VML* – A Family of Languages for Variability Management in Software Product Lines[1]

Steffen Zschaler[1], Pablo Sánchez[2], João Santos[3], Mauricio Alférez[3], Awais Rashid[1],
Lidia Fuentes[2], Ana Moreira[3], João Araújo[3], Uirá Kulesza[3]

[1] Computing Department, Lancaster University, Lancaster, United Kingdom
{zschaler, awais}@comp.lancs.ac.uk
[2] Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga, Spain
{pablo,lff}@lcc.uma.es
[3] Computer Science Department, Universidade Nova de Lisboa, Lisbon, Portugal
{jps, mauricio.alferez, amm, ja}@di.fct.unl.pt, uirakulesza@gmail.com

Abstract. Managing variability is a challenging issue in software-product-line engineering. A key part of variability management is the ability to express explicitly the relationship between variability models (expressing the variability in the problem space, for example using feature models) and other artefacts of the product line, for example, requirements models and architecture models. Once these relations have been made explicit, they can be used for a number of purposes, most importantly for product derivation, but also for the generation of trace links or for checking the consistency of a product-line architecture. This paper bootstraps techniques from product-line engineering to produce a family of languages for variability management for easing the creation of new members of the family of languages. We show that developing such language families is feasible and demonstrate the flexibility of our language family by applying it to the development of two variability-management languages.

Keywords: Software Product Lines, Family of Languages, Domain-specific Languages, Variability Management.

1 Introduction

Software Product Lines Engineering (SPLE) is seen as a promising approach to increasing the productivity and quality of software, especially where essentially similar software needs to be provided for a variety of contexts and customers each requiring customizations and variations for their specific conditions [1-2]. In SPLE, features [3] are used to capture commonalities or discriminate among products, i.e. capture variabilities, in an SPL. SPL features are often modelled using feature models [3-4]. Management of variability throughout the product line is a key challenge in SPLE.

An important part of variability management is to make explicit the relation between the variability model (e.g., the feature models referred to in the previous para-

graph) and other models and artefacts of the SPL. Once this relation has been explicitly represented, it can be used for a number of purposes, most importantly to automatically derive product instances based on product-configuration specifications, but also for other purposes such as trace-link generation and consistency checking of SPL models. Due to its relevance, this topic is currently an area of intensive research and a number of approaches have been proposed [5-9]. Initial research focused on using general-purpose model transformations to encode product derivation [10-11]. Later it was argued that this placed too heavy a burden on SPL engineers, as they would now also have to learn the intricacies of model transformations. Consequently, a number of approaches that hide the model transformations from the SPL engineers have recently been developed [6-7, 12].

Czarnecki et al and Heidenreich et al [6-7] propose generic techniques that associate features with arbitrary combinations of model elements and generate a standard model transformation for product derivation from this. In contrast, we have argued before [12] [13] that transformation actions that are specific to the types of models used for describing the SPL are more useful, as they provide a terminology already known to SPL engineers, allow consideration of model semantics in the definition of transformations, and allow avoiding some inconsistencies (e.g., dangling references) in product models by design.

This requires new languages to be developed for each type of model that may be used in describing an SPL—a costly and error prone task. To make development of such languages feasible, this paper proposes VML*[2], a family of languages—or a language product line—for variability management, showing that developing such languages is a feasible goal. Individual members of the family are described using a domain-specific language (DSL). Based on such a specification, a generator produces the complete infrastructure for the specified language. Such a generative approach has the added benefit of making it easier to support other evaluations beyond product derivation: they can be implemented in additional code generators from the language specification.

The key contribution of this paper is, thus, in the domain of software-language engineering, where it applies ideas from SPLE and model-driven development to the development of VML* languages. This enables us to efficiently build new VML* languages for new SPL contexts, and thus improves over our previous work [12], which was limited to copy-and-paste-based reuse, limiting efficiency and increasing error-proneness of language development. A secondary contribution is that this new approach to language development allows us to support additional evaluations for VML* languages, such as generation of trace links or SPL consistency checking.

Section 2 further discusses the motivation for building custom languages instead of one generic language and derives a set of challenges to be overcome to enable efficient development of such languages. Section 3 then presents how we applied SPLE techniques to construct a family of languages for variability management and is followed by Sect. 4, which shows how concrete languages have been developed based on our approach. Section 5 reviews some related work and Sect. 6 concludes the paper and points out directions for future work.

---

[2] For Variability Management Languages

2 Motivation

This section describes the motivation that led to the creation of the VML* family of languages. First, we provide some background on VML languages and then we present the motivation of this paper.

2.1 Managing Variability Using Target-Model–Specific Languages

This section explains why we choose to model SPL variability using target-model–specific languages rather than a single generic language. We use as an example an architectural model of a lock control framework for a Smart Home Software Product Line (SPL) [1, 14]. Smart Home applications aim at automating and controlling houses and buildings in order to improve the comfort and security of their inhabitants. The lock control is placed on doors of rooms whose access must be controlled. Several options are available to end users acquiring a specific Smart Home software installation:

- Different authentication mechanisms can be used: identification cards, fingerprint scanners or a simple numeric keypad.
- Doors are opened manually and users have a time period to authenticate before triggering the alarms. Optionally, it is possible to select a computer-controlled door lock control (Automatic Lock), which will be released upon successful authentication.
- Automatic sliding doors can also be used (Door Opener). This option requires that the Automatic Lock control of the door lock be selected.
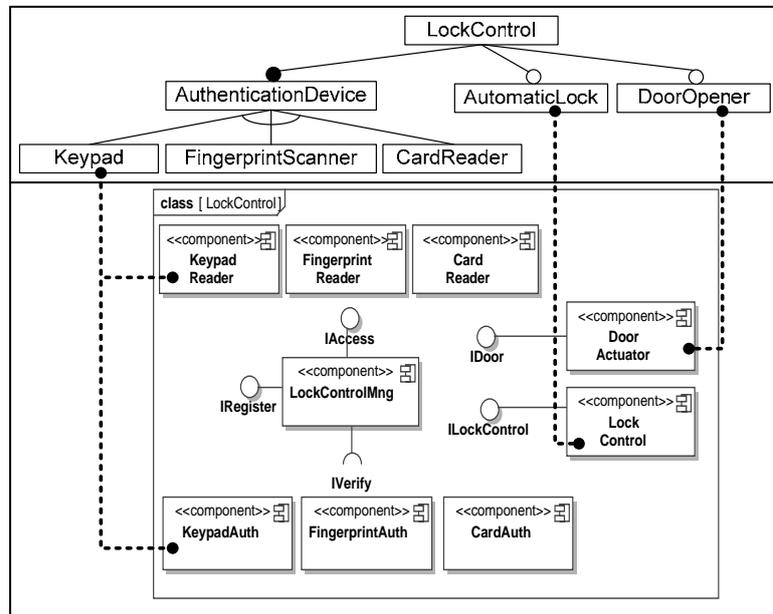

Figure 1 A software architecture for the lock control framework

Figure 1 depicts a software architectural design for this lock control framework. This architectural design is comprised of three different parts, which are explained in the following.

Firstly, variability inherent to the domain is expressed using a feature model [4, 15] (Fig. 1 (top)). This feature model represents variability specification or problem space. It specifies which features of the system are variable and the reasons why. For instance, the `AuthenticationDevice` to be used is a variable feature because there are several alternative devices available but only one must be selected. `Automatic-Lock` and `DoorOpener` are variable features because they are options that may be included in a specific lock control application or not.

Secondly, once variability has been identified, the software architecture is designed using the component model of UML 2.0 (Fig. 1 (bottom)). This represents variability realization or solution space. The mechanism selected for supporting variability in the architectural design is plugin components. The `LockControlMng` component is the central component of this architecture. Each alternative for authentication is designed as a pair of plugin components: one for controlling the physical device that serves to authenticate users (e.g. `KeypadReader`); and the other one encapsulating the logic of the authentication algorithm (e.g. `KeypadAuth`). These plugin components communicate with the `LockControlMng` through the `IAccess` interface, in the case of reader components, and the `IVerify` interface, in the case of authenticator ones. All plugin components must register in the `LockControlMng` component using the interface `IRegister`. The `LockControlMng` receives data from the reader components and, with the data received, it calls the authenticator component. The latter is in charge of checking if the user has access to the room or not. If the user is authentic, the `Lock-ControlMng` component invokes the `LockControl` component, which releases the lock. This invocation is placed only if the automatic lock control option has been selected. If the door is a sliding one, the `LockControlMng` should also invoke the `DoorActuator` component for automatic opening of the door.

Thirdly, we must specify the links between variability specification and variability design, or problem space and solution space, indicating how the components of the architectural model must be composed according to the selected features. In our case, for instance, when a specific authentication device is selected, the corresponding reader component must be connected to the `LockControlMng` through the `IAccess` interface. In the same way, the `LockControlMng` component must be connected, to the corresponding authenticator component though the `IVerify` interface. Both the authenticator and the reader components must also be connected to `LockCon-trolMng` through the `IRegister` interface. The components corresponding to non selected alternatives must simply be removed. Similarly, the `DoorActuator` and `LockControl` components are adequately connected if the corresponding optional features are selected; otherwise, they should be removed.

These relationships can be expressed using general purpose model transformation languages, such as demonstrated in [10-11]. Nevertheless, as previously discussed in [10], these have the following shortcomings:
- *Metamodel Burden.* A model transformation language is often based on abstract syntax manipulations. According to Jayaraman et al. [16], "Most model

Table 1 Part of the VML4Arch Specification for Smart Home

```
01    import features <"/SmartHome.fmp">;
02    import core <"/SmartHome.uml">;
03
04    ...
05
06    variant for FingerprintScanner {
07        connect("FingerprintReader","LockControlMng","IAccess");
08        connect("FingerprintReader","LockControlMng","IRegister");
09        connect("FingerprintAuth","LockControlMng","IRegister");
10        connect("LockControlMng","FingerprintAuth","IVerify");
11    } // Fingerprint scanner
12
13    variant for not (FingerprintScanner) {
14        remove('FingerprintReader');
15        remove('FingerprintAuth');
16    } // not FingerprintScanner
```

developers do not have this knowledge. Therefore, it would be inadvisable to force them to use the abstract syntax of the models".

- *Language Overload and Abstraction Mismatch.* There are different kinds of model transformation languages [16], and each of them is based on a specific computing model. They range from rule-based languages (e.g. ATL [17]) to expression-based languages (e.g. xTend [18]) and graph-based languages (e.g. AGG [19]). When employing a model transformation language, software product line engineers must also understand the underlying computing style (e.g. rule-based) and learn the language syntax. As a result, software product line engineers are forced to rely on abstractions that might not be naturally part of the abstraction level at which they are working.

To overcome these shortcomings, we proposed [12] to create dedicated languages, for specifying product derivation processes; that is, for specifying how features map to software models. These dedicated languages must follow a very basic computation style, where based on a selection of features, small sequence of simple commands are executed. These commands, moreover, must use syntax familiar to the modeler, using concepts of the concrete syntax of the model rather than their abstract syntax. These user-friendly high-level specifications are then translated into a set of low-level general purpose model transformations, which support the automation of the product derivation process. So, the SPL engineer can enjoy the benefits of using model-driven techniques but without paying the associated cost, i.e. without needing to learn the intricacies of model transformation languages. Table 1 provides an example of such a dedicated language for manipulating UML component models.

This specification establishes that whenever the `Fingerprint` option is selected (lines 06-11), the `KeypadAuth` and `KeypadReader` components must be connected to the `LockControlMng` component through the corresponding interfaces, as previously described. The `connect` operator is an intuitive composition mechanism to specify that two components must be connected using the interface specified as a parameter. The first parameter of the connect operator is the component that requires the interface while the second parameter is the component that provides it. In the case where the `Fingerprint` variant is not selected (lines 03-16), the `FingerprintAuth`

and the `KeypadReader` components are removed from the architecture, using the `remove` operator.

2.2 Automating the generation of new VML languages

Beyond the language from Figure 1, a wide range of languages for managing variability in any kind of target modeling language need to be constructed. For instance, we need to develop a dedicated language with specific operators for managing variability in use cases models, activity models, business process models or any other kind of architectural description language. Developing such languages is cost-intensive and error-prone, especially as so far there is no support for reuse between different such languages beyond a copy-and-paste approach. This is a serious barrier to the adoption of our approach in SPL projects.

To make developing such languages feasible, we need to solve the following three challenges:

1. *Support of reuse between different languages.* The support infrastructure should be easily reused for new languages. Reuse should not be based on copying an existing language implementation and adjusting it, removing unneeded actions and adding new actions. Otherwise, if errors are found and fixed in the infrastructure for one language, these corrections would have to be manually transferred into all other language infrastructures. The same would be true for new features of the infrastructure, for example, new evaluations of specifications other than product derivation.

2. *Allow the type of variability models to vary.* Different approaches to modelling variability have been proposed: very often, feature trees [4] or cardinality-based feature models [20] are used. However, DSLs have also been used to represent variability [21]. Any variability management language should be easily adapted to any type of variability model.

3. *Support for easy customisation of target-model element access.* Target-model model elements need to be accessed from a specification based on a textual reference (e.g., their fully qualified name or some pattern matching a number of names). Depending on the target model different forms of such textual references may be useful. The evaluation of such textual references should be implemented separately from the individual actions to allow for easy exchange and customisation of this feature.

In this work, we present a generative infrastructure for creating new VML languages for a concrete target model that tackles these issues.

3 The VML* Family of Languages

In response to the challenges identified in the previous section, we propose to bootstrap SPLE techniques using a model-driven and generative approach for creating the infrastructure (e.g., parser, editor, evaluation engine) for a specific VML* language. To this end, we have developed the VML* family of languages, which consists of:
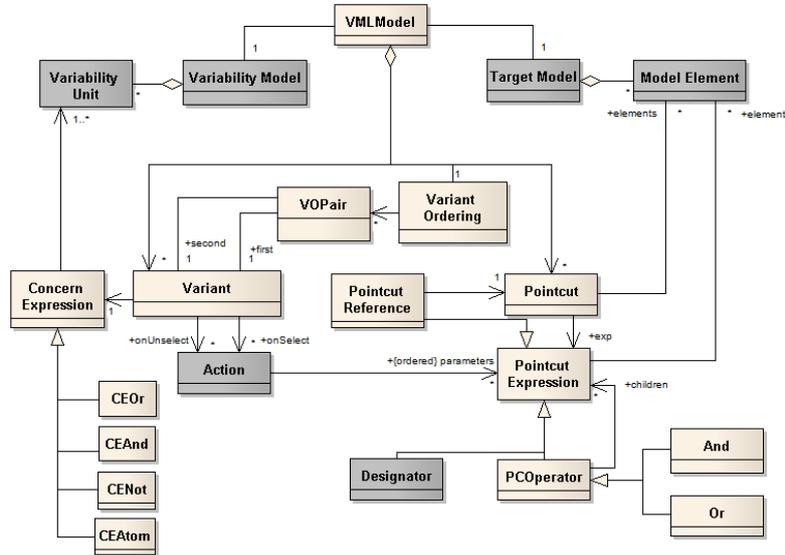
Figure 2 Common metamodel for VML languages. Variation points have been highlighted in dark grey

1. A common metamodel for VML* languages including variation points that can be customised for describing specific VML* languages. This provides the concepts common to all VML* languages.
2. A DSL for specifying the choices a specific language makes for each variation point.
3. A generator-based infrastructure that can instantiate all custom elements of the process from [12] for any VML* language.
   A working prototype of this system is available as a set of Eclipse plugins [22].

3.1 A common metamodel for VML* languages

Figure 2 shows the general concepts required for expressing variability in product-line models. This metamodel has been developed as a generalisation of the metamodels of VML4Architecture, or simply VML4Arch [12-13] and VML4Requirements, or simply VML4RE [23-24], two variability management languages we have previously developed. VML4Arch is a language for relating feature models and UML2.0 architectural models of an SPL. VML4RE is a language for relating feature models and UML2.0 use case and activity models. These languages have been developed in parallel, but independently. They have a number of differences, but they also share a large number of commonalities, enabling us to derive a common metamodel for VML* languages.

The metamodel shown in Figure 2 is independent of both the specific models used for variability modelling (e.g., feature models, domain-specific languages) and the specific target models (e.g., UML, architecture description models, generation work-

flow models). Consequently, a number of concepts are abstract in this metamodel. To apply the metamodel for a specific combination of target model and variability model, these concepts (highlighted in dark grey in Figure 2) need to be specialised (how to specify such specialisations will be discussed in the next section). In the following, we discuss each of the metamodel concepts in more detail.

VMLModel. A VML model relates a variability model and a target model, using a set of variants to describe how the target model needs to vary as each of the concerns of the variability model is selected or unselected.

VariabilityModel. A variability model is the central artefact in variability modelling. `VariabilityModel` and `Variability Unit` serve as adapters to the specific form of variability modelling employed in a specific scenario.

Variability Unit. These are the units of variability in variability modelling. A variability model describes what variability units a potential product may have and what constraints govern the selection of combinations of variability units for individual products. From the perspective of variability management, we are mainly interested in the name of a variability unit and whether it has been selected for a specific product configuration. Notice that for the purposes of our metamodel we do not care about how variability units are expressed in a variability model. They may be represented as explicit features in a feature model [4] or more implicitly in a DSL [21], or in any other form that is convenient for modelling variability in a specific project. To enable our metamodel to relate to all these different kinds of representations, we standardise on the common notion of Variability Unit and require adapters that extract these from any of the representations discussed above.

TargetModel. Target models describe a product line. There are a large number of potential target models—for example, requirements models, architecture models, or code-generation-workflow models.

ModelElement. Model elements represent arbitrary elements of the target model. This concept serves as an adapter to actual model elements and needs to be specialized for each kind of target model (thereby defining the concrete model elements available). The model elements are typed using metaclasses imported from the target metamodel.

Variant. A variant describes how the target models must be varied when a certain combination of variability units is selected or unselected. Notice that for product derivation it is sufficient to provide a variant for each non-mandatory variability unit, as we can assume the unvaried target model to represent the model for all the mandatory variability units. For some other evaluations (e.g., trace-link generation), however, a variant must be provided for each variability unit including mandatory ones. Each variant defines two sets of actions for its variability units: a set of *onSelect* actions defines how to vary the target model when the variability units are selected; a set of *onUnSelect* actions defines what to do when the variability units are not selected.

ConcernExpression. For certain use cases it is not sufficient to map variability units directly onto modifications of the target model, as has also been previously discussed in the literature [6-7]. Therefore, we define variants for so-called concern expressions, logic expressions over variability units. We support *And*, *Or*, and *Not* expressions as well as atomic terms.

VariantOrdering. Sometimes the order in which the actions of different variants are executed during product derivation is important, as actions for one variant may

rely on model elements created by actions for another variant. VariantOrdering provides SPL developers with a means of defining a partial order of execution over variants using pairs of variants. The infrastructure will guarantee that all actions of the first variant in a pair are executed before any action of the second variant of that pair is executed.

Action. Actions are used to describe modifications to the target model. These need to be customised for each kind of target model, depending on the kinds of variations that make sense at the level of abstraction the target model covers. For example, if the target model is a use case model, one particular action may be to connect an actor and a use case, while for an architectural model a possible action could be to connect two components. Actions may add, update or remove model elements in the target model and may create, update or remove links between existing or newly added model elements.

PointcutExpression. A pointcut expression is an expression that identifies a model element or a set of model elements. It is constructed from atomic designators, pointcut references and combining operators (*Not, And*, and *Or*).

Pointcut. A pointcut identifies a model element or set of model elements. The model elements are denoted by a pointcut expression. The main purpose of the Pointcut concept is to allow particular pointcut expressions to be named. A named Pointcut can then be reused using a PointcutReference.

PCOperator. Operators enable the construction of pointcut expressions combining the set of elements returned from more than one element pointcut. Here, we define only two operators, namely *and* and *or*, which represent intersection and union of the sets of model elements of their element expressions, respectively.

Designator. A designator is a piece of text that is used to identify a model element or a set of model elements. It may be a name (possibly qualified), a signature, a wildcard expression, or anything else that makes sense in the target model. As resolution of designator text into actual model elements is specific to the target model, the designator concept needs to be customised for each target model.

3.2 A DSL for specifying individual VML* languages

To enable succinct description of the specificities of a certain VML* language, we have defined a metamodel and concrete syntax for language-instance description. Figure 3 shows the key concepts. Based on an instance of this metamodel—a VML* language description—we can then generate an appropriate infrastructure customised for that specific VML* language.

The individual concepts in the language-description metamodel are:

LanguageInstanceModel. The central metaclass of VML* language descriptors, binding together the other parts of a VML* language descriptor.

VariabilityModelImport. This provides information about the type of variability model to be supported by the VML* language. The key interface between VML* and a variability model is the set of features defined. The language descriptor, therefore,
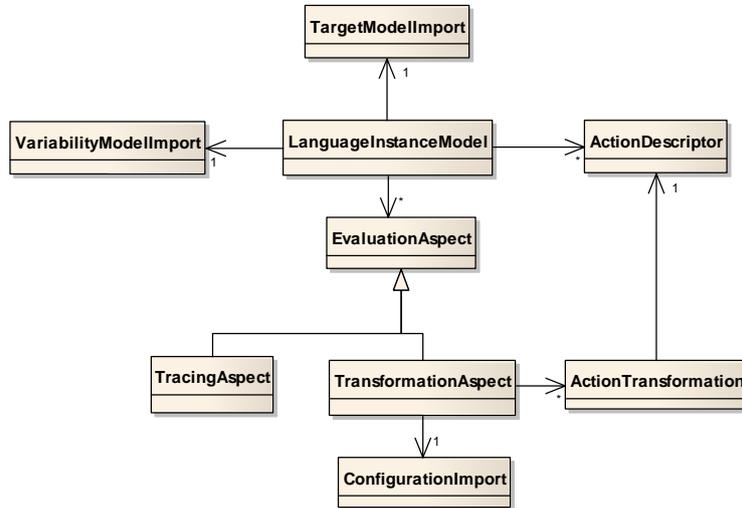
Figure 3 Metamodel for VML* language instance descriptions.

contains a snippet of model-query code[3] that serves as an adapter between the variability model and a VML* specification. This snippet is the only place where knowledge about the variability-model metamodel is located in a VML* language descriptor.

TargetModelImport. This provides information about the type of target model to be supported by the VML* language. Mainly, this defines how pointcut designators should be evaluated for a specific target model. Depending on the specific kind of target model, different pointcut designators may be required. While, for example, use-case models require only simple qualified names (possibly using wildcards for quantification) to identify individual actors, use cases, or activities, architectural models may additionally require pointcut designators for operation signatures or component provided or required interfaces. Therefore, both the syntax of pointcut designators and their interpretation is specific to the kind of target model. In all VML* languages, pointcut designators are syntactically represented as simple string values. They are then passed to a piece of model-query code interpreting them to return a set of model elements from a given target model. This piece of code is defined for a specific VML* language using TargetModelImport.

ActionDescriptor. Each action descriptor provides general syntactic information about one action. This includes the name of the action and the number of parameters it takes. The concrete syntax for action invocation in the generated VML* language will be '<action_name> (param$_1$, ..., param$_n$)'. For each parameter, users of the VML* language will be able to provide a pointcut expression.

---

[3] Our prototype uses openArchitectureWare's (oAW) xTend language to express model queries and model transformations. These xTend snippets can be kept as operations in a separate xTend file and referenced from the language instance descriptor, allowing language designers to take full advantage of oAW's checking capabilities.

EvaluationAspect. Every evaluation aspect describes one form of evaluation of a VML* specification. The VML* family can be extended with a number of these evaluation aspects (currently only one aspect—product derivation—has been implemented, but we are working on an implementation for trace-link generation and are planning to work on consistency evaluation), which can be supported for every concrete VML* language, but not all VML* languages will need support for all evaluation aspects. A VML* language description can, therefore, include only those evaluation aspects that are actually required for this VML* language, providing an additional opportunity for optimisation. Notice that making such a selection manually based on the architecture presented in the previous subsection can be very difficult, as the different evaluation aspects actually overlap in some elements of the architecture (for example, in plugin configuration files). The model-driven approach not only allows a selection of one aspect or another, it additionally allows this selection to be changed flexibly, even experimentally.

TransformationAspect. If present, it enables product-derivation for target models. For each `ActionDescriptor` this defines an `ActionTransformation` specifying the model transformation encapsulated by this action. Furthermore, a `ConfigurationImport` defines an adapter for configuration models.

ConfigurationImport. For the construction of models for specific products, the VML* infrastructure requires access to the set of features selected in a specific product configuration. To avoid polluting the VML* infrastructure with knowledge about the inner structure of product configurations, `ConfigurationImport` provides a snippet of model-query code that serves as an adapter to product-configuration specifications by extracting the set of selected features from a product configuration.

ActionTransformation. Provides additional information for an action pertaining to the transformation of target models by this action. For every `ActionDescriptor` there needs to be a corresponding `ActionTransformation` instance. In particular, this includes a snippet of model-transformation code that implements the action. In this code, the parameters can be referenced as 'param$_1$' thru 'param$_n$'. The type of each parameter is defined in the `ActionTransformation`.

TracingAspect. If present, it enables the generation of trace links from a VML specification. Such trace links connect selected features and added or removed model elements of the target model. The tracing aspect is specified by naming the model-transformation operations that create or remove model elements; wildcards may be used to provide these names. VML* will then generate an aspect for the model transformation that advises these operations and creates appropriate trace links using the AMPLE Tracing Framework (ATF) [25].

3.3 Generation of VML* language infrastructure

Instances of this metamodel can be defined using a textual concrete syntax. Table 2

Table 2 Excerpt from the language descriptor for VML4RE

```
01   vml instance vml4req { // Define a new language called vml4req
02       // This section defines the type of variability model and
03       // how to access it
04       features {
05           metamodel  "/bin/fmp.ecore"
06           // Extracts all variability units from a variability model
07           function   "getAllFeatures"
08       }
09
10       // This section defines the type of target model and how to
11       // access it
12       target model {
13           metamodel   "UML2"
14           type        "uml::Package" // Metamodel type of a model
15           // Function to interpret pointcut designators
16           function    "dereferenceElement"
17       }
18
19       // Importing plugins and external specifications
20       bundles: "unl.vml4req", "ca.uwaterloo.gp.fmp", ...
21       extensions: "unl::vml4req::library::umlUtil"
22
23       // Syntactical definition of available actions
24       actions:
25           createInclude {
26               params "List[uml::UseCase]" "List[uml::UseCase]"
27           }
28           insertUseCase {
29               params "String" "uml::Package"
30           }
31           ...
32
33       // Definition of available evaluation aspects
34       aspects:
35           transformation { // Evaluation for product derivation
36               // Defines adapter for product-configuration access
37               features {
38                   type  "String"
39                   function "getAllSelectedFeatures"
40               }
41               // Definition of the semantics of actions as
42               // model transformations
43               createInclude {
44                   function   "createIncludes"
45               }
46               insertUseCase {
47                   function   "createUseCase"
48               }
49               ...
50           }
51   }
```

shows an excerpt of the language descriptor for VML4RE (cf. Sect. 4). Mapping this concrete syntax to the abstract syntax discussed above is rather straightforward so that we will not discuss it in any more detail here. It is worth noting, though, that this language descriptor does not contain complicated model-transformation code; all that is specified are the names of some functions. These functions with the actual model-transformation code are contained in an external file[4], allowing standard editors and error highlighting to be used when writing the code. Including the fully qualified name of the external file in the list after the "extensions" keyword ensures that the extension can be accessed from all relevant places in the generated code. Similarly, the "bundles" keyword lists other plugins that should be made available to any generated plugins. Here we include the plugin project containing our extension and the FMP plugin [26] providing support for cardinality-based feature models.

Furthermore, we have developed a generator that takes language descriptors such as shown in Table 2 and generates a set of Eclipse plugins containing the infrastructure for this language. The operational prototype can be obtained from [27]. The code generated by this generator is based on the work previously presented in [12]. The generation is completely automatic; the only manual input provided by language developers is the language instance descriptor and the implementations of the actions provided in a separate file. The complete infrastructure for editing, compiling, and executing specifications of the new VML language is encapsulated in the generator and can, thus, be reused for each new language.

4   Example languages from the VML* family

We have re-implemented both VML4Arch and VML4RE based on our new infrastructure. As VML4Arch has already been discussed extensively in [12], here we will focus on VML4RE. For VML4Arch we will only give a brief discussion of what needed to be changed to make it compatible with VML*. Both implementations can be downloaded from [24].

4.1   VML4RE

Requirements are most recurrently documented in a multi-view fashion [28-29]. Their description is typically based on considerably heterogeneous languages, such as use cases, activity diagrams, goal models, and natural language. Initial work on compositional approaches for early development artefacts does not clearly define composition operators for combining common and varying requirements based on different views or models. Therefore, a key problem in SPLE remains how to specify and apply the composition of elements defined in separated and heterogeneous requirements models.

With the Variability Modelling Language for Requirements (VML4RE) [23] we propose an initial solution for this problem by introducing a new requirements composition language for SPLs. VML4RE is a textual language with two main goals:

---

[4] An oAW xTend file for our prototype.

Table 3 Selected VML4RE actions for Use Case Models

Action Signature	Description
insertUseCase (String name, Package p)	A new use case named name is inserted into package p.
insertPackage (String name, Package p)	A new package named name is inserted into package p.
createActorToUseCaseLink (List[Actor] actors, List[UseCase] usecases)	A new connection is created between each of the actors and each of the use cases.
createInclude (List[UseCase] source, List[UseCase] target)	A new <<include>> dependency is created between each of the source use cases and each of the target use cases.

(i) to support the definition of relations between SPL features expressed in feature models and requirements expressed in multiple views (based on a number of UML diagram types, such as use case diagrams and activity diagrams); and (ii) to specify the compositions of requirements models for specific products of a SPL. VML4RE supports composition operators for UML use cases and activity models. It has been applied to case studies in domains such as home automation [23] and Mobile Applications [30]. It has shown great flexibility to specify composition rules and references to different kinds of elements in heterogeneous requirements models. The results of these experiments are encouraging and comparable with other approaches that support semi-automatic generation of trace-links relationships and composition between model elements in SPLs.

Table 3 shows an overview of some of the available actions of the VML4RE language for use cases. A more complete list can be found in [23]. VML4RE provides another set of actions for activity models, which are not shown here due to space restrictions.

Table 2 shows an excerpt from the language descriptor for VML4RE. It has been defined to map from feature models expressed using the FMP metamodel [26] to UML2 use case and activity models. This is expressed in the two sections named 'features' and 'target model', respectively, which also reference the functions to adapt to the feature model and to dereference pointcut designators in the target model. The real dereferencing code is implemented in the extension referenced through the 'extensions' keyword. The full language descriptor also specifies a tracing aspect. This is not shown in Table 2 for lack of space.

Finally, Table 4 shows an excerpt of a VML4RE specification for the Smart Home case study [23]. Lines 7 to 20 show the additional use cases needed when the Security feature is selected in a product configuration. Notice the use of wildcards on Line 13 to select all use cases in a package. If, and what, wildcards are supported and how they are evaluated is defined in the dereferenceElement operation invoked from the language instance descriptor in Table 2 on Line 16. Further, notice the use of a slightly more complex pointcut expression on Lines 16 to 19 of Table 4. This pointcut

Table 4 Part of the VML4RE Specification for Smart Home

```
01   import features <"/SmartHome.fmp">;
02
03   import core <"/SmartHome.uml">;
04
05   ...
06
07   variant Security {
08
09        insertPackage ("Security", "");
10        insertUseCase ("SecureTheHouse", "Security");
11        insertUseCase ("ActivateSecureMode", "Security");
12        createActorToUseCaseLink (
13           "Inhabitant", "Security::.*");
14        createInclude (
15           "Security::SecureTheHouse",
16           or (
17             "Notification::SendSecurityNotification",
18             "WindowsManagement::" _
19             "OpenAndCloseWindowsAutomatically"));
20   }
```

expression results in a set of two use cases: Notification::SendSecurityNotification and WindowsManagement::OpenAndCloseWindowsAutomatically.

4.2 VML4Arch

Re-implementing VML4Arch based on the VML* infrastructure proved surprisingly easy. However, as any product line requires a certain amount of stream-lining be-tween individual products to maximise reuse, there were some minor adjustments we had to make to fit VML4Arch into the family of languages. These adjustments, how-ever did not affect the functionality provided by VML4Arch. In detail, we had to:

- *Adjust the syntax of some VML4Arch operators.* VML4Arch originally had some operators like `connect c1, c2 using interface i`, which used a concrete syntax slightly different from the standard concrete syntax for VML* operators. We had to adjust the concrete syntax of these operators to fit the standard scheme generated by VML*. For example, the `connect` operator from above now is expressed as `connect (c1, c2, i)`.

- *Extend some operator definitions to allow for the use of pointcut expressions as parameters.* VML4Arch originally used direct references to model elements rather than pointcut expressions. This meant that we had to modify some of the operator definitions so that they would be able to deal with receiving sets of model elements as parameters rather than individual model elements only.

5 Related Work

The work presented in this paper is related to work in two areas of research: systematic development of families of languages and support for variability management in SPLE. As the main focus of this paper is on constructing a family of languages, we will begin by discussing literature from this area.

A number of research projects—for example, CAFÉ, Families, or ESAPS—have explored the notion of software system families (or product lines). In this work, we are extending these ideas to families of software languages, specifically for the case of VML languages.

Families of languages have been presented in the research literature for a range of domains: Voelter presents an approach for a family of languages for architecture design at different levels of abstraction [31], Akehurst et al. [32] present a redesign of the Object Constraint Language as a family of languages of different complexity, Visser et al. [33] present WebDSL, a family of interoperating languages for the design of web applications. All approaches, including ours presented in this paper, use very different kinds of technologies for their specific case: Voelter uses conditional compilation to construct an appropriate infrastructure, Akehurst et al. use a special parser technology that enables modular language specification, Visser et al. use rewriting of abstract syntax trees and our approach generates a monolithic infrastructure for each language. Equally, all approaches focus on different purposes of the language family: the different members of the family presented by Voelter are architectural languages at different levels of abstraction. The family presented by Akehurst et al. modularises different features of the OCL language, so that specific languages can be constructed as required for a project. WebDSL is a set of interoperating languages with purposes ranging from data modelling to workflow specification. The family of languages presented in our paper consists of languages that share a common set of core concepts, but adapt these to different languages with which they interface. At this point, an overview of the different potential uses of families of languages begins to emerge. What is needed next, is research into systematic development of such language families beyond individual examples.

Ziadi et al. [10] and Botterweck et al. [11] both propose the implementation of product derivation processes as model transformations. Their proposal relies on the realization of product derivations via a model transformation language. This strategy requires SPL engineers to deal with low-level details of model transformation languages. Our approach provides syntax and abstractions familiar to the SPL engineers. This eliminates the burden of understanding the intricacies associated with model transformation languages and metamodels. A VML* specification is automatically compiled into an implementation of the product derivation process in a model transformation language, but SPL engineers need not be aware of this generation process.

In [12] we have presented a process for developing variability management languages. The structure of these languages has some similarities to the languages developed using VML*, in fact VML4Arch was previously developed based on this process. However, focusing on process rather than infrastructure, [12] falls short of solving the issues discussed in the introduction. In particular, reuse between individual languages is only possible based on a copy-and-paste approach, variability-model and target-model access are closely intertwined with the other infrastructure code

making it difficult to modify them independently. In contrast, in this paper we have presented an infrastructure, which tackles all of these issues. The code generated for a specific VML* language is partially based on code developed for VML4Arch following the process from [12].

Czarnecki and Antkiewicz [6] present an approach similar to ours based on using feature models to model variability. They create a template model, which models all products in the product line. Elements of this model are annotated with so-called presence conditions. Given a specific configuration, each presence condition evaluates to true or false. If a presence condition evaluates to false, its associated model elements are removed from the model. Thus, such a template-based approach is specific to negative variability, which might be critical when a large number of variations affect a single diagram. Our approach can also support positive variability by means of actions such as connect or merge. Moreover, presence conditions imply introducing annotations into the SPL model. Therefore, the actions associated with a feature selection are scattered across the model, which could also lead to scalability problems. In our approach, they are well-encapsulated in a VML* specification, where each variant specifies the actions to be executed. FeatureMapper [7] is another approach similar to that of Czarnecki and Antkiewicz and our approach, avoiding the pollution of the SPL model with variability annotations. FeatureMapper is generic for all EMF-based models and generically integrates into GMF-based editors. In contrast, our approach is based on languages that are specific to a kind of feature model and a kind of target model. Genericity is achieved through a generative approach to creating the infrastructure for these languages from a set of common core concepts. The actual variability model in FeatureMapper is created implicitly by the designer selecting model elements in an editor and associating them with so-called feature expressions determining when the model element should be present in a product model. Negative variability is easily supported by this approach, as model elements can be easily removed if their feature expression is not satisfied by a specific configuration. Positive variability is more difficult to implement: instead of mapping features to target model elements, they need to be mapped to elements of a model transformation, again requiring SPL designers to have sufficiently detailed knowledge of that model-transformation language and the metamodels involved. In contrast, in our approach, designers of a specific VML* language can provide powerful actions that can support both negative and positive variability (or any mixture of the two) in a systematic manner. Finally, Haugen et al. [34] define the common variability language (CVL), which is a generic extension to DSLs for expressing variability. It provides three generic operators, but using these to express variability can lead to comparatively complex models. On the flip side, a VML* language is potentially less flexible than the two other approaches discussed in this paragraph, as it can only support the variability mechanisms for which a corresponding action has been defined.

A completely different approach to SPLE is followed in the feature-oriented software development community. Here, features are directly related to separate modules implementing each feature, where these feature modules can be understood as program or model transformations (e.g., [35]). This implies that no mapping from features to target models is required. Instead, the programming or modelling language must be sufficiently powerful to support modularizing of features as coherent well-encapsulated units of compositions. In another publication [36], we have presented a

feature-oriented approach towards SPL development. In this context, we also noted that a pure feature-oriented approach can lead to a large number of small feature modules negatively impacting scalability and comprehensibility of the approach, especially where features are often associated with small-grain changes to the architecture or implementation. Thus, for such cases, an approach with an explicit mapping may be beneficial.

Generally, all SPL approaches face the problem of ascertaining that only consistent and well-formed product models and implementations can be constructed. This problem becomes even worse when several interconnected types of models representing different views of the system are used—for example, activity diagrams and class diagrams. As a consequence, there is a need to analyse the changes of each view and the inconsistencies that these may cause with other views when instantiating a product model. In our work on VML*, we have not discussed this issue so far, but some previous work on this topic exists from other groups—for example, [37-38].

6 Conclusions

This paper presented a generative approach to building a family of languages for specifying the relationship between variability models and other models in software-product-line engineering. Our experience shows that the proposed infrastructure is powerful enough to support generating different language instances (in addition to the two languages presented here, we are currently developing VML* languages for mapping to openArchitectureWare workflows as well as a number of project-specific DSLs) and that it can reduce the effort required to learn about the support infrastructure for such languages. Specifically, regarding the challenges we identified in Sect. 2.2, our generative approach to the family of VML* languages provides the following solutions: reuse is substantially improved over a copy-and-paste approach as all reusable parts of the infrastructure are encoded in the generator and all variable parts are explicitly configured through language descriptors (Challenge 1). Because all dependencies on varying variability and target models have been made explicit in the language descriptor, model access code could be completely disentangled from the actual model manipulation code (Challenges 2 and 3).

In implementing our prototype, we identified a need for aspect-oriented code generation beyond what is offered by current code-generation engines. Our system is structured such that the code generators for the basic VML* infrastructure and for each evaluation aspect are kept in separate modules. This is sensible because evaluation aspects can be included or excluded from a specific VML* language as required. For some files generated (for example, for plugin descriptors contained in `plugin.xml` files) there is a conflict between code generators for the evaluation aspects: each evaluation aspect needs to contribute to the final contents of the file. Using separate code-generation templates for each evaluation aspect would result in a file containing only the contributions from one evaluation aspect. Aspect-oriented code generation could provide a solution here: it effectively allows the results of two or more different generators to be merged into one output file. However, all current aspect-oriented code generators [18, 39] only support asymmetric aspect orientation.

This requires one template to be declared as the base template while the other templates are aspect templates. These aspect templates can then manipulate generation rules in the base template, providing before, after, and around advice for code generation. For our purposes this is not appropriate; because evaluation aspects may be included or excluded as required, we cannot rely on any one of them being present. Consequently, no template defined for an evaluation aspect can be made into the base template. As the basic VML* generator does not provide a template for `plugin.xml`, this can also not be designated as the base template. For our prototype, this problem has been solved by breaking the encapsulation of evaluation-aspect code generators in a controlled way. However, a cleaner solution using a more symmetric approach to aspect-oriented code generation remains for future work.

References

[1]   K. Pohl, *et al.*, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer, 2005.

[2]   P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2002.

[3]   K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*: ACM Press/Addison-Wesley Publishing Co., 2000.

[4]   K. Kang, *et al.*, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Technical report, CMU/SEI-90-TR-0211990.

[5]   M. Alférez, *et al.*, "A Model-Driven Approach for Software Product Lines Requirements Engineering," in *proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, San Francisco Bay, USA, July 2008, pp. 779-784.

[6]   K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, Tallinn, Estonia, September-October 2005, pp. 422-437.

[7]   F. Heidenreich, *et al.*, "FeatureMapper: mapping features to models," presented at the Companion of the 30th international conference on Software engineering, Leipzig, Germany, 2008.

[8]   D. Batory, *et al.*, "The Objects and Arrows of Computational Design," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, France, Toulouse, 2008, pp. 1-20.

[9]   S. Soares, *et al.*, "Supporting software product lines development: FLiP - product line derivation tool," presented at the Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, Nashville, TN, USA, 2008.

[10]   T. Ziadi and J. M. Jézéquel, "Software Product Line Engineering with the UML: Deriving Products," *Software Product Lines 2006,* pp. 557-588.

[11]   G. Botterweck, *et al.*, "Model-Driven Derivation of Product Architectures," in *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*, Atlanta (Georgia, USA), November 2007, pp. 469-472.

[12]   P. Sánchez, *et al.*, "Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines," presented at the Software Language Engineering 2008, Toulouse, France, 2008.

[13]   N. Loughran*, et al.*, "Language Support for Managing Variability in Architectural Models," in *Proc. of the 7th Int. Symposium on Software Composition (SC)*, March 2008, pp. 36-51.

[14]   M. Voelter and I. Groher, "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development," in *Procceedings of the 11th International Software Product Line Conference (SPLC)*, Kyoto (Japan) September 2007, pp. 233-242.

[15]   K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*: Addison-Wesley, 2000.

[16]   P. Jayaraman*, et al.*, "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis," in *Proc. of the 10th Int. Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Nashville, (Tennessee, USA) September-October 2007, pp. 151-165.

[17]   F. Jouault and I. Kurtev, "Transforming Models with ATL," in *Satellite Events at the MODELS 2005 Conference*, Montego Bay, Jamaica, 2005, pp. 128-138.

[18]   *OpenArchitectureWare*. Available: http://www.openarchitectureware.org/

[19]   G. Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software," in *Proceedings of the 2nd Int. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, Charlottesville, (Virginia, USA), September-October 2003, pp. 446-453.

[20]   K. Czarnecki*, et al.*, "Staged Configuration Using Feature Models," in *Proceedings of the 3rd International Software Product Line Conference (SPLC 2004)*, Boston, MA, USA, pp. 266-283.

[21]   M. Volter and T. Stahl, *Model-Driven Software Development*. Glasgow, UK: Wiley, 2006.

[22]   *VML* Download*. Available: http://www.steffen-zschaler.de/publications/vmlstar/

[23]   M. Alférez*, et al.*, "A Metamodel for Aspectual Requirements Modelling and Composition,"
http://ample.holos.pt/gest_cnt_upload/editor/File/public/AMPLE_WP1_D13.pdf, AMPLE D1.3, 2007.

[24]   M. Alférez*, et al.*, "Multi-View Composition Language for Software Product Line Requirements," in *Proceedings of the 2nd Int. Conference on Software Language Engineering (SLE)*, Denver, USA, 2009.

[25]   A. Sousa. (2008, *AMPLE Traceability Framework Frontend Manual.* Available: http://ample.di.fct.unl.pt/Front-End_Framework/ATF%20Front-end%20Manual.pdf

[26]   *Generative Software Development Group, U. Waterloo, Feature Modelling Plugin (FMP) for Eclipse*. Available: http://gsd.uwaterloo.ca/projects/fmp-plugin/

[27]   (2009, *VML* Download.*

[28]   G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*: John Wiley, 1998.

[29]   S. Ian and S. Pete, *Requirements Engineering: A Good Practice Guide*: John Wiley and Sons, 1997.

[30]   T. Young, "Using AspectJ to Build a Software Product Line for Mobile Devices - www.cs.ubc.ca/grads/resources/thesis/Nov05/Trevor_Young.pdf," University of Waterloo, 2005.

[31]   M. Voelter, "A Family of Languages for Architecture Description," presented at the Conference on Object-Oriented Programming, Systems, Languages, Orlando, Florida, 2008.

[32]   D. H. Akehurst*, et al.*, "Supporting OCL as part of a Family of Languages," in *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, 2005.

[33]   E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," presented at the I*nternational Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Heidelberg, October 2008.

[34]   Ø. Haugen*, et al.*, "Adding Standardized Variability to Domain Specific Languages," in *Proceedings of the Conference on Software Product Lines (SPLC'08)*, 2008, pp. 139-148.

[35]   D. Batory*, et al.*, "Scaling Step-Wise Refinement," in *IEEE Transactions on Software Engineering*, 2003, pp. 355-371.

[36]   L. Fuentes*, et al.*, "Feature-Oriented Model-Driven Software Product Lines: The TENTE approach," in *Proceedings of the Forum of the 21st International Conference on Advanced Information Systems (CAiSE)*, Amsterdam, The Netherlands, 2009.

[37]   S. Thaker*, et al.*, "Safe Composition of Product Lines," in *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE)*, Salzburg, Austria, 2007, pp. 95-104.

[38]   M. Janota and G. Botterweck, "Formal Approach to Integrating Feature and Architecture Models," in *Fundamental Approaches to Software Engineering (FASE)*, Budapest, Hungary, 2008, pp. 31-45.

[39]   *MOFScript*. Available: http://www.eclipse.org/gmt/mofscript/