Introduction

Software systems are central for the infrastructure of modern society. To justify the huge investment made to build such systems, they need to live for decades. This requires that the software is highly adaptable; software systems must support a high degree of variability to accommodate a range of requirements and deployment scenarios, and to allow these to change over time. A major challenge facing software construction is addressing high adaptability combined with trustworthiness. A limitation of current development practices is the missing rigour of models and property specification. Without a formal notation for distributed, component based systems, it is impossible to achieve automated consistency checking, security enforcement, generation of trustworthy code, etc. Furthermore, it does not suffice to simply extend current formal approaches.

Work done in the HATS project will make software product line engineering (SPLE) [30] into a more rigorous approach. SPLE addresses the development of software products sharing a number of commonalities, while differing in other aspects. Fig. 1 depicts the workflow in SPLE. Product variability can be expressed by *features*, which are user-visible product characteristics. The set of products is represented by a feature model [22, 4], describing valid combinations of features. Given a set of software artefacts associated to these features, a final product is built by selecting the desired features and combining the artefacts.

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

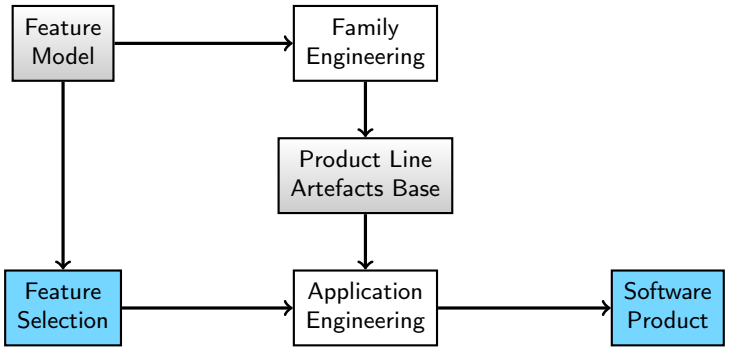

Fig. 1. Stages of product line development

The HATS project aims at developing a model-centric methodology for the design, implementation and verification of highly configurable systems, such as software product lines, that have high demands on dependability and trustworthiness. The HATS methodology is centred around the Abstract Behavioural Specification language (ABS) and its accompanying tool suite[4] that allows precise specification and analysis of the abstract behaviour and variability of highly configurable software systems. ABS is designed to fill the niche between design-oriented formalisms such as UML [27] and feature description language FDL [13], on one hand, and implementation-oriented formalisms such as Spec# [2] and JML [8], on the other hand. In this paper, we focus on the linguistic concepts of the ABS to represent anticipated system variability.

ABS [15] comprises a core language, called *Core ABS*, with specialised language extensions addressing system variability. Core ABS is a class-based, object-oriented language based on the active object concurrency model of Creol [21, 6], which uses *asynchronous method calls* and *cooperative multi-tasking* between concurrent object groups of one or more ABS objects that share a computation resource; i.e., there can be at most one activity running inside the group.

The full ABS modelling framework extends Core ABS by four specialised languages to represent variability of Core ABS models. The *micro textual variability language* (μTVL), based on TVL [7, 11], expresses variability via feature models (Section 2). The *Delta Modelling Language* (DML), based on the concept of delta modelling [32], expresses the code-level variability of ABS models (Section 3). In delta modelling, a set of products is described by an initial core module, which is a Core ABS model, together with a set of product deltas specifying transformations to this core module (additions, removals, or modifications). The *Product Line Configuration Language* (CL) defines the relationship between the feature model and product deltas and thus forms the top-level specification of a product line of Core ABS models (Section 4). The *Product Selection Language* (PSL) represent the actual products by providing a selection of the product features

[4] http://tools.hats-project.eu/

and their attributes along with initialisation code for the product (Section 5). Fig. 2 depicts the relationship between these languages. The process of generating a software product from a software product line specification is explained in Section 6. Related work is presented in Section 7 and Section 8 concludes.

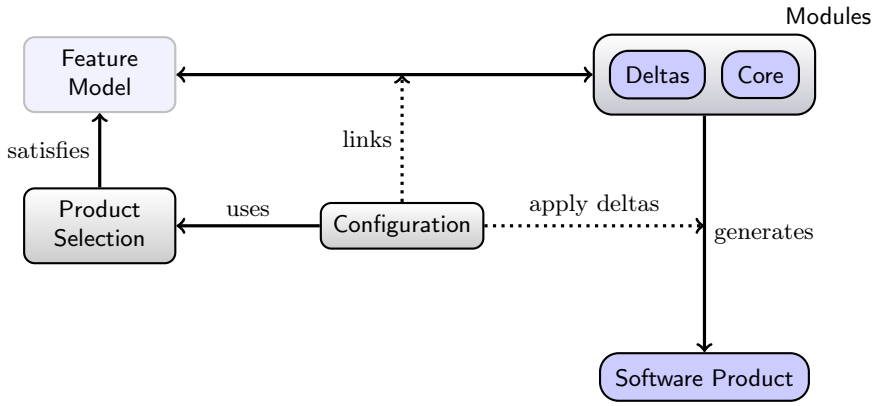

Fig. 2. Relationship Between Ingredients

2 Feature modelling

This section introduces the μTVL text-based feature modelling language, pronounced either *micro textual variability language* or simply *mu tee vee ell*, an extended subset of TVL [7, 11]. TVL was developed at the University of Namur, Belgium, to serve as a reference language for specifying feature models. It is textual, as opposed to diagrammatic, and aims to be scalable, concise, modular, and comprehensive, and thus, serves as a suitable starting point for our purposes. A feature model is represented textually as a tree of nested features, each with a collection of boolean or integer attributes. Additional cross-tree dependencies can also be expressed in the feature model.

μTVL is designed to be deliberately smaller than TVL in order to capture the essential feature modelling requirements and to simplify the manipulation of feature models. The simplification allows reducing a number of semantic constraints imposed by TVL to syntactic constraints. μTVL enables a feature model with multiple roots (hence, multiple trees) to express orthogonal variability [30], which is useful for expressing application models and platform models in an orthogonal fashion (even in different files). Support for attributes of enumerated types have been dropped, but our tools support checking of satisfiability of integer attributes. Finally, in μTVL features can only be extended (in *FeatureExtension* clauses) by adding new constraints, but not by introducing new features. Even though TVL syntax is used (with a few variations), the tools for

μTVL have been developed from scratch and integrated with the ABS language tool suite.

2.1 Concrete Syntax

The grammar of μTVL is given in Fig. 3. Text in `monospace` denote terminal symbols. Assume the presence of two global sets: FID of feature names and AID of attribute names.

$$
\begin{array}{rcl}
\mathit{Model} & ::= & (\texttt{root}\ \mathit{FeatureDecl})^*\ \mathit{FeatureExtension}^* \\
\mathit{FeatureDecl} & ::= & \mathsf{FID}\ [\texttt{\{}\ [\mathit{Group}]\ \mathit{AttributeDecl}^*\ \mathit{Constraint}^*\ \texttt{\}}] \\
\mathit{FeatureExtension} & ::= & \texttt{extension}\ \mathsf{FID}\ \texttt{\{}\ \mathit{AttributeDecl}^*\ \mathit{Constraint}^*\texttt{\}} \\
\mathit{Group} & ::= & \texttt{group}\ \mathit{Cardinality}\ \texttt{\{}\ [\texttt{opt}]\ \mathit{FeatureDecl}\texttt{,}\ ([\texttt{opt}]\ \mathit{FeatureDecl})^*\ \texttt{\}} \\
\mathit{Cardinality} & ::= & \texttt{allof} \mid \texttt{oneof} \mid \texttt{[}n_1\ \texttt{..}\ *\texttt{]} \mid \texttt{[}n_1\ \texttt{..}\ n_2\texttt{]} \\
\mathit{AttributeDecl} & ::= & \texttt{Int}\ \mathsf{AID}\ \texttt{;} \mid \texttt{Int}\ \mathsf{AID}\ \text{in}\ \texttt{[}\ \mathit{Limit}\ \texttt{..}\ \mathit{Limit}\ \texttt{]}\ \texttt{;} \mid \texttt{Bool}\ \mathsf{AID}\ \texttt{;} \\
\mathit{Limit} & ::= & n \mid * \\
\mathit{Constraint} & ::= & \mathit{Expr}\ \texttt{;} \mid \texttt{ifin:}\ \mathit{Expr}\ \texttt{;} \mid \texttt{ifout:}\ \mathit{Expr}\ \texttt{;} \\
 & & \mid \texttt{require:}\ \mathsf{FID}\ \texttt{;} \mid \texttt{exclude:}\ \mathsf{FID}\ \texttt{;} \\
\mathit{Expr} & ::= & \texttt{True} \mid \texttt{False} \mid n \mid \mathsf{FID} \mid \mathsf{AID} \mid \mathsf{FID.AID} \\
 & & \mid \mathit{UnOp}\ \mathit{Expr} \mid \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \mid \texttt{(}\ \mathit{Expr}\ \texttt{)} \\
\mathit{UnOp} & ::= & \texttt{!} \mid \texttt{-} \\
\mathit{BinOp} & ::= & \texttt{||} \mid \texttt{\&\&} \mid \texttt{->} \mid \texttt{<->} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{+} \mid \texttt{-} \mid * \mid \texttt{/} \mid \texttt{\%}
\end{array}
$$

Fig. 3. Grammar of μTVL; n ranges over integers.

Attributes and values in μTVL range either over integers or booleans. The *Model* clause specifies a number of 'orthogonal' root feature models along with a number of extensions that specify additional constraints, typically cross-tree dependencies. The *FeatureDecl* clause specifies the details of a given feature, firstly by giving it a name (FID), followed by a number of possibly optional sub-features, the feature's attributes and any relevant constraints. The *FeatureExtension* clause specifies additional constraints and attributes for a feature. This is particularly useful for specifying constraints that do not fit into the tree structure given by the root feature model. The *Cardinality* clause describes the number of elements of a group that may appear in a result. The *AttributeDecl* clause specifies the declaration of both integer (bounded or unbounded) and boolean attributes of features.

The *Constraint* clause specifies constraints on the presence of features and on attributes. An **ifin** constraint is only applicable if the current feature is selected. Similarly, an **ifout** constraint is only applicable if the current feature is not selected. A **require** clause specifies that the current feature requires some other feature, whereas **exclude** expresses the mutual incompatibility between the current feature and some other feature. The *Expr* clause expresses a boolean constraint over the presence of features and attributes, using standard boolean

and arithmetic operators. Features are referred to by identity (FID). Attributes are referred to either using an unqualified name (AID), for in scope attributes, or using a qualified name (FID.AID) for attributes of other features.

Example 1. The following is a feature model of a *multi-lingual Hello World* product line, which describes software that can output "Hello World" in multiple languages some number of times.

```
root MultiLingualHelloWorld {
  group allof {
    Language {
      group oneof { English, Dutch, German }
    },
    opt Repeat {
      Int times in [0..1000];
      ifin: times > 0;
} } }
```

```
extension English {
  ifin: Repeat ->
        (Repeat.times >= 2 &&
         Repeat.times <= 5);
}
```

The *multi-lingual Hello World* product line in the example above has two main features, *Language* and *Repeat*, under the root feature and joined with the **`allof`** combinator. The *Language* feature requires one out of three possible features: *English*, *Dutch*, or *German*. The *Repeat* feature is optional, it has no associated sub-features, and it has an attribute *times* which ranges between 0 and 1000, with an added condition that it must be strictly greater than 0. In this example an extension for the *English* feature is given. When the *English* and the *Repeat* features are present, the attribute `times` must be between 2 and 5, inclusive.

2.2 Abstract Syntax

The abstract syntax tree for μTVL programs is presented in Fig. 4, where $f \in \mathsf{FID}$, $a \in \mathsf{AID}$, and $n \in \mathsf{Int}$. The translation from the concrete tree to the abstract tree is straightforward and hence omitted. Local attribute names are expanded to fully qualified names. Bounds are placed on all integer attributes. The semantics of μTVL is given as the solutions of the integer constraints defined inductively on the abstract syntax of feature models.

2.3 Semantics

The semantics of a feature model in μTVL are defined by translation into constraints over integers whose solutions correspond to valid feature and attribute selections. Boolean variables are treated as integers in the standard manner: 0 corresponds to false, and 1 to true. The function $[\![\ ]\!]$ encoding feature model M as an integer constraint is given in Fig. 5. The notation $\overline{x}$ represents a sequence of elements $x_1 \cdots x_n$. Within the context of a given feature f, function $[\![\ ]\!]_f$ translates constraints relative to that feature. In the translation, $f^\dagger$ is a unique name based on name f. If f is an optional feature, $f^\dagger$ can freely be set to 1 to count the optional feature, even when f is absent. For example, when dealing with an

M	$::= F^*$	feature model	C	$::= e \mid$ `ifin` $e \mid$ `ifout` $e \mid$	
F	$::= f\ [G]\ A^*\ C^*$	feature (extension)		$\mid$ `require` $f \mid$ `exclude` f	constraint
G	$::= c\ N^*$	group	lt	$::=$ `true` $\mid$ `false`	
N	$::=$ `opt` $F \mid$ `mand` F	feature node		$\mid n \mid f \mid f.a$	literal or variable
c	$::=$ `allof` $\mid$ `min` n		U	$::=$ `neg` $\mid$ `not`	unary operator
	$\mid$ `rng` $n\ n$	cardinality	B	$::=$ `or` $\mid$ `and` $\mid$ `implies`	
A	$::= f.a\ T$	attribute declaration		$\mid$ `equiv` $\mid$ `eq` $\mid$ `neq`	
T	$::=$ `bool` $\mid$ `int` $L\ L$	type and domain		$\mid$ `lt` $\mid$ `gt` $\mid$ `lteq`	
L	$::= * \mid n$	domain limit		$\mid$ `gteq` $\mid$ `plus` $\mid$ `minus`	
e	$::= lt \mid U\ e \mid B\ e\ e$	expression		$\mid$ `mult` $\mid$ `div` $\mid$ `mod`	binary operator

Fig. 4. Abstract syntax of μTVL.

allof constraint, it is required that all children are present; some may however be optional, so as far as the **allof** constraint is concerned, optional children are counted, though the corresponding features may not be included. Expressions e are encoded into constraints, denoted ϕ_e. Their encoding is straightforward and therefore omitted (see [11]). Boolean operations are mapped to a conjunctive set of integer operations over the values 0 and 1 where, for example, $a \rightarrow b$ is a shorthand for $a \leq b$. Finally, we assume a lower bound *MIN* and an upper bound *MAX* on the values of integer variables.

Given a feature model *FM* in μTVL, the set of solutions of the integer constraints $[\![FM]\!]$ provides our semantics for *FM*. Such a solution will specify values for all attributes even when the corresponding feature is not selected. Such assignments should have no effect.

The semantics also enforce that each feature is selected either zero or one times, in spite of cardinality conditions which may appear to allow more instances of a feature. Cardinality conditions specify the number of selected sub-features from a group. Note that optional features can only appear under the **allof** cardinality; otherwise there would be a fragile interaction between cardinality conditions and optional features [5].

Example 2. Below is the encoding into integer constraints of the Hello World feature model introduced in Example 1.

$$
\begin{aligned}
&0 \leq \texttt{MultiLingualHelloWorld} \leq 1 \ \wedge \\
&\texttt{Language} \rightarrow \texttt{MultiLingualHelloWorld} \ \wedge \ \texttt{Repeat}^{\dagger} \rightarrow \texttt{MultiLingualHelloWorld} \ \wedge \\
&\texttt{Language} + \texttt{Repeat}^{\dagger} = 2 \ \wedge \\
&0 \leq \texttt{Language} \leq 1 \ \wedge \\
&\texttt{English} \rightarrow \texttt{Language} \ \wedge \ \texttt{Dutch} \rightarrow \texttt{Language} \ \wedge \ \texttt{German} \rightarrow \texttt{Language} \ \wedge \\
&1 \leq \texttt{English} + \texttt{Dutch} + \texttt{German} \leq 1 \ \wedge \\
&0 \leq \texttt{English} \leq 1 \ \wedge \ 0 \leq \texttt{Dutch} \leq 1 \ \wedge \ 0 \leq \texttt{German} \leq 1 \ \wedge \\
&0 \leq \texttt{Repeat}^{\dagger} \leq 1 \ \wedge \\
&\texttt{Repeat} \rightarrow \texttt{Repeat}^{\dagger} \ \wedge \\
&0 \leq \texttt{Repeat} \leq 1 \ \wedge \ 0 \leq \texttt{Repeat.times} \leq 1000 \wedge \texttt{Repeat.times} > 0 \ \wedge \\
&\texttt{English} \rightarrow (\texttt{Repeat} \rightarrow (\texttt{Repeat.times} \geq 2 \wedge \texttt{Repeat.times} \leq 5)).
\end{aligned}
$$

$$
\begin{aligned}
[\![\overline{F}]\!] &= \textstyle\bigwedge_{x\in\overline{F}} [\![x]\!] \\
[\![f\ [G]\ \overline{A}\ \overline{C}]\!] &= (0 \le f \le 1) \wedge [\![[G]\,]\!]_f \wedge [\![\overline{A}]\!] \wedge [\![\overline{C}]\!]_f \\
[\![\texttt{allof}\ \overline{N}]\!]_f &= \mathit{tree}(f, \overline{N}) \wedge \textstyle\sum \overline{N} = \#\overline{N} \wedge [\![\overline{N}]\!] \\
[\![(\texttt{min}\ n)\ \overline{N}]\!]_f &= \mathit{tree}(f, \overline{N}) \wedge n \le \textstyle\sum \overline{N} \wedge [\![\overline{N}]\!] \\
[\![(\texttt{rng}\ n_1\ n_2)\ \overline{N}]\!]_f &= \mathit{tree}(f, \overline{N}) \wedge n_1 \le \textstyle\sum \overline{N} \le n_2 \wedge [\![\overline{N}]\!] \\
[\![\texttt{opt}\ (f\ [G]\ \overline{A}\ \overline{C})]\!] &= f \to f^\dagger\ \wedge\ [\![f\ [G]\ \overline{A}\ \overline{C}\,]\!] \\
[\![\texttt{mand}\ F]\!] &= [\![F]\!] \\
[\![f.a\ \texttt{int}\ L_1\ L_2]\!] &= \mathit{val}_{min}(L_1) \le f.a \wedge f.a \le \mathit{val}_{max}(L_2) \\
[\![f.a\ \texttt{bool}]\!] &= 0 \le f.a \le 1 \\
[\![e]\!] &= \phi_e
\end{aligned}
\qquad
\begin{aligned}
[\![\texttt{ifin}\ e]\!]_f &= f \to [\![e]\!] \\
[\![\texttt{ifout}\ e]\!]_f &= \neg f \to [\![e]\!] \\
[\![\texttt{require}\ f']\!]_f &= f \to f' \\
[\![\texttt{exclude}\ f']\!]_f &= \neg(f \wedge f')
\end{aligned}
$$

$$
[\![\ [X]\ ]\!] = \begin{cases} [\![X]\!] \text{ if } X \text{ is present} \\ \mathit{true} \text{ otherwise} \end{cases}
\qquad
\begin{aligned}
\mathit{feat}(\texttt{opt}(f\ \_\ \_\ \_)) &= f^\dagger \\
\mathit{feat}(\texttt{mand}(f\ \_\ \_\ \_)) &= f \\
\mathit{val}_x(n) &= n \\
\mathit{val}_{min}(*) &= \mathit{MIN} \\
\mathit{val}_{max}(*) &= \mathit{MAX}
\end{aligned}
$$
$$
\begin{aligned}
\#(N_1 \cdots N_n) &= n \\
\textstyle\sum(N_1 \cdots N_n) &= \mathit{feat}(N_1) + \cdots + \mathit{feat}(N_n) \\
\mathit{tree}(f, N_1 \cdots N_n) &= \textstyle\bigwedge_{1\le i\le n}\ \mathit{feat}(N_i) \to f
\end{aligned}
$$

Fig. 5. Semantics of μTVL.

Every declaration of a new feature or attribute x is converted into a constraint of type $\mathit{min} \le x \le \mathit{max}$, and, in the case of booleans and feature names, $\mathit{min} = 0$ and $\mathit{max} = 1$. The tree structure of the feature model is captured by implications between the children and their parents, as shown in the second line of Example 2. The optional feature `Repeat` is split into two variables: `Repeat` and `Repeat`†. The latter is used only to address the cardinality of the parent `MultiLingualHelloWorld`, and they are connected by the implication `Repeat` $\to$ `Repeat`†, similar to how child features are related to their parent. Cardinalities are encoded as constraints that add the 0-1-integer value of the feature variables and check whether they belong to a specific domain, as shown in the third and seventh line of the example. Constraints over attributes are simply interpreted as integer constraints.

3 Delta Modelling

Delta-oriented programming was introduced by Schaefer et al. [32, 34, 33] as a novel programming language approach for software-based product lines, and as an direct alternative to feature-oriented programming [3]. Both approaches aim at automatically generating software products for a given feature selection by providing a flexible and modular technique to build different products that share

common code. In feature-oriented programming, software modules are associated to features, and product generation consists of composing the modules for a feature selection. In delta-oriented programming [32], *application conditions* over the set of features and their attributes, are associated with modules of program modifications (add, remove or modify code), called delta modules. The collection of applicable delta modules is given by the application conditions that are true for a particular feature and attribute selection. By not associating the delta modules directly with features, a degree of flexibility is obtained, resulting in better reuse of code and the ability to resolve conflicts caused by deltas modifying the code base in incompatible ways [10]. The flexibility offers benefits for managing the evolution of product lines, by allowing versions to be implemented using software deltas.

The implementation of a software product line in delta-oriented programming [32] is divided into a *core module* and a set of *delta modules*. The core module consists of the classes that implement a complete product of the corresponding product line. Delta modules describe how to change the core module to obtain new products. The choice of which delta modules to apply is based on the selection of desired features for the final product. Schaefer et al. described and implemented delta-oriented programming for *Java* [32], introducing the programming language DeltaJava. This language has strongly influenced our design, though we further separate deltas from features by moving application conditions out of deltas and into a product line configuration language, as pursued in [34, 33]. Delta modelling is included in the ABS language to implement variability at the source code level of abstraction.

3.1 Syntax

Figure 6 specifies the ABS syntax related to delta modelling. Nonterminals written in purple (gray) refer to core ABS symbols, whose intended meaning should be immediate.

The *DeltaDecl* clause specifies the syntax of delta modules, consisting of an unique identifier, a list of parameters and a body containing a sequence of class and interface modifiers. The *ClassOrIfaceModifier* clause describes the syntax of modifications at the level of classes and interfaces. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The *ImplModifiers* clause describes how to modify the interfaces a class implements or an interface extends, either by adding new or removing existing interfaces.

The *Modifier* clause specifies the modifications that can occur within a class or interface body. These include (where relevant) adding and removing fields and method signatures (from interfaces), and modifying methods, which amounts to replacing a method with a new one, but enabling the original method to be called using the **`original`** keyword. The aim of **`original`** is to enable the method being replaced to be called from the delta module that replaces it. This is implemented by renaming the original method, and replacing the call via keyword **`original`** with a call to the renamed method. The semantics of calling **`original()`** as

```
DeltaDecl ::= delta TypeId [DeltaParams] { ClassOrIfaceModifier* }
ClassOrIfaceModifier ::= adds ClassDecl
                       | modifies class TypeName ImplModifier* { Modifier* }
                       | removes class TypeName ;
                       | adds InterfaceDecl
                       | modifies interface TypeName ImplModifier* { Modifier* }
                       | removes interface TypeName ;

ImplModifier ::= adds TypeName
               | removes TypeName

Modifier ::= adds FieldDecl
           | removes FieldDecl
           | adds MethDecl
           | modifies MethDecl
           | removes MethSig

DeltaParams ::= ( DeltaParam ( , DeltaParams )* )
DeltaParam ::= Identifier HasCondition*
             | Type Identifier

HasCondition ::= hasField FieldDecl
               | hasMethod MethSig
               | hasInterface TypeName
```

Fig. 6. ABS Grammar: Delta Modules.

shown in the above example are essentially the same as `Super()` from feature-oriented programming [3], and `proceed` from context-oriented programming [19], and similar to ordinary `super` calls in standard object-oriented languages, as well as `around` advice from aspect-oriented programming [23], except without quantification.

In contrast to deltas presented in the literature [32, 34, 33], delta modules in the HATS ABS language can be parameterised both by attribute values, which ultimately flow from the feature model selection, and by class names, to enable the application of a single delta module in more than one circumstance. Finally, the *HasCondition* describes constraints on class arguments to which a delta may be applied. These constraints consist of descriptions of the methods and fields such a class implements and any interfaces it is expected to have.

Example 3. Following is the implementation of the Hello World product line with the feature model shown in Example 1. Delta modules specify the variable behaviour.

```
interface Greeting {
    String sayHello();
}
class Greeter implements Greeting {
    String sayHello() {
        return "Hello world";
}   }
class Application {
    String s = "";
    Unit run() {
        Greeting bob;
        bob = new Greeter();
        s = bob.sayHello();
}   }
delta Nl {
    modifies Greeter {
        modifies String sayHello() {
            return "Hallo wereld";
} } }
```

```
delta De {
    modifies Greeter {
        modifies String sayHello() {
            return "Hallo Welt";
} } }

delta Rpt (Int times) {
    modifies Greeter {
        modifies String sayHello() {
            String result = "";
            Int i = 0;
            while (i < times) {
                String orig = original();
                result = result + " " + orig;
                i = i + 1;
            }
            return result;
} } }
```

In the example above the interface `Greeting` and the classes `Greeter` and `Application` form the core module of the implementation, written in the core ABS language. There are three delta modules: `Nl`, `De`, and `Rpt`. The delta module `De` has a single class modifier for `Greeter`, which in turn has a single method modifier. This method modifier replaces the method `sayHello` to return the German text "Hallo Welt". The delta module `Rpt` has a single parameter for the number of times that the greeting should be repeated. It replaces the method `sayHello()` inside `Greeter` with new ABS code, allowing the original method to be called via **`original`**`()`.

Example 4. The following diagram illustrates both the use of parameters and of the **`original`** keyword. A parameterised delta, such as `Rpt`, must have its arguments provided before it can be applied. The arguments are substituted into the body of the delta module prior to application.

```
class Greeter {
    String sayHello() {
        return "Hello";
    }
}
```

`Rpt(10)`

```
class Greeter {
    String original_sayHello() {
        return "Hello";
    }
    String sayHello() {
        String result = "";
        Int i = 0;
        while (i < 10) {
            String orig = original_sayHello();
            result = result + " " + orig;
            i = i + 1;
        }
        return result;
}   }
```

3.2 Formal Semantics

Applying a delta module Δ to a core ABS program P yields a new core ABS program. Thus a product is constructed by successively applying delta modules, one at a time, to a core module. This section presents a formal semantics of

delta modules based on the more abstract presentation of Clarke et al. [10]. That work also describes the composition of delta modules with each other, which is essential for reasoning about conflicting delta modules, but this feature is elided from the current presentation. ABS programs, classes and delta modules will be represented in terms of finite maps from identifiers to the corresponding contents of the program, class, or delta module, in order to more cleanly present the semantics. The semantics only describes the modifications of methods; dealing with fields and so forth is a straightforward extension. Parameters are omitted. These will be treated when dealing with configurations in Section 4.

Let *Identifier* be the set of identifiers, let *MethBody* be the set of method bodies, including the parameter and return types, and let *MethBodyWrap* be the set of method bodies with an explicit call to **`original`**. In the following domains, Replace, Update, and Remove are used to tag the various branches of sum data types. Finally, let *Error* denote that an error has occurred. Errors occur if one attempts to wrap a method that is not present. Other irregularities, such as attempting to update a class that is not present, can be given a sensible semantics, so long as no wrapping occurs.

$$
\begin{aligned}
\mathit{Program} &= \mathit{Identifier} \rightharpoonup \mathit{ClassBody} \\
\mathit{ClassBody} &= \mathit{Identifier} \rightharpoonup \mathit{MethBody} \\
\mathit{Delta} &= \mathit{Identifier} \rightharpoonup \mathit{DeltaBody} \\
\mathit{DeltaBody} &= \mathsf{Replace}\ (\mathit{Identifier} \rightharpoonup \mathit{MethBody}) \\
&\quad \uplus\ \mathsf{Update}\ (\mathit{Identifier} \rightharpoonup (\mathit{MethBody} \uplus \mathit{MethBodyWrap} \uplus \mathsf{Remove})) \\
&\quad \uplus\ \mathsf{Remove}
\end{aligned}
$$

A program is a map from class names to classes, which themselves are collections of named method bodies. A delta module is a map from class names to delta bodies, which consist of three different types of modification: Replace either adds or replaces the class with the specified contents; Update modifies a class in place, where the three elements within an update clause correspond to replacing a method with a new body from *MethBody*, wrapping the method with a body from *WrapMethBody* or removing the method; and finally, Remove denotes the removal of the class.

Notation 1 *Let $f : X \rightharpoonup Y$ denote a partial function from X to Y. If $f(x)$ is undefined for $x \in X$, write $f(x) = \bot$, where $\bot \notin Y$. For set A, let $A_\bot$ denote $A \cup \{\bot\}$, where $\bot \notin A$. We freely shift between partial functions $X \rightharpoonup Y$ and functions $X \to Y_\bot$. If $\odot : A_\bot \times B_\bot \to C_\bot$, define the lifting of $\odot$ to partial functions over index set I as*

$$
\begin{aligned}
- \mathbin{\overline{\odot}} - &: (I \rightharpoonup A) \times (I \rightharpoonup B) \to (I \rightharpoonup C) \\
(f \mathbin{\overline{\odot}} g)(i) &= f(i) \odot g(i), \qquad \text{where } i \in I
\end{aligned}
$$

Given class update $f : \mathit{Identifier} \rightharpoonup (\mathit{MethBody} \uplus \mathit{MethBodyWrap} \uplus \mathit{Remove})$*, define function* $f^* : \mathit{Identifier} \rightharpoonup (\mathit{MethBody} \uplus \mathit{Error})$ *as follows. For* $i \in \mathit{Identifier}$*:*

$$f^*(i) = \begin{cases} \bot & \text{if } f(i) = \mathit{Remove} \\ f(i) & \text{if } f(i) \in \mathit{MethBody} \\ \mathit{Error} & \text{if } f(i) \in \mathit{MethBodyWrap}. \end{cases}$$

Notation 2 *In the following definition, the notation* $w[\ ]$ *denotes a wrapper method from* MethBodyWrap*, where the hole* $[\ ]$ *denotes that the original method is unknown. Notation* $w[b]$ *denotes the wrapping of method body* b *with wrapper* w*, thus the* `original` *call can be successfully bound. The resulting method* $w[b]$ *is considered to be an element of* MethBody*.*

Definition 1 (Delta module application). *The application of a delta module to a program is specified by the following functions:*

$$\begin{aligned} \mathit{apply} &: \mathit{Delta} \times \mathit{Program} \rightharpoonup \mathit{Program} \\ \mathit{apply}(d, p) &= d \mathbin{\overline{\odot_c}} p \end{aligned}$$

where

$$\begin{aligned} - \odot_c - &: \mathit{DeltaBody}_\bot \times \mathit{ClassBody}_\bot \to \mathit{ClassBody}_\bot \\ \bot \odot_c x &= x \\ (\mathit{Replace}\ g) \odot_c \_ &= g \qquad (\mathit{Update}\ f) \odot_c \bot = f^* \\ \mathit{Remove} \odot_c \_ &= \bot \qquad (\mathit{Update}\ f) \odot_c h = f \mathbin{\overline{\odot_m}} h \end{aligned}$$

and

$$\begin{aligned} - \odot_m - &: (\mathit{MethBody} \uplus \mathit{MethBodyWrap} \uplus \mathit{Remove})_\bot \times \mathit{MethBody}_\bot \\ &\to (\mathit{MethBody} \uplus \mathit{Error})_\bot \\ \bot \odot_m x &= x \\ w[\ ] \odot_m b &= w[b] \qquad m \odot_m \_ = m \\ \mathit{Remove} \odot_m \_ &= \bot \qquad w[\ ] \odot_m \bot = \mathit{Error} \end{aligned}$$

where $m \in \mathit{MethBody}$ *and* $w[\ ] \in \mathit{MethBodyWrap}$*.*

Notation 3 *If* $m \in \mathit{Identifier}$ *then* $w[m]$ *denotes the wrapper with each call to* **`original`** *replaced by a call to* m*.*

In our implementation the method body is not inlined. Instead, if the resulting class C has an element $m \mapsto w[b] \in C$, the following post-processing steps are performed before applying another delta module:

1. generate a fresh method name $m' \notin C$,
2. remove $m \mapsto w[b]$ from C, and
3. add $m \mapsto w[m']$ and $m' \mapsto b$ to C.

Example 4 illustrated this process with concrete code. The modified method in that example is `sayHello()`. Before replacing the method, it was renamed to a fresh name such as `original_sayHello()`. The new method was then added to the class, with its body modified so that `original` is replaced by the renamed

Configuration ::= **productline** *TypeId* { *Features* ; *Deltas* }

Features ::= **features** FID (, FID)*

DeltaClauses ::= *DeltaClause* (, *DeltaClause*)*
DeltaClause ::= **delta** *DeltaSpec* [*AfterCondition*] [*ApplicationCondition*] ;

DeltaSpec ::= *TypeName* [(*DeltaArgs*)]
DeltaArgs ::= *DeltaArg* (, *DeltaArg*)*
DeltaArg ::= FID | FID.AID | *DataExp*

AfterCondition ::= **after** *TypeName* (, Name)*
ApplicationCondition ::= **when** *Expr*

Fig. 7. Product Line Configuration Grammar.

method's name `original_sayHello`. The stipulation that the method name is fresh is required in the case that multiple delta modules are applied to the same class, each wrapping the same method. In such a case, the first renaming would result in method `original_sayHello`, for example, the second in name `original_sayHello2`, and so forth.

4 Product Line Configuration

This section describes the product line configuration language CL which links feature models specified in μTVL (Section 2) with delta modules (Section 3), to specify the variability in a product line. This approach is similar to the product line specification proposed in delta-oriented programming [34, 33].

A product line configuration consists of a set of features assumed to exist and a set of *delta clauses*. Each delta clause specifies a delta and the conditions required for its application, propositional formulas over the set of known features and attributes called *application conditions*, and a partial ordering relation with respect to other deltas. When the propositional formula holds for a given product, the delta is said to be active. The partial order states which deltas, when active, should be applied before the current delta.

4.1 Syntax

The syntax of the product line configuration language is given in Fig. 7. The *Configuration* clause specifies the name of the product line, the set of features it implements, and the set of delta modules used to implement those features. The feature names are included so that certain simple self-consistency checks can be performed. The *DeltaClause* clause is used to specify each delta module, linking it to the feature model. Each *DeltaClause* has a *DeltaSpec*, specifying its name and its parameters, an *AfterCondition*, specifying the delta modules that the current delta must be applied after, and an *ApplicationCondition*, specifying an arbitrary predicate over the feature and attribute names (see Fig. 3) that describes when the given delta module is included in the product line.

Example 5. The Hello World product line is configured, connecting the features and attributes defined in the feature model to delta modules.

```
productline MultiLingualHelloWorld {
    features English, German, Dutch, Repeat;

    delta Rpt(Repeat.times) after De, Nl when Repeat;
    delta De when German;
    delta Nl when Dutch;
}
```

The example above first names the set of features from the feature model in Example 1 used to configure this product line. The **`delta`** clauses link each delta module to the feature model through an application condition (**`when`** clause); in this case, a delta module is applied simply when the specified feature is selected (e.g. "`De` **`when`** `German`"). There is no delta module corresponding to the feature `English`, as the core module provides support for the English language by default. In addition, `Rpt` has to be applied **`after`** `De` and `Nl`. `Rpt`'s argument is `Repeat.times`, the `times` attribute feature `Repeat`; its value (defined by product selection, see Section 5) is propagated to the `Rpt` delta.

4.2 Semantics

A CL script specifies how the feature model relates to the delta modules that are to be applied to the core module. It does so by specifying the parameters and application conditions for each delta module, and an ordering on the deltas.

Each delta module referred to in a configuration file is modelled by an element of the following type:

$$\mathit{Delta} \times \mathit{Params} \times \mathit{AppCondition}$$

where *Delta* is the semantic domain of delta module bodies, defined in Section 3.2,

$$\mathit{Params} \;=\; \mathit{Var} \rightharpoonup \mathsf{FID} \uplus (\mathsf{FID} \times \mathsf{AID}) \uplus \mathsf{Int}$$

models the substitution of actual parameters, which may be attributes or constants, defined in the CL script with the formal parameters of the corresponding delta module, and *AppCondition* is the syntactic category of application conditions. Class parameters to delta modules are not modelled.

A configuration script can be modelled as a partial order over the declared delta modules (with their parameters and application conditions), where the partial order is determined by the reflexive, transitive closure of the **`after`** clauses. This is given by the following domain, where $\mathit{PO}(-)$ denotes the collection of all partial orders over a given set.

$$\mathit{Config} = \mathit{PO}(\mathit{Delta} \times \mathit{Params} \times \mathit{AppCondition})$$

The semantics of a configuration script $\mathit{conf} \in \mathit{Config}$ is a function of type

$$[\![\mathit{conf}]\!]\_ : \mathit{ProductSelection} \to \mathcal{P}(\mathit{Delta}^*)$$

which maps a product selection—the interpretation of a PSL script (see Section 5.2)—to the delta modules to apply, in the order they should be applied. Note that many orders may exist if the **after**-order is underspecified. A product selection is an assignment from feature names to true or false (1 or 0) and from attributes to values, given by the domain *ProductSelection*:

$$\mathit{ProductSelection} = (\mathsf{FID} \uplus (\mathsf{FID} \times \mathsf{AID})) \rightharpoonup \mathsf{Int}$$

We now develop the ingredients making up function $[\![\mathit{conf}]\!]\_$.

Firstly, assume that a notion of substitution exists for delta modules, respecting the scoping of variables, to replace parameters with appropriate values:

$$\begin{aligned} \mathit{Subst} &= \mathit{Var} \rightharpoonup \mathsf{Int} \\ \mathit{applySubst} &: \mathit{Subst} \times \mathit{Delta} \to \mathit{Delta} \end{aligned}$$

Next, we define the composition of the parameter specifications of delta modules with a product selection, giving a mapping from formal parameters of delta modules to values (Int), which will be used to refine the delta modules with the configuration parameters specifying in the product selection:

$$\begin{aligned} \circ &: \mathit{ProductSelection} \times \mathit{Params} \to \mathit{Subst} \\ \sigma \circ p &= \{v \mapsto x\sigma \mid v \mapsto x \in p\} \\ \text{where } x\sigma &= \begin{cases} v & \text{if } x \in \mathsf{FID} \uplus (\mathsf{FID} \times \mathsf{AID})) \text{ and } x \mapsto v \in \sigma \\ x & \text{if } x \in \mathsf{Int} \end{cases} \end{aligned}$$

Now the function taking a product selection $\sigma \in \mathit{ProductSelection}$ and giving the collection of delta modules to apply is computing as the composition of the following steps:

1. Select applicable deltas by applying $\mathit{select}\_ : \mathit{Config} \to \mathit{PO}(\mathit{Delta} \times \mathit{Params})$

   $\mathit{select}_\sigma(D, \prec) = (D', \prec|_{D'})$,
   where $D' = \{(d, p) \mid (d, p, \phi) \in D, \sigma \models \phi\}$ and $\prec|_{D'}$ is $\prec$ restricted to D',
   and $\models \subseteq \mathit{ProductSelection} \times \mathit{AppCondition}$ is the satisfaction relation.

2. Specialise deltas using the function $\mathit{specialise} : \mathit{ProductSelection} \times \mathit{PO}(\mathit{Delta} \times \mathit{Params}) \to \mathit{PO}(\mathit{Delta})$

   $\mathit{specialise}_\sigma(D, \prec) = (D', \prec|_{D'})$, where $D' = \{\mathit{applySubst}(\sigma \circ p, d) \mid (d, P) \in D\}$.

3. Order deltas using the function $\mathit{order} : \mathit{PO}(\mathit{Delta}) \to \mathcal{P}(\mathit{Delta}^*)$

   $\mathit{order}((D, \prec)) = \{[d_1, \ldots, d_n] \mid d_1, \ldots, d_n \text{ is a linear extension of } (D, \prec)\}$.

Finally, the semantics of a CL script can be interpreted as a function

$$\begin{aligned} [\![\_]\!]\_ &: \mathit{Config} \times \mathit{ProductSelection} \to \mathcal{P}(\mathit{Delta}^*) \\ [\![\mathit{conf}]\!]_\sigma &= \mathit{order}(\mathit{specialise}_\sigma(\mathit{select}_\sigma(\mathit{conf}))). \end{aligned}$$

Note that this process may be ambiguous when multiple orderings of delta modules are possible. This should be resolved either by adding more elements to the '**after**' order or by introducing conflict-resolving deltas [10].

5 Product Selection

A product selection needed to generate a product from a product line is specified using the *product selection language* (PSL) A product selection states which features are to be included in the product and by sets attributes of those features to concrete values. In addition, some core ABS code is provided to initialise the selected product. As depicted in Fig. 2, a product selection is checked against a μTVL feature model for validity. It is then used by the configuration file to guide the selection and application of deltas during the generation of the final software product.

5.1 Syntax

Fig. 8 specifies the grammar of the ABS product selection language. The *Selection* clause specifies a product by giving it a name, by stating the features and optional attribute assignments that are included in that product, and by specifying an initialisation block. An initialisation block can be any core ABS block, but typically will be a simple call to some already present *main* method. Initialisation blocks are specified in the product selection language to enable product lines with multiple entry points to start execution.

Example 6. Products of the Hello World product line are product selections.

```
// basic product with no deltas
product P1 (English) {
    new Application();
}

// apply delta De
product P2 (German) {
    new Application();
}
```

```
// apply deltas De and Repeat
product P3 (German, Repeat{times=10}) {
    new Application();
}
// apply deltas En and Repeat, but it
// should be refused because "times > 5"
product P4 (English, Repeat{times=6}) {
    new Application();
}
```

In the example above we specify four products: `P1`, `P2`, `P3`, and `P4`. In the case of the product `P1`, the parameter `English` means the product consists of this feature and of the features implied by the constraints over the feature model. In this case the implied features are `Language` and the root `MultiLingualHelloWorld`, according to the model in Example 1. In `P3` and `P4` the parameters also include

$$
\begin{aligned}
\mathit{Selection} &::= \texttt{product}\ \mathit{TypeId}\ \texttt{(}\ \mathit{FeatureSpecs}\ \texttt{)}\ \texttt{\{}\ \mathit{InitBlock}\ \texttt{\}} \\
\mathit{FeatureSpecs} &::= \mathit{FeatureSpec}\ (\texttt{,}\ \mathit{FeatureSpec})^* \\
\mathit{FeatureSpec} &::= \mathsf{FID}\ [\mathit{AttributeAssignments}] \\
\mathit{AttributeAssignments} &::= \texttt{\{}\ \mathit{AttributeAssignment}\ (\texttt{,}\ \mathit{AttributeAssignment})^*\ \texttt{\}} \\
\mathit{AttributeAssignment} &::= \mathsf{AID}\ \texttt{=}\ \mathit{Literal} \\
\mathit{InitBlock} &::= \mathit{Block}
\end{aligned}
$$

Fig. 8. PSL Grammar

attribute values, in these cases assigning a value to the attribute `times` from the feature `Repeat`. The block of ABS code associated to each product provides its initialisation code. Every product in our example instantiates an `Application` object and executes its `run` method.

5.2 Semantics

There are two components of interest in a PSL product selection such as

```
product P (Feature1 {attribute1_1 = value1_1, ...},
           Feature2 {attribute2_1 = value2_1, ...}, ...)
{ InitBlock }
```

- An assigment $\sigma \in \mathit{ProductSelection}$ defined as follows:
  - for each $\texttt{Feature}_i$, $\sigma(\texttt{Feature}_i) = 1$.
  - for each $\texttt{attribute}_{i,j} = \texttt{value}_{i,j}$ clause in $\texttt{Feature}_i$, $\sigma(\texttt{Feature}_i.\texttt{attribute}_{i,j}) = \texttt{value}_{i,j}$.
- The initialisation block.

The assignment is not complete as it does not specify the values for unselected or implicitly-selected features. An example of an implicitly-selected feature occurs when a leaf feature is selected, requiring that its ancestors in the tree need to be selected too. In addition, the variable $f^\dagger$ introduced to count optional feature f is set to 1. Finally, values of attributes for unselected features are set to some arbitrary value so that the all variables appearing in a constraint are defined (required to test satisfaction). The following steps add the missing elements to an assignment. We call this the *completion* of the product selection. Assume that $f \in \mathsf{FID}$, $a \in \mathsf{AID}$, and feature model FM is encoded as constraints given by $\psi = [\![FM]\!]$.

1. Iterate the following steps until a fixed point is reached:
   (a) If $f \in \mathit{dom}(\sigma)$ and f' is the parent of f, then set $\sigma(f') = 1$.
   (b) If $f \in \mathit{dom}(\sigma)$ and $f^\dagger$ appears in ψ, then set $\sigma(f^\dagger) = 1$.
2. If $f \notin \mathit{dom}(\sigma)$ and f appears in ψ, then set $\sigma(f) = 0$
3. If $f.a \notin \mathit{dom}(\sigma)$ and $f.a$ appears in ψ, then set $\sigma(f.a) = v$, where v is an arbitrary (integer) value within the range specified for $f.a$.

A product selection σ is *valid* whenever for all completions σ' we have $\sigma' \models \psi$.

Example 7. The product `P3` from Example 6 results in the following initial variable assignment

$$\sigma(\texttt{German}) = 1 \qquad \sigma(\texttt{Repeat}) = 1 \qquad \sigma(\texttt{Repeat.times}) = 10.$$

In the context of the feature model in Example 2. The remaining variables are `English`, `Dutch`, and `MultiLingualHelloWorld`, which is the parent of `Language`

and `Repeat`, and there are no other attributes. The completion of σ includes the following additional elements:

$$\sigma(\texttt{MultiLingualHelloWorld}) = 1 \qquad \sigma(\texttt{English}) = 0$$
$$\sigma(\texttt{Language}) = 1 \qquad \sigma(\texttt{Dutch}) = 0$$

The resulting completed assignment σ satisfies the constraints specified in Example 2. In contrast to this, the constraints would not be satisfied for product `P4`, where $\sigma(\texttt{English}) = 1$, $\sigma(\texttt{Repeat.times}) = 6$, and $\sigma(\texttt{Repeat}) = 1$, due to the clause $\texttt{English} \rightarrow (\texttt{Repeat} \rightarrow (\texttt{Repeat.times} \geq 2 \wedge \texttt{Repeat.times} \leq 5))$.

6 Product Generation

This paper introduced four language extensions to core ABS: the μTVL language to represent feature models, the delta modelling language (DML) to represent delta modules, the product line configuration language (CL) to associate deltas to products and to establish the order of application of the deltas, and the product selection language (PSL) to describe the desired products. From a global perspective, these are used in the generation of a final software product as follows.

Given a core ABS module P, a set of delta modules Δ, a product line configuration C, a feature model FM, and a product selection p, the following steps are performed to build the final software product:

Check that the product selection p is satisfied by the feature model FM, as explained in Section 5.2.

Select the delta modules from Δ with valid application condition according to p, as described in Section 4.2.

Apply the deltas to the core module P, in the prescribed order, as described in Section 3.2. Add the initialisation block from the product selection—this will be the 'main' method.

Application of the deltas yields the final software product, a core ABS program.

7 Related Work

Existing approaches to express variability in modelling languages can be classified in two main directions [37]: annotative (or negative) and compositional (or positive). A third main approach for representing variability of development artefacts are model transformations.

Annotations. Annotative approaches consider one model representing all products of the product line. Variant annotations, e.g., using UML stereotypes in UML models [38, 14] or presence conditions [12], define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) proposed in Pohl. et al. [30] models the variability of product line artefacts in a separate model where links to the artefact model take the

place of annotations. Similarly, decision maps in KobrA [1] define which parts of the product artefacts have to be modified for certain products. In the Koala component model [28], the variability of a component architecture containing all possible components is expressed by component parameterisation that is instantiated depending on the product features.

Composition. Compositional approaches, such as delta modelling [35, 31, 32, 34], associate model fragments with product features that are composed for a particular feature configuration. A prominent example of this approach is AHEAD [3], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques [17, 37, 26]. Apel et al. [36] apply model superposition to compose model fragments.

Transformations. The common variability language (CVF) [16] represents the variability of a base model by rules describing how modelling elements of the base model have to be substituted in order to obtain a particular product model. In [20], graph transformation rules capture artefact variability of a single kernel model comprising the commonalities of all systems. In [18], architectural variability is represented by change sets containing additions, removals or modifications of components and component connections that are applied to a base line architecture. Perrouin et al. [29] obtain a product model by model composition and subsequently refinement by model transformation.

Delta modelling. The notion of program deltas was introduced by Lopez-Herrejon [25] to describe the modifications of object-oriented programs. Schaefer et al. [35, 31] introduced delta modelling as a means to develop product line artefacts suitable for automated product derivation. The conceptual ideas of delta modelling have also been applied the programming language level in an extension of Java with core and delta modules allowing the automatic generation of Java-based product implementations [32]. In recent work, Schaefer et al. [34, 33] propose a version of delta-oriented programming where products are generated only from delta modules applied to the empty product. Furthermore, in this version the application conditions and the application ordering are specified separately from the delta modules in a product line specification in order to increase the reusability of the delta modules and to enable compositional type checking.

8 Conclusion

This paper presented the variability modelling fragment of the HATS ABS modelling framework, realised by languages μTVL, DML, CL, and PSL. Together these languages can specify all the variability of a product line of core ABS models, with PSL scripts specifying the eventual products that can be derived.

The presented variability modelling concepts only target spatial variability. However, an ABS product line must also safely evolve over time in order to accommodate necessary changes after the deployment of the products; e.g., bug

fixes, feature extensions or modifications, or changes in user requirements. In order to facilitate the modelling of temporal variability for core ABS models, it is crucial that evolution is expressed at the abstraction level of the modelling language. Hence, in the future, also within the scope of the HATS project, we are planning to extend the presented variability modelling concepts which are based on delta modelling with dynamic delta models to capture variability in space as well as variability in time.

A description of the core ABS language [15] and the proposed component model [24], along with a tutorial of the full ABS language and HATS tools suite [9] are available. In addition, the HATS tool suite, documentation, as well as several case studies are available from http://www.hats-project.eu.

References

1. Atkinson, C., Bayer, J., , Muthig, D.: Component-Based Product Line Development: The KobrA Approach. In: SPLC (2000)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T. (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004). Lecture Notes in Computer Science, vol. 3362, pp. 49–69. Springer-Verlag, New York, NY (2005)
3. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Eng. 30(6) (2004)
4. Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated analysis of feature models: challenges ahead. Commun. ACM 49(12), 45–47 (2006)
5. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems (2010)
6. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: European Symposium on Programming (ESOP'07). Lecture Notes in Computer Science, vol. 4421, pp. 316–330. Springer-Verlag (2007)
7. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29. pp. 159–162. University of Duisburg-Essen (January 2010)
8. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) 7(3) (Jun 2004)
9. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Bernardo, M., Issarny, V. (eds.) SFM. Lecture Notes in Computer Science, vol. 6659, pp. 417–457. Springer (2011)
10. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract Delta Modeling. In: Proceedings of the ninth international conference on Generative programming and component engineering. pp. 13–22. GPCE '10, ACM, New York, NY, USA (Oct 2010)
11. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming (Nov 2010)
12. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: GPCE (2005)

13. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10(1), 1–18 (2002)
14. Gomaa, H.: Designing Software Product Lines with UML. Addison Wesley (2004)
15. Hähnle, R., Johnsen, E.B., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification, In this volume.
16. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: SPLC (2008)
17. Heidenreich, F., Wende, C.: Bridging the Gap Between Features and Models. In: Aspect-Oriented Product Line Engineering (AOPLE'07) (2007)
18. Hendrickson, S.A., van der Hoek, A.: modelling product line architectures through change sets and relationships. In: ICSE. pp. 189–198 (2007)
19. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology (March/April 2008)
20. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: MoDELS. pp. 151–165 (2007)
21. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling 6(1), 35–58 (Mar 2007)
22. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute (1990)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 1241 (1997)
24. Lienhardt, M., Lanese, I., Bravetti, M., Sangiorgi, D., Zavattaro, G., Welsch, Y., Schäfer, J., Poetzsch-Heffter, A.: A component model for the ABS language (2011), In this volume
25. Lopez-Herrejon, R.E., Batory, D.S., Cook, W.R.: Evaluating Support for Features in Advanced Modularization Technologies. In: European Conference on Object-Oriented Programming (ECOOP'05). Lecture Notes in Computer Science, vol. 3586, pp. 169–194. Springer-Verlag (2005)
26. Noda, N., Kishi, T.: Aspect-Oriented Modeling for Variability Management. In: SPLC (2008)
27. OMG: Unified modelling language, infrastructure and superstructure (version 2.2, OMG final adopted specification) (2009)
28. van Ommering, R.C.: Software reuse in product populations. IEEE Trans. Software Eng. 31(7), 537–550 (2005)
29. Perrouin, G., Klein, J., Guelfi, N., Jézéquel, J.M.: Reconciling Automation and Flexibility in Product Derivation. In: SPLC (2008)
30. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
31. Schaefer, I.: Variability Modelling for Model-Driven Development of Software Product Lines. In: Proc. of 4th Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010) (2010)
32. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proc. of 15th Software Product Line Conference (SPLC 2010) (Sep 2010)
33. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: 10th International Conference on Aspect-Oriented Software Development, AOSD 2011. pp. 43–56. ACM (2011)

34. Schaefer, I., Damiani, F.: Pure Delta-oriented Programming. In: FOSD 2010 (2010)
35. Schaefer, I., Worret, A., Poetzsch-Heffter, A.: A Model-Based Framework for Automated Product Derivation. In: Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009) (2009)
36. Sven Apel, Florian Janda, S.T., Kästner, C.: Model Superimposition in Software Product Lines. In: International Conference on Model Transformation (ICMT) (2009)
37. Völter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: SPLC. pp. 233–242 (2007)
38. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML Profile for Software Product Lines. In: Workshop on Product Familiy Engineering (PFE). pp. 129–139 (2003)