



# SINTEF REPORT

## SINTEF ICT

Address: P.O.Box 124, Blindern  
0314 Oslo NORWAY  
Location: Forskningsveien 1  
0373 Oslo  
Telephone: +47 22 06 73 00  
Fax: +47 22 06 73 50

Enterprise No.: NO 948 007 029 MVA

TITLE

## A Generic Language and Tool for Variability Modeling

AUTHOR(S)

Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, Gøran K. Olsen, Andreas Svendsen, Xiaorui Zhang

CLIENT(S)

Research Project Mosis (directed by ICT Norway)

REPORT NO. <b>SINTEF A13505</b>	CLASSIFICATION <b>OPEN</b>	CLIENTS REF. Research Council of Norway project number 180110/I40	
CLASS. THIS PAGE <b>OPEN</b>	ISBN 978-82-14-04457-7	PROJECT NO. <b>90B246</b>	NO. OF PAGES/APPENDICES <b>20</b>
ELECTRONIC FILE CODE <b>N/A</b>		PROJECT MANAGER (NAME, SIGN.) <b>Øystein Haugen</b>	CHECKED BY (NAME, SIGN.) <b>Arnor Solberg</b>
FILE CODE <b>N/A</b>	DATE <b>2009-12-03</b>	APPROVED BY (NAME, POSITION, SIGN.) <b>Bjørn Skjellaug, Research Director</b>	

### ABSTRACT

This paper presents an approach to variability modeling where variability models are made in a separate, generic language CVL (Common Variability Language) that works with any other language defined by a metamodel. CVL models specify both variabilities and resolutions of these, and by executing a CVL model the base product line model is transformed into a specific product model. Our CVL tool is a generic tool in the sense that the supported transformations work on any model in any language defined by a metamodel. We show how the well known notation for variability, feature diagrams, can be subsumed under CVL as its (partial) concrete syntax. We also demonstrate the use of a simple, but powerful means for parameterization. Furthermore we give a worked-out example from the real domain of train signaling.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	ICT, modeling	IKT, modellering
GROUP 2	Design	Design
SELECTED BY AUTHOR	Variability modeling, feature diagram	Variabilitetsmodellering, feature diagram
	Common variability language	Common variability language
	Software product line	Software product line

# A Generic Language and Tool for Variability Modeling

Franck Fleurey<sup>1</sup>, Øystein Haugen<sup>1</sup>, Birger Møller-Pedersen<sup>2</sup>, Gøran K. Olsen<sup>1</sup>,  
Andreas Svendsen<sup>1</sup>, Xiaorui Zhang<sup>1</sup>

<sup>1</sup>SINTEF

<sup>2</sup>University of Oslo

{name}@sintef.no ; birger@ifi.uio.no

**Abstract.** This paper presents an approach to variability modeling where variability models are made in a separate, generic language CVL (Common Variability Language) that works with any other language defined by a metamodel. CVL models specify both variabilities and resolutions of these, and by executing a CVL model the base product line model is transformed into a specific product model. Our CVL tool is a generic tool in the sense that the supported transformations work on any model in any language defined by a metamodel. We show how the well known notation for variability, feature diagrams, can be subsumed under CVL as its (partial) concrete syntax. We also demonstrate the use of a simple, but powerful means for parameterization. Furthermore we give a worked-out example from the real domain of train signaling.

## 1 Introduction

Variability modeling is intended to capture the essence of how one product is similar, but still different from another. Products that are sufficiently similar are often defined to form a product line. Product line modeling is different from modeling only one singular product. A product line model must contain information on product variants and dependencies. Furthermore the modeling of a product line involves other competences than those of a software specialist. Typically there are domain experts that take part in the modeling of the product line, and also in the configuration of the individual product models.

The competence of domain experts of one (or more) product lines can be synthesized into a domain specific language (DSL). A DSL contains concepts specific to the area implying more effective specification by those knowledgeable in the domain. However, a DSL with merely concepts from the domain will often be too restricted. Therefore DSLs grow to contain abstraction mechanisms found in general purpose languages. Our work presented here is about whether the general concepts related to modeling variability can be defined in a generic language working seamlessly with any other domain specific language. To achieve this we are creating CVL – the Common Variability Language.

The product line communities use feature diagrams to depict the user-centric and domain specific definition of the product line. There is a rich literature on feature

diagrams and feature models. The different dialects have in common that they give a description of the alternatives that need to be bound and the constraints to be fulfilled for a valid product configuration. However, feature diagrams and the corresponding feature models are user-centric and do not cover the details of the transformations required to be implemented in order to get a specific product model according to a configuration of a feature model. Therefore the connection from the feature models to the actual software of the product is often defined ad hoc. We want means to describe any variability formally such that executing the description will yield a product model. This will then formalize the product creation with the added advantage of analyzability and maintainability. The CVL language semantics is defined as a transformation of an original (e.g. a product line model) into a configured, new product model.

In this paper we present the advanced abstraction mechanisms, the mechanisms that support feature modeling, and show how the corresponding tool provides a flexible support of the concrete syntax of CVL.

CVL combines user-centric feature diagrams with an automation-centric approach to the production of product models. Thereby CVL can serve as the language for variability for those who design product lines and configure products as well as those who do the required modeling to produce the right products automatically.

Through a running example we shall show the following:

1. Abstraction mechanisms for reuse, parameterization and choice in CVL.
2. Feature diagram notation as notation for CVL (but recognizing that feature diagrams do not cover the whole CVL).
3. A generic way to combine the CVL tool with base language editors.

The paper is organized as follows: Section 2 presents background on feature diagrams, the Train Control Language (TCL) and CVL basic concepts through our running example. Section 3 presents the general concepts introduced in CVL to make variability models flexible and reusable. Section 4 discusses the challenges related to the concrete syntax of CVL, and also presents the CVL prototype tool. Section 5 presents related work and Section 6 concludes and gives future work directions.

## 2 Introducing the Running Example and Background

Our running example is from the domain of signaling in train stations using TCL – Train Control Language [24]. Our example is somewhat simplified relative to real stations to fit into the format of this paper.

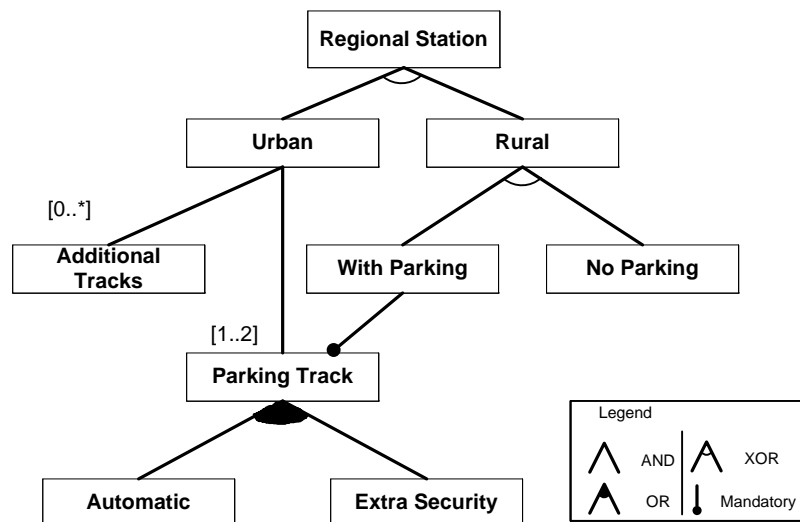
We have designed a feature diagram (Fig. 1) of an imaginary regional station product line with the variability and commonalities of stations within a geographical region. CVL is then introduced, and we provide some of the detailed model fragments used to define substitutions of the original TCL base model. We superimpose the CVL fragments on base models in the concrete TCL notation (Figures Fig. 4, Fig. 5 and Fig. 6). The depiction of placements and replacements is only illustration and the solid and dashed borders are not actual concrete syntax for CVL.

## 2.1 Feature Diagrams Exemplified by the Regional Station Product Line

Feature modeling as a technique for defining features, and their dependencies has been widely adopted in the Software Product Line community. It was originally introduced by Kang as part of Feature-Oriented Domain Analysis (FODA) [15]. There a feature is defined for the first time as a “prominent or distinctive user-visible aspect, quality, or characteristic of a software system”. Feature modeling is a means to reflect user choices and requirements in an early phase of the product design. Therefore we recognize that to relate to the bulk of work done on feature diagrams we should make sure that our CVL subsumes the feature models.

Features are typically modeled in the form of tree-like feature diagrams along with cross-tree constraints, e.g. require/exclude a feature based on the selection of another feature. A feature diagram is considered to be an intuitive way to visually present the user choices of the features, and is therefore widely used and extended. In the FODA notation, a hollow or solid circle in the connection end indicates whether the child feature is optional or mandatory. Sub-features connected with solid arcs represent ‘OR’ relationship, such that at least one of the sub-features must be selected. On the other hand, hollow arcs mean alternative (XOR), i.e. at most one of the sub-features must be selected.

A large number of extensions of FODA has been proposed ([16][12][8][25][1][2][22][5]). The cardinality-based feature diagram proposed by Czarnecki et al [5] integrates several extensions of the FODA notation. They annotate features with cardinalities, such as e.g. [1..\*] or [2..2]. Mandatory and optional features can be considered as features with the cardinalities [1..1] and [0..1], respectively.



**Fig. 1. Feature diagram of regional station product line**

Fig. 1 shows a feature diagram of our running example the Regional Station product line. We distinguish between Urban and Rural stations where Urban stations

can have several additional tracks. Urban stations will have 1 or 2 parking tracks while rural stations may be either with a parking track or without parking.

## 2.2 Train Control Language

The Train Control Language (TCL) is a Domain Specific Language (DSL) that is used to specify the interlocking system of train stations [7, 24]. Fig. 2 shows the TCL concrete syntax as it appears in the TCL editor.<sup>1</sup> Based on the models from the TCL editor the source code, documentation and interlocking tables can be automatically generated.

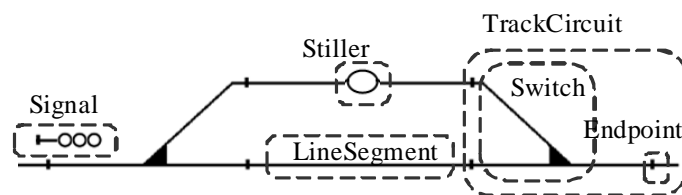


Fig. 2. Train Control Language constructs

The interlocking system is for avoiding train collisions and safety critical issues, by coordinating the availability of the train routes, the signaling system and the direction of the switches at a station, which is handled automatically.

## 2.3 Common Variability Language

As pointed out in [13] the Common Variability Language approach has focused on specifying variability in a model separated from the base product line model. There may be several variability models applying to the same base product line model and the base model is unaware of the variability models (there are only links from the CVL model to the base model). Several product resolutions can apply to the same variability model. CVL is executable, by having specified resolutions of variabilities, the CVL tool automatically produces the specific product model.

The core concepts of the CVL are substitutions. Models are assumed to consist of model elements in terms of object that are related by means of references. The CVL model points out model elements of the base product line model and defines how these model elements shall be manipulated to yield a new product model. There are three kinds of substitutions: *value substitution*, *reference substitution* and *fragment substitution*. The substitution replaces base model elements defined as a *placement* by base model elements defined as a *replacement*. The most elaborate kind of substitution is that of the fragment substitution as illustrated in Fig. 3. The product model on the right side is produced from the base model to the left by means of a fragment substitution. The fragment is defined in CVL by a set of boundary elements

<sup>1</sup> Note that the names in Fig. 2 are the terms of the concepts and the dashed lines are only for explaining their extension and are not part of the TCL syntax as such.

(circles named  $x^p$  and  $x^r$ ) recording the references in and out of the fragment. In Fig. 3 the boundary elements and the corresponding placement and replacement are superimposed onto a base model. In the models this is realized by pointers from the CVL elements to base model elements. A fragment substitution will define a Placement Fragment (solid-drawn circle around 2 and 3) and one or more corresponding Replacement Fragment (dashed line around 5, 6, and 7). The idea is that substitutions define a transformation from a product line model to a product model. Provided the resolution model, the execution of the CVL transformation will bind the boundary elements of the placement to the boundary elements of the replacement e.g. binding  $a^p$  to  $a^r$  and  $b^p$  to  $b^r$ . The transformation is generic and relies on reflection; therefore it can be applied to any language defined by a metamodel.

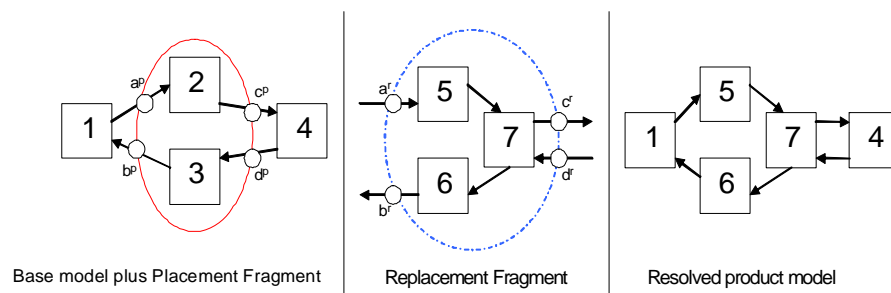


Fig. 3. A simple fragment substitution

### 2.4 CVL Fragments Superimposed on TCL Models

As a starting point for our product line we have the base model in Fig. 4. It is a simple station with two tracks, signals and stillers. We have also defined 3 placement fragments (depicted by superimposed solid-drawn circles). This notation is just for illustration and the boundary elements have been omitted here. These placements are candidates for substitutions where a compatible replacement fragment can be inserted.

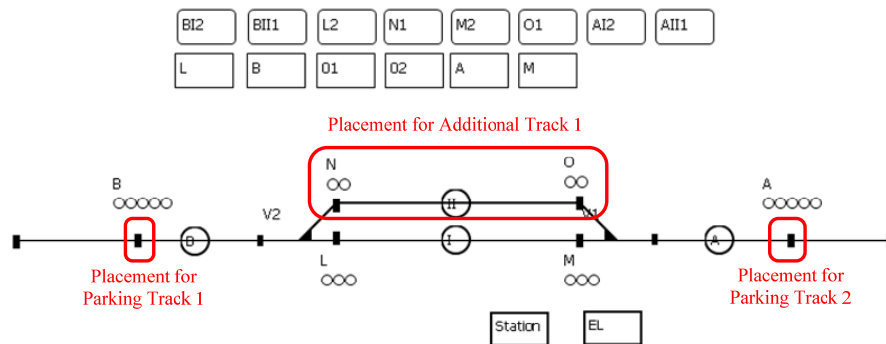
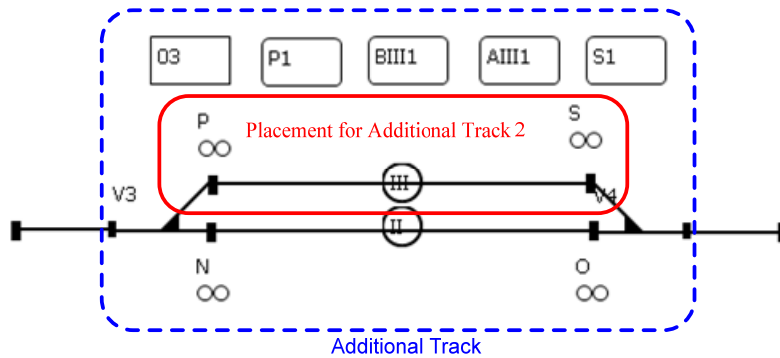


Fig. 4. Initial Base Model with three placement fragments

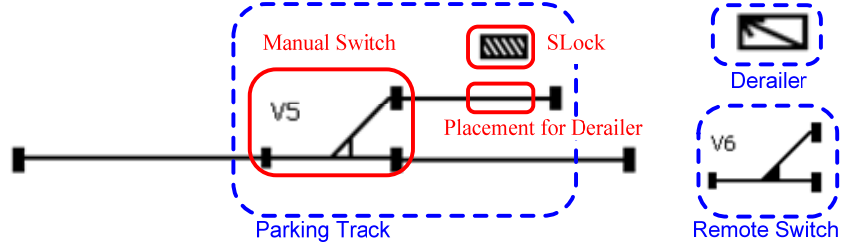
In Fig. 5 we see the replacement fragment Additional Track, which is compatible with the placement Additional Track 1 in Fig. 4. This fragment will be used to add any number of additional tracks to the station. The replacement fragment contains a composite placement fragment that can be replaced by an instance of the Additional Track recursively, creating e.g. a station as presented in Fig. 7 station C.



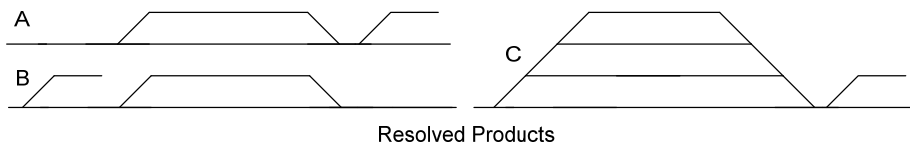
**Fig. 5. Replacement fragment Additional Track with one composite placement**

Note that the replacement fragment in Fig. 5 also could have been defined on the initial base model in Fig. 4, but for clarity we have defined it on its own.

Parking Track, Derailer and Remote Switch in Fig. 6 are replacement fragments concerned with the ability to add parking capabilities to the station.



**Fig. 6. Replacement fragment Parking Track, Derailer and Remote Switch**



**Fig. 7. Example of produced station**

Fig. 7 shows three different products that can be produced based on the base model fragments.

### 3 CVL Abstraction Mechanisms

In Section 2 we presented the user-centric feature diagram and the detailed model fragment substitutions. We will now introduce the CVL language concepts beyond those presented in [13] defining user-centric entities such as “Parking Track” that is a pattern that may occur in Urban or Rural stations.

Recognizing that feature diagrams are user-friendly notations for expressing variability and since our aim is to provide a complete language for variability we must also have concepts that logically cover what feature diagrams can express. Still we do not want to mirror every symbol of the feature diagrams as metaclasses in our metamodel. We want to find the minimal set of concepts that are expressive enough.

#### 3.1 CVL Language

In Fig. 8 we have extracted the most significant concepts of the CVL metamodel. The CVLModel, which is the root element, can contain a variability specification, representing the variability model, and one or more resolution specifications, representing the resolution models. The variability specifications are divided into two groups; Executable Primitive and Declaration. The executable primitives form the flow of the CVL execution, while declarations are used as input for these primitives. For grouping elements we use two mechanisms that can be nested; Composite Variability and Iterator. The main difference between these elements is that while all executable elements in a composite variability will be executed, the iterator represents a choice, where the resolution model can make a selection of the contained executable primitives.

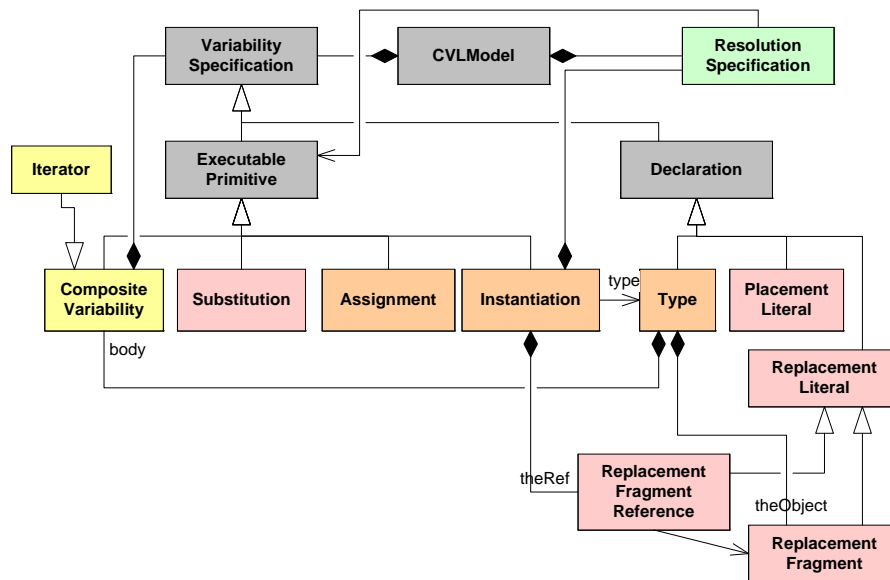


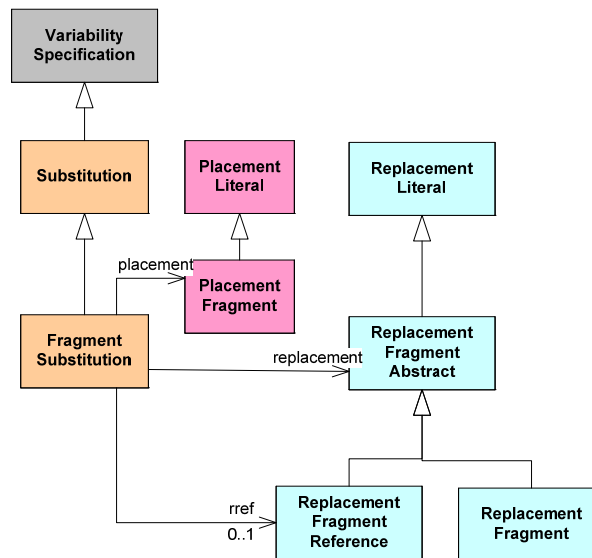
Fig. 8. CVL core metamodel excerpts

The substitution is the simplest form for operation in CVL, and it basically replaces a placement literal with a replacement literal. The type with the corresponding instantiation allows structuring variability specifications for reuse.

The semantics of CVL is described generically by a model-to-model transformation that relates to the actual base language through reflection mechanisms provided by the modeling platform. This transformation has been implemented in an extended version of MOFScript [17] that in addition to model-to-text generation capabilities also supports model-to-model transformations.

As mentioned earlier CVL includes three kinds of substitutions as the core concept: value substitution, reference substitution and fragment substitution. When executing a value substitution the valid object is fetched from the base model and the attribute in this object is retrieved by the property name using reflection. The property is then updated with the new value. Reference substitution is similar, but instead of changing an attribute, it changes a reference.

Fragment substitution is more advanced since it replaces a set of objects (placement fragment) by another set of objects (replacement fragment). The CVL metamodel for fragment substitution is shown in Fig. 9.



**Fig. 9. Fragment Substitution metamodel excerpts**

When executing a fragment substitution the model elements in the placement fragment are first retrieved and deleted. The bindings between the boundary elements of the placement fragment and replacement fragment are then traversed. The references of objects inside and outside the fragments are then updated as follows: References to the placement fragment are updated to refer to objects in the replacement fragment, and references from the replacement fragment are updated to refer to the objects that are referred by the placement. We demand that the references of the placement and the corresponding references of the replacement point to objects

of the same base language type. Thus, we avoid dangling pointers and keep the structure of the model consistent.

### 3.2 CVL Concept for Reuse

Typical concepts for reuse are functions, types and templates. In CVL we have settled for a type concept and corresponding instantiations. During the execution of the CVL specification the instantiation of a type will return a replacement fragment, and instances of the type are therefore replacement fragments.

The body of the type definition is a composite variability such that any transformation expressible through a series of substitutions can be expressed by the type. The body is similar to a constructor in C++ and Java since the substitutions will work on some initial replacement fragment that is eventually returned when the type has been instantiated during the execution. In other words, the substitutions contained by the type customize the replacement fragment. The customization is made possible by the fact that replacement fragments may contain placements.

In order to be able to instantiate a type several times to reuse the composite CVL substitutions, we need to distinguish between different instances of the type. Since it is necessary to refer to these instances in subsequent substitutions, CVL defines replacement fragment reference to store these instances. Other operations, such as assignment, support dynamically update of the references to these replacement fragment instances.

Assume that we have a type Parking Track that we instantiate several times. These parking tracks have to be customized to be automatic, have extra security or both. The type concept in CVL therefore supports customization of replacement fragments. We must also express in full detail how these parking are connected to the available placements in the station. CVL therefore has references that are dynamically updated, and it has replacement fragment references and the associated assignment exactly for that purpose.

### 3.3 CVL Parameterization

Our parameterization of types recognizes that the atomic building block of our variability descriptions in CVL are the substitutions where replacements are put in the place of placements. Thus, placements are similar to formal parameters, and replacements to actual parameters. By allowing replacements (our objects of change) to contain placements we let the objects contain the possibilities for subsequent change. Semantically we require that a placement can only be replaced once which simplifies the semantics and the intuition. Obviously adding yet another train track in our station implies connecting it to the existing ones by substituting some placements that are ready for such connection and left there for that purpose.

While traditional parameter transfer brings the values of the actual parameters into the function prior to its actions, our approach to parameterization is slightly different. Every instantiation of one of our CVL types will return identical objects, but then subsequently their internal placements may be replaced. This corresponds to the

parameter transfer, but it happens after the type instantiation has returned its replacement fragment.

Let us recapitulate how this may look from the perspective of a replacement fragment defined before the CVL description is executed. First, the replacement fragment is defined through selection of pieces of some base model. Let us assume this definition is to define the starting point for a type. Then the substitutions will start modifying a copy of this replacement fragment through replacing the contained placements. Such substitutions may also in fact introduce new placements into the structure. Finally the type instantiation completes and the replacement fragment is returned. This is in turn picked up by the substitutions following the type instantiation and remaining placements may be replaced by new replacement fragments.

Placements do not necessarily need to be replaced. The default is that they remain untouched.

### 3.4 CVL Concept for Choice

Feature diagrams are focused on choices: Which variants are to be chosen? We have defined *one* concept, the *Iterator*, with a lower and upper limit to define the multiplicities of the choice, and one bit of information that we call “isUnique”. Together with the grouping construct of composite variability we can cover what feature diagrams can express, as presented in more detail in Section 4.1.

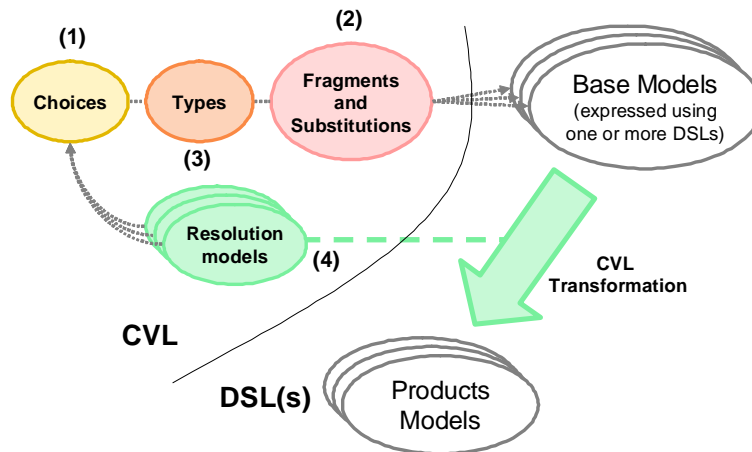
The semantics of the iterator is as follows: The execution of CVL walks through a resolution specifications for every iterator. The resolution specification points out executable primitives contained in the iterator. The number of selected execution primitives is between the lower and upper limits defined in the iterator. If the iterator specifies “isUnique” each selected primitive can only be selected once. The selected execution primitives are then subsequently executed.

## 4 Using CVL with Feature Diagrams

Thanks to the mechanisms presented in Section 3, CVL provides a complete solution to variability modeling. Fig. 10 presents an overview of how CVL is applied to a base model (or a set of base models). The CVL description can be divided in four:

1. Choices capture a logical, user-centric description of variability. The *CompositeVariability* and *Iterator* classes of the CVL metamodel are the key concepts for capturing choices. CVL choices can be depicted by traditional feature diagrams.
2. Fragments and Substitutions are the core concepts of CVL to link variability to the base models. Fragments allow capturing any arbitrary base model pieces ranging from a link or an object to a group of object of any complexity. Substitutions allow substituting placement fragments with alternative replacement fragments or values.

3. Types are used to provide reusability and flexibility by decoupling the choices and substitution/fragments structures. Using types, the structure of the feature diagram can be mapped to fragments and substitutions.
4. Resolution models capture the decisions associated with choices. Each resolution model can be executed by the CVL transformation in order to automatically produce the product model corresponding to a set of decisions.



**Fig. 10. Making feature models executable**

To make CVL usable, concrete syntax elements need to be associated with the CVL concepts. Section 4.1 details how feature diagrams can be used as a concrete syntax for choice and invocations. Section 4.2 discusses the concrete syntax for defining fragments and substitutions and details the connection to the DSL tools. For types and resolution models, the current implementation of the CVL tools uses default syntax (a tree-view editor). These elements do not pose any major challenges for concrete syntax definition since they are pure CVL elements. In future work we will investigate the possibility of using a textual concrete syntax or an extension of the feature diagram notation to represent types.

#### 4.1 Feature Diagrams as CVL Concrete Syntax

CVL includes the Iterator concept as a general choice mechanism and the CompositeVariability concept in order to compose these choices. These two mechanisms allow capturing all common feature diagrams notations. Fig. 11 presents the mapping between feature diagram symbols and their representation in CVL. All these symbols are uniquely represented in CVL which make feature diagrams well-suited as a graphical concrete syntax for that part of the CVL metamodel.

The rationale for defining a few general concepts such as the Iterator in CVL, rather than reusing an existing feature diagram metamodel with a metaclass for every symbol, is to remain general and concise and be able to support various feature diagram notations.







Semantics	Symbol	CVL Element	Comment
Mandatory		<code>:CompVar</code>	<i>CompositeVariability</i> makes the resolution of a node mandatory for any instantiation
Optional		<code>:Iterator</code> Lower = 0 Upper = 1	The multiplicity [0..1] reflect the fact that the sub-node can be chosen or not.
AND		<code>:CompVar</code>	<i>CompositeVariability</i> makes the resolution of all sub-nodes mandatory for any instantiation
OR		<code>:Iterator</code> Lower = 1 Upper = -1 IsUnique = true	The multiplicity [0..*] means that any number of sub-nodes can be chosen. <i>IsUnique</i> specifies that each sub-node can only be chosen once (which corresponds to the usual OR semantics)
XOR		<code>:Iterator</code> Lower = 1 Upper = 1	The multiplicity [1..1] means that one and only one of the sub-nodes can be chosen (which corresponds to a classic XOR semantics).
Multiplicity		<code>:Iterator</code> Lower = x Upper = y IsUnique = true	The multiplicities associated to a choice are mapped to an Iterator containing the same multiplicities

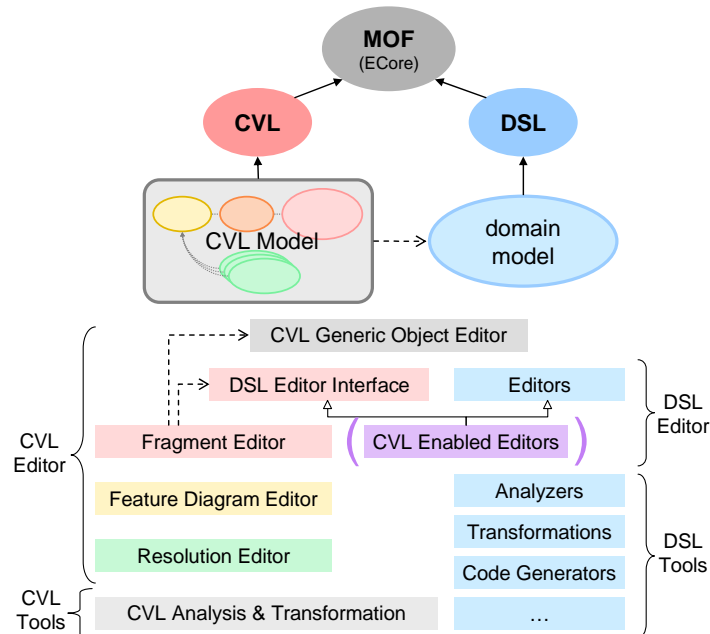
Fig. 11. Mapping between feature diagrams and CVL metamodel

#### 4.2 CVL Concrete Syntax

CVL fragments and substitutions are closely linked to base models elements that are expressed using a DSL. CVL is designed to be generic with respect to the DSLs used to express base models and to keep the variability information fully separated from the base models. These requirements are fundamental in order to keep a good separation of concerns. However, from a user point of view, the integration of the variability description with the base model should be as smooth as possible. Even though the CVL description is kept separate from the corresponding base model it is by no means independent from the given base model. The base model is oblivious to the CVL model while the CVL model refers to objects of the base model. Especially, the definition of fragments and substitutions on base models should be intuitive to the users who are familiar with the base models. From a concrete syntax point of view, it means that the syntax of CVL elements in direct relation with the base model needs to adapt to the concrete syntax of particular base models DSLs.

The process of going from a fragment defined in the concrete syntax of a DSL to the CVL representation involving boundary elements has to be transparent to the user. To achieve this, defining a fixed CVL concrete syntax for fragments and boundary elements is not an option: all the representation of base model elements should use the base model concrete syntax. The solution proposed by CVL is tight tool integration with the DSL tools. This way, the usual features of the DSL tools can be used to

define, connect and visualize fragments. In practice, this solution is viable if the development effort to connect exiting DSL editors with CVL tools is small.



**Fig. 12. Feature Diagrams as concrete syntax of CVL**

Fig. 12 presents the way CVL tools and DSL tools are integrated. The right side of the figure presents a particular DSL and the set of tools associated with it. No particular assumption is made on the DSL except that, for integration with the prototype implementation of CVL, the metamodel of the DSL should be defined using MOF [18] and more specifically the Eclipse implementation of it (ECore). Other than that, the concrete syntax of the DSL can be textual, graphical or any intermediate between these two. For editing fragments and substitutions, the CVL tool defines an interface which must be implemented by the DSL editor in order to allow defining, connecting and visualizing fragments.

```

public interface ICVLEnabledEditor {
    // Highlight in the editor the object identified by xmi_id
    public void highlightObject(String xmi_id, int color);
    // Remove highlighting for all object in the editor
    public void clearHighlighting();
    // Get the editor selection
    public ArrayList<Object> getSelectedObjects();
    // Set the selection of the editor
    public void selectObjects(ArrayList<String> objects);
}

```

**Fig. 13. Java interface for CVL editor integration**

The complete CVL interface for DSL editors is presented in Fig. 13. This interface allows getting and setting the base model objects selected in the DSL editor as well as highlighting base model objects. The CVL editor interoperates with any editor

implementing this interface. The selecting capabilities are used to make selections and links to the base model objects, and the highlighting capabilities allow displaying the variability with the DSL concrete syntax. The advantages of this interface are:

1. It requires only typical editor features. Almost any DSL editor will already include a selection mechanism and some mechanisms for highlighting objects.
2. It is suited for any kind of concrete syntax and easy to customize. Selection and highlighting can be implemented for textual and graphical editors.

In total the interface counts 4 operations. To implement the CVL enabled version of the DSL editor we extend the DSL editor by implementing the interface. This interface allows building fragments from selections in the base model editor. The fragments boundary elements are automatically created by traversing the base model and identifying pointers going from and to the selected objects. The interface then also allows visualizing fragments by highlighting corresponding objects in the base model editor. Finally, for the definition of bindings for substitutions, the interface is used to choose and visualize how placements and replacements should be connected.

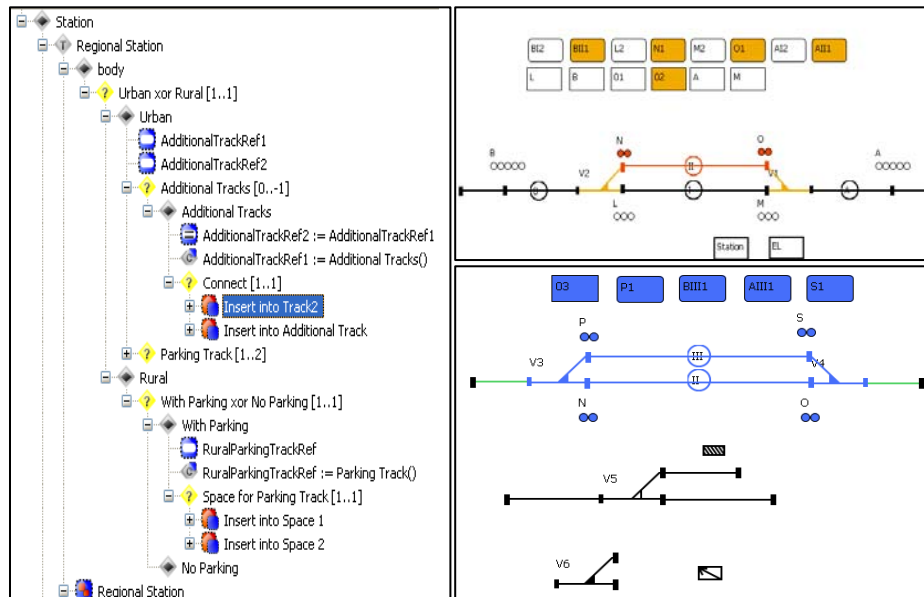
### 4.3 Validating the Feasibility of the CVL Approach

To validate the feasibility of the CVL approach including the tooling support, the CVL interface has been implemented for 4 different languages and editors: TCL [7, 24], UML, APRiL (a DSL for payroll reporting [26]) and the generic object editor provided with the Eclipse modeling framework. The editors for the TCL and APRiL have been built using GMF [10] and subsequently extended to implement the interface. In all the three cases the extension was simple and required around 150 lines of Java code. Because both TCL and APRiL editor are based on GMF/GEF, the integration code is shared and can be reused with any other GMF based editor. The result of this integration with the TCL editor is presented in Fig. 14. The CVL editor is in the left pane and interacts with the TCL editors in the right panes.

CVL does not require the union of all products to be available as model elements in one model, as e.g. FeatureMapper [14] or PureVariants' Enterprise Architect integration [21]. The product line in Fig. 14 has been specified using two different TCL models, the top right is the starting point and the bottom right provides building blocks that can be (re)used. This capability of using fragments from different models enables extensive reuse, and several CVL descriptions can utilize the same base models. Different fragments can also intersect with each other. One of the enablers for this is the fact that CVL is described in a separate model with links only from CVL to the DSL.

Selecting a CVL element in the left side CVL editor, will highlight the corresponding TCL elements in the right side editors. The placement and replacements have been automatically generated from selections in the TCL editor. In the CVL enabled TCL editor we have used coloring to display the CVL concrete syntax on TCL. Selecting the *Insert into Track2* CVL element will highlight the elements in the TCL editor. The placement (elements that will be deleted) are colored red and the connecting elements orange, the elements of the replacement will be blue

and the connecting elements will be colored green. The CVL concrete syntax is in this way a combination of CVL and TCL. A default concrete syntax is provided, but this can be changed to better fit the target DSL. The visualization can be used as assistance when defining the actual bindings between the boundary elements (when this is necessary).



**Fig. 14. Screenshot of the CVL editor together with the TCL graphical editor**

For the integration with UML, the interface was implemented on top of the Papyrus UML editor [19].

#### 4.4 The Example Revisited

Having introduced the mechanisms of CVL, we now return to our running example and recapitulate how we come from a feature diagram (shown in Fig. 1) to the automatic production of a station product (say: an urban station with an additional track and a parking track where the parking track will have extra security).

The feature diagram shown in Fig. 1 can be created without any knowledge of the underlying TCL. It represents an overview of the intuitive variants of stations that are available in our product line, but it does not contain enough information to generate the products. To produce a product we need a resolution model and a well-defined relation to base models (in TCL).

The CVL model defining the product line expressed in the feature diagram is illustrated in Fig. 15a. It contains a variability model (Station) and a resolution model specifying the resolution for a product.

The Station variability model in CVL is initially defined by the feature diagram of Fig. 1,<sup>2</sup> and Fig. 15b and c illustrates this. Fig. 15b shows that the variability model contains three types representing the reusable elements (Regional Station, Additional Tracks and Parking Track) and Fig. 15c shows the choices by iterators depicted as diamonds with question mark. We see the choice between Urban and Rural stations and in the Urban station the choice concerning additional tracks and parking tracks.

The CVL model shown in Fig. 15b and c contains more than what can be derived from the feature diagram. There are substitutions with corresponding placements and replacements plus definition and usage of replacement fragment references. These CVL elements define the precise relationship with the base model(s) defined in TCL.

The CVL elements defining placements, replacements and substitutions have been created with our CVL tool. TCL elements are selected in the CVL-enhanced TCL editor shown in Fig. 14, and CVL constructs created to refer the selection. Feedback is given through configurable highlighting mechanisms with the TCL editor.

To create the substitutions it is also needed to connect the boundary elements properly and our boundary element editor is helping to perform this without too much intricate knowledge of the DSL's metamodel specifics. Fig. 15d shows the definition of the Regional Station replacement fragment shown in Fig. 4 including also the necessary boundary elements. (Please refer to Fig. 3 for explanation of boundary elements.)

Furthermore we show how dynamic instantiations of Additional Tracks are defined. The Additional Tracks iterator (expanded in Fig. 15e) contains one alternative (Additional Tracks composite) that can be chosen several times (isUnique is false). This composite includes an assignment and an instantiation of the Additional Tracks type. The assignment stores the last inserted track to be able to build a fragment with arbitrarily many tracks. An iterator then gives the choice whether the returned replacement fragment should be inserted directly onto track 2 (Fig. 4) or onto another additional track (Fig. 5).

Details of the Parking Track are given in Fig. 15f and g, where the Parking Track type is instantiated and the returned replacement fragment substituted either onto the left (Space 1) or right (Space 2) side of the station.

This concludes our walkthrough of the variability model that defines a product line. By defining resolution models specific products of this product line can be produced. The resolution model illustrated in Fig. 15h specifies a three track station with a parking track on the left side of the station. The first iterator (Urban xor Rural) is resolved by choosing Urban. The Additional Tracks iterator is resolved by choosing the Additional Tracks composite only once. The Connect iterator is resolved by inserting the track onto Track 2. The Parking Track iterator is resolved by choosing the composite Space 1. At last the Extra Security composite is chosen in the iterator Automatic or Extra Security. Different products can be produced by making other selections in the resolution model.

Then we execute the CVL model comprising the variability model and the resolution model. The execution traverses the variability model and for every iterator it looks for the selected choices in the resolution model. The result is a TCL model of

---

<sup>2</sup> Our tool is not finished on this point and the figure is made with Visio and the corresponding CVL model produced manually.

the Urban station with three tracks and a parking track with extra security. This station can then be fed into TCL-specific tools such as code generators or validators.

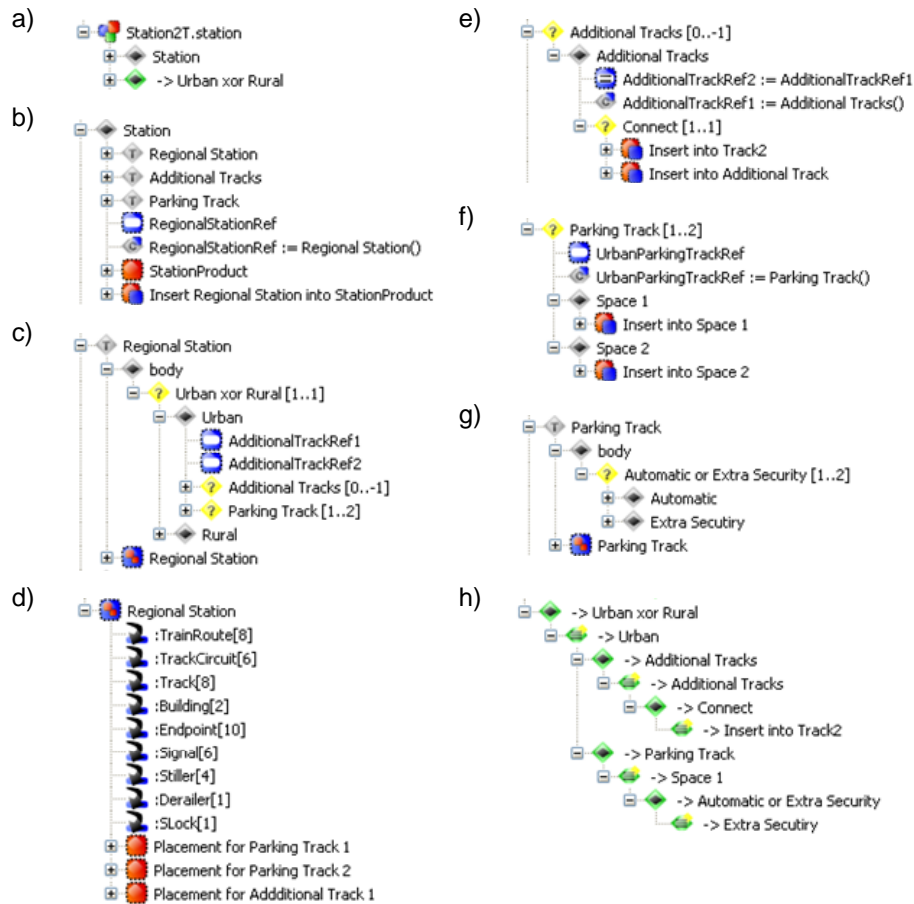


Fig. 15. Regional Station product line in CVL

## 5 Related Work

The main idea behind CVL as a language for specifying variability models has been related to other approaches in the SPLC'2008 paper on CVL, [13]. It is not a new idea to make variability models orthogonal to the product line models see e.g. [20], and [3] where the main reason for making a separate variability model is for tracing variabilities across different artifacts, but the new thing with our approach is that elements of the variability models have unidirectional links to the elements of the base product line model that are subject to variation. Other approaches have the links the other way, see e.g. [6] and [4], combined with annotations of the product line

model elements, while the CVL approach does not require any annotation of the base product line model. Other approaches to variability just make variability annotations of UML models, e.g. by means of profiles with stereotypes (e.g. [9], [27] and [11]), and thus have no explicit, separate variation models.

Feature diagrams as such have been surveyed in a number of papers. Feature diagrams is not the main subject of this paper, but rather the use of them for generating specific product models. The use of feature diagrams falls in two main categories: as stand-alone specifications of features as part of requirements specification, and as specifications that are linked to base product line models and are executable in the sense that selecting a configuration of features may produce a specific model.

The most primitive way of making feature diagrams executable is by translating them to models/programs in either some general purpose model transformation language or just to a general purpose programming language. Formalizing feature diagrams (as e.g. [23]) helps in this respect, but may also just have the purpose of making stand-alone feature diagrams more precise. Feature diagrams are supposed to specify features of a product at a domain specific level and at a level where the detailed transformations are not provided - with the implication that the above translation must be comprehensive. If the links to the base models are not part of the feature diagrams, then these relations must be provided as part of the translation.

The FeatureMapper [14] approach is similar to the CVL approach in that the tool maps features to elements of the models. It also applies to arbitrary languages defined by metamodels, it has support for recording of modeling steps associated with a feature, but it lacks the support for model fragments with boundary elements as supported by CVL.

A more specific approach is the template-based approach in [4]. This approach has many similarities with our CVL approach, in that feature diagrams are separate from the base product line model, and in that the approach applies to any model in any language defined by a metamodel in MOF. However, it links from the base product line model to the feature diagrams, and it still annotates the base product line model. The CVL approach has the links the other way around, with the implications that there may be more than one feature diagram for each base product line model. The same model may therefore be used as the base for many product lines, without having to consider that when making the base model. Furthermore, as CVL primitives may produce new models based upon fragments of the base model, the base model does not have to include the union of all features.

The reason that the template approach has annotations of the base model is that often more than one model element have to be removed or exchanged by a given configuration, and the feature diagram modeller has to specify this by means of annotating all the affected model elements. In the CVL approach, selecting a model fragment as subject for variation (consisting of several base model elements), the tool will find the boundary elements to the rest of the base model.

The type concept of CVL may be compared with the notion of feature diagram references [6]. While the semantics of a feature diagram references amounts to a macro-like expansion of the referenced diagram in the place of the reference, CVL supports instantiations of types.

Another main difference from other approaches is that CVL may in fact express the construction of models based upon base model fragments, while other approaches rely on the base model being a model that has the union of all features. Feature models thereby simply specify possible configurations of features, and the transformations are simply removing model elements and possible setting values of model element properties.

## 6 Conclusions and Further Work

We have shown how the CVL approach solves variability modeling in a generic and flexible way. CVL comprises feature diagrams as (partial) concrete syntax. The CVL tool has devised a simple, yet powerful way to let the variability modeling be performed in the look-and-feel of the base model language and tools. CVL has few but powerful abstraction mechanisms with a novel parameterization approach where replacements contain placements for further substitutions. We have validated the approach by a set of experiments on several different base language tools.

CVL is executable, and with feature diagrams as concrete syntax for parts of CVL, feature diagrams also become executable. The semantics of CVL is now given precisely by the execution of the CVL tool, but we will go on to provide a formal semantics on which analysis tools can be based. The idea is that analysis performed on the base product line model will not have to be done in its entirety for specific product models, as these have come out of the restricted transformations in CVL and not out of general transformations. In the future we will include more general constraints between features, and make more experiments with how to visualize the type and instantiation concepts in the generic CVL tool.

## 7 References

1. Batory, D., Chen, G., Robertson, E., and Wang, T.: Design Wizards and Visual Programming Environments for Genvoca Generators. *IEEE Transactions on Software Engineering*. 26, 441-452 (2000)
2. Benavides, D., Trinidad, P., and Corté, A.R.: Automated Reasoning on Feature Models. In: *Lecture Notes in Computer Science*, vol. 3520, pp. 491-503. Springer (2005)
3. Berg, K., Bishop, J., and Muthig, D., "Tracing Software Product Line Variability – from Problem to Solution Space," SAICSIT 2005, (2005)
4. Czarnecki, K. and Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *Generative Programming and Component Engineering*. *Lecture Notes in Computer Science*. Springer (2005)
5. Czarnecki, K., Helsen, S., and Eisenecker, U.: Formalizing Cardinality-Based Feature Models and Their Specifications. *Software Process Improvement and Practice*. 10(1), 7-29 (2005)
6. Czarnecki, K., Helsen, S., and Eisenecker, U.: Staged Configuration Using Feature Models. *Software Process Improvement and Practice*. 10(2), 143-169 (2005)
7. Endresen, J., Carlson, E., Moen, T., Alme, K.-J., Haugen, Ø., Olsen, G.K., and Svendsen, A., "Train Control Language - Teaching Computers Interlocking," *Computers in Railways XI (COMPRAIL 2008)*, Toledo, Spain, (2008)

8. Eriksson, M., Börstler, J., and Borg, K., "The Pluss Approach : Domain Modeling with Features, Use Cases and Use Case Realizations," 9th international conference of Software Product Line, Rennes, France, (2005)
9. Fontoura, M., Pree, W., and Rumpe, B.: The Uml Profile for Framework Architectures. Addison-Wesley, (2001)
10. GMF, "Eclipse Graphical Modeling Framework (Gmf)." <http://www.eclipse.org/modeling/gmf/>
11. Gomaa, H.: Designing Software Product Lines with Uml: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, (2004)
12. Griss, M.L., Favaro, J., and d' Alessandro, M., "Integrating Feature Modeling with the Rseb," the 5th International Conference on Software Reuse, Vancouver, BC, Canada, (1998)
13. Haugen, O., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., and Svendsen, A., "Adding Standardized Variability to Domain Specific Languages," SPLC 2008, Limerick, Ireland, (2008)
14. Heidenreich, F., Kopicsek, J., and Wende, C., "Featuremapper: Mapping Features to Models," ICSE 2008, Leipzig, Germany, (2008)
15. Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A., "Feature-Oriented Domain Analysis (Foda) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA Tech. Report CMU/SEI-90-TR-21, Nov., (1990),
16. Kang, K., Kim, S., Lee, J., and Kim, K.: Form: A Feature-Oriented Reuse Method. *Annals of Software Engineering*. 5, 25 (1998)
17. MOFScript. <http://www.eclipse.org/gmt/mofscript/>
18. OMG: Meta Object Facility (Mof) Core Specification. Version 2.0 (Available Specification) Omg Document: Formal/06-01-01. (2006)
19. Papyrus: Papyrus Open Source Tool for Graphical Uml2 Modelling: [Http://Www.Papyrusuml.Org](http://www.papyrusuml.org). (2007)
20. Pohl, K., Bökle, G., and Linden, F.v.d.: *Software Product Line Engineering—Foundations, Principles and Techniques*. Springer, (2005)
21. PureSystems, "Pure::Variants Home-Page - [Http://Www.Pure-Systems.Com/Pure\\_Variants.49+M54a708de802.0.Html](http://www.pure-systems.com/pure_variants.49+m54a708de802.0.html)," (2009)
22. Riebisch, M.: Towards a More Precise Definition of Feature Models - Position Paper. In: *Modelling Variability for Object-Oriented Product Lines*, pp. 64-76. BookOnDemand Publ. Co. (2003)
23. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y.: Generic Semantics of Feature Diagrams. *Computer Networks*. 51, 456-479 (2007)
24. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.-J., and Haugen, O., "The Future of Train Signaling," *Model Driven Engineering Languages and Systems (MoDELS 2008)*, Toulouse, France, (2008)
25. van Gurp, J., Bosch, J., and Svahnberg, M., "On the Notion of Variability in Software Product Lines," *The Working IEEE/IFIP Conference on Software Architecture (WICSA)*, (2001)
26. Zhang, X., Lin, Y., and Haugen, Ø., "April: A Dsl for Payroll Reporting," in *The 1st International Workshop on Future trends of Model-Driven Development in conjunction with the 11th International Conference on Enterprise Information Systems*. Milan, Italy, (2009)
27. Ziadi, T., Hérouët, L., and Jézéquel, J.M., "Towards a Uml Profile for Software Product Lines," *PFE*, (2003)