MODELLING COMPOSITIONS OF MODULAR EMBEDDED SOFTWARE PRODUCT LINES

Wolfgang Friess
AUDI AG
wolfgang.friess@audi.de

Julio Sincero
University Erlangen-Nuernberg
sincero@informatik.uni-erlangen.de

Wolfgang Schroeder-Preikschat
University Erlangen-Nuernberg
wosch@informatik.uni-erlangen.de

ABSTRACT
Coping with lots of variants is a challenging task in the field of embedded software development. Due to the restricted hardware resources in this domain, it is essential for the embedded system software to be highly adaptable to the specific needs of the application and no unused functionality is implemented. Configurable system software can realise this adaption, but it brings the problem of variant management in this domain. Currently, software product line methods are in the focus of research to cope with high amounts of software variants. But current methods lack of support for systems composed of several subsystems, configured independently. However, such modular systems are very common in the domain of embedded software. This paper introduces a concept for modelling compositions of several software product lines, like the composition of an application software product line and an operating system product line, for example. With this concept, it is possible to model not only single software product lines but also compositions of several ones. This supports the development of modular software systems with high variability. After that, an implementation of a tool for verifying compositions based on this concept is presented.

KEY WORDS
Software product lines, Embedded systems, Reusability, Software Tools

1 Introduction

Embedded systems are becoming more and more complex and distributed. The embedded software of a modern car for example is distributed over a large number of electronic control units and the running code size is up to many mega bytes. To cope with the increasing cost pressure and shortenend development time, the systems are often composed of several commercial off-the-shelf software modules, for example, an embedded operating system or a network driver. Because of the restricted hardware ressources due to the high cost pressure these embedded software modules have to be optimally adapted to the specific use case to avoid waste of resources. This adaption is normally done by static configuration. This means that the adaptions are done before compile time so that unused functionality is not included in the binary file. Examples of static configured off-the-shelf software modules are the eCos operating system [11] or any OSEK-compliant operating system [1] in the automotive domain. Typical configuration parameters for operating systems are the number and priority of tasks.

To handle the configuration variants, software product line methods and especially the usage of feature modelling is very promising. But current feature models lack of support for describing systems, that are created by the composition of several subsystems. Modular systems are very common in embedded systems and different configurable modules often come from different vendors. Therefore, describing modular systems is essential for handling variants in the embedded domain.

Another example for a modular system is a modern car. It consists of many embedded subsystems like radio, navigation system, driver assistance systems, and so on. Each of this subsystems can be seen as a product line described by a feature model. When composing this product lines into a car, some rules must be fulfilled, like that every car has only one kind of radio or that a driver assistance system needs an apropriate way to give information to the driver. Modelling this kind of composition rules is in focus of our work.

This paper shows our work on using composition models in the field of feature modelling and presents a tool for verifying compositions of different software product lines in the embedded domain. In the following section, the current state of feature modelling tool support is shown. After that, composition models to describe modular systems with variants are introduced. Finally, the usage of composition models and a tool for composition checking is demonstrated.

2 Feature modelling - State of Art

Feature modelling is the activity of identifying and organizing the variable and the common characteristics of a software system. The variability is captured in entities called *features* which are organized in an hierarchical structure known as feature model. The graphical representation of a feature model is called *feature diagram*. The use of feature models is manifold. It provides an abstract and concise view of the domain of study, and therefore, it can be applied in any stage of software design [6]. However, the development of a software product line takes advantage on feature modelling mainly at two stages. First, during the *domain analysis*, the first step of the domain engineering, feature

modelling is of great value as it encompasses all features identified during the domain study. Second, as far as configuration in software product lines is concerned, the use of feature models is very beneficial. The variation points of a software system are clearly identified in the feature model, which can be used for the instantiation of a specific product by the selection of the features. There are available different notations and tools for the representation of feature models. Although there are several tools discussed in literature, we will concentrate on two feature modelling tools. We focused on these tools, because they provide not only mechanisms for the design of feature models but also support for a feature-based configuration of software product lines.

2.1 FeaturePlugin

FeaturePlugin is a feature modelling and feature-based configuration plug-in for the Eclipse development platform. It integrates feature modelling into a development environment, which facilitates the support for modelling variability in different artifacts [2].

The plug-in implements the *cardinality-based feature modelling* [8]. This notation extends the FODA [10] notation with feature and group cardinalities, feature attributes, feature diagram references, and user-defined annotations. Moreover, a *feature model* can be comprised of one or more *feature diagrams*.

Besides the ability to design feature models, which is done by defining feature cardinalities, attributes and annotations, the plug-in also offers the possibility to derive concrete configurations which conform to the feature diagram. During this process, optional and group features can be selected by means of check boxes. Features with cardinalities, whose upper bound is greater then 1, can be *instantiated*, which means that the feature and its entire sub-tree are replicated. A configuration wizard is also provided as an alternative configuration mechanism to support the user during the configuration process. The result of the configuration is supposed to be used either as input for code generators or used by a system as runtime configuration.

In order to provide *staged configuration* [7], the plug-in implements specialization of feature diagrams. With the specialization of a feature diagram it is possible to make only a few configuration decisions. The result is a second feature diagram that represents a subset of the configuration options denoted by the first one.

In addition to the specifications of feature diagrams, the activity of feature modelling often requires that other information like feature priorities, binding times, implementation status, etc., must also be recorded as annotations and attached to the feature model itself or its elements. However, this information is very project-dependent. Therefore, the plug-in implements a user-extensible meta-model of the feature notation, which allows the user to edit it as any other regular feature model. However, the system defined features in the meta-model can not be changed.

2.2 pure::variants

pure::variants is a commercial tool for the variant management of software product lines developed by the company pure::systems[1][4]. The tool is an Eclipse plug-in which extends the Eclipse IDE in order to support the development and deployment of software product lines.

By using the tool, software product lines are developed as a set of integrated models [3]. *Feature Models* are responsible for describing the problem domain, *Component Family Models* for describing the problem solution and *Variant Description Models* for specifying individual products from the product line. According to this approach, *Feature Models* and *Component Family Models* are separately and independently captured in order to improve the reusability of models by different projects. The tool uses the information of the different models (product specification) in order to make the correct selection of modules that must be present in the final product.

pure::variants can be seen as an integrated development environment to support the individual phases of the software product line development process. It is an open framework that facilitates the integration with other tools and types of data such as requirements management systems, object-oriented modelling tools, configuration management systems, bug tracking systems, code generators, compilers, UML os SDL descriptions, documentation, source code, etc.

Both, feature models and family models are hierarchical tree structures comprised of *elements* (tree nodes). Elements are typed and may have any number of associated attributes. Moreover, each element may be guarded by a number of restrictions. Relations among elements are also allowed, examples of currently supported relations are: *requires*, *requires-for*, *conflicts*, *recommends*, *discourages*, *cond_requires* and *influences*. Additional information can be associated to an element by the use of *attributes*. An attribute is always typed and named, it is able to represent the information allowed by its type. The assignment of values to attributes can be fixed, non-fixed and also calculated by a *calculation expression* defined in Prolog.

Feature models can be constructed using the elements as explained above. However, the elements of a feature model are the specific *mandatory*, *optional*, *or-*, *alternative* features. The *model evaluation* is responsible for checking if all relations and restrictions are valid in a variant description and also for calculating attribute values defined by expressions.

2.3 Problem Analysis

The introduced software product line environments are examples of feature modelling tools. They offer the possi-

[1] www.pure-systems.com

bility to model variations of software systems using feature models. But current tools only have limited support for modelling compositions of different software product lines. In order to create modular systems by means of feature models, the following requirements are necessary:

- model compositions of feature models created by different tools and
- to model compositions of feature models using different feature model notations.

Therefore, the following section introduces a concept for modelling compositions of separate feature models.

3 The Concept of Composition Models

Composition models are used to describe modular software systems. Different kinds of composition models are used in software development to describe such systems. For example object-oriented or component-based software systems are often modelled with UML composition models[2]. The basic relationships between objects in general are 'uses' and 'part-of' relations [5]. These relation types are used in combination with other information like the cardinality of the objects to describe the system structure. An example for a composition rule might be: *'One or two A are part of B'*. These relation types are also used in the following.

To model compositions of several software product lines the following information is necessary:

- which software product lines are composed
- which composition rules have to be fulfilled by the composition

To describe the composition rules for product lines it is necessary to consider the variability of different products of a product line. In the following the term 'instance' is used for a single product of a software product line.

It must be possible to specify a group of possible instances defined by certain conditions that all instances of the group must fulfill. In software product lines, the possible instances are identified by the features and attributes they have. So the features and attributes can be used to describe a certain group of instances by certain conditions. In our composition model, these groups are described by feature configurations.

A feature configuration represents a selected feature with specified attributes, selected and configured sub-features.

Figure 1 shows the meta-model of a feature configuration. As a feature configuration can contain other feature configurations, it is possible to describe all kinds of structures in a feature model. Every instance, which

[2]www.uml.org

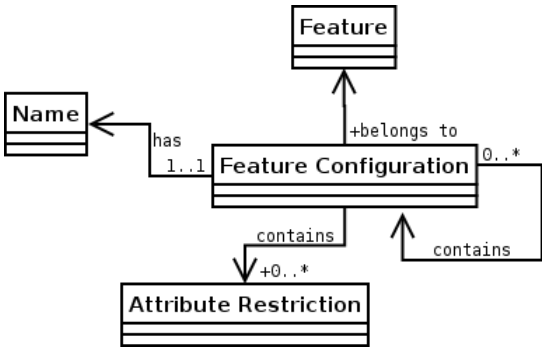

Figure 1. feature configuration meta-model

fits to this structure is part of the group defined by the feature configuration. An example for such a nested feature configuration, consisting of two single feature configurations, is: *'every instance of product line X, which have the features A and B'*. The first feature configuration belongs to feature A and it contains the second feature configuration which belongs to feature B.

As in common feature model notations, features are described more detailed with attributes, it is necessary to add additional attribute restrictions to a feature configuration. An attribute restriction can define for example that the value of an integer attribute must be greater than 10. In that case, not all instances of a product line are part of the group defined by the feature configuration, but only the instances with an attribute value greater than 10. Depending on the data type the value of an attribute can have, different types of attribute restrictions are possible. Which data types are possible in a certain case, depends on the used feature model notation. Common data types are for example string, boolean or integer. The attribute restriction 'greater than 10' for example, would not make sense on a boolean data type.

With feature configurations and attribute restrictions it is possible to define groups of instances by describing specific conditions that every instance of the group must fulfill. But there are also conditions, not only an instance but the whole group must fulfill. This conditions are called 'group conditions'. For example, *every instance of product line X with feature A selected*, is a description of a group of instances. A group condition could be: *feature A has an attribute NAME and the value of this attribute must be unique in all instances of the group*. This group condition must be fulfilled by the whole group defined by the conditions described by feature configurations and attribute restrictions.

To describe composition rules of software product lines, these feature configurations, attribute restrictions and group conditions are combined with the basic relationships 'used', 'part-of' and cardinalities. *'One instance of product line A with the feature 1 is part of every instance of product line B'*, is an example for a composition rule. Figure 2 shows a simplified meta model for composition models.

A composition model consists of feature models and

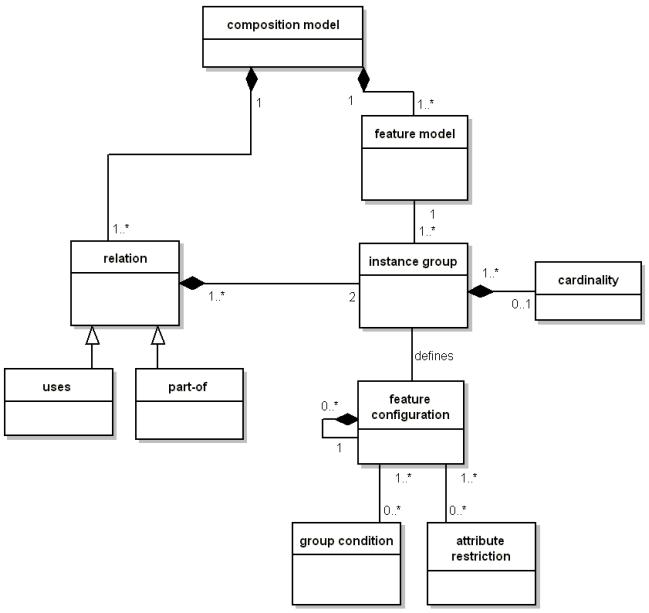

Figure 2. composition meta-model

relations. Possible relations are 'uses' and 'part-of' relations. For each feature model several instance groups can be defined by means of feature configurations. Finally, a cardinality can be defined for each instance group.

The composition models for software product lines combine variability with the basic relationships 'used' and 'part-of'. In the following section, the usage of the composition models is shown in a tool for checking compositions of instances of several software product lines.

4 Implementation of a Composition Checker

To get more experience with compositions of software product lines, a prototype implementing the introduced composition model notation was developed. The prototype allows to create composition models and checks if a given set of instances satisfies the composition rules defined in the composition model. The composition checker is implemented in JAVA with a Prolog[3] evaluation core. The benefit of using a prolog engine for evaluating the composition rules is the extendability. For example, if the necessity arises to add some additional group conditions this can be done easily by adding new prolog clauses to the composition checker.

The feature models and the instances are described by an intermediate format to allow compositions of different feature model notations and tools. This is necessary to allow checking compositions of software product lines from different tools working with different feature model notations. The intermediate format for a feature model is called 'concept interface' and is a list of all possible features in the feature model stored in a XML file. Listing 1 shows an example of such a concept interface.

Listing 1. feature model description

```
<?xml version="1.0" encoding="UTF-8"?>
<concept_interface name="TASK" version="0.1
    " author="User">
  <feature name="TaskSettings">
    <attribute name="Prio" type="int"></
        attribute>
    <attribute name="Name" type="string">
        </attribute>
    <attribute name="Stacksize" type="int
        "></attribute>
    <attribute name="Activation" type="
        int"></attribute>
  </feature>
  <feature name="Type"/>
  <feature name="Basic"/>
  <feature name="Extended"/>
  <feature name="Schedule"/>
  <feature name="Full"/>
  <feature name="None"/>
</concept_interface>
```

The intermediate format of an instance on the other side is a list of all the selected features with the attributes extended by an attribute value. This intermediate format is also stored in a XML file.

As the evaluation core is prolog-based, the feature configurations and the composition rules are represented by prolog clauses. Listing 2 shows an example for such a composition rule.

Listing 2. example for composition rule

```
f_c('fc0', 'OS',['fc1'],[]).
fc_card('fc0',1,1).
f_c('fc1', 'OS-CC-BCC1',[],[]).
f_c('fc2', 'TASK',['fc3'],[]).
fc_card('fc2',0,0).
f_c('fc3', 'TASK-Extended',[],[]).
part-of('fc1','fc2').
```

This rule consists of two nested feature configurations 'fc0'/'fc1' and 'fc2'/'fc3'. The meaning of this rule is, that each instance of 'OS' with the feature 'OS-CC-BCC1' must not include an instance of 'TASK' with the feature 'TASK-Extended'.

The group conditions are also described by prolog clauses. Listing 3 shows two examples for group conditions already implemented in the composition checker. The first one is the uniqueness condition mentioned in the previous section. The second one checks if the sum of the values of the attribute STACK of all instances in the defined instance group is smaller than 150.

Listing 3. examples for group conditions

```
uniqueness('OS', 'Name').
```

[3] www.swi-prolog.org

```
sum('fc1', 'Stacksize', <, 150).
```

When starting the evaluation, the composition checker generates a knowledge-base representing the information from the composition model and the given set of instances. This knowledge-base represents an instanciated feature diagram of the overall composition. That means, if three instances of product line X are part of the composition, three subtrees representing the three instances are integrated into the instanciated feature diagram. For every different set of instances, a different knowledge-base will be generated. So the instanciated feature diagram is an exact representation of the composition defined by the given set of instances. After that, the evaluation core tries to prove the correctness of the composition rules defined in the composition model with the generated knowledge-base. In case a rule was broken, the evaluation will fail. Figure 3 shows the logical architecture of composition checker.

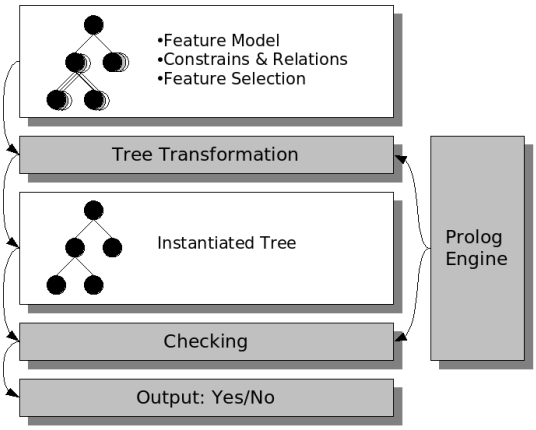

Figure 3. architecture of the composition checker

The evaluation output is a log file describing either which rule made the evaluation fail or true if the evaluation was successful.

5 Related Works

Another approach for verifying composition of modular embedded systems comes from the PECOS project [9]. It supports the construction of software systems out of several modules and includes a component meta-model capable to add composition rules to the components. As this approach is based on component technology, it offers less capability to model variants than an approach based on feature models.

An approach for a formalized description of rules in feature models is described in [12]. It uses the Object Constraint Language (OCL) to decribe complex constraints between features and feature attributes. This constraints can be used for verifying a single variant of the feature model. This approach offers a formalized way to describe rules but does not support composition of several feature models.

6 Conclusion and Outlook

Modelling modular systems with variabilities is essential for using software product line methods in scenarios where many different companies supply software modules which will be integrated into a software system. The development of electronic control units in the automotive industry is an example for such a scenario. The composition models introduced in this paper allow to model such modular systems and are a step towards using software product line methods for embedded systems.

The next step in our project is using the composition checker to evaluate and to improve the usage of composition models on real-world use cases. Furthermore, we work on improving the usability of the composition checker by adding a graphical user-interface. We also plan to integrate the composition of cardinality-based feature models in the next version of the composition checker.

References

[1] *OSEK/VDX Operating System Specification 2.2.3*. www.osek-vdx.org, Februar 2005.

[2] M. Antkiewicz and K. Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM Press.

[3] D. Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, Otto-von-Guericke Universit Magdeburg, 2003.

[4] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.

[5] G. Booch. *Objektorientierte Analyse und Design - Mit praktischen Anwendungsbeispielen*. Addison-Wesley, 1994.

[6] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Computer Science Department, Technical University of Ilmenau, 1998.

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *SPLC*, pages 266–283, 2004.

[8] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[9] T. Genssler and C. Zeidler. Rule-driven component composition for embedded systems. In *4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, 2001.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, 1990.

[11] A. Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[12] D. Streitferdt, M. Riebisch, and I. Philippow. Details of formalized relations in feature models using ocl. In *ECBS*, pages 297–304, 2003.