Adding Standardized Variability to Domain Specific Languages

Øystein Haugen
SINTEF and Univ. of Oslo
Oystein.Haugen@sintef.no

Birger Møller-Pedersen
University of Oslo
birger@ifi.uio.no

Jon Oldevik
Univ. of Oslo and SINTEF
jonold@ifi.uio.no

Gøran K. Olsen
SINTEF
Goran.K.Olsen@sintef.no

Andreas Svendsen
SINTEF
Andreas.Svendsen@sintef.no

Abstract

We show how a common language of variability can be used to enhance the expressiveness of a Domain Specific Language (DSL). DSLs have been proposed as a mechanism for expressing variability. Variability between models in a given domain or of a family of systems is captured by language constructs, implying that all possible models in this language are the allowed variations. We explore the possibility of expressing variability in a language independently of the base modeling language. We explore how this works for small DSLs as well as for general purpose languages like UML. Implications of this approach are that the variability language can be standardized, and that DSLs do not have to include variability mechanisms.

1. Introduction

Variability models for system families come in two very different forms: as pure feature models that are independent of any design or implementation model, or as variability models that are related to a base model: elements of the base model will become elements of specific models (or not) according to resolutions of related variability models.

In this paper we focus on the second form for which two main approaches have emerged:

- Annotating the base model by means of extensions to the modeling language, e.g. UML [16] profiles with stereotypes, see e.g. [6] and [7]. The annotated models are unions of all specific models in a family of models.
- Making separate, orthogonal variability models that apply to a single base model. Examples are [13] and [1]. In [1] this approach has been coined the BVR-approach: Base-Variation-Resolution.

The advantage with the annotation approach is that model elements subject to variability is clearly marked, while the disadvantage is that base models are cluttered with variability specifications. The disadvantage with the BVR approach is that base model elements subject to variability are not clearly marked, however, the main advantage is that there may be more than one variability model for each base model, as indicated in Figure 1.

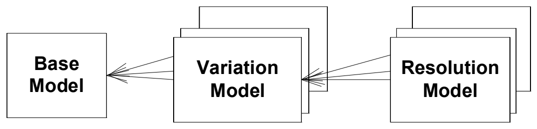

Figure 1 BaseVariationResolution - BVR

The BVR approach described in [1] was developed within the Families project [5]. The main focus in that work was variability models for base models in general purpose modeling languages like UML, so some of the variability mechanisms in the language for making variability models relied on the existence of certain base model language mechanisms. This paper reports on work within the ITEA project MoSiS (ITEA 2 - ip06035) to apply (and thereby further elaborate) the Families variability language also to DSLs that may not have the language mechanisms of general purpose languages. The aim is to come up with a variability language that may enhance both DSLs and general purpose languages.

2. Approaches

We have identified two different approaches to the combination of a DSL (or any language) and a variability language.

- Amalgamated language
- Separate languages

To exemplify our approaches we use a very simple domain specific language called ARI that can model arithmetic expressions. The metamodel for ARI is given in Figure 2.

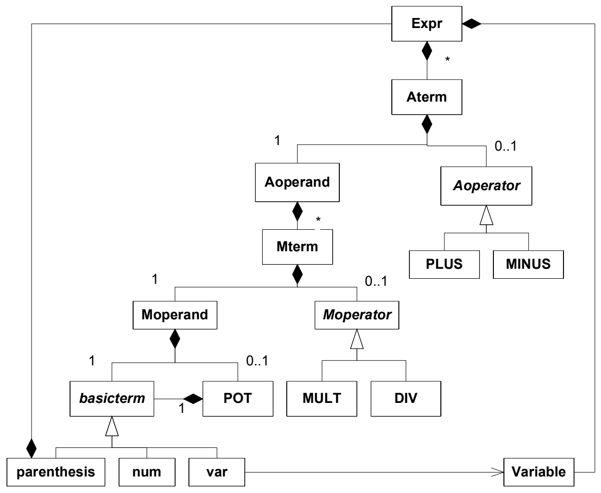

Figure 2 ARI metamodel

A textual concrete syntax with infix notation for the operators gives us the following examples:

- 3*x^2 – 4*x + 1
- 3*x*y + 5

Whenever an Aoperator is absent from an Aterm logically a PLUS is assumed and whenever a Moperator is absent from an Mterm a MULT is assumed.

2.1 The amalgamated approach

The amalgamated language is formed by the DSL and the variability language being combined into one language, e.g. by importing the meta model of one of the languages into the meta model of the other language. The combined language either has a new, combined syntax, or a syntax based on the DSL extended by the syntax of the variability language (e.g. profiling UML).

In practice the situation is typically that the creators of the DSL realize that even though the DSL is expressive enough to describe the whole domain, the language is not quite expressive enough to achieve effective reuse and simple model maintenance. Often the DSL designers will look for more general language constructs to achieve this.

Regarding the simple ARI language, to express more general patterns that could be reused, it would be tempting to reach for a concept of a function, see Figure 3.

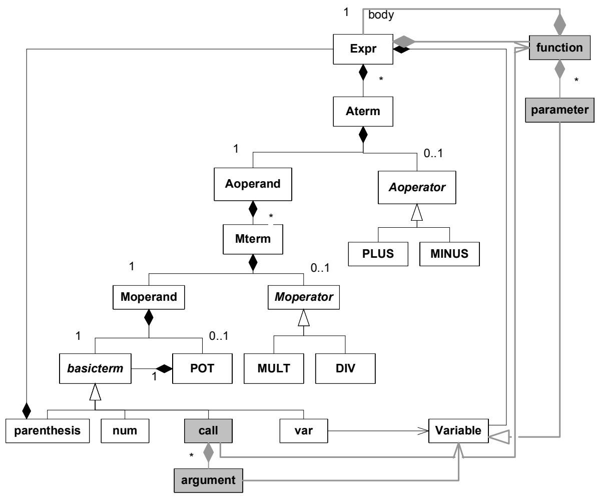

Figure 3 Adding functions to ARI

The concepts added to ARI to achieve reuse are all very generic and can be seen in many languages. Now we are able to express such terms as:

- f(x) = 3*x^2 – 4*x + 1
- g(y) = f(y)*R

We could have imported the four “function-oriented” metaclasses from a meta-package, but we would have had to associate these metaclasses with those of the DSL. As an example the ‘call’ metaclass has to be defined as a subclass of the metaclass ‘basicterm’, while this may not be the case for the general ‘call’ metaclass. The competence needed to include general concepts into a domain specific one is quite high, and when having included these constructs into the DSL you have also included the obligation to implement the new concepts in your code generator.

Finally your DSL is no longer fully domain specific as there are several concepts that are more general than domain specific. Another challenge for the language designer is how to handle that the metamodel appears cluttered over time and the dedicated domain specific concepts are obscured.

2.2 The separated-languages approach

The separated-languages approach means combining a Common Variability Language (CVL) with the domain specific DSL. CVL model elements will relate to DSL elements by simple references. The two languages will each have their own concrete syntax, but the combination may still appear as an integrated concrete syntax.

In this paper we explore the separate-languages approach. DSLs are often made simple and should stay simple. Variability can be considered a domain of its own and therefore a candidate for a simple DSL, just focusing on variability. Standardization of how to model variability also calls for a variability language separate from the base modeling language(s).

We shall go through some of the elements that should be contained in the variability language. We start by variability on simple values.

Take our ARI language and assume that we add to it a variability language that simply has the ability to express the *variability of values*. The combination of ARI and a variability language will now have the ability to express template-like concepts that are *similar* to what may be expressed by introducing a function concept. It is only similar because we by introducing a template-like mechanism do not introduce function calls with its dynamic semantics. If we want to express as a concept a polynomial in x of second order where the coefficients may vary, we can start by any second order polynomial term such as "3*x^2 – 4*x + 1" as our base model. Then we would describe in the variability language three substitutions, each one suited to substitute the coefficients 3, 4 and 1 with other values. We could make up a simple parameterized notation for this by annotating every sub-term in the ARI-term that should be modified by a subscript that refers to a parameter value:

- vary-coeffs(A,B,C): 3_A*x^2 – 4_B *x + 1_C

Intuitively it is obvious what this means, as any instantiation of the term will be an ARI-term where the coefficients have been modified to be those of the given values of terms A,B,C. Our resolution model could be the term

- vary_coeff(6, 5, 7)

and the resolved model would then be

- 6*x^2 – 5_B *x + 7_C

Let us now continue to introduce another simple form of variability, namely the choice between variant model elements (in terms of meta objects – objects according to the ARI meta model). In our simple ARI model of the second degree polynomial, we could be interested in choosing between using MINUS or PLUS for the Aoperator. In the ARI language (Figure 2), MINUS and PLUS are two different metaclasses and in the representation of the formula, one meta object might need to be exchanged by another. This is quite similar to value variability, and we therefore call it *reference variability*. The changed value is a reference value and in addition a new (replica of an) object must be created.

Using a common notation for choice already used in BNF (a domain specific language for grammars), we can write our second order polynomial like:

- polynom2(A,B,C): 3_A*x^2 {–|+} 4_B *x {–|+} 1_C

In general there may be many references between objects of a repository for a model defined by a DSL, and substituting sets of model element objects with other model element objects may be desirable. In general we should then need a concept in the variability language that would express such groups of model element objects (constituting a fragment of the model) with an interface to elements of the rest of the model. We shall use a slightly more elaborate DSL to illustrate this after briefly showing how we have evolved the Families variability model into the CVL metamodel. As indicated above, the CVL metamodel will consist of a metamodel for expressing variability and a metamodel for expressing resolution.

3. BVR revisited

The BVR approach is defined by a language (in terms of a meta model) for specifying Variation Models and Resolution Models, see Figure 4. The Base Model with Model Elements represents the meta model of some modeling language in which base models are made and therefore not subject for definition by the BVR approach. Variation Models will for short be called VAR-models, while Resolution Models will be called RES-models.

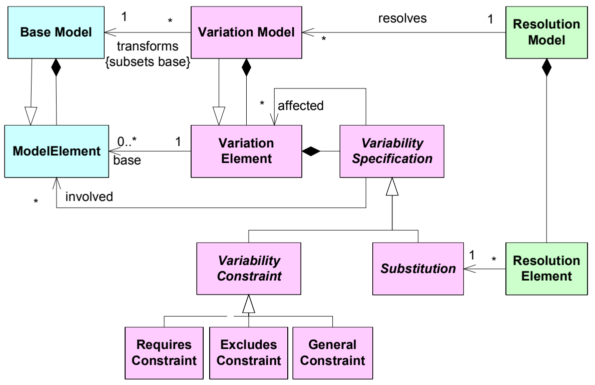

Figure 4 BVR detailed

A variation model applies to a base model, and through variation elements with associated variability specifications, the variation model specifies variants of the base model that may be specified by resolution models. In this paper we concentrate on the Substitutions, but the Variability Constraints of course play an instrumental role in the specification of the variability.

Each Variation Element references a set of Model Elements in the DSL-repository. These are the elements that will be involved in the compound transformation defined by the contained Substitutions.

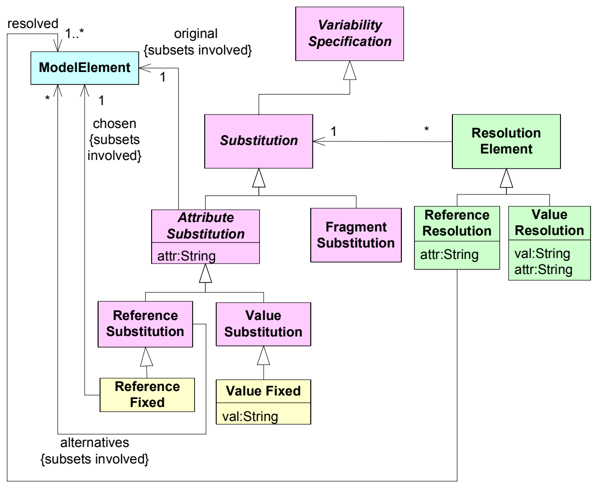

Figure 5 Variability Specifications

Figure 5 shows the value and reference substitutions that we introduced intuitively in the earlier section. Notice that we have introduced two specialized Substitutions, ValueFixed and ReferenceFixed. They represent the situation when the resolution model has been applied to value and reference substitutions, but before the final transformation on the base model has been performed.

We illustrate this two stage transformation process in Figure 6. First the VAR-model and the associated RES-model are fed into the generic Resolution Transformation and the result is another VAR-model where all the resolutions have been incorporated and represented by the Fixed elements. In fact, we may have RES-models that only resolve some of the variabilities, and then the resolution process will produce a VAR-model that cannot be fed into the final Variability Transformation.

If all the variability is resolved, then the VAR-model together with the referred base model (the DSL-model) are fed into the generic variability transformation that will result in a product model defined in the DSL.

For each resolution model the resolution elements specify which substitutions should be applied and thereby transform the base model into a new (product) model expressed in the DSL (see Figure 6). Substitutions have to adhere to the variability constraints.

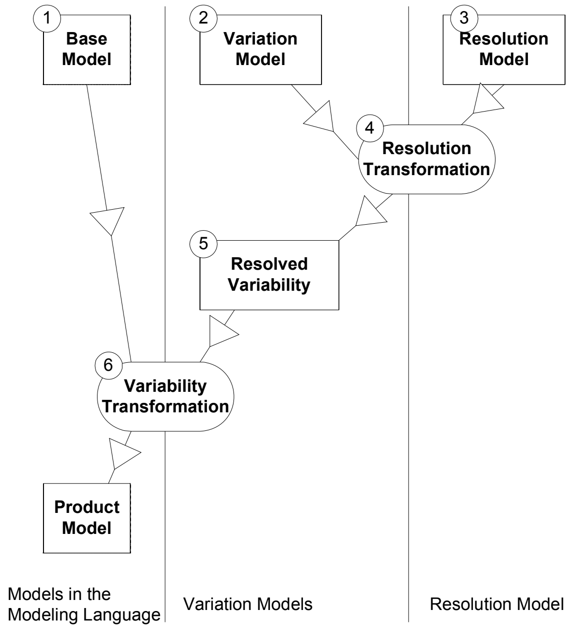

Figure 6 Resolution Process

The approach relies on both base languages and the variability language (for variation models and resolution models) to be defined by means of a common meta-meta modeling language. The references between variation model elements and base model elements and between resolution elements and specific variation model elements will thereby be in terms of references to model elements as defined by this common meta-meta modeling language. In this paper we will consider EMOF [11] as this language, but the approach as such can be applied to settings where other meta-meta modeling languages are used.

We indicated above that the ARI example may be too simple for the reader to appreciate the need for flexible means to express structural variability where fragments of models (in terms of integrated sets of objects of the repository) are replaced and augmented by other such fragments and hooked up. To describe this situation we have been inspired by concepts of composite structures e.g. in UML 2 [16] where the interfaces of the structures are given by ports or gates.

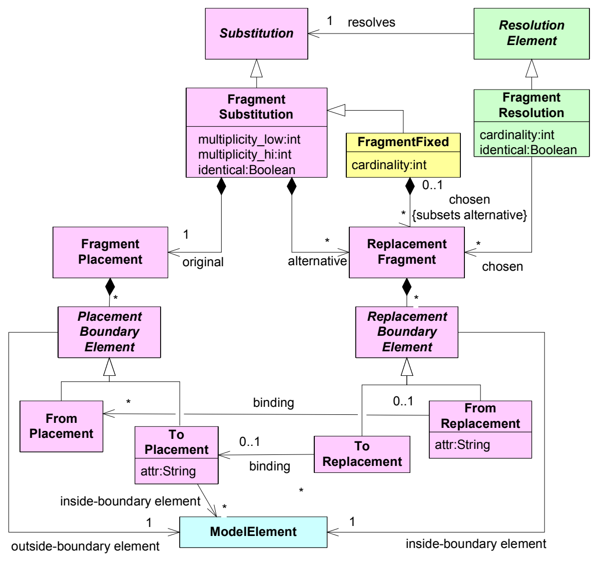

Figure 7 Fragment Substitution in CVL

Figure 7 shows the CVL metamodel concepts relating to the Fragment Substitution concept. A placement fragment (original) of the base model is the fragment of the model that may be replaced by so-called (alternative) replacement fragments. A fragment is defined by a set of boundary elements that give the boundary between the fragment and the rest of the model. When replacing a fragment by another fragment, these boundary elements tell which references between model elements (in terms of meta objects) shall be updated in order to have a model according to the metamodel of the base langauge.

We use the Fragment Substitution concept to express *iteration* similar to multiplicity on parts in UML composite structures, and it is even possible to use substitutions to express *choice* between variants since the substitution may point out more than one model fragment as alternatives and these may be interpreted as separate choices. In the case where the resolution model defines more than one chosen alternative for a substitution, this means that a copy of each replacement fragment should be included. The Fragment Substitution concept is also applied to express *options*. When the FragmentFixed has no chosen fragment this means that all the involved objects of the original fragment are simply removed (and the hole closed by the replacement boundary elements).

The fragment substitution concept can be seen as a generalization of several of the central feature model concepts and it also incorporates the notion of staged configurations with feature diagram references [3].

We shall illustrate the fragment substitution with boundary elements as overlays to the concrete syntax of a simple graphic language for the description of train stations in Section 4.2.

4. Three languages combined with CVL

As mentioned above, our goal is to define a variability language such that it may be combined with different kinds of DSLs (from the simplest textual language like ARI – the language expressing arithmetic formulas, to more general DSLs) and with general purpose languages like UML. In this Section we shall consider how our variability language combines with three languages. In addition to ARI and UML we shall introduce the Train Control Language, TCL.

4.1 ARI

ARI is a DSL for simple arithmetic expressions. We used ARI in Section 2.1 to highlight the difference between the amalgamated approach and the separate-languages approach.

While ARI can express any arithmetic expression and therefore any polynomial, it cannot express the concept of a polynomial. This subtle distinction is important not only formally, but also to understand the difference in thinking and the difference in methodology between only using a DSL and using a DSL combined with CVL.

The DSL purists may claim that there is no need for CVL since everything can be described in the DSL. In the ARI case this means that every polynomial may be defined in ARI and why would you need CVL?

The arguments are threefold. First, there is the argument of abstraction levels. Every polynomial can be expressed, but the abstract notion of a polynomial cannot be expressed. By adding CVL, it is possible to define concepts like 'second order polynomial' (as pointed out in Section 2.2) and even the generic notion of 'polynomial' by applying the fragment substitution concept indicated in Figure 7.

Second, there is the argument of implementation. If you insist using only the DSL, you will soon reach a situation where you would need to add concepts for abstraction (like functions in ARI). Once you add these concepts, you need to implement them throughout your modeling framework, and your DSL metamodel has changed with all the migration challenges that this involves.

Third, we have the argument of standardization. A common CVL language and CVL tool should make it simpler and faster to define a domain specific language, but still be a very expressive language and tool environment for product line development.

4.2 Train Control Language

A Train Control System describes which train routes that are available for a train to use at a given time. This is to avoid two trains on a collision course. A train route is defined as a route between two main signals, but for simplicity we exclude signals from this example.

The Train Control System is described by a DSL that is called the Train Control Language (TCL). Some of the concepts of this DSL are illustrated in Figure 8. The base language contains the concept *TrackCircuits*. *TrackCircuits* consist of *LineSegments* and *Switches* with *Endpoints*.

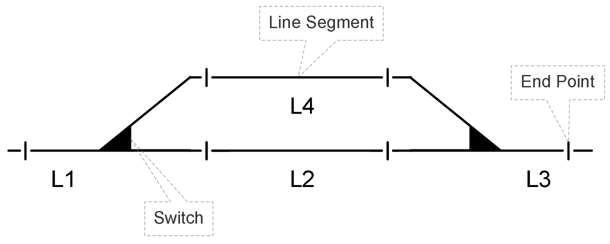

Figure 8 TCL

Assume that we have a station with only a single track, and another station with double tracks. These two stations can both be expressed by TCL. We want to show that these two stations can be seen as variants of the same model by introducing the variability language defined in Section 3. We start by defining a generic train station. The elements to vary can be represented in a variation model, and we can then decide how many tracks the station should have by a resolution model.

We assume that the generic station only has one track. We want to use the variability language to derive a new station with two tracks connected by two switches as shown in Figure 8.

The variability model defines the varying elements. To be able to perform the transformation from one track to a double track station, we have to substitute a fragment of the original model with a (replacement) fragment that represents the double track. We want to substitute the *SingleTrack* of the Placement Fragment in Figure 9 with the *DoubleTrack* of the Replacement Fragment in Figure 10.

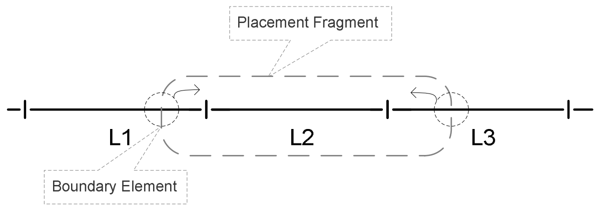

Figure 9 Generic Train Station (with SingleTrack)

Every element we want to add to the model has to be part of the base model. This implies that DoubleTrack in Figure 10 has to be part of the base model as well as the original station in Figure 9. Notice that we have defined the notion of *Fragment Placement* in Figure 9 by an overlaid notation giving the involved model elements by the rounded rectangle and the boundary elements by two circles.

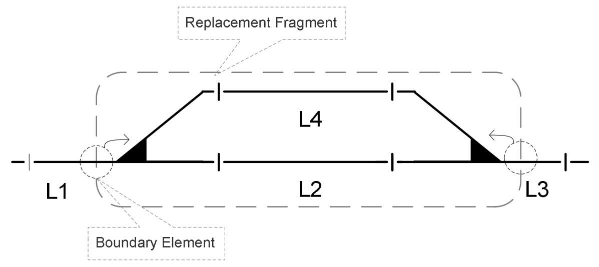

Figure 10 DoubleTrack

In Figure 10 we have defined the Replacement Fragment by a rounded rectangle and circles as *Replacement Boundary Elements*. Please notice that the dashed-line notation is for the graphic variability language and it overlays the TCL notation.

Since we want to substitute one model fragment with another, and this fragment has other elements referring to it, we have to bind these Boundary Elements in a special way. By applying Boundary Elements (see Figure 9 and Figure 10), we are able to express this substitution. *LineSegments* L1 and L3 in Figure 9 refer to Boundary Elements that refer into *SingleTrack*. We are able to substitute *SingleTrack* with *DoubleTrack* by updating the *Placement Boundary Elements* with information from the corresponding *Replacement Boundary Elements* shown in Figure 10.

When we perform the resolution process, we need a placement where we may insert the *DoubleTrack*. We also need a reference to the *DoubleTrack*, which is to be inserted. The elements for *DoubleTrack* are copied and the references between model elements are updated. This makes it possible to insert more than one *DoubleTrack*, if that should be our intention. Note that we use the boundary elements in the variability model to connect the elements together. This makes it possible to make the necessary reference updates on the copy.

We have illustrated the boundary element concepts with concrete syntax rather than through repository examples, and the reason for that is that the repositories quickly get too large for a figure in this kind of paper. Still we should bear in mind that our transformations are in fact done on the repositories of the CVL model and the base model.

In Section 4.1 we gave three reasons for why DSL+CVL is a good idea. Our experiences from designing and evolving TCL add one more reason, namely that separating the DSL and the variability language also provides a good separation of concerns and of developing efforts. The domain experts can concentrate almost exclusively on defining a good DSL for the domain. In this case the railroad engineers and train control experts could focus on making TCL such that every interesting rail station can be defined.

They can also in turn concentrate on defining code generators for TCL while the variability experts work on showing how the CVL can be seamlessly integrated with TCL and giving examples for how it may work.

4.3 UML

UML 2 is a general language that includes quite a number of mechanisms to express abstractions and variability. In [1] Bayer et al. have described how variability is expressed in UML 2 and in [9] Haugen and Møller-Pedersen have elaborated on the difference between direct modeling and generative modeling.

It may not be obvious what can be gained from combining CVL with an advanced language like UML. What reasons already stated in Sections 4.1 and 4.2 still apply?

The first reason does not apply as we know that UML has most of the abstraction constructs that you can find in any language, and the second reason is also dubious as there are already many tools that provide UML implementations. Finally the third reason is hardly true since the standardization of UML must have taken more efforts than most languages. In Section 4.2 we added the reason of separation of labor between the DSL designers and the CVL-specialists. This reason is at best absolutely irrelevant since when using the UML there are no DSL language designers.

As is pointed out in [9] there are variability challenges that are not easily handled in UML. The separation of the generic product line and the different product variants can be hard to distinguish in UML. The separation of labor following the UML+CVL combination is potentially fruitful. There is no reason why selecting product variants should need detailed UML knowledge. Neither is there any reason why domain knowledge (of the product domain) should be overridden by UML knowledge.

We propose a profession of product configurators that have strong product domain knowledge and adequate command of the variability language. They will develop resolution models based on an elaborate CVL model that in turn refers to UML models.

The CVL model is made by experts with both domain knowledge and detailed knowledge of the UML system. They may still not need to have the same kind of expertise that the engineers that actually made the UML system had. These CVL-experts see the UML system on fairly high abstraction level; one might perhaps say that they see the domain specific concepts having been defined in UML. Upon these they build the variability.

Another reason that the combination UML+CVL may be fruitful is that products defined by UML are often rather large products, and it is not always clear from the start what kind of variability the customers will want in the future. In fact their configuration wishes may be crosscutting the design principles of the UML definition of the product line. The CVL model can be seen to handle such crosscutting concerns very similar to aspect oriented modeling [10] since the variability primitives represent transformations on the generic original.

5. The model transformations perspective

One of the main requirements for advocating the DSL+CVL approach is that CVL can be made independent of the DSL. In order to support product resolution generally, it is desirable to have generic *domain-independent* transformations from product line to product. We have implemented such domain-independent transformations that based on variability, resolution, and base models generate a new and resolved product model in the base language. These transformations have been implemented by a two stage process, as illustrated by Figure 6.

The *Resolution Transformation* takes a Variation model and a Resolution model as input and produces a *resolved Variation-model*. The *Variability Transformation* takes the resolved Variation-model and a domain-specific model as input and produces a *new*, resolved domain-specific base model. The domain-specific metamodel may be any EMOF-based metamodel, such as ARI, TCL, or UML metamodels.

5.1 The resolution transformation

The feature resolution transformation generates a new and modified Variation-model, which contains new elements representing the resolutions made. The resolved model is populated with new transformer objects of classes ValueFixed, ReferenceFixed, or FragmentFixed that define the resolutions as defined in Section 3.

The population of the resolved Variation-model is a straight-forward transformation: for each resolution defined in the RES-model, a corresponding *fixed Substitution* is added to the VAR-model by copying the information provided by each resolution.

5.2 The domain variability transformation

The domain variability transformation for the product line model is done by processing the resolved VAR-model and executing the corresponding transformation on the product line base model. The inputs for the transformation are the VAR-model and implicitly the base model(s) that are referenced by the VAR-model.

The strategy chosen for producing the actual product model is by manipulating a copy of the original base model referenced by the variation model through the association *transforms*. From that, model elements are deleted or populated according to the resolved VAR-model, resulting in a new product configuration.

The basic processing iterates all variation elements in the resolved VAR-model. All elements that contain a resolved *Substitution* (*ValueFixed*, *ReferenceFixed*, or *FragmentFixed*) are processed as part of the product configuration. The associated base model element are modified, added or deleted in the new model.

5.2.1 Modifying the base model

As we want our transformation to be general, it cannot explicitly use concepts from the DSL metamodel. Base model properties must be accessed by reflection in order to be modified. The transformation uses the information provided by the resolved (*fixed)* substitutions to determine the access to the base model. For example, an *AttributeSubstitution* defines the name of the property that should be accessed in order to set the resolved value for the transformation.

5.2.2 Setting Property Values

The *ValueFixed* or *ReferenceFixed* types require the transformation to set property values or references on a domain object. To do this, the transformation needs information on which property to access. The setting of property values is illustrated by the transformation rule in Program 1, where attribute names are fetched from the *attr* property of the *AttributeSubstitution* in the VAR-model.

```
vMdl.ValueFixed::setPropertyValue
(obj:ecore.EObject, value:String) {
  if (self.attr != null) {
    var class:ecore.EClass = obj._getClass();
    var attr:ecore.EAttribute = class.
         eAttributes->select(a:fMdl.EAttribute
         | a.name.equals(self.attr))
    if (attr != null) {
       obj._setFeature(self.attr, value);
    }
  }
}
```

Program 1 Setting Property Values

5.2.3 Deleting Objects

Deletion is taking place in the course of a *Fragment Substitution*, where base model elements identified by a *Fragment Placement* are deleted to be replaced by elements in a selected alternative *Replacement Fragment.* The transformation iterates and deletes the elements defined as *inside boundary elements.* Deletion is done on the *copy* of the original base model. Deleting an object from the base model copy is handled by invoking a delete operation on the object, which deletes the object and references to it. The deletion operation does not require any knowledge of the product line metamodel.

5.2.4 Moving Object References in Fragment Substitutions

In order to implement *Fragment Substitutions*, model element references are modified, and model elements might be moved from one element container to another.

The set of model elements from the base model to be moved are defined by the *Replacement Fragment* selected by a *FragmentFixed* instance, in its set of *Replacement Boundary Elements.* The transformation processes each *to/fromReplacement* instance and modifies the object structure as follows: for each *toReplacement*, the corresponding *outside-boundary element* of its binding is modified by setting the feature defined by the binding's *attr* property to reference the model element referenced by its *inside-boundary-element*. Correspondingly, for each *fromReplacement*, the element referenced by its *inside-boundary-element* is modified to point to the element defined by its binding's *outside-boundary element* using the *attr* property. Program 2 shows the transformation code for modifying references for *Replacement Fragments.*

```
property atoRep:vMdl.toReplacement = plbe
property atoPla:vMdl.toPlacement =
       atoRep.binding
property bObj:ecore.EObject =
       atoPla.outside-boundary-element
property bCopy:ecore.EObject = getCopy(bObj)
property bCl:ecore.EClass =bObj._getClass()
property bFeature:ecore.EReference =
       bCl._getFeature(atoPla.attr)
property insideTarget = atoRep.inside-
       fragment-element
property insideCopy = getCopy(insideTarget)
var feature:Object = null
if (bFeature != null and bFeature.changeable)
{
  feature = bCopy._getFeature(bFeature.name)
  if (bFeature.upperBound == -1 ||
     bFeature.upperBound > 1) {
    feature.addOrg(insideCopy)
  } else {
    bCopy._setFeature(bFeature.name,
      insideCopy)
  }
```

Program 2 Modifying Model Element Fragments

Our experiments so far show that setting values, deleting model elements, and moving model element references for creating a new domain model can be defined in DSL-independent terms. We have focused on the basic aspects of the transformations, and need to extend and improve our transformations to fully support the expressiveness of the CVL language.

5.3 The implementation

The transformations have been implemented as model to model transformations in an extended version of MOFScript ([4], [12]), as illustrated by Program 1.

MOFScript supports an imperative style of transformation, and although it is designed for model to text transformations, recent extensions provide model to model transformation capabilities as well. Reflection is supported by built-in operations which are delegated to the reflection operations available in on EMF objects.

Since we describe our transformations from an instance of an EMOF metamodel to an instance of an EMOF metamodel, the transformations will most certainly also be definable in QVT [14].

6. Related work

In [2], Czarnecki and Antkiewicz describe an approach for mapping features to models and generating model instances (configurations). They use model templates with feature annotations that are matched and evaluated against a feature configuration. The approach is generally applicable to any model domain based on EMOF, which is the same for our approach. The implementation of their approach, however, seems dependent on the domain, which is avoided in our approach. Czarnecki and Antkiewicz, on the other hand, handle presence conditions described in the model template, which is not addressed in our approach.

Voelter and Groher [17] describe an approach combining aspect-oriented and model-driven development. They define aspects both at the modelling level, the transformation level, and the implementation level. They outline support for positive variability at the modelling level represented by aspects, and employ model weavers to support composition. They do not address representations of the variation model independent from the base model.

In [8], they describe the support for model weaving in XWeave, where they represent the product line as an EMF metamodel and a specific product as an instance of that metamodel. They address the use of model aspects to compose product configurations, but do not address the variability representation or the resolution process.

In [15], Sanches et al define a metamodel for software architecture in aspect-oriented software product lines. The focus is on variability representation and links to an architectural model using the *Variability Pointcut Language* (VPL). This approach provides variability representation that provide cross cutting capabilities (querying) into the base model. However, the approach is restricted to architectures defined in the UML metamodel. The idea of a query-based relationship to base model elements could be a useful extension to our variability language.

Pohl et al [13] define the Orthogonal Variability Model (OVM) for representing variability. In OVM, variability can be described independently of base models, but there is no formalized way of specifying the links to the base model elements, as we propose in the BVR approach.

7. Conclusion

We have presented an approach to product line engineering based on a combination of language design and applying standardized notions of variability.

Through the separation between Domain Specific Languages (DSLs) and a Common Variability Language CVL (for specifying both Variation and Resolution Models), we obtain the following potential advantages:

1. Domain experts can concentrate on domain language concepts only.
2. The DSL thus becomes compact and simple to support both with editors and code generators.

3. The CVL language can be standardized and CVL tools can be made available.
4. The CVL language approach can be taught in general courses.
5. The BVR approach supports division of labor and separation of concerns.

We have given examples to show the usability of the approach and have investigated that the approach is feasible by implementing general domain-independent transformations to resolve the variability within a DSL.

Acknowledgments. The work presented here has been developed within the MoSiS project ITEA 2 – ip06035. MoSiS is a project within the ITEA 2 – Eureka framework. The TCL has been developed in cooperation with project members from ABB Norway.

8. References

[1] Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.-P., and Widen, T.: *Consolidated Product Line Variability Modeling*, in *Software Product Lines: Research Issues in Software Product-Lines*, Käkölä, T. and Dueñas, J. C., Eds.: Springer, 2006.

[2] Czarnecki, K. and Antkiewicz, M.: *Mapping Features to Models: A Template Approach Based on Superimposed Variants*, http://dx.doi.org/10.1007/11561347_28, 2005.

[3] Czarnecki, K., Helsen, S., and Eisenecker, U.: *Staged Configuration Using Feature Models*, SPLC, 2004, vol. LNCS 3154, Springer-Verlag.

[4] Eclipse: *MOFscript*, http://www.eclipse.org/gmt/mofscript/, 2006.

[5] Families: *Families*, in *Eureka S! 2023 Programme, ITEA project ip02009*, 2004.

[6] Fontoura, M., Pree, W., and Rumpe, B.: *The UML Profile for Framework Architectures*: Addison-Wesley, 2001.

[7] Gomaa, H.: *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, vol. 736: Addison-Wesley, 2004.

[8] Groher, I. and Voelter, M.: *XWeave: models and aspects in concert*, ACM, pp. 35--40, 2007.

[9] Haugen, Ø. and Møller-Pedersen, B.: *Modeling Variability - From Direct Modeling to Generative Modeling*, NIK2006 - Norsk Informatikkonferanse, 2006, TAPIR.

[10] Heidenreich, F. and Wende, C.: *Bridging the Gap Between Features and Models*, Second Workshop on Aspect-oriented Product Line Engineering, Lancaster, 2007, vol. COMP-005-2007, Lancaster University.

[11] MOF: *MOF 2.0 (Meta Object Facility)*, http://www.omg.org/spec/MOF/2.0/, 2006.

[12] Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., and Berre, A.: *Toward Standardised Model to Text Transformations*, European Conference on Model Driven Architecture - Foundations and Applications, Nuremberg, 2005.

[13] Pohl, K., Bökle, G., and Linden, F. v. d.: *Software Product Line Engineering - Foundations, Principles and Techniques*: Springer, 2005.

[14] QVT: *MOF QVT*, OMG http://www.omg.org/cgi-bin/doc?ptc/2007-07-07, 2007.

[15] Sanchez, P., Gamez, N., Fuentes, L., Loughran, N., and Garcia, A.: *A metamodel for designing software architectures of aspect-oriented software product lines*, AMPLE Deliverable D2.2, AMPLE: Aspect Oriented, Model-Driven, Product Line Engineering - Specific Targeted Research Project: IST- 33710, 2007.

[16] UML: *Unified Modeling Language 2.1.2*, http://www.omg.org/spec/UML/2.1.2/, 2006.

[17] Voelter, M. and Groher, I.: *Product Line Implementation using Aspect-Oriented and Model-Driven Software Development*, 11th International Software Product Line Conference, SPLC, 2007.