An eclipse-based Feature Models toolchain

Luca Gherardi, Davide Brugali

Dept. of Information Technology and Mathematics Methods, University of Bergamo
luca.gherardi@unibg.it, brugali@unibg.it

Abstract. Feature models are used in software engineering for modeling all the possible configurations of the software of a specific domain. They capture the commonalities and the variabilities among these software and offer a formalism to clearly represent these properties in a separate way. Features represent end-user characteristic of the software and can be optional or mandatory. A selection of a number of these features defines one specific configuration of the software and could be used for example for configuration purpose.
Along the years, at least two extensions of the feature models have been proposed in order to improve the original proposal, but only few attempts of providing a set of graphical tools can be found in literature. Moreover some of those are not open source.
In this paper we propose a tool chain completely based on the Eclipse Modeling Framework and the Graphical Modeling Framework. It consists of: (i) a meta-model that describes the rules for defining feature models, (ii) a graphical tool for creating feature models and defining constraints between the features, (iii) a graphical tool for selecting one possible configuration and checking if it satisfies all the constraints.

1 Introduction

Feature models were introduced for the first time two decades ago and are today widely spread in the context of the Software Product Lines [13]. They are really generic and for this reason they can be used in various different fields. In this paper we describe the meta-model and the tools that we have designed in the context of the European project BRICS [1] in our attempt of introducing feature models in the robotics field. Despite our work is more focused on robotics, in the paper we will keep our attention on the feature models and we will use a generic example because it allows us to describe the feature models without having to introduce a number of terms that are specific of the robotics field. In the description of the tools we will show a more complex example related to the robotics field without going into details of the meaning of the terms.

Feature models are a useful formalism, which allows the developers to model and define all the possible configurations of a family of software in such a way that is independent from the implementation mechanism. These configurations share a set of properties and functionality (aka features) that are always present

in the software of the specific domain and that are called *commonalities*. On the other side some features are present only in some specific configurations and so they define a subset of the domain. They are called *variabilities*. Feature models capture these commonalities and variabilities among a family of software and offer a method to clearly represent them in a separate way.

Feature models are typically used in two phases of the software development process:

- During the specification of the requirements they allow the designers to define which configurations of the software will be supported by the product line. In this stage they give to the designers a view on the entire family of systems.
- During the composition of the software instead, they allow the developers to select a particular configuration of the system and so they provide a view on a specific application. In this stage it is particularly interesting the use of the defined configuration as input for an automatic toolchain, which is in charge of deploying the final product.

The two stages of the development process also suggest two different users of these models: the system designer and the application developer. The first one is in charge of designing the models and defining the allowed configurations. The second one instead has the role of selecting one available configuration and developing the corresponding application.

The work described in this paper is part of a work package of the BRICS project that aims to provide a set of tools that allow the management of the software variability in robotics. In particular our final goal is to make as easier as possible the process of designing and transforming feature models in order to realize an input for the application deployer. That means to provide to the system designers and the application developers with the possibilities of specifying all the possible configurations, selecting one of them, transforming it in a configuration file and finally easily deploying the desired application.

For achieving this goal we have to model the variability from three different points of view:

- Variability specification. That means describing which are the variabilities of a system (variation points) and how they can be solved (variants).
- Variability implementation. That means specifying how the different variants will be implemented in the final applications in terms of architectural elements.
- Variability resolution. That means defining a function that maps each variant to one or more architectural elements.

In this paper we focus on the first point and we propose two completely eclipse-based tools that allow the specification of the variability through the use of the feature models. We started the implementation of a new plugin because no one of the open source eclipse-based tools that we analyzed provides all these features:

– Conformity to the standard specification of the feature models that was defined in literature (see section 2) for what regards the features, the containments and the constraints.
– The presence of a graphical tool that allows the system designers to define feature models in form of feature diagrams in a simple and user-friendly way.
– The presence, in the same graphical editor, of a functionality that allows the application developers to select one available configuration directly from the same feature diagram defined by the system designer.

In particular for what regards the last point we found out that most of the available plugins allow the selection of a configuration only by using a tree-view of the model or by recreating a new feature diagram in which the user has to insert some of the features defined in the feature model. Let's think about a use case in which the model defines more than hundred features. In that case those ways of selecting configurations can be very uncomfortable since the user doesn't have a complete view of the available features and of the structure of the model. In our opinion it would be more user-friendly and quick reusing the diagram defined by the system designer and having a toolbar button that allows the selection of the features that have to be included in the configuration.

During the development of our plugin we tried to take in account all these points and at the same time to keep the graphical interface and the feature model design flow as simple as possible.

The paper is structured as follows.

Section 2 describes the fundamental concepts of the feature models and their extensions that have been proposed along the years. Section 3 briefly presents some attempts of providing eclipse based feature model tools. Section 4 describes the tools that we have developed and in particular the meta-model on which they are based. Finally section 5 draws the relevant conclusions.

2 Feature Models

Feature models were proposed for the first time in 1990 in the context of the Feature Oriented Domain Analysis (FODA) approach [15]. FODA aims to identify the properties and the functionality of a software, which are commonly present in applications of a specific domain, and separate them in two groups: those that are always present and those that are present only in some applications. For representing these properties and functionality FODA proposes a formal method: *the feature model*. In this section we will describe its original proposal and the two major extensions that were proposed along the years.

2.1 Basic Feature Models

A feature model is an hierarchical composition of features. A *feature* defines a software property and represents an increment in program functionality [10]. Compose features, or in other words select a subset of all the features contained

in a feature model, corresponds to define a possible configuration of a software that belongs to the application domain described by the model. This selection is usually called *instance*.

On the base of the feature models, the FODA experts have defined a graphical representation called *feature diagram*. In this section we will refer to the feature diagram depicted in figure 1 in order to exemplify the characteristics of feature models. The diagram describes the functionality of a family of geographical maps websites.

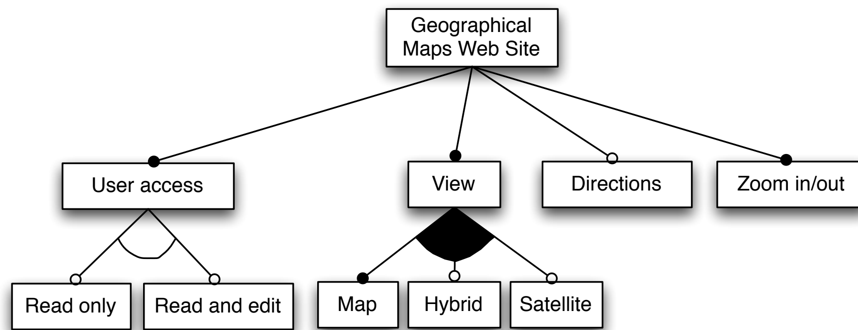

Fig. 1. An example of Feature Diagram of a geographical maps website

Feature models are organized as a tree and the root feature, also called *concept*, defines the application domain. Features are represented by means of white boxes, which contain the feature names, and are connected to the children features by means of edges, which represent containment relationships [9].

Features can be discerned in two main categories:

- ***Mandatory***. Mandatory features **have to** be present in every possible configuration of a software that belongs to the domain described by the model. They usually define the core of the software and represent functionality or properties that are fundamental in the specific domain: the commonalities. In the feature diagrams they are depicted by means of a black circle on the top. In the example the *Zoom in/out* functionality is a mandatory feature. Every maps website in fact allows the user to execute zooming operations in order to explore the maps.
- ***Optional***. Optional features **can** be present but they are not mandatory. They represent functionality or properties that characterize a specific configuration of the software: the variabilities. In the feature diagrams they are depicted by means of a white circle on the top. In the example the *Directions* functionality is an optional feature, which means that not all the maps websites provide it.

Features can be connected to their children features by means of three types of containment relationships:

- ***One-to-one*** *containment*. It is the simplest relationship and it means that the parent feature can (or has to) contain the child feature. In the example it is represented by the containment between the root and the *Directions* feature and by the containment between the root and the *Zoom in/out* feature.
- ***Or*** *containment*. It is a relationship between the parent feature and a set of children features. It means that from the children features **at least one** has to be present in a possible configuration of the software. In the example it is represented by the containment between the *View* feature and its children. This relationship is depicted by means of the black semi-circle that connects the edges.
- ***Alternative*** *containment (X-Or)*. It is a relationship between the parent feature and a set of children features. It means that from the children features **only one** can be present in a possible configuration of the software. In the example it is represented by the containment between the *User access* feature and its children. This relationship is depicted by means of the white semi-circle that connects the edges.

The basic feature models also define two kinds of constraints between the features: *requires* and *excludes*. These constraints allow the definition of a subset of valid configurations. They are typically expressed in the form *A kind_of_constraint B*, where *A* and *B* can be a simple feature or a composition of features by means of logical operators (AND, OR, XOR, NOT).

- ***Requires*** *constraint*. It means that if a feature A is selected to be part of a configuration, then also a feature B **has to** be selected. If A and/or B represent logical rules the constraint imposes that if A is true, then also B has to be true. To be noticed that for solving the logical rules the value of a feature has to be considered true if the feature is selected.
- ***Excludes*** *constraint*. It means that if a feature A is selected to be part of a configuration, then a feature B **cannot** be selected. If A and/or B represent logical rules the constraint imposes that if A is true, then B has to be false.

2.2 Cardinality-Based Feature Models

The cardinality-based feature models propose to replace the properties *optional* and *mandatory* and the containments *or* and *alternative* with a cardinality-based annotation. In particular these ideas are proposed in two different works:

- [14] proposes a ***feature-cardinality*** approach. The idea consists of marking each feature with a lower bound and an upper bound. The upper bound defines the maximum number of times that the feature can be present in an instance.

– [16] proposes a ***containment-cardinality*** approach. The idea consists of marking each containment with a lower bound and an upper bound. The lower bound defines the minimum number of sub-features that have to be present in an instance whereas the upper bound the maximum number. According to this approach the containment relationships "*or*" and "*alternative*" are respectively replaced by $[1 \ldots *]$ and $[1 \ldots 1]$.

2.3 Extended Feature Models

The extended models propose to attach some information to the features by means of attributes [14]. The purpose of the attributes is to allow a more concise representation of feature models. In fact the idea is to use the attributes for representing information that are important but not so relevant to be represented as features. Attributes are defined by means of a name, a type and a value.

3 Related Works

Despite along the years feature models have gained popularity, only few attempts of providing a set of tools, which support the design of these models, can be found in literature. Moreover some of those are not open source. In this section we will focus on the eclipse-based tools, because with respect to the standalone tolls they can be more easily integrated with other tools. From this point we will use the term graphical editor for defining an editor which allows the representation of the feature models in form of feature diagrams (see figure 1).

The Eclipse Modeling Framework Technology project (EMFT)[4], which aims to provide a set of new tools that extend or complement the Eclipse Modeling Framework (EMF)[2], proposes a feature model plugin called EMF Feature Model [3]. This plugin is still in the incubation stage and it is based on two meta-models. The first one describes the rules for modeling the variability and designing feature models, whereas the second the rules for creating instances of the feature models. The major drawback of this plugin is the dependency to pure::variants [11] for providing a graphical representation and the constraints evaluation. Pure::variants is another well done plugin but we avoided using it due to the commercial license. Indeed, it is free available only in a limited version.

Another interesting plugin, which is still in development, was designed at the university of Waterloo [6]. In contrast to the proposal of [3], this plugin uses a single meta-model for representing both the feature model and its configurations. They provide an XPath support for the definition of the constraints and also a constraints evaluation engine. This plugin is completely open source but it doesn't provide a graphical editor.

The P&P Software GmbH and the ETH-Zurich developed an open source feature model plugin called XFeature [12]. This plugin provides a graphical editor, an XPath constraints definition support and a constraints evaluation engine. The editor is in general very complete but in our opinion the editing of the feature model is not really user friendly. For example a tools palette is not provided

and the user has to edit the model by continuously using the context menu. Furthermore the plugin allows the users to define an instance only by creating a new feature diagram in which only the features defined in the feature-model can be inserted.

Finally one of the most complete and open source plugin that we found in literature is FeatureIDE [5]. It provides a lot of functionality, for example a graphical editor which is compatible with the standard notation (as far as we know it is the only one), a statistics view that collects information about the number of features, the number of variants and the number of possible configurations and last but not least the possibility of comparing the current version with the last saved version in order to reason about how the changes in the model will affect the product line. Despite this editor is very complete, in our opinion it has some drawbacks: (i) like the previous plugin it doesn't provide a tools palette for editing the model, (ii) it provides a tree representation for the definition of the feature model instances but it doesn't allow the users to do this operation by using the standard feature diagram representation, which is more intuitive, (iii) it doesn't explicit allow the definition of the exclude constraint (the workaround is defining "*A excludes B*" in the form "*not (A and B)*").

4 The meta-model and the tools

The tools that we are going to propose in this section are based on the Eclipse Modeling Framework (EMF) and on the Graphical Modeling Framework (GMF) [7]. EMF and GMF are two eclipse projects that allow the development of domain specific languages (DSLs) and graphical editors for their visual representation. Every DSL is based on a set of rules that are defined by means of a formal meta-model. During the development of the eclipse plugins these meta-models are typically expressed through the Ecore format [17], which is a small and simplified subset of UML.

4.1 The Feature Meta-model

Figure 2 depicts the Ecore meta-model that represents the core of our tools. Due to the fact that it was designed by following the feature models specification, which is a standard, it is quite similar to the meta-models of other tools presented in the previous section.

The root entity of this model is the class ***FeatureModel***. It represents the feature model and contains a set of *Features*, *Constraints* and *Instances*.

- ***Feature***. The features are defined by a name and by the boolean attribute *required*. When its value is true the feature is mandatory, otherwise the feature is optional. Features also have another boolean attribute (*root*, true only for the root feature) and two integer attributes (*lowerBound* and *upperBound*) which allow the definition of the feature cardinality (see sub-section 2.2). Finally features contain *Attributes*, *CompositeFeatures* and the sub-features that are their direct children (one-to-one containment).

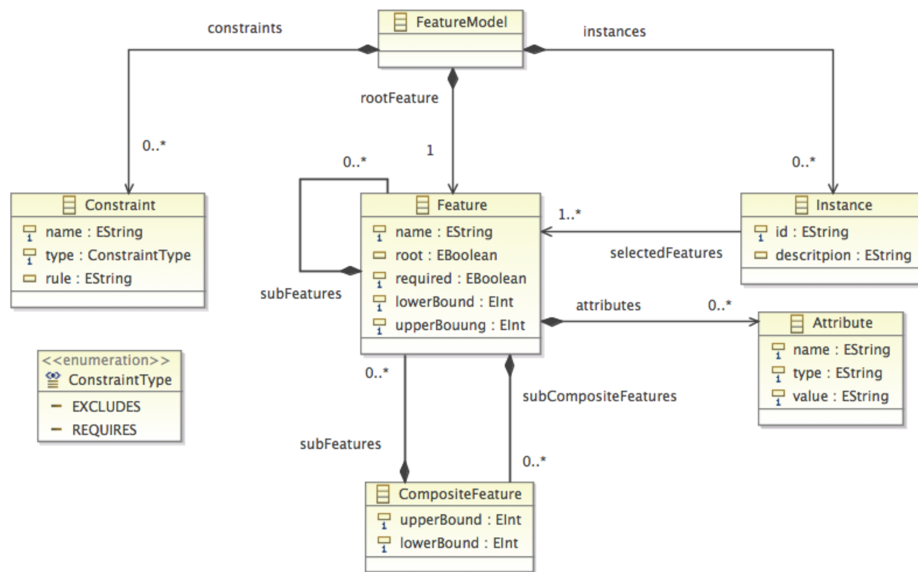

Fig. 2. The Meta-Model

- ***Attribute***. The attributes are defined by a name, a type and a value. They allow the representation of the extended models (see sub-section 2.3).
- ***CompositeFeature***. The composite features are used for representing the group containments. They are defined by two integers (*lowerBound* and *upperBound*) that allow the representation of the standard containments (alternative and or) and of the cardinality containments. The composite features contain the features that are part of the containments.

– ***Constraint***. The constraints are defined by a name and a rule. The rule is a string and it is composed of two members separated by the ***ConstraintType***, which can assume two values: *requires* or *excludes*. The members can be defined as any possible logical combination of the features.

– ***Instance***. The instances represent a possible configuration of the model and are defined by a name and a description. An instance contains references to the features that are selected for being part of the configuration. More instances can have references to the same feature.

The presented model defines the rules for creating feature models that are conforming to the specification described in the section 2. After having defined this model in EMF, it is already possible to create feature models by means of a simple tree-view editor. In the following sub-section we will introduce the editor that we have developed in order to provide a graphical representation.

4.2 The feature model editor

The feature model editor allows the creation of feature models that are conforming to the meta-model presented in the previous sub-section. It is a graphical editor realized by means of the eclipse GMF tools. Figure 3 depicts the editor and shows how the feature models are represented. The figure reports a feature model that describes the possible configurations of a motion planning software. It is a software that involves different functionality in order to plan a path for moving a robot along a collision free trajectory.

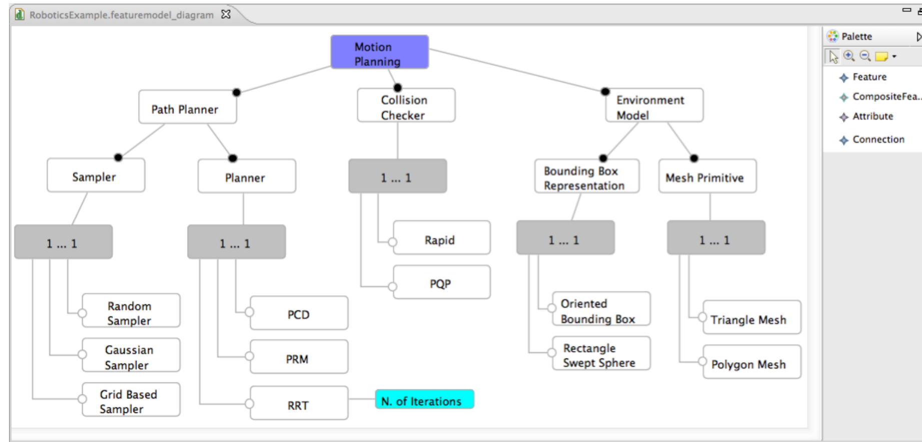

Fig. 3. The feature model Editor

The graphical representation is conforming to the standard convention of the feature diagrams, except for the group containments. In particular we represent the different entities defined in the model in the following way.

- Features are represented by means of a white box that contains the name of the feature. The blue box represents the concept (root feature).
- Mandatory features are represented by means of a black circle on the top whereas optional features by means of a white circle.
- Group containments are represented by means of a grey box that shows the lower and the upper bound of the cardinality.
- Attributes are represented by means of a cyan box that contains the name of the attribute.

The editor also provides the possibility of defining constraints by means of the dialog window depicted in figure 4. It forces the user to create rules that are syntactically correct, for instance it doesn't allow him to insert two logical operators without a feature between them. The constraint presented in the example means that if the feature *Rapid* is selected then also the *Oriented Bounding Box* and *Triangle Mesh* features have to be selected.

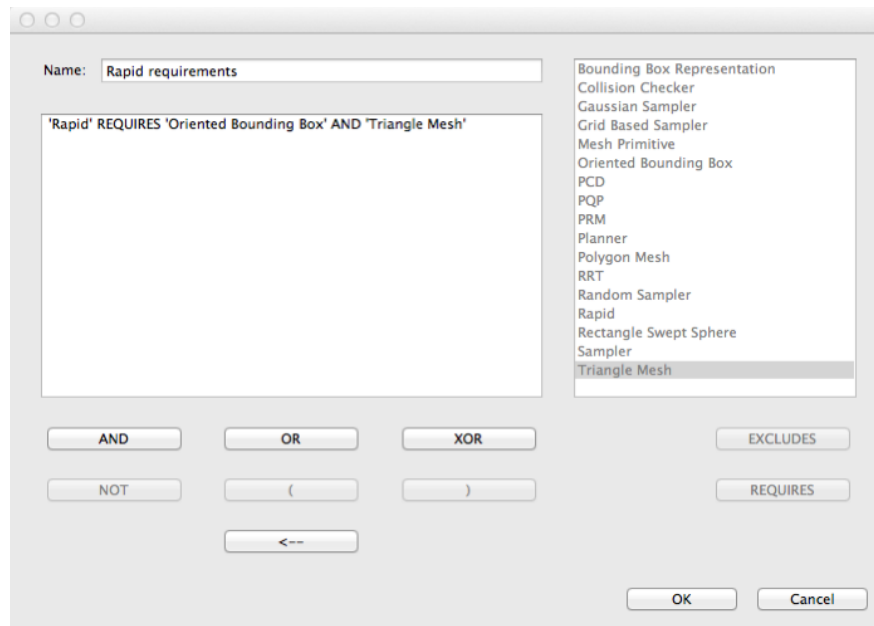

Fig. 4. The Constraints Editor

4.3 The instance editor

The instance editor allows the user to create instances of an existent feature model. That means select a sub-set of features from all the features that are defined in the model. This selection has to contain all the mandatory features, satisfy the cardinality of the containments and respect all the constraints defined on the model. Figure 5 depicts the instance editor. It is composed of two parts: the visual representation of the model and the instances view.

The instances view shows the information about the existent instances and offers a set of commands that allow the user to create, remove and select instances. By looking at the third column of the view it is possible to find out which instance is currently showed in the visual representation. In the presented example it is the "Config 1".

In the visual representation the selected features are drawn in green. The user can select a feature for being part of the instance by simply clicking on it and pressing a button in the toolbar.

Once the user has completed the selection of the features, the instance can be validated. The process of validation checks if the instance satisfies the following requirements.

- All the mandatory features have to be selected to be part of the instance.
- All the cardinalities of the containments have to be satisfied, which means that the number of the selected sub-features of a containment has to be

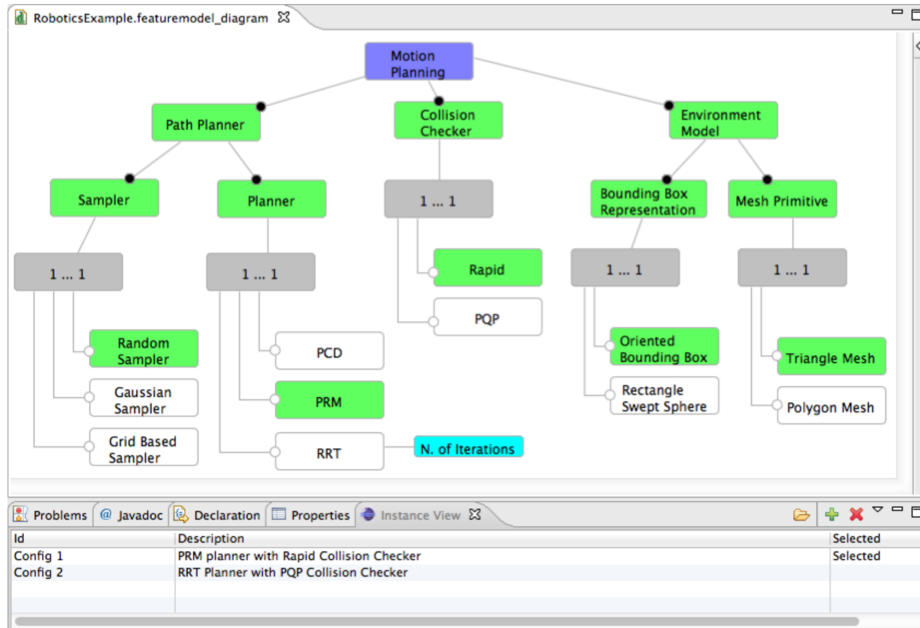

Fig. 5. The Instance Editor

greater than or equal to the lower bound and less than or equal to the upper bound.

– All the constraints have to be respected. The constraint checking is done by using an external open source library called MVEL [8], which resolves the two members of the constraint rule. After that, if the constraint type is "require", the constraint checker controls that both the members are true. In case of an "exclude" constraint instead when the first member is true the constraint checker controls that the second is false.

Finally the instances can be saved on an XML file, which will be the input for the next tools that we are going to develop (see section 1).

5 Conclusions and future works

In this paper we have introduced the feature models, their syntactic and semantic and their typical use in the software development process. We have also described the tools that we have developed and the meta-model that describes the rules for the definition of the feature models.

Our plugin provides a simple and intuitive tool for designing feature models that are conforming to the syntactic that was presented for the first time in 1990 and then extended in different works described in literature. The plugin also provides a tool for creating and validating instances of the designed models.

These instances describe concrete applications and can be saved as an XML file in order to be reused by other tools. We have tested the plugin by using it for representing the possible configurations of robotics software and validating them according to the constraints imposed on the models.

In the next months we plan to design a new meta-model that allows the representation of the variability resolution (sec 1). In particular we plan to associate to each variant described with a feature model one or more of the architectural elements that compose a component based system and that define how these variants are implemented (variability implementation). By using this mapping we finally plan to develop a tool that helps the developers in the process of deploying the applications defined with the Instance editor.

6 ACKNOWLEDGMENTS

The work described in this paper has been funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics). The authors would like to thank all the partners of the project for their valuable comments.

References

1. BRICS, Best Of Robotics. http://www.best-of-robotics.org/.
2. EMF: Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/.
3. EMF Feature Model. http://www.eclipse.org/modeling/emft/featuremodel/.
4. EMFT: Eclipse Modeling Framework Technology Project. http://www.eclipse.org/modeling/emft/.
5. FeatureIDE, an Eclipse plug-in for Feature-Oriented Software Development. http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/.
6. Fmp: Feature Model Plug-in. http://gsd.uwaterloo.ca/fmp.
7. GMF, the Eclipse Graphical Modeling Framework. www.eclipse.org/gmf/.
8. MVEL, MVflex Expression Language. http://mvel.codehaus.org/.
9. OMG, Common Variability Language Wiki. http://www.omgwiki.org/variability/.
10. Pamela Zave, FAQ Sheet on feature interaction. http://www2.research.att.com/ pamela/faq.html.
11. Pure::variants. http://www.pure-systems.com/.
12. XFeature - Feature Modelling Tool. http://www.pnp-software.com/XFeature/.
13. P. Clements and L. Northrop. *Software product lines.* Addison-Wesley, 2001.
14. K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Generative Programming and Component Engineering*, pages 156–172. Springer, 2002.
15. K. Kang. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.
16. M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with UML multiplicities. In *6th Conference on Integrated Design & Process Technology (IDPT 2002), Pasadena, California, USA.* Citeseer, 2002.
17. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, 2008.