Extending a Model-Driven Engineering Environment to Support Product Line Engineering

Dolev Dotan, Tomer Amarilio, Gilad Saadoun, Tali Yatzkar-Haham

IBM Research – Haifa, Israel
{dotan, tomeram, giladsa, tali}@il.ibm.com

Abstract. In order to make model-driven product line engineering a reality, variability and feature modeling techniques are necessary, but not sufficient. Adequate tool support is of equal importance, and presents its own set of challenges. In this paper, we present an approach to the activities of modeling product lines, defining and verifying products, and exporting product models. Our approach pioneers innovations such as the *active product view* and an efficient *decision propagation logic*, and supports advanced concepts such as *incomplete product definitions*. We have implemented our approach on top of a UML tool.

1 Introduction

In recent years, the concept of software product lines has moved from principle to practice. One particular area where this new method is needed most is the embedded software space. In this space, some UML-based tools provide world-class Model-Driven Engineering environments. While several efforts exist to add feature and variability modeling to UML (e.g. see refs [1]-[4]), they fall short on providing or even suggesting adequate tool support. Thus, we had set out to explore the modeling concepts and tool support that would be needed for making this approach usable. This paper describes the results of this work – a comprehensive extension to a UML tool to support Model-Driven Product Line Engineering.

Using our extension, users may (1) define a UML model containing ***variation points***, (2) define ***local features*** on which variation points may be conditioned, (3) specify ***product definitions*** which resolve variation points, (4) quickly view the implications of these resolution by going to ***product mode***, and ultimately (5) employ a model transformation to ***export*** product models (or sub-product-line models, if the chosen product definition is incomplete).

To support these activities, we have extended UML with several modeling constructs, which are described In Section 2 along with their semantics and user interface. A novel *decision propagation logic* was designed and implemented, and is explained in Section 3. In Section 4 we introduce several mechanisms that verify that, taken together, a model and a given product definition define a valid product model. Section 5 describes how we tackle one of the main goals of product lines engineering – exporting product models. Section 6 describes active product mode, which provides immediate resolution feedback while editing the model. Finally, Section 7 concludes.

2 Product Line Modeling

Modeling a product line is a complex activity. Thus, we strived to provide an easy and seamless modeling experience, which will feel natural for users of the tool. For that end, we have used a combination of "lightweight" UML extensions[5] (using stereotypes in a profile) and "heavyweight" UML-metamodel-extensions, as well as comprehensive GUI support. This section describes the new modeling constructs, their semantics, as well as the user-interface mechanisms we added.

2.1 Variation Points

To specify variability in a project the user defines *variation points*. A variation point is a point of choice on some UML elements implying products may differ with respect to those elements. A variation point may also have a *condition* (see Section 2.3), in which case the decision would be derived by its evaluation (this is called a ***derived*** *choice*). Otherwise, the variation point will be decided upon explicitly by the user (*user choice*). Currently, we support three different kinds of variation points (variability patterns), which are described below.

2.1.1 Optional Element

An optional model element (marked with the «optional» stereotype, as in [1]) is an element which may exist in some products but not in others. Any kind of model element may be declared optional, and may be given a condition using the Variability tab of the element's Features Dialog (see Figure 1).

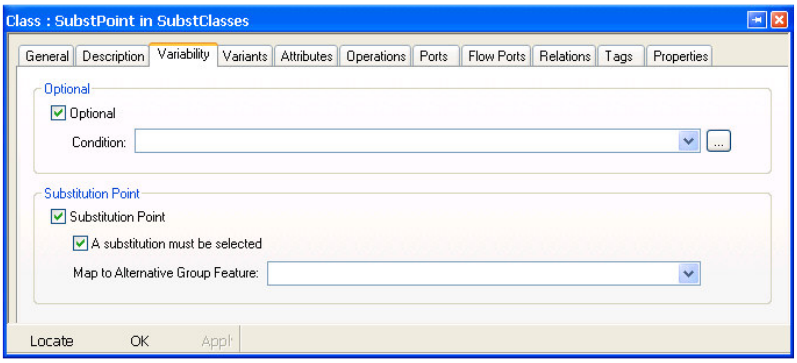

Figure 1. Variability tab

The un-inclusion of an optional element may propagate to other elements, not necessarily marked optional. For example, if a class is not included in a product, all associations in which it participates (both outgoing and incoming) will also not be included in the product. The decision derivation algorithm is described in Section 3.

2.1.2 Optional Code Fragment

A UML model may contain snippets of code in the target programming language (e.g. using OpaqueBehavior). These define the behavior of the owning UML elements. For example, operation body, transition effect and state entry elements may all have associated code, without which they would have no effect.

We have added language-neutral keywords that allow delimiting *optional code fragment* – a part of the code snippet which may exist in some products but not in others.

```
#optional [condition]
optional code fragment
#endoptional
```

2.1.3 Substitution and Substitution Point

A Substitution (a dependency with the stereotype «Substitutes») indicates that its source element may replace its target element.The elements supported for substitution are classes and interfaces, as well as some UML extensions such as "implicit object" (a part typed by an anonymous class). Both source and target must be of the same meta-type, and must be substitutable as defined below.

Note that optionality takes precedence over substitution, i.e. if a substitution source or target is an unselected optional then the substitution will not take place.

2.1.3.1 The Semantics of Substitution

Substituting C for P (i.e. replacing P by C) means putting C in the *context* of P and then deleting P. The context of an element is all relations (associations, links, dependencies, flows, etc.) in which it is involved, regardless of direction (See Figure 2 and Figure 3).

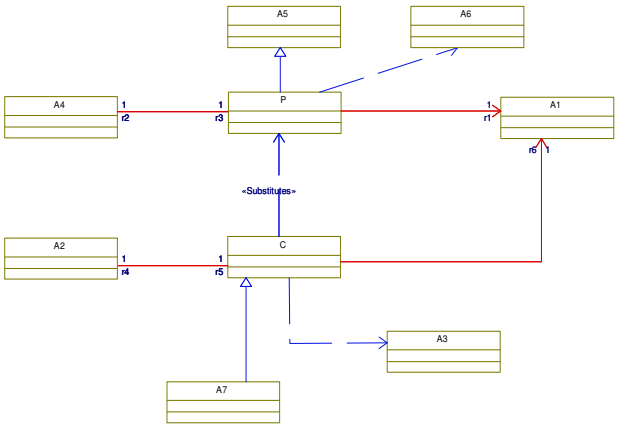

Figure 2. Class substitution (before)

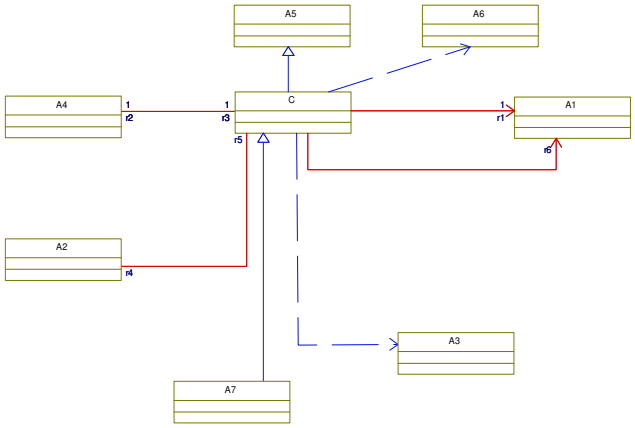

Figure 3. Class substitution (after)

Substitution is transitive, i.e. if the substitution target is also the source of another «Substitutes» dependency, then its chosen variant will also replace the element in the target of the dependency (if both dependencies are selected in the given product).

2.1.3.2 Merge - combining substitution with static inheritance

Our target UML tool extends the notion of inheritance with "static inheritance" (using «static» stereotype on a generalization) which allows copying the content of one element (class, interface or implicit object) to another. If this construct is combined with substitution, the substitution source is effectively *merged* with the target (see the figures below). This is because the content of the general is copied to the specific and then substitution is performed. Note that substitution changes the *context* of the substituting element whereas static inheritance changes its *content*.

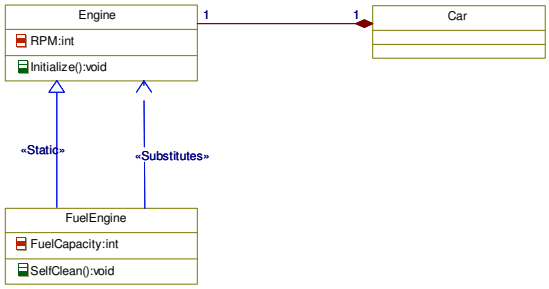

Figure 4. Substitution with static inheritance (before)

Figure 5. Substitution with static inheritance (after)

2.1.3.3 Substitution Points

The target of a substitution is called a *substitution point* (similar to a "variation point" in [1]). Users may apply the «Substitution Point» stereotype (directly or using the appropriate checkbox in the Variability tab – see Figure 1) as a visual reminder that the element may be substituted. When the stereotype is applied, the following additional specification may be added using the Variability tab. A UML tool may check these as part of product validation, hence providing greater safety for the design.

1 "Substitution must be selected": when this checkbox is checked, the substitution point is considered resolved only if a variant is selected for substitution. This is implemented using the isSubstitutionRequired stereotype attribute.

2 "Map to Alternative Group Feature": This will require that all the substitution point's incoming substitutions be conditioned on the (alternative) sub-features of the pointed group feature (see below). This is implemented using the associatedAlternativeGroup stereotype attribute.

We have extended the dialog for substitution points with the Variants tab, which allows easy handling of all the substitutions of that substitution point from a single place.

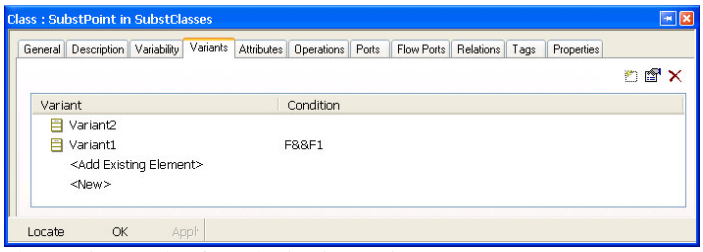

Figure 6. The Variants tab of a substitution point

2.1.3.4 Variants

A variant of a substitution point may optionally be adorned by the stereotype «Variant». Any element so marked is designed with the sole purpose of being a variant. Thus, not selecting it for substitution will entail not selecting it as an element as well. That is, if not selected for substitution it will not exist in the product model or generated code. If not marked as «Variant», the element will be kept in the product model even if not selected for substitution.

2.2 Local Features

We have extended UML with support for modeling *Product Line Features,* which are elements used to model the client-level view of the system's functionality. For example, a car may have features like gasoline engine, ABS system, GPS, audio system, etc. Features are organized as a hierarchy, that is, features may have sub-features. For example, a feature **Transmission** may have sub-features **Automatic** and **Manual**.

Like regular UML model elements, Product Line Features may be optional, which means they may be selected for some products but not others. Taking from FODA[6] notations, we represent optional features in the model browser with an icon that has an empty circle, whereas mandatory features have filled circles (see Figure 7).

A feature may be defined to be an *Alternative Group Feature*. This means that if selected, at most one of its sub-features must be selected. **Transmission** above is an alternative group feature. In addition to the above basic selection constraint, the user may add another constraint, that "an alternative must be selected". These constraints are checked as part of model validation. In keeping with the FODA notation, we use an arc in the icon to denote that a certain feature is an Alternative Group Feature.

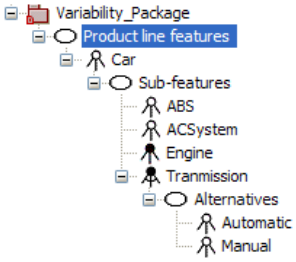

Figure 7. Features in the browser.

Features as variability parameters/"system constants". The local features may be viewed as an abstraction of the variability in a UML model, or they may be viewed as parameters constituting the "variability interface" of a model. The latter view is similar to *system constants* in AUTOSAR. Local features may indeed be used to implement AUTOSAR's system constants. They are more general than that, however, since they support a *hierarchy* with particular semantics, such as alternatives.

2.3 Conditions

For each optional element and for each substitution, the user may define a condition – a Boolean expression over local features. The decision whether or not to select the conditioned element in a given product is not explicit, but instead derived by evaluating the condition. In condition evaluation, each operand (a reference to a feature) is evaluated according to the inclusion value of the referenced feature in the given product definition.

We have implemented conditions using a UML metamodel extension. Each condition is parsed immediately and stored as an expression tree, with the operands holding references to the features (this tree is invisible to the user). Besides the obvious performance gains (no need to parse the condition on each evaluation), this allows us to provide intelligent editing behavior, in a fashion which the users of our target tool have come to expect. For example, conditions are always synchronized with changes in feature name or location; a feature may not be deleted if it is used by any condition; and so on.

2.4 Product Definitions

To enable the derivation of products from a product line model the user defines *product definitions*. These are implemented using (yet another) metamodel extension we added to UML, and may be shown and handled in the browser as any other element. The content of a product definition is a set of *decisions* regarding the variation points and features in the product line.

The decisions are entered using the features dialog for Product Definitions (see Figure 8). In this window, a browser is shown. Note that since substitutions are always alternatives to one another, they are shown under the substitution point, much like alternative group features. This is the *choice tree* for the project.

Elements for which a *user choice* should be made (variation points with no condition and optional features) have a checkbox next to them, which may take three states: positive decision (e.g. optional is included), negative decision (e.g. optional is not included), and undecided (the default).

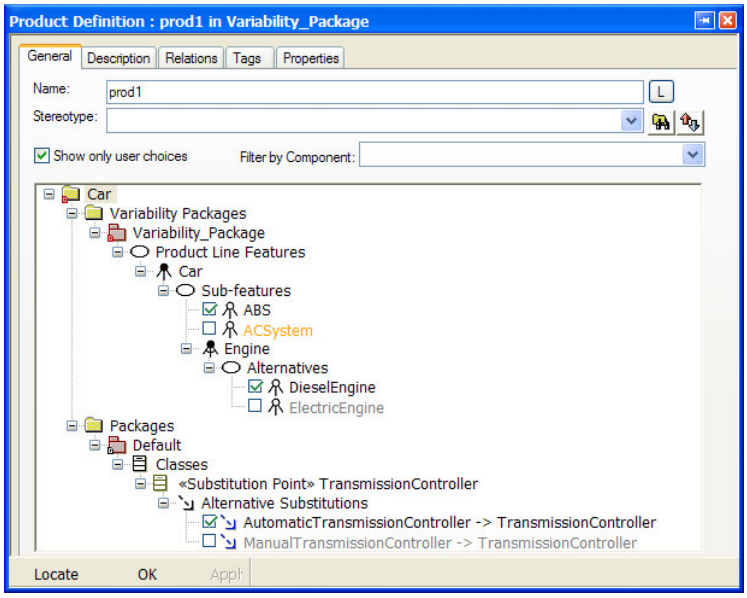

Figure 8. Feature dialog for a Product Definition, showing the choice tree

The decisions and their effects are immediately shown in the choice tree using the following color scheme:

- Unselected element – grayed out
- Selected elements – normal appearance
- Undecided element – orange

Note that since some decisions ***derive*** other decisions (see Section 3), the color scheme above is not simply a 1-1 reflection of the decision check boxes.

In addition to using the features dialog, decisions may be set directly from within the model when in active product mode (see Section 6).

3 Decision Propagation Logic

Each model element, even those that are not variation points, may be affected by the variability in a product-line model. For example, any class in an optional package will not exist in the product model if the package is not included in the exported product. Similarly, a transition is not included if its source or target states are not-included (even though the transition is not owned by the states). For each model element, a 3-state decision (included, not-included, undecided) may be derived using the decision logic described in this section.

3.1 Approach

Due to performance reasons, it would be un-realistic to propagate all decisions to the entire model every time a decision, or the model, changes. Thus, all our decision calculations are done "on-demand". This has repercussions on the decision logic, as shown below.

Our decision propagation logic follows one distinctive guideline: while negative (not included) decisions propagate down the hierarchy and across different references, the "unspecified" and "included" decisions do not propagate. Thus, when looking at the contents of an undecided package, they will not be undecided themselves. Instead, each will have its own decision.

Putting this in terms of (3-state) logic, instead of calculating the decision using:

product-decision(element):= product-decision(owner) $\wedge$ user-decision(element) (**1**)

We state that:

product-decision(element):= if (decision(owner) = Not-Included): Not-Included (**2**)
else: user-decision(element)

Definition: Boolean operator hierarchical-AND: For brevity, we shall call the above if-else structure "hierarchical- AND" and mark it with $. Thus, the above can be written:

decision(element):= decision(owner) $ user-decision(element) (3)

Notice that this operator is *not* symmetric, is left-associative, and is transitive.

The benefits of using the hierarchical-AND for our purposes are two-fold: First, it allows us to pinpoint (in the product definition dialog and in active product view – see section 6.1) the undecided variation points and features using the color coding. Second, it reduces the calculation performance for "on-demand" decision derivation dramatically: Since the derivation is not symmetrical, there is no need to go down the hierarchy and see if anything should be deduced from the decisions of descendant nodes – which is costly since it may span the whole model tree.

3.2 The Decision Logic

The above formula (3) is actually a simplified version or the actual calculation:

```
product-decision(element) := decision(owner) $                    (4)
                             type-dependent-decision(element) $
                             local-decision(element)
```

where local decision is either the condition evaluation or the user decision:

```
local-decision(element) :=                                        (5)
        if is-variation-point(element):
            Let x = if has-condition(element): evaluate(condition)
                    else: product-user-decision(element)
            if is-variant(element):
                    return x $ variant-decision(element)
            else: return x
        else: return Included
```

and the type-dependent decision differs according to the UML type:

type-dependent-decision(element):= (6)

UML element type	**Calculation**
Dependency, generalization, hyperlink	product-decision(target)
Object	product-decision(type)
Association end	product-decision(target) $ hierarchy-decision(association-class)
Association class / element	hierarchy-decision(source) $ hierarchy-decision(target)
Link, flow, transition, edge, message	product-decision(source) $ product-decision(target)
pseudostates	pseudostate-decision(element)
Substitutes dependency	product-decision(target) $ sibling-substitution-decision(element)
Alternative Feature	alternative-feature-decision(element)
Others	Included

Hierarchy decision is an auxiliary formula defined as:

```
hierarchy-decision(element) := decision(owner) $ local-decision(element)   (7)
```

Our goal was to match the deletion semantics of the tool (that basically says that context elements are deleted) as much as possible, to provide a familiar user interface. However, this was not always possible. Thus, in the decision derivation for transition we propagate the Not-Included decision from the transition's source and target.

Due to space constraints, we will not show here the full derivation logic. It is worth mentioning that variant-decision supports the variant logic described in Section 2.1.3.4. Pseudostate-decision provides support for composite-transitions. Finally, the sibling-substitution-decision allows the automatic deduction that a substitution is not selected if it has a selected sibling (and similarly alternative-feature-decision for decisions about alternatives).

4 Product Validation

The process of modeling a product-line and defining products is very complicated and thus extremely error-prone. A wrong choice or a missing variation point can easily result in an invalid model. Thus, it is important to verify that the decisions made in the product definition result in a well-defined product which may be exported and for which code can be generated.

For that end, we have developed a set of verification rules that check the legality of the product. Implemented using a "checks" mechanism, they are automatically checked before product model export, before code generation, and upon user request (from the browser context popup menu of the product definition). The results are seen clearly in the result pane, and users can easily navigate both to the problematic model element and to its relevant entry in the product definition, where they can change the decision.

Some of the notable product checks are:

1) **No references exist from any included element to any not-included element.** For example, if a class C is not-included in the product, but some attribute a:C is included, the attribute will not have a legal type after export.
2) **A single substitution is selected for the substitution point.**
3) **A single alternative is selected for the alternative group feature.**
4) **The chosen variant is *substitutable* for the substitution point.** We check there are no conflicts which will prevent moving the context elements of the substitution point to the variant, e.g. an existing association-end owned by the variant, which has the same name as the one owned by the substitution point, but a different target (otherwise the UML tool might merge them). We rely on element reparenting policies – i.e. whenever a user can safely manually move an element from the substitution to the variant, there is no conflict.
5) **The chosen variant is *mergeable* with the substitution point**. Much like substitutability, except we now check there are no conflicts which will prevent moving the *content*, rather than the *context*.
6) **No substitution-generalization circles exist.** This is to avoid generalization loops in the exported product.

5 Export a Product Model

An important end-result of the product-line model is to be able to export models that represent a particular product, preferably without any trace of variability (if the product is fully-defined, i.e. leaves no "undecided" variation points). We provide an export capability which operates as described below:

- Each **not included** element / optional code fragment is removed from the resultant model.
- Each **undecided** element / optional code fragment survives as a variation point in the exported product. When possible, its condition is *pruned*: all positively-

decided features are removed from it and operators are removed to simplify it as needed.

- **Included** elements:
  - Optional element: kept without the «optional» stereotype.
  - Optional code fragment: kept without the #optional syntax
  - Substitution: a substitution (as defined in Section 2.1.3) is performed. In addition, any textual reference to the substitution point in code snippets is replaced by a reference to the variant.
  - Product Line Feature: kept only if it has some descendant which is undecided (we remove included features otherwise, since we want to remove any hint to there being a variation regarding that feature). If kept, such feature becomes mandatory.

6 Product Activation

To see the effect of resolving a certain product without triggering the full export mechanism, the user may *activate* that product definition. This puts the project in *active product mode*. In this mode, the effect of the decisions made in the product definition are visualized via a color scheme and dashed lines, and code or reports may be generated for the product without first exporting it. Users can continue to work on the product line model in this mode, and immediately see the implications of their changes on that particular product.

6.1 Visualization of Decisions

In active product mode, the decisions are visualized as follows:

- Not-included elements appear dashed and gray in diagrams and gray in the browser.
- Undecided optional elements appear dashed and orange in diagrams and orange in the browser. Orange is an indication that a decision needs to be made to resolve variability.
- Optional included elements appear normal (since after the decision there is no difference anymore between them and their non-optional siblings).

6.2 Code Generation

When trying to generate code that is similar to that which will be generated for the exported product model, a model-transformation approach (which effectively performs the export before code-generation) looks like the obvious choice. However, due to performance and other reasons, we were currently not able to use such an approach. Instead, we were able to work around this limitation using the following modifications to code generation capabilities:

- For not-included elements, no code is generated.

- Optional code fragments go through the same processing as in export.
- For substitutions, the following occurs:
  - The substitution point's *context* elements are generated under the variant.
  - If a static inheritance exists, the substitution point's *content* is generated under the variant.
  - Other references to the substitution become references to the variant, by generating the following code in the substitution point's h file:

```
typedef Variant SubstitutionPoint
```

  - For C code generation, in which each class method generated is prefixed by the class name, the following is also generated:

```
#define SubstitutionPoint_method Variant_method
```

  - Finally, for substitution of implicit objects, the code generation function generates attributes / variables only for the variant and not for the substitution point.

7 Conclusions and Future Work

We presented here modeling and tool-support concepts for enabling product-line engineering in a UML-based Model-Driven Engineering tool. While this is, in our opinion, a major step forward, we hope to explore further variation point types, product line verification, support for binding times other than model export, and integration with other lifecycle tools.

References

1. T. Ziadi, L. Hélouët and J.-M. Jézéquel. “Towards a UML Profile for Software Product Lines”. Proc, of the 5th Int. Workshop on Product Family Engineering (PFE), F. van der Linden (editor), LNCS 3014, 129-139, Siena, (Italy), November 2003
2. M. Clauss. “Generic Modelling using UML extensions for variability”. Workshop on Domain-specific Visual Languages (DSVL), 16 Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 11-18, Tampa Bay (Florida, USA), June 2001.
3. H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.
4. Hassan Gomaa: Object Oriented Analysis and Modeling for Families of Systems with UML. Software Reuse: Advances in Software Reusability, 6th International Conference, ICSR-6, 2000: 89-99
5. Selic, B. A Systematic Approach to Domain-Specific Language Design Using UML. ISORC 2007 2-9.
6. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)