Questionnaire-Based Configuration of Product Lines in FeatureIDE

Jens Wiemann, Stephan Dörfler, Otto-von-Guericke-Universität Magdeburg,
{jens.wiemann, stephan.doerfler}@st.ovgu.de

***Abstract*—Variability management is an essential part of working on product lines. As an established way to simplify the process of product configuration out of software product-lines, feature models are used to describe the set of features and constraints contained in a given software product-line. This paper proposes a method for automatically generating feature models out of descriptive files and naming conventions. Furthermore existing methods of configuration are considered in this work and to develop an alternative based on questionnaires to enable users or customers to configure a product on their own and to allow experts to design the questionnaires according to their domain knowledge.**

***Index Terms*—FeatureIDE, Feature Model, Extraction, Configuration, Questionnaire.**

I. Introduction

Developing feature oriented, although bringing with it some overhead, facilitates the creation of software when it is planned on custom tailoring it for a multiple number of clients. Feature oriented means developing and maintaining single features that can be combined with others to create a whole product. All of the individual features make up a product line, whereas a subset of those features create a variant of this product. Developing software product lines can result in a large amount of variants, when customizing the software to each customers needs. By developing with a feature-oriented approach the configuration of a single variant can be done by selecting the features a customer needs, automatically including its dependencies. The configuration is based on a Feature Model, that defines the available Features and its relations to one another. A feature model documents the features of a product line and their relationships[1].

Feature Models are essential structures for the feature-oriented development and later configuration of software product lines (SPL). There are projects being developed in a feature-oriented manner, but don't have a feature model yet. Implementing it on top of a given Project can result in a complex task due to the amount of its features and the constraints between them. Nevertheless the benefits of a correct feature model justify the effort to extract one out of a live project. This work aims at automatically generating it out of descriptive files and naming conventions, to simplify a big part of the feature model creation.

Although the feature configuration gives developers the ability to create custom variants, there still has to be a consultant explaining the features to the customer, trying to figure out his current and future needs. As the software grows and gets more features, this process gets difficult, as one can no longer explain all the features, but you still have to figure out if the customer needs them or not. This paper considers existing methods of configuration and tries to come up with a better alternative based on questionnaires to enable users or customers to configure a product on their own and to allow experts to design the questionnaires according to their domain knowledge.

FeatureIDE is an open source IDE for feature-oriented-software development. It provides all the functionality needed to programmatically generate and work with feature models. This includes the logic behind configurations, data structures and the visualization and processing of feature models. The underlying Eclipse enables us to implement the feature extraction and the questionnaire as a plugin.

This work demonstrates the feature model extraction and questionnaire creation based on a real life Project. The Project, an open source ERP system called *Odoo*[1] is predestined to be developed feature-oriented due to a huge amount of features and complex constraints. It is currently being developed and structured feature oriented, although it doest't contain a feature model yet. Furthermore, *Odoo* reached a critical amount of features where a salesman cannot consult a customer by going through all of the features anymore, but has to come up with a more effective way.

Contributions:

- Automatic feature model generation based on descriptive files and naming conventions as a foundation for the thereon based questionnaire.
- Eclipse plugin, that enables the creation of a questionnaire based on a configuration file and the associated feature model. The liberties of the configuration file specification allows very personalized questionnaires.
- The questionnaire changes as choices are made based on the logic behind the configuration file and the dependencies of the feature model. This enables a responsive questionnaire that prevents unnecessary questions.
- Exiting the questionnaire always results in a valid configured variant to always ensure a valid configuration.

II. Basics for simplification of the configuration process

In the scope of supporting the configuration of a given product line some specific tools and techniques are used. This section gives an overview of what is used in this work to archive the simplification.

[1] https://www.odoo.com/

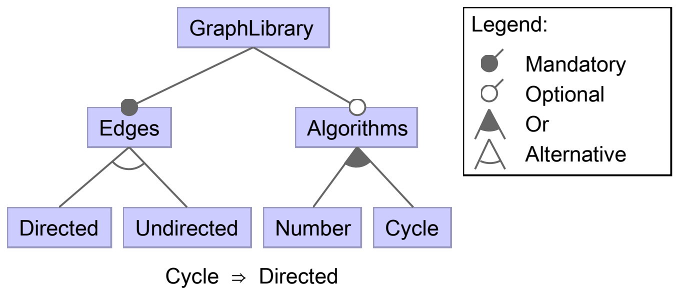

Fig. 1. A simple example of a feature model

A. *Feature Models*

Feature models are multi-purpose structure for product lines. One of their benefits is the visualization as a feature diagram, providing an overview of the containing features, and their hierarchy and dependencies. In addition it classifies the features and their dependencies by Ordinality (Optional/mandatory), logical operator (Disjunction/exclusive disjunction) and whether it is abstract or concrete. Furthermore it contains all information needed to provide a configuration of Features and its validation.

Although feature models can be represented as a boolean formula in CNF, it is typically represented in form of a feature diagram. The feature diagram is structured as a tree. In that manner feature models map the hierarchy of features. The possible relationships of features with a common parent feature are *or*, *alternative* and *and*. In the case of an *or*-group, at least one of the child features has to be selected if the parent feature is selected. Within an *alternative*-group exactly one of the child feature has to be selected if the parent feature is selected. When grouped with an *and*-relation, any number of child features can be selected, if the parent feature is selected. In addition, child features in an *and*-group can be marked as either mandatory or optional. As features' relations may be of higher complexity than just parent-child relations additional constraints can be noted within a feature model. Constraints can contain any propositional expression.

Figure 1 shows an example of a simple feature model. The node *GraphLibrary* is the root feature. Its descendants are the node *Edges*, that has to be selected due to the mandatory marker, and *Algorithms* that is marked optional. The implemented edge types are *Directed* and *Undirected* from which exactly one has to be chosen as they are mutually exclusive. From the algorithms at least one has to be chosen whenever *Algorithms* is selected. The constraint beneath Figure 1 implies that if the feature *Cycle* is selected, the edge type needs to be directed.

B. *Product configuration*

The variability of a product line is represented by it's feature model, which itself is a set of features with specific interrelations. The process of configuration of a product line describes the steps to derive a product from the product line. To archive this, a user has to select a subset of all the possible features within the product line to meet his requirements. However, not all subsets of features result in a valid product. The interrelations of the features restrict the possible combinations of features resulting in a *valid* configuration. If only one of the requirements from the interrelations between the features is not met, no product can be created and the configuration as such is considered *invalid*.

During configuration a user selects or deselects features to his needs. This process requires a lot of domain knowledge on the one hand and detailed information about the implementation of each single feature on the other. With growing numbers of features and thus growing numbers of possible products configuration requires more effort. However, the even greater problem arises from the possible interactions and interrelations of features, which one has to keep track of during configuration. The automatic selection or deselection of a features doe to propagation of a made selection always results in a valid configuration. Nevertheless the propagated selections can be confusing for the user.

A non-final configuration, that is a configuration that still has some feature choices left to be made, is called a *partial configuration*. A *valid* partial configuration can be useful, as it narrows down the number of possible resulting variations and choices left to make.

C. *Satisfiability*

A feature model can be expressed as a propositional statement. The mapping between the individual groups (sub-trees) and the corresponding propositional expression is shown in the following list:

TODO: Rules for

- Or
- Alternative
- And

These particular groups are joined via a logical *AND*. For the example feature model shown in Figure 1 the corresponding propositional expression is as follows:

$$\begin{aligned} & GraphLibrary \ \land \ Edges \\ & \land ((Directed \ \land \neg Undirected) \\ & \lor(\neg Directed \ \land \ Undirected)) \end{aligned} \tag{1}$$

As the feature model not only contains the feature tree, but also additional constraints, these constraints have to be considered in the boolean expression representing the complete feature model. They can be linked to one another via logical *AND*s. The connection to the previously created propositional expression of the feature tree is also made through a logical *AND*. Thus, the complete resulting propositional expression for the feature model shown in Figure 1 is:

$$\begin{aligned} & GraphLibrary \ \land \ Edges \\ & \land ((Directed \ \land \neg Undirected) \\ & \lor(\neg Directed \ \land \ Undirected)) \\ & \land (Algorithms \ \Rightarrow \ (Number \ \lor \ Cycle)) \\ & \land (Cycle \ \Rightarrow \ Directed) \end{aligned} \tag{2}$$

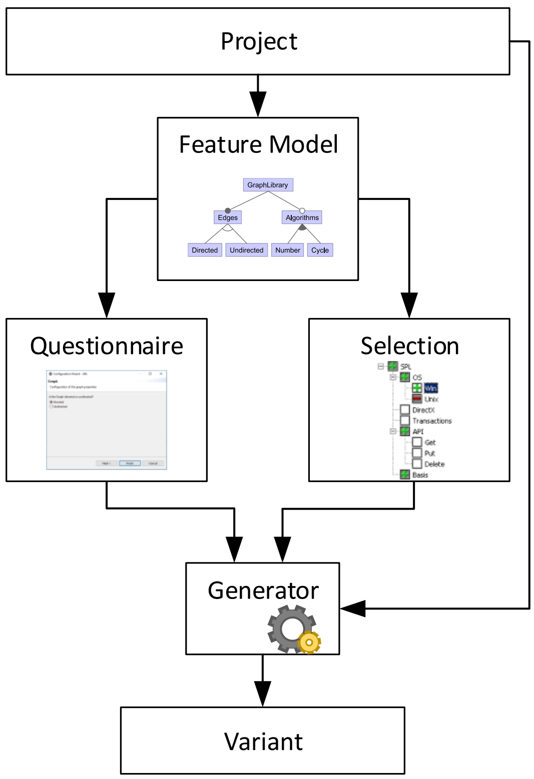

Fig. 2. Diagram showing the general workflow

This formalism allows a configuration to be checked for validity. To do so each selected feature is appended with a logical *AND* and each specifically unselected feature is also appended with a logical *AND* but gets negated. The resulting expression is then evaluated by a SAT-solver to check for satisfiability. Even during configuration this process can be applied to check for invalid partial configurations after each decision.

III. Workflow based on an existing software product line

This chapter describes the general workflow we aim to archive within this work. This workflow can be seen in Figure 2. To start off, some kind of software product line is required. It's source code has to be structured in a way that allows for an algorithm to recognize the individual features as well as their interrelations, especially the hierarchical dependencies.

The two major steps of extraction of the feature model and the creation and usage of the questionnaire are described in detail in the following sections.

A. Extraction of a Feature Model

Out of the given structure we extract the hierarchies of the features as well as additional dependencies. These are the input for the automated generation of a feature model. This allows for overview over the product line and it's variability. During our efforts to implement an automated generation of a feature model we encounter some problems for which we have to find solutions/workarounds.

The first problem consists of finding the structures of the given source code and processing them algorithmically. Most software complies to conventions regarding the naming and structuring of directories. We try to make use of these conventions. But as they are just conventions and not rules, in most source code one will find violations of the conventions. Instead of interrupting the whole process, we implement error handling. The result is a notification for the user to alert the developers about the error. Also, the remaining information, which aren't immediately affected by the error, are still extracted and used for the generation of the feature model, if possible.

Another challenge lies within the order of processing the features during the creation of the feature model. To correctly reproduce the hierarchy of the features, features have to be declared as child features of their parents. If the declared parent feature doesn't already exist, the creation of a feature model fails. So we have to ensure for each feature, that it gets processed after it's parent feature.

Another error we found is one ore more features having a parent feature which doesn't exist as a code artifact itself. Thus, the parent feature doesn't exists in the feature model and it's child features can't be placed in the hierarchy in the correct place. To still include these features in the feature model we create an abstract feature for the parent feature. The abstract feature is a direct child of the root feature. But as it doesn't have any describing code, it only has a name and no additional descriptive details.

The extracted feature model is also used in the following step of configuration. There, it visualizes the interrelations of the features and supports the understanding of the configuration steps.

B. Questionnaire Approach

Configuration is a challenging tasks within the scope of software product lines. The resulting feature model from the automated generation yields a better overview over the possible features and their interrelationships. Although the formalism of a feature model allows for tool support for the process of configuration, still domain knowledge is required to be able to find the right combination of features for a given use-case.

This work therefore introduces a method to allow experts to apply their knowledge and understanding to a whole product line during the development. Thus, users are enabled to draw on this knowledge whenever a configuration is taking place.

In this work, we made the decision to use a questionnaire-based approach. In the progress of implementation, experts also design a questionnaire. They do so in such a way, that a user has to answer a given amount of questions to perform the configuration of a product. Depending on the implementation of the questionnaire, a partial or even complete configuration can be archived by a user through just answering the questions of the questionnaire.

The selection of features gets lifted to a higher level of abstraction. The user only has to decide between the possible answers presented to him in the questionnaire to best fit to his use-case. Internally the selected answers are mapped to a specified (un-)selection of features so the user avoids the hassle of considering implementation-details of each feature. This effectively redesigns the process of configuration in such a way, that a user is independent of the knowledge about the details of the implementation and can focus on tailoring the product line to his specific use-case.

This highly depends on the implementation of the questionnaire during development. Our work therefore introduces a set of tools to easily integrate such a questionnaire. The following paragraphs will give an overview over the the possible definitions of a questionnaire.

Each page of the questionnaire is defined independently. It always contains at least a Question and more then one possible answer. The answers can be grouped analogous to the grouping of features in a feature model: *or* (at least one answer has to be selected), *alternative* (exactly one answer has to be selected) and *and* (any number of answers can be selected).

Each answer internally has a mapping to a set of features. This set of features defines which features are selected or specifically unselected in the case of that answer being chosen by the user.

Each answer can also have an indicator to define which page of the questionnaire is to be displayed next. An answer can also indicate the end of the questionnaire. If no next page is defined the questionnaire will continue with the next page within it's definition. This allows a dynamic conditional design of the questionnaire so the user is only confronted with the exact set of questions needed to configure the variant for his specific use-case. This also allows the user to skip questions or cancel the configuration before finishing it and thus creating a partial configuration.

We also introduce a data structure to hold the definition of a questionnaire. To archive easy integration we decided for a definition in XML. We defined the necessary tags to create a questionnaire which are displayed in the following code snippet:

Tag	attributes	child tags
configurationSurvey		projectName, section, page
section	id	name, description
page	id, sectionId	question, answers
answers	type	answer
answer	nextPageId	label, description, dependencies
dependencies		feature
feature	selection	

The individual tags are explained as follows:

- `configurationSurvey`: The root tag to contain all other tags for the questionnaire.

```
<configurationSurvey>
  <projectName>Name</projectName>
  <section id="0">
    <name>Section Name</name>
    <description>Section description</description>
  </section>
  <page id="0" sectionId="0">
    <question>Question for the user</question>
    <answers type="alternative">
      <answer nextPageId="1">
        <label>Answer label</label>
        <description>
          Answer description
        </description>
        <dependencies>
          <feature selection="false">
            Unselected feature
          </feature>
          <feature>selected feature</feature>
        </dependencies>
      </answer>
    </answers>
  </page>
</configurationSurvey>
```

Fig. 3. Examplary XML file for a questionnaire

- `section`: Enables grouping of question-pages. Also displays the name and description at the top of every page.
- `page`: Contains a question and the corresponding answers. Also has an indicator for a section
- `answers`: Contains the individual possible answers and groups them in the specified manner.
- `answer`: Defines the displayed text of an answer as well as the corresponding features. Can also have an indicator for the next page.
- `dependencies`: Defines the (un-)selection of features, if the corresponding answer gets selected.

IV. Examplary Scenario

State the process of FM-Extraction on the example of *Odoo*.

We applied these steps to a total of [A] files, thus extracting [B] features for *Odoo*. These features are arranged in a total of [C] hierarchical levels. Additionally, [D] constraints limit the possible valid configurations of *Odoo*.

Also state how some of the questions and their mapping in the questionnaire were developed.

Finally, show the process of configuration through running a fictinal user over the questionnaire and document his decisions as well as the resulting configurations.

V. Conclusion and Future Work

This is where the work is concluded. In this section there will be a description of the way we did things and the experiences we made during it. An emphasis will be on the insights and the findings from the scenarios will get outlined.

Here will be a summary of the new questions that were raised in this work. Also there will be topics for further research. Particularly the problems we encountered and couldn't solve with our concept and why will be pointed out and first approaches will be suggested.

REFERENCES

[1] C. K. G. S. Sven Apel, Don Batory, *Feature-Oriented Software Product Lines*. Heidelberg: Springer-Verlag Berlin, 2013.