On the Implementation of a Tool for Feature Modeling with a Base Model Twist

Peyman Shakari
Birger Møller-Pedersen
University of Oslo
Gaustadalleen 23
NO-0316 Oslo, Norway

Abstract

The paper provides a feasibility study of a new approach, BVR: Base-Variation-Resolution, to the modeling of feature variations in system families/software product lines. While existing approaches either capture variations by means of annotations (like 'optional', 'alternative', etc) in a model that is a *union* of all systems within the family, or by *completely separate* feature models, the BVR approach has *separate* feature models, but *relative* to a base model. This approach relies on the establishment and maintenance of relations between feature models and elements of a base model. The feasibility of this is established by implementing a prototype feature modeling tool as an Eclipse plugin with base models in terms of Java programs.

1. INTRODUCTION

Software System Families (also called Software Product Lines) have gained growing interest over the last years. While frameworks capture the commonalities between all systems made from these frameworks, the additional idea with system families is to capture the variations in so-called feature models. A feature may be mandatory (common to all systems in the family), a feature may be optional or it may be one out of several alternatives. Deciding on the features of a given systems is called resolution.

This paper is a feasibility study of an approach called the BVR approach: Base, Variation and Resolution, by implementing a prototype implementation of a feature modeling tool according to this approach.

Traditionally variations in system family models have either been expressed by independent, separate feature models (excellently covered in [3]), or they have been expressed by annotations to a system family model which then have to contain the union of all specific models (see e.g. [5]). [6] is a comparison of these different approaches, in addition to the use of domain specific languages for system family modeling.

The main benefit of independent, separate feature models is that the feature models may be used for generation of models in whatever language is desired, however, they are bad on exploiting existing (base) models Annotations have the drawback that the large (union) system family model becomes cluttered; in addition the approach is bad on unforeseen features, as potential features have to be part of the system family model.

The BVR approach is characterized by Variation models (similar to feature models) being defined relatively to a Base model, and Resolution models which in turn apply to a given Variation model, see Figure 1. Typically the Base model is not an adequate model, but rather a collection of model fragments. These fragments can of course be whole frameworks or components, but they do not have to be restricted to that. A Variation model captures the variabilities that may be applied to the Base model and its elements.

This paper was presented at the NIK 2006 conference. For more information see http://www.nik.no/

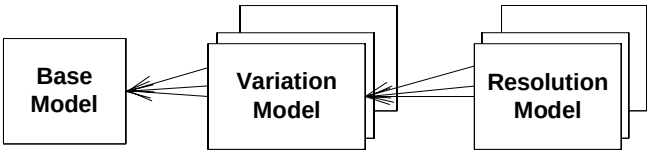

Figure 1 The BVR approach

As indicated in Figure 1 one of the obvious benefits of the BVR approach is that it is possible to have several Variation models for each Base model, while this is difficult with annotated models. The fact that one may have several Resolution models for one Variation model is not specific for BVR.

The BVR approach relies on the establishment and maintenance of relationships between Base models, Variation models and Resolution models. As indicated in Figure 6 these relationships are not only between models, but between individual elements in the various models.

A prototype implementation of a tool for BVR has been developed in order to show that it is possible to establish and maintain these relationships, and to exploit them. The BVR approach is independent of the languages in which the different models are made. For the purpose of this prototype Java was chosen as the language of the Base model (and thereby of the specific models), while the notation for the Variation/Resolution models is a standard feature diagram notation. The prototype tool is implemented as an Eclipse plugin.

We will use a well-known example of a digital watch in order to highlight the difference between the BVR approach and existing approaches. The example will also be used to illustrate the relationships between the different models.

Chapter 2 gives an overview of existing feature modeling approaches, at the same introducing the notion of feature modeling and feature diagrams. Chapter 3 describes the BVR approach in details. Chapter 4.3 describes the prototype tool, how it supports the idea of BVR and how it is implemented, especially how the relationships between Base and Variation models are implemented.

2. FEATURE MODELING APPROACHES

2.1 Feature Modeling Notations

FODA [7] is a domain analysis method that was developed at the Software Engineering Institute (SEI). The idea of domain in the context of this method is related to family systems. The primary goal of FODA is to develop products that are generic and widely applicable within a “feature oriented” domain.

The terms “feature” and “feature modeling” were originally introduced by FODA. FODA defines features as prominent and distinctive characteristics that are externally visible to various stakeholders. In the context of domain engineering, features represent reusable, configurable requirements, and each feature has to make a difference to someone, such as a stakeholder or a client program.

FODA organizes features hierarchically in feature models. A FODA feature model consists of four key elements:

- Feature Diagram: a hierarchical decomposition of features, including feature types, see Figure 2.
- Feature definition: a description of all features.
- Composition rules: rules (constraints) indicating interdependencies between features, and also which features are valid and which are not.

- Rationale of features: indicates the reason for choosing or not choosing a given feature.

A feature diagram in FODA has the form of a tree in which the root represents the product being described and the remaining nodes (leaves) denote features.

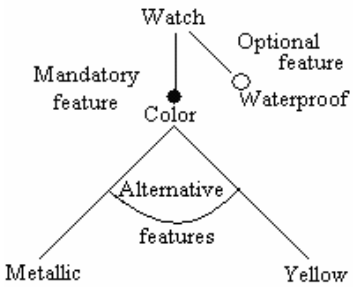

Figure 2 FODA-style feature diagram

FODA has three kinds of features:

- Mandatory features, which means that at least one and at most one instance of the feature, must be related to the product. In Figure 2 all watches have a Color, because Color is mandatory.
- Optional features, which means that a product may or may not have a certain feature. In Figure 2 Watch may or may not have a Waterproof feature.
- Alternative features, which means that a product can possess one feature at a time, for example either Yellow or Metallic in Figure 2.

The Czarnecki-Eisenecker (CE) notation is an extension of the FODA notation, especially with new kinds of alternative features. Below are the new extensions.

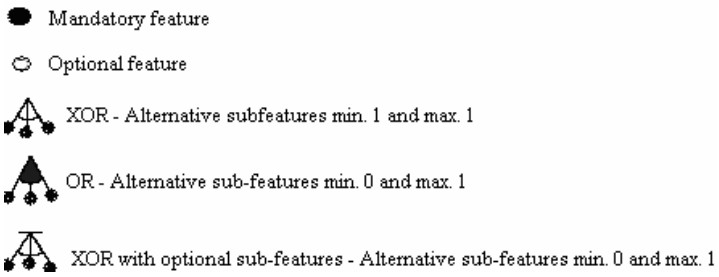

Figure 3 Czarnecki-Eisenecker Notation

Alternative features come in three flavors, XOR, OR and XOR with optional sub-features. The first corresponds to FODA's alternative feature type, while the two others are additional alternative feature kinds. Further, CE presents children of a feature as sub-features of that feature. In the original FODA all features were features.

Czarnecki and Eisenecker have extended their original notations to conform to UML cardinalities and yet being compatible with the FODA notation. The ECE notation, (also called cardinality-based notation) is based upon object orientation and bring new possibilities.

- Feature cardinalities: A feature can be annotated with a cardinality that will express the type of the feature.
- Group cardinalities: Alternative features in the FODA notation can be viewed as a grouping mechanism. In ECE the alternative feature type from FODA is represented as a <1-1> exclusive-or group cardinality. In addition ECE proposes two new group cardinalities: <0-n> inclusive-or-optional group cardinality and exclusive-or-optional group cardinality (<0-1>).
- Attributes: ECE proposes attributes as a way to represent choice of a value from a large or infinite domain such as integers or strings. The idea is to associate a feature

with a type. A collection of attributes can be modeled as a number of sub-features, where each is associated with a desired type.

- Relationships: Relationships between features are modeled using notations such as entity-relationship or class diagrams.

2.2 Annotated models

An alternative to separate feature modeling notations is to annotate the system family model with variation specifications. The result is a model that contains all possible variants. Figure 4 (from [2]) is an example in UML2.0.

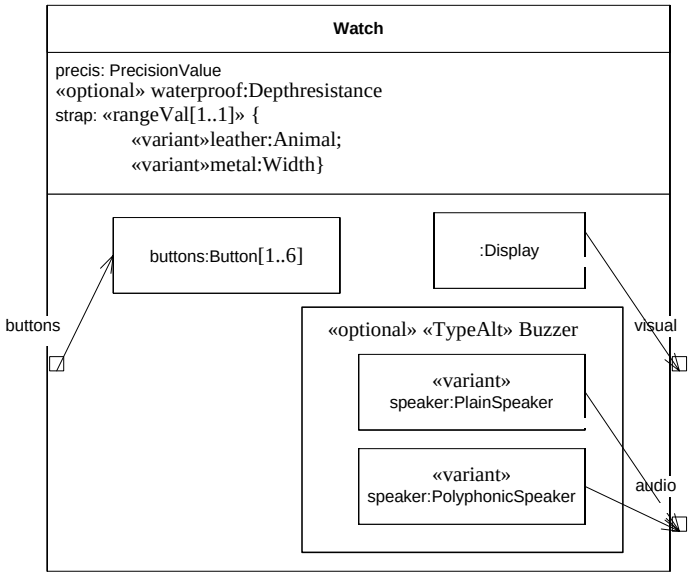

Figure 4 Example of annotated system family model (UML)

The notion of composite structure of classes is used to model the design of a watch. The UML as such supports the notion of classes containing specification of sets of objects (‘buttons:Button[1..6]’, ‘:Display’, ‘speaker:PlainSpeaker’ and ‘speaker:PolyphonicSpeaker’, while «optional» and «variant» are annotations and as such not part of the UML. A feature modeling tool will, based upon a resolution saying e.g. that the plain speaker variant is chosen, generate a valid UML model, where the ‘speaker:PolyphonicSpeaker’ part is removed together with the annotations.

2.3 Feature Modeling Tools

Since FODA first introduced feature modeling in 1990, a number of feature modeling tools have been launched. Almost all of these tools are based on the FODA definition of feature modeling. The FODA notation has been extended to object oriented notations; however, few tools are based upon these extensions. This section will describe two typical tools, FeaturePlugin and V-Manage.

FeaturePlugin [1] is an Eclipse plug-in for Feature modeling. The basic structure of FeaturePlugin is generated using the generator for tree-oriented model editors provided by the Eclipse Modeling Framework (EMF). EMF is a Java framework and code generation facility for building tools and other applications based on structured models. Integrating feature modeling as a part of a development environment such as Eclipse helps to optimally support modeling variability in different artifacts.

FeaturePlugin is based on the cardinality-based notations (ECE). It supports cardinality-based feature modeling, specialization of feature diagrams and configuration based on feature diagrams.

V-Manage from European Software Institute (ESI) is a tool that supports system family engineering in the context of MDA (Model Driven Architecture) from OMG. The feature modeling notation is FODA-inspired. The objective of V-Manage is to support users in implementing feature models (in V-Manage called decision models) and application models. V-Manage consists of three modules:

- V-Define: Decision models (V-Manage's term for feature model) are made using V-Define. V-Define's specific languages are XSD and HTML.
- V-Resolve is used to make application models by setting values to the decisions defined in the decision model. It is here we make a suitable configuration of the general decision model. V-Resolve's specific languages are XML and HTML.
- V-Implement: When marking a decision, V-Implement links the variation parameters attached to the marked decision to some external components or to other defined dependencies from V-Resolve. V-Implement generates the result of a configuration to HTML-, PLC-code, UML Model or Requirements document. The specific language of V-Implement is XSLT.

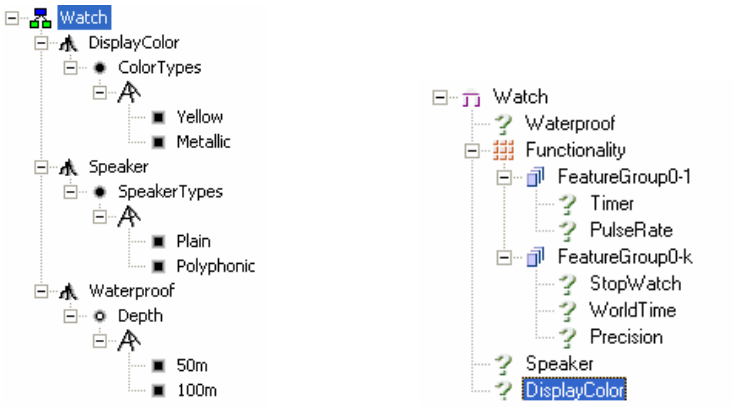

Figure 5 Watch feature model in FeaturePlugin and in V-Manage

The elements of the feature model of V-Manage (Figure 5) are HTML links. Following such a link gives the additional requirements (attributes) of that element. This process (selecting an element and the attributes will be displayed) is similar to when you click a link in a web-browser.

FeaturePlugin draws the additional requirements of a feature into the feature model. For instance the feature Waterproof in Figure 5 is expressed with its additional requirements mandatory and choices (decisions to be made to a feature) 50 and 100. The feature choices are actually implemented as alternative features of the feature Depth. This is typically a FODA approach to feature modeling, where every characteristic and properties of characteristics are modeled as features.

In FeaturePlugin we have some "extra" characteristics such as ColorTypes, Depth and DisplayTypes. These characteristics are hidden or abstracted away in the V-Manage model in Figure 5. As mentioned before, FeaturePlugin is based upon the FODA's conceptual modeling notation. Therefore it is important that all additional requirements are visible in the model. In V-Manage HTML is used to present the decision. Additional requirements are abstracted away and are displayed whenever their respective decision is clicked.

By organizing features as characteristic trees the way FeaturePlugin does, feature models can get very large and heavily detailed, as the number of characteristics can increase very easily. In V-Define much of these details can be abstracted from the feature model which results in a more readable model.

Although these two representative tools have differences, they are similar in the sense that they solely focus on feature and resolution models. Types and/or additional properties of elements in the feature models are either included in the feature model (FeaturePlugin), or they are specified separately, but still as part of the feature model (V-Manage).

What about the situation where we have a base of existing classes defining types and properties of elements of watches, and maybe even a class that defines the overall structure of watches? One answer could be to leave feature modeling all together and just use framework technology in order to express variations on watches. However, as there are still advantages to feature and resolution modeling, the answer could also be to include the base model in feature/resolution modeling. This is exactly what the BVR approach is about.

3. THE BVR APPROACH

3.1 Model

The BVR approach has been developed in the Families project [4] and documented in [2]. Figure 6 is a more detailed model of the BVR approach. A similar approach is found in [8].

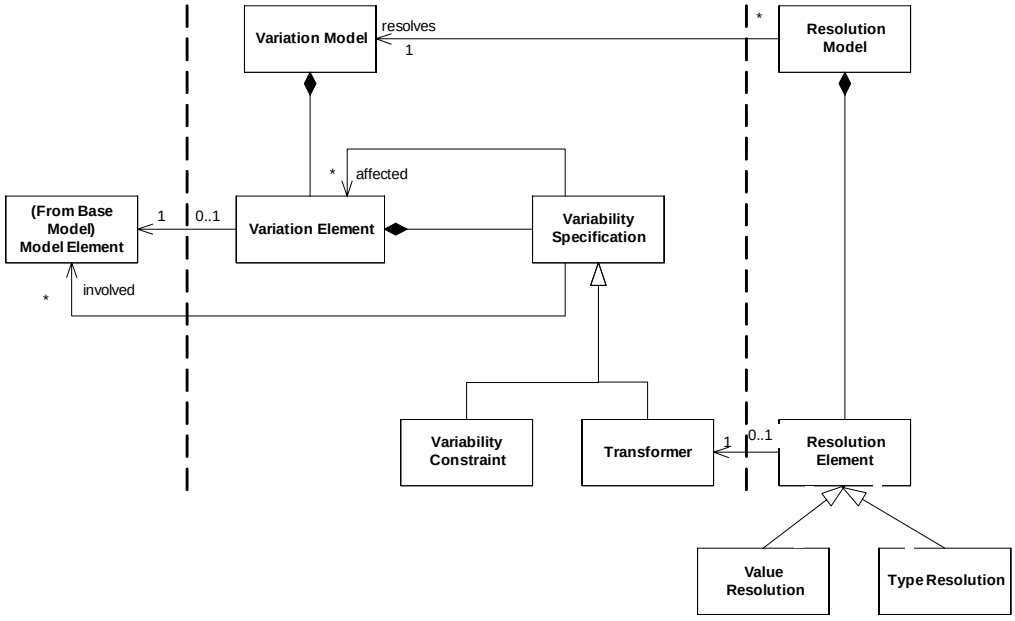

Figure 6 BVR in details

The approach is defined by a meta model divided into three parts.The Base Model will be any model in a given language. The Variation model will contain Variation Elements, where each element refers to the Base Model Element that is subject to variation (implying that those that are not related are not subject to variation). This relationship has a zero-to-one cardinality, as not all Model Elements are affected by variability. Variation Elements only contain the information that the referenced model elements may be affected by variations; the information containd in the Base Model element is not duplicated.

Variation is specified in a Variability Specification: it may in general involve other model elements, and affect a number of variation elements. Variability Specification comes in two kinds: Variability Constraint represents constraints on valid resolutions and distinguishes between valid resolution models and invalid ones; Transformers have

concrete transformations associated with them. When values are bound to transformers (from the Resolution Element), this defines the transformation of the Variation Model and the Base Model into a specific model.

Resolution Model defines resolutions of variabilities for a system family model. It is a collection of resolutions that references variability specifications in a system family model. A Resolution Model represents a binding of variability specifications, which can be used to derive a new, more specific model. A Resolution Model that contains resolutions for all variability specifications of a model represents a derivation of a system model. Resolution Element represents a binding of variability. This is either a complete binding in which all variability is resolved, or a partial one in which some variability still is present. A resolution has a number of effects, which represents the effects a resolution has on the model.

The model in [2] is more detailed (especially on Variability Constraint and Transformer) than the one in Figure 6, but Figure 6 is adequate for the purpose of this presentation. The important thing to illustrate the implementation of is the relationship between Base Model and Variation Model.

3.2 BVR Feature modeling

System families within a domain are most often based upon a domain model. A domain model lends itself to a basic design model in terms of a number of classes. The example introduced above belongs to a domain, where some of the elements readily may be represented by a number of Java classes:

```
class Watch {
  Color color;
  Waterproof waterproof;
  Depth depth;
  ...
}
class Color {...}
class Yellow extends Color {...}
class Metallic extends Color {...}
class Depth {...}
class 50m extends Depth {...}
class 100m extends Depth {...}
```

Instead of making feature models that re-model the facts represented by these classes, the variation models of the BVR approach only defines the variations, and it does so by relating these variations to elements of the base model.

Even though there is a major difference between the BVR approach and other approaches, it is still desirable to present feature models to the users. Figure 7 gives an indication of what kind of feature models the BVR approach will have: instead of feature models where every element is a feature, the BVR approach recognizes e.g. that color is a feature, while Yellow and Metallic are possible values (indicated by different symbols).

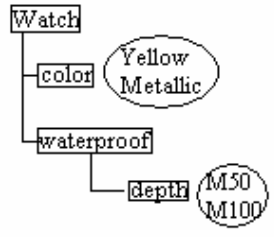

Figure 7 Fragment of BVR feature model

The reason that it becomes like this in BVR is that it relates variations to elements in a base model in a given language. It may therefore distinguish e.g. between classes, objects, attributes of objects and possible values for these attributes. Other feature model approaches are designed without any relation to base models. Therefore they have to stick to features as the only mechanism (i.e. possible values of attributes also become features), or they have to 'invent' mechanisms often found in languages, e.g. types and values.

In addition to the division into base and variation models, the BVR approach also implies that it is only possible to include in feature models (i.e. variation models) elements that are meaningful with respect to the language in question. This is important when it comes to generate the specific system models (in this case specific Java programs).

4. BVR tool and its implementation

The BVR approach relies on the possibility of establishing and maintaining the relations between the variation models and the base model, and between the resolution and variation models. In order to demonstrate that this is possible, we have built a prototype tool called Object-Oriented Feature Modeler (OOFM). For the purpose of this prototype tool implementation we have used Java as the language of the Base models – it could just as well have been e.g. UML. The principles would be the same. In addition we have restricted ourselves to a selected set of variation mechanisms.

4.1 Base-Variation model relation

The OOFM prototype tool was made in parallel with the development of the BVR approach. Therefore its Variation Model has a slightly different set of meta classes, see Figure 8.

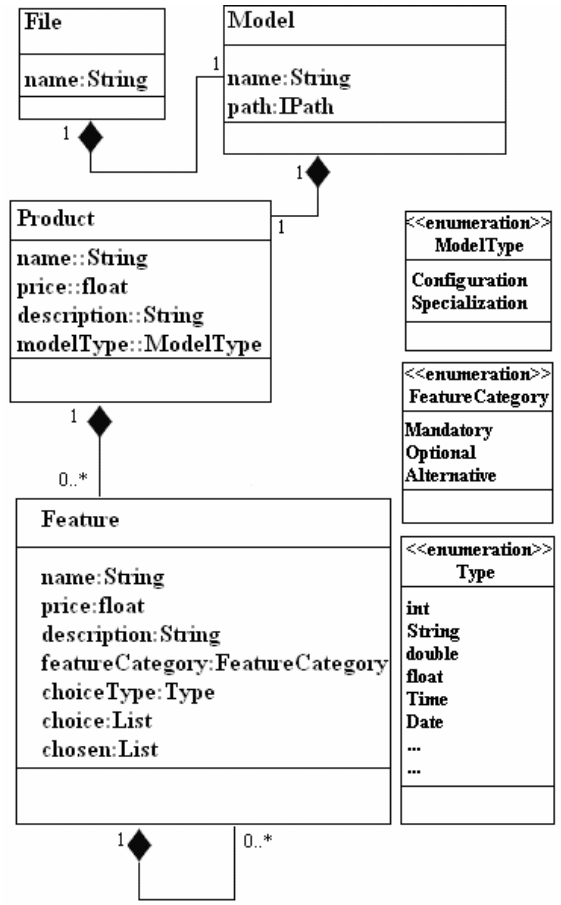

Figure 8 Variation model behind the OOFM tool

According to the variation model of OOFM, a Model contains exactly one Product. Product may have zero or many Features, each containing zero or many Features. That is, a Feature can contain other features.

Waterproof is an example of a feature that contains two other features: Depth and Time. Feature cardinality is represented as mandatory ([1...1]), optional ([0...1]) and group feature cardinality as alternative (<1-n>). A Feature is mandatory, optional or alternative.

Feature choices are stored in a List in the container feature object. For example the sub-feature Depth has the choices 50 and 100, which are kept in the choice List in the feature object Depth. Depth choices state the waterproof depth (in meters) of a watch. Similarly, the sub-feature Time has a choice List that contains the choices 0, 5, 10 and 15. Time choices (in hours) tell us how many hours a waterproof watch can be under water before it no longer can resist water.

Figure 9 illustrates how Variation Models and Resolution Models are linked to the Base Model.

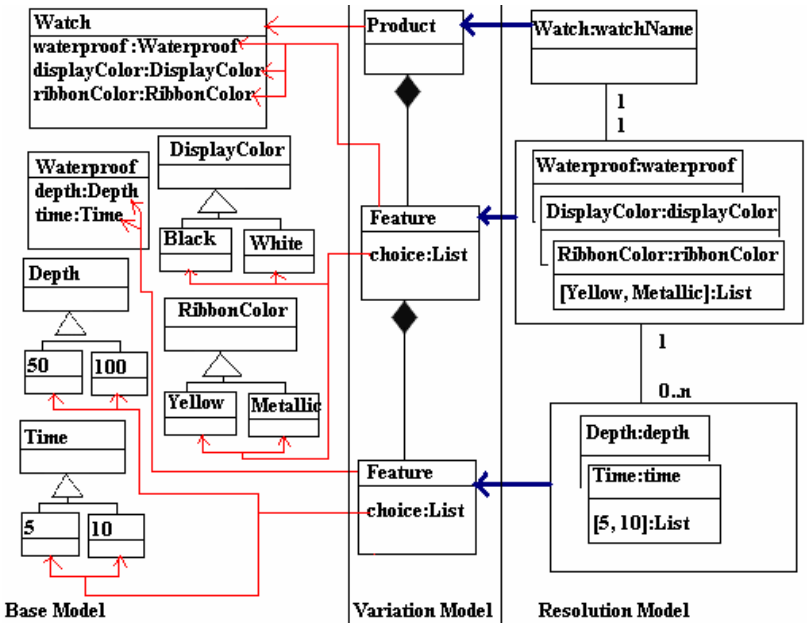

Figure 9 Base, Variation, Resolution Models

The lines between the Variation Model and the Base Model indicate to which element of the Base Model the variation applies. Feature definition in OOFM is not totally automatic. OOFM has the ability to recognize and display all object-fields as possible features. But it needs assistance from the users to decide which object fields it can define as features. The resulting resolution model will contain the variable features from the variation model and those object fields of the base model that were not defined as features. Note that Figure 9 illustrates how the tool maintains the different models and their relations; this is not what is presented to the user. The user will still see the Base model (i.e. in this case Java program text) and a feature model corresponding to the underlying Variation model.

4.2 Feature modeling

OOFM support feature modeling from base models and (as a special case, because it is also a feature modeling tool) feature modeling regardless of base models. OOFM may produce three kinds of models: feature model, configuration model and specialization model.

Feature model is a model of a product based on the variation meta model (Figure 8) of OOFM. The tool has the advantage of being able to make feature models based upon a base model (in terms of Java classes), by recognizing selected elements of Java and create a feature for each of these elements.

In order to be able to model from a base model we need to link our variation model to a base model, that is to set a base model. Base model is set by selecting “Set Base Model” from the toolbar menu (Figure 10), or by right-clicking on the root of the feature model and selecting the “Set Base Model” choice. A list of base-models will be displayed, from which one can be chosen. In our example we would select the collection of Java classes with Watch, Color, etc.

Setting the root of the feature model to a Java class, the tool can then define features according to the base Java class. Each feature may in turn have other features, which in turn may have features, etc. The feature that may be at a specific place in the feature model is determined by the link to the element of the Base Model. As an example, when the root feature model object is linked to the class Watch, possible features may be

'Color', 'Speaker' and 'Functionality'. The tool will provide the possible features by consulting the base class Watch (Figure 10).

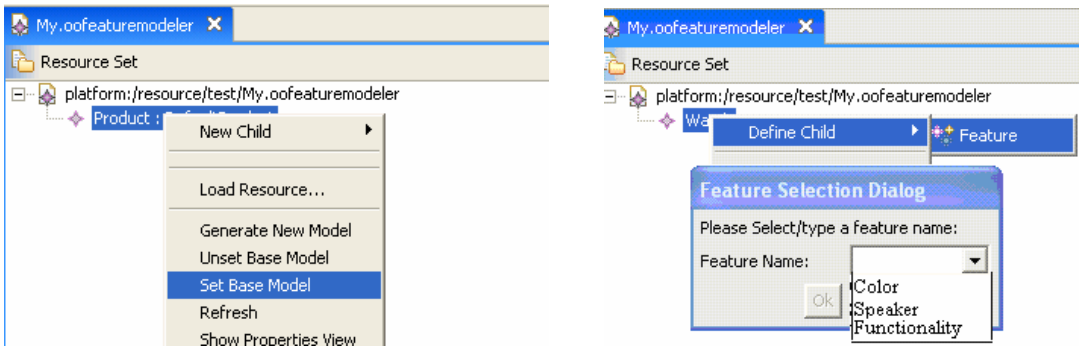

Figure 10 Setting the Base Model, and defining features based upon an element of the Base Model

Features have a Choice List and a Chosen List. The Choice List contains possible choices a feature can have. The Chosen List contains zero, one or all the elements of the Choice List depending on the feature's cardinality.

When the model is linked to a base model, the Choice List of the model-element will be filled with subclasses of the class that is linked to from that model-element. In the base model, each subclass has a Boolean value "choice". If this Boolean value is set to true, then the subclass that contains the value will be added to the model-element's Chosen List. If the model is *not* linked to a base model, then the Chosen List can be generated by defining new features or feature values.

A feature can be one of three feature kinds Mandatory, Optional or Alternative. An Optional feature has a [0...1] feature cardinality, which means that this feature's Chosen List can contain none or exactly one of its Choice List elements. A Mandatory feature has a [1...1] feature cardinality. Therefore the feature's Chosen List must contain exactly one of its Choice List elements.

An Alternative feature has a <1-n> group feature cardinality, where n is the number of elements in the feature's Choice List. The Chosen List of an Alternative Feature can contain at least one of its Choice List elements and at most all of them. The Choice List of an Alternative Feature contains its child-features (sub-features), since this kind of feature does not have choices.

The tool supports two kinds of resolution models. Configuration -and Specialization models. In both cases the tool will generated new Java classes (based upon the base Java classes and the feature resolution). The following rules are applied when making Configuration and Specialization models:

- Configuration: configuring a model
  - Features can be added only if the model is linked to a Base Model.
  - New features can not be created, only those that are found in the Base Model.
- Specialization: creating a new model from an existing:
  - New features can be added to the model.
  - The Choice List can be changed (elements can be removed and new elements can be added).

Both kinds of models may be used as new feature models (where some of the decisions have been taken, in order to make a sub-family of systems), or the tool may generate the Java program for use in a specific product.

4.3 IMPLEMENTATION

The tool is made as an Eclipse plugin and based upon the Eclipse Modeling Framework (EMF). The feature modeling editor is based upon a meta model made according to the

Variation model part of the meta model in Figure 8. This is done by defining the meta model in terms of annotated Java classes and using the generator for tree-oriented model editors provided by the EMF.

The Java Development Technology (JDT) is used to represent Base Java programs and Java programs that are generated on the basis of a resolution of a Variation model. The resolution process generates Java programs in term of objects according to JDT.

5. CONCLUSION

The paper has demonstrated that it is feasible to implement a feature modeling tool that does not make separate and independent feature models, but rather feature models that are defined relatively to a base model. The paper has also demonstrated how such relative feature models can exploit that they are based upon a base model, both when it comes to the definition of sensible features of feature models, when it comes to both configure and specialize such feature models, and when it comes to the generation of specific models based upon resolved feature models. The tool is a prototype tool made in order to illustrate the relation between a base model and feature models. The next step will be to implement more kinds of features and especially constraints between features.

6. REFERENCES

[1] Antkiewicz, M. and Czarnecki, K.: *FeaturePlugin: Feature Modeling Plug-In for Eclipse*, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, Vancouver, British Columbia, Canada., 2004,

[2] 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 *Research Issues in Software Product-Lines*, Käkölä, T., Ed., Springer, 2006.

[3] Czarnecki, K. and Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*, vol. 864: Addison-Wesley, 2000.

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

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

[6] Haugen, Ø., Møller-Pedersen, B., and Oldevik, J.: *Comparison of System Family Modeling Approaches*, SPLC 2005, Rennes, France, 2005, LNCS 3714, Springer

[7] 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.

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