Vrije
Universiteit
Brussel

FACULTY OF SCIENCE
Department of Computer Science
Web & Information Systems Engineering

Feature Models Visualization Based on Ontology Framework

Thesis submitted in fulfilment of the requirements for the degree of Master Computer Science.

Jose Evelio Martinez Saiz

Academic Year 2008-2009

Promotor : Prof. Dr. Olga De Troyer
Supervisor : Lamia Abo Zaid

Abstract

The goal of this project is the design and development of a tool for creating and visualizing software variability by means of visualizing the development of feature models. Allowing for different user perspectives and thus using different views for visualizing feature models. Visualization of feature models is not an easy task. The feature models is something more than hierarchy of concepts due to the enrichment of role relations between features and the various attributes related to each concept. For these reasons, it is not simple to create a visualization that will display effectively all this information and will, at the same time, allow the user to perform easily various operations on the feature model.

All the information presented in this document is a research and study of the actual visualization techniques that can be applied in the field of feature modeling visualization. In addition, we studied some existing tools that allow the user to visualize and modify feature models; this gave us overview about the state of the current situation in feature modeling visualization.

To sum up, this deep analysis is expanded with all the information related on the design of a feature modeling tool and based on the decisions taken during the design of our application. There does not exist a standard feature model technique, so this study is approached from a global point of view based on the basic feature model requirements.

Declaration

I declare that this document and the accompanying code have been composed by myself, and describe my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed,

Acknowledgments

I would first like to thank my project supervisor, Lamia Abo Zaid, for her support, advice and help during the entire project. Thanks as well to her for proof reading parts of this thesis and correcting some of the grammatical mistakes.

Thanks also to the head of the WISE Laboratory, Prof. Dr. Olga De Troyer, my promoter, for her kindness, her willingness to help me all the time, and allowing me to collaborate in the research of the VariBru project.

I would also like to thank my flatmates Marko Rezonja and Brett Terrell for giving me their point of view in some details of this thesis, for being by my side in tough times and for being my friends.

Finally, I need to thank all my friends here, specially Andrea, Gianluca, Mari Carmen, Paola, Rossella, Michela, Ole, Oliver, Susana, Sara, Tamara, Ann-Sofie, Laura … that have been my family in Brussels during this four month., But mostly, I want to express all my gratitude to my mother and my sister, for their infinite love and for giving me the opportunity to experience this great and unforgettable experience, the Erasmus experience...

Jose Evelio Martinez Saiz

Content

List of figures

List of tables

Chapter 1

Introduction

1.1. Variable Software Modeling

Feature modeling [1] is a popular domain analysis technique, which analyzes commonality and variability in a domain to develop highly reusable core assets for a product line. . A software product line is defined as a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [2]. The development of software product lines emerged from the field of software reuse (i.e. the process of creating software systems from predefined software components) [3]. This arise was caused because the more obtaining benefits from reusing software architectures (abstract concepts of elements) instead of reusing individual software components (elements). Feature modeling techniques supports requirements analysis and domain engineering in software product lines.

The need of this technique lies in the fact that the number of possible configurations quickly grows as the variability increases, resulting in an increasing complexity. Therefore organizations need support for the development of these kinds of products. The existence of many articles related to this technique gives us a global idea about the importance of the technique for modeling software product lines. However, there are only a few tools supporting variability modeling. Although feature models have been around since the

1990's, visualizing feature modeling using GUI tools is quite recent. In this thesis, we describe the design and implementation of a GUI tool for creating and maintaining feature models in a visual way.

1.2. Motivation

The result of the applying the feature modeling technique to model a certain variability problem is a feature model. Feature models [4] are crucial for the development of variable software, correct feature models lead to correct products. Thus there is a need for tools that allow specifying, verifying, and correcting feature models. The best way to support the user is by providing a tool that allows to create and to manipulate feature models in a visual way. However, the visualization of feature models is not an easy task. In industrial cases, the number of features can be very large and then it becomes difficult to provide a good visualization of such a model. The more features are needed in the visualization of the model, the more complicated the drawings will become and the more difficult it will be to understand and to manipulate the content of the model. In addition to the large amount of features, the visualization should also support the specification and visualization of all necessarily details (attributes, feature dependencies, feature and/or/alternative hierarchies), so the problem becomes more complicated because showing all this information together may result in diagrams that are too complex to understand and manipulate and therefore unusable.

In this project our objective was to identify, from a HCI perspective, the best method to visualize feature models and interact with them. In addition, for the sake of interoperability and ease of use in a distributed setting it was required to develop such a tool using Rich Internet Application (RIA) technology [5]

1.3. Problem statement

This work relates both feature modeling techniques and information visualization techniques. We try to provide the best possible view(s) that visualize feature models and thus ease the modeling process. Visualizing and a representing feature models in industrial models, where the number of components included sometimes becomes very large, which is not an easy task. Therefore we foresee visualization scalability as an important issue for any good feature modeling visualization tool.

As commented previously (see *section 1.2.*), the number of components in our model tends to grow, so the need to achieve a good representation model for visualizing clearly the global contents of the model rises. Furthermore, the tool should allow for flexible user interaction the underlying model.

The visualization of the feature model also must match the needs required for the stakeholders. Due to the fact that a concept can be represented by different valid models; another problem confronting modelers is to decide which of them is better for satisfying their requirements. In addition, sometimes a good way to represent the model uses the combination of various visualizations. This fact increases the visualization problem in terms of complexity because of the necessity to extend the study with all the possible combinations of visualization views providing the synchronization between them.

1.4. Objectives

The main goal of this project is to create an effective and useful application that solves the problems indicated previously, running in a browser as a RIA and with a good response time. In particular, two specific objectives can be defined for the tool:

Usable in that with this application (tool) the user can create, modify, save and load feature models in a visual way. The creation means the addition of features, the links between

features (hierarchical links and feature dependencies), and the addition of attributes for features.

Efficient in that with the visualization of the model the user can understand in an easy manner all available information. Also the interaction between the user and the different components of the model should be supported by the visualization.

1.5. Overview

The remainder of the document is distributed in five chapters as follows:

- *Chapter 2: Background and related work,* provides the information about what is a feature model, and the components related to it. The chapter also includes a report about the research of different tools in feature modeling and other visualization tools such as ontology visualization.
- *Chapter 3: Visualization and interaction of feature models*, shows the requirements related to the design of our feature modeling tool. In addition, the chapter includes an extensive explanation about the design of the tool and justification of the decisions taken during this process.
- *Chapter 4: Tool Design*, includes the architecture of the system, the Uses Cases of the tool and its UML structure.
- *Chapter 5: Implementation,* gives an overview of the technology used for the implementation and provides the tool design structure.
- *Chapter 6: Discussion and conclusion*, in this section we discuss the lessons learned from this project and give some recommendations for future work.

Chapter 2

Background and related work

2.1. Feature Models

In a global context, we can consider a feature model as an approach for capturing variable properties and functionalities of concepts and commonalties in software systems and product lines. The function of the feature model is to express the requirements regarding the variability using a high-level of abstraction.

Variability is an abstract concept difficult to explain theoretically. We can imagine a factory that creates products with certain aspects that can be variable. These products with variable aspects are called product lines [6]. By changing some of these aspects in a product, we obtain a different *product variant*. In case the factory needs to offer specializations of the products to achieve certain requirements at the end of its development, the need of variability study arises. Each aspect of the product that can vary is called as variation point [6] of the product. So, the usefulness of variability is when configuring this *variation point*, the product line ends as a different *product variant*. For example, consider a car product line in the case of automobile companies. If different cars are produced with different door types and numbers as an example (e.g. *2-Door Coupe, 4-Door Sedan* …) then we call the door a *variation point* and each different type of door is called a *variant.* We can obtain different products (cars) from the configuration of this *variation point.*

Continuing with the theme, a feature model tries to represent the variability requirements with certain abstraction. The interdependencies between the common and variable properties (called features) and the organization of them into a coherent model are one of the key points to structure in a feature model. If requirements and solutions are isolated structures, then the modeling can be a bridge which connects these structures together. Depending on the accuracy and precision of the feature definition, the model can become more robust. Because of this, also the description of the relations between the features becomes more precise in the model.

More formally, we can say that a feature is a common intention of a concept or a general idea derived or inferred from a specific product line [7]; that is as a standard definition, a distinguishing characteristic of a software item (e.g., performance, portability, or functionality) [8]. The relations between these instances give the meaning of the global concept, and the properties of these instances add information to the elements.

Feature models are usually represented in a graphical way by means of a tree like diagram.

2.1.1. Components of a Feature Model

To understand a feature model it is essential to comprehend the components that form a feature model. The basic structure of feature models is a tree representation, where primitive nodes are leaves linked with compound interior nodes [9]. The representation of these links varies depending on the notation that the modeler uses. In the next subsections, the descriptions of the three basic components used in the construction of a Feature Model are described: features, attributes and feature dependencies.

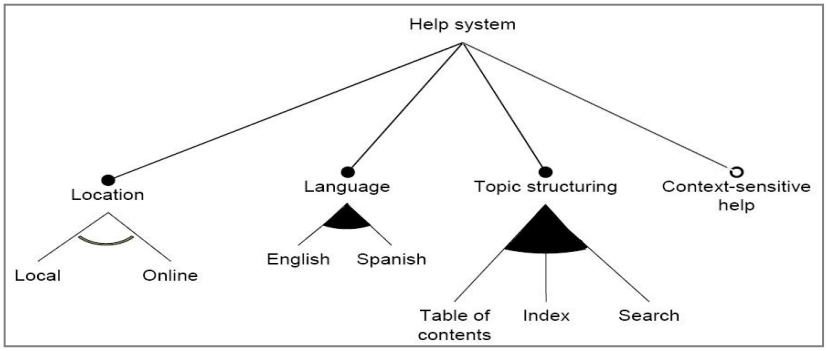


Figure 1. Example of feature model [10].

2.1.2. Features

Each concept of the model represents a feature. It is a concept that gives a part of the information of the complete model and it is considered as one of the basic components of a feature model. Features are involved in a net of connections or arcs that represents relations between the different features and gives the global meaning of the model. Some of the connections represent the composition hierarchy of the model which provides information about the specialization of a product-line [9]. From the root, a reader of the model gets the general product. Then, when starting to explore descending to the next level of nodes in the tree, more details of the product components and functions and variations are explored.

Figure 1 is an example of a feature model; the root 'Help System' is a generalized description of the required product, descending in the component hierarchy of the model, and arriving to the leaves of the tree (each composition of constraints is represented as a node) more specific descriptions of the product are shown. Thus, the root of the tree, as the name indicates, is a more global concept than the interior features.

In addition, the sub-features get a type depending on their type of involvement in the hierarchy. Every feature can have only one hierarchy relation that gives the type. In our case we consider four types of features, which are depending of the hierarchy: mandatory, optional, alternative and or. In the majority of the feature model representations, the

hierarchy relation used to be represented using the notation Czarnecki-Eisenecker [11] (see *Figure 2*).

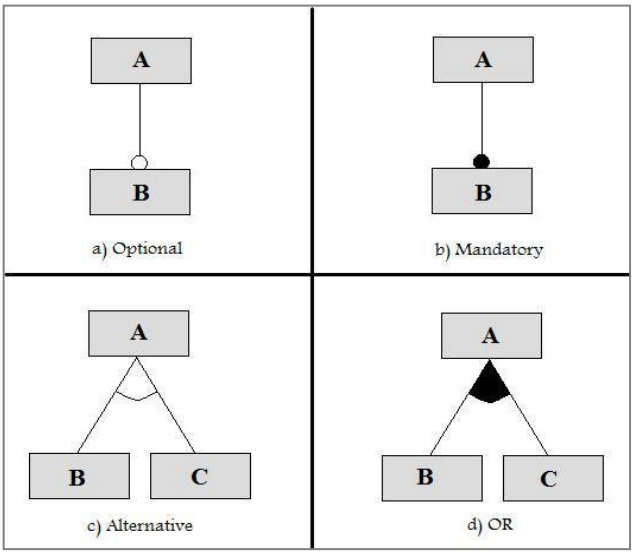


Figure 2. Czarnecki-Eisenecker notation [11].

An *optional feature* is the specialization of the feature that may or may not be chosen in the final product when its parent is included in the final product. In the graphical representation, this type of features is represented by a simple edge from the parent feature optionally ending with an empty-fill circle (see *Figure 2*, *a*).

In the second type, the *mandatory feature*, the specialization of the feature must be included in the final product if its parent feature is included. The no inclusion of the mandatory feature in the final product implies also that its parent feature is not included. In the graphical representation, this type of feature is represented by a simple edge from the parent feature ending with a black-fill circle (see *Figure 2*, *b*).

The *alternative feature* only gets reason of existence if it is included in a set of alternative options. This means that this set of components represent the different alternatives of their parent concept (every feature apart of the group has no meaning). In case the parent of the set is included in the product definition; then exactly one of the features from the alternative group is included as a specialization of the parent's concept. In the graphical

representation, each set of alternative features are representing by an arc that encompasses all the edges that connects the parent feature with the different features (see *Figure 2, c*).

Finally the *or-feature* is meaningful only if the component is grouped in a set of *or-features*. If the parent feature is included in the final product definition, then any non-empty subset from the or-group is also included in the product. In the graphical representation, each set of *or-features* is representing by edges that connects the parent feature with the or-features and a black-fill arc that encompasses all these connections (see *Figure 2, d*).

Features used to have characteristics. These characteristics give extra information and are called attributes, so each feature may contain a list of attributes (or no attributes).

2.1.3. Attributes

Attribute is any characteristic related to features that adds information about the description of the feature and can be measured [12].

There are many ways to classify the attributes in types depending on the needs of the model. In our case, the modeler can distinguish between three types of attributes depending of its functionality and meaning:

- The first type, called *quality attribute* [13], is the inherent or distinguishing characteristic which is measurable in an abstract manner. In addition, the modeler has to provide rules for measuring this type of attribute (e.g. security, usability …).
- The second type, *quantity attribute*, is the characteristic that is measurable mathematically (e.g. weight, age …).
- The *extra-functional attribute* [12] of a feature is the characteristic defined by the relation of one or more attributes related to this feature (e.g. ‘age = 24’, ‘height/width > 0.5’ …).

- To finalize, the *external attribute* is the characteristic that gets its value from an external application or device.

2.1.4. Feature Dependencies

In addition of the connections that represent the composition hierarchy of the model, it may be possible to add additional relations between features to increase the global meaning of the model. In fact, these connections can also be considered as attributes of features, because they also add additional information to a feature. However, as they involve more than one feature they are expressed explicitly between features by means of dependencies.

There are a lot of possible types of feature dependencies, in principle as many as the user needs. Each relation has got a meaning to use, so when a feature is linked with another feature (is not allow to link the same feature) the global context of the model is modified and gets a new vision. The following table gives the explanation of the types of dependencies considered in our research.

Dependency name	**Meaning**
Excludes	Feature1 excludes Feature2 when Feature1 and Feature2 cannot occur together. *Ex.* "Raining day" *[excludes]* "Sunny day".
Extends	Feature1 extends Feature2 if Feature1 adds to the functionality of Feature2. *Ex.* "Full registration" *[extends]* "Simple registration".
Includes	Feature1 includes Feature2 if Feature1 contains Feature2. *Ex.* "Add username" *[includes]* "Check user name exists".
Incompatible	When Feature1 is mutual exclusive due to a conflict with Feature2, it is considered that Feature1 is incompatible with Feature2. *Ex*. "Advanced graphics" *[incompatible with]* "Basic graphic controller"
Requires	Feature1 requires Feature2 when Feature1 is functionally dependent on Feature2. *Ex*. "Advanced editor" *[requires]* "Spelling checker".
Uses	Feature1 uses Feature2 then there is a dependency relation, so logically if Feature1 is required then Feature2 should also be required. *Ex*. "Search" *[uses]* "Provide hints".
Same	Constraint used to indicate that two features are the same. *Ex*. "Advanced graphics" *[same]* "AG"

Table 1. Feature dependencies description [14].

Two features cannot be given the same type of dependency between them, because of the strict meaning of each role. On the other hand, two features can get different type of dependency, for example: “Add username" *[includes]* "Check user name exists” / “Add username" *[uses]* "Check user name exists”.

2.2. Related Visualization Tools

There exist some tools that allow the manipulation and visualization of feature models. The purpose of this section is to present and study some techniques of visualization based on the tools studied. In addition, these tools are analyzed with the requirement to achieve an effective view model in mind. The goal of this study was to gain insight on the visualization problem and to obtain some useful ideas for the design of our tool.

The analysis presented in the next sections is based on how the visualization of the different components is done and how the user can manipulate these components. Speaking about components, we mean speaking about features, dependencies or attributes.

Therefore, the focus of the study is on the visualization and manipulation of features, attributes, dependencies and hierarchy, as these are the basic components of a feature model. In *Table 2*, there is an overview of the relevant details to study about the tools.

Component	**Important details to analyse**
Feature	Representation of the feature in the visualization, interaction with the user.
Attribute	Is it visualized in the model? How is it visualized?
Dependency	Is it visualized in the model? Differentiation between the different types.
Hierarchy	Is it visualized in the model? Differentiation between the different types.

Table 2. Focus of the analysis

2.2.1. Existing tools of Feature Model visualization

The next sections present some tools according to different visualization categories. In fact, the presentations of the characteristics related to each tool are a deep study of different visualization categories. For example, the research the Feature Modeling Tool [15] [16] is an overview of the combination of the tree representation and the indented list. Other tools presented are Feature Modeling Plug-in for Eclipse [17] [18], Pure::variants [19] [20] or XFeature [21].

2.2.1.1. Feature Modeling Tool

Feature Modeling Tool [15] [16] allows creating feature models from inside Visual Studio IDE. The representation of feature models is based in two visualizations: the indented list and the tree structure. In the left side of the window (see *Figure 3*) is presented the feature model's hierarchy as an indented list where the nodes represent the features.

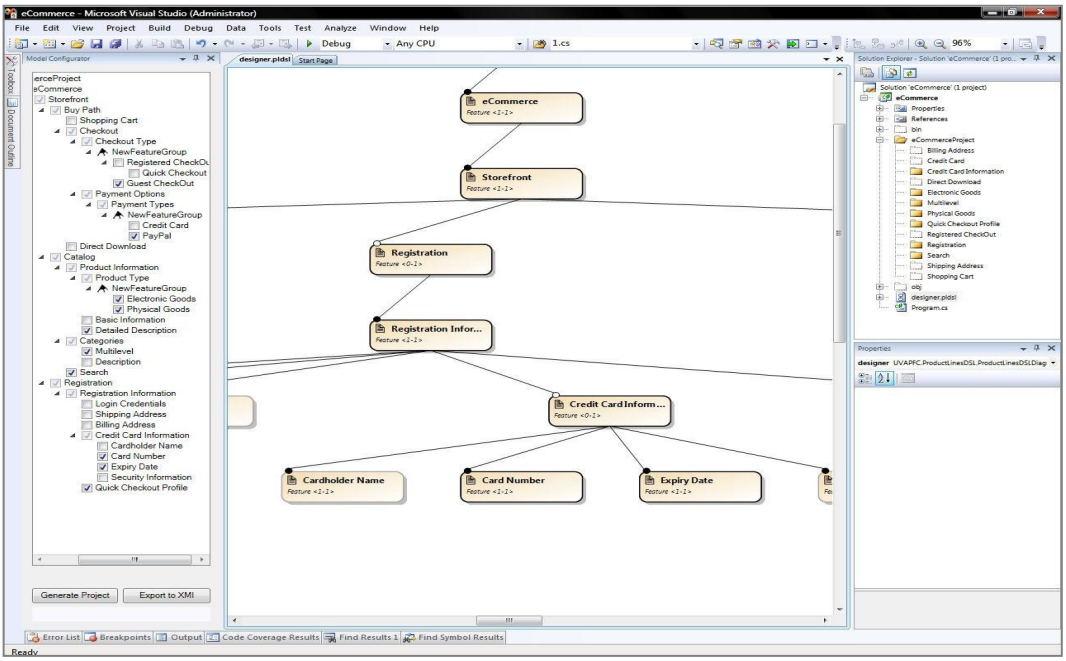


Figure 3. Feature Model Tool main window [15]

The modeler can recognize the hierarchy composition from this representation and also distinguish or-groups and alternative-groups due to the use of a new node in the list that contains as child the features of the related group. However, the introduction of a new node that represents the existence of a group of features can be confused with the actual features when comprehending the hierarchy composition (it also means adding an extra level to the indented list).

The central window represents the modeler's design where it is allowed to add/modify/delete features. The representation is based on a tree, where the links symbolize the hierarchy component and the nodes symbolize features. All the changes produced on this representation will propagate automatically to the other views.

One inconvenience of this view relates on the non-visualization of the feature dependencies, so the main functionality of the representation lies in the basis to show the information in a hierarchical form, as well as in the indented list. One of the tools disadvantages is the excessive use of two similar views that represent the information in a hierarchical manner. The tool could have placed more emphasis on representations that show other information.

In addition, this tool can be considered more a feature configuration tool than a feature modeling tool. The leave nodes of the indented list provide a check box to select different combinations of features with the aim of obtaining different product lines. This means that the design for representing the model is based on other requirements than in our case (we focus more on providing aid for creation of feature models). Therefore, it is assumed that this type of representation probably is not the most appropriate solution for our problem.

2.2.1.2. Pure::variants

Pure::variants [19] [20] is a feature modeling tool created by pure-systems GmbH setup in 2001 as a spin-off from the Otto-von-Guericke-Universität Magdeburg and the

Fraunhofer Instituts Rechnerarchitektur und Softwaretechnik. The application is based on Eclipse, an open source community whose projects are focused on building an open development platform comprised of extensible frameworks [22], and the main functionality of it is to be used as a framework for the design of product line architectures.

The modeling development on the tool is based on hierarchical structures, consisting of items related to the different components (feature – attribute - dependency) those make up the final model. These logical items can be augmented based on the user needs and preferences.

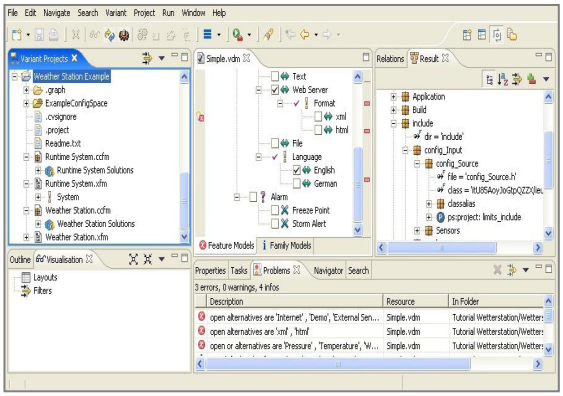


Figure 4. Pure::variants Family Model Editing [20]

Similar to one of the representations of the previous tool, the items are situated as nodes in an indented list. Each check-box placed near each component is used to configure a product line from the feature model. Thus, the user is allowed to display a final result of a product line, if he or she selects some configuration by the use of these check-boxes (see *Figure 4*).

Pure::variants adds the possibility to represent the model by graph visualizations [19]. Although some common editing operations (editing - deletion) are supported by the tool, this view is primary intended for understanding the model and printing the solution.

The model presents the different representation of features by boxes containing the name of the feature and an associated icon for differentiating features by their type. On the other hand, the hierarchical component representation uses arrows to make the parent-child links.

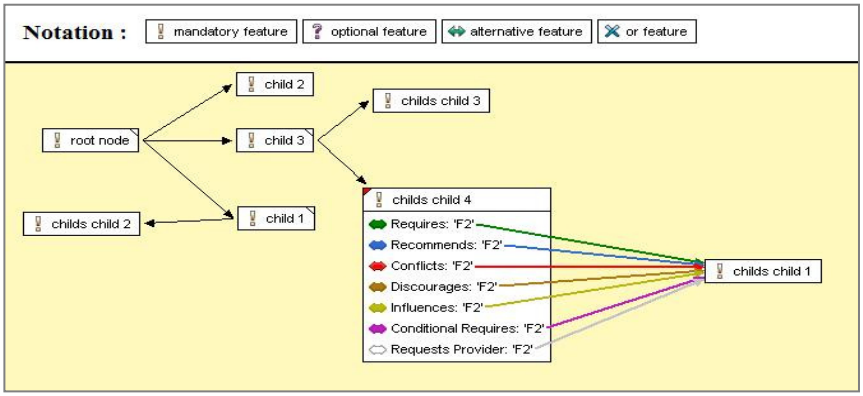


Figure 5. Pure::variants graph visualization

Although the representation seems to be poor in content, the graph visualization adds an interesting functionality to visualize the feature dependencies. The boxes can be expanded to visualize the dependencies using colored connection lines between the related elements, where the color of the connection line depends on the relation (see *Figure 5*).

The possibility to visualize these dependencies gives the model more information meaning better overall understanding of the representation. On the other hand, the model could become unintelligible if there are many expanded nodes, because of this big representation.

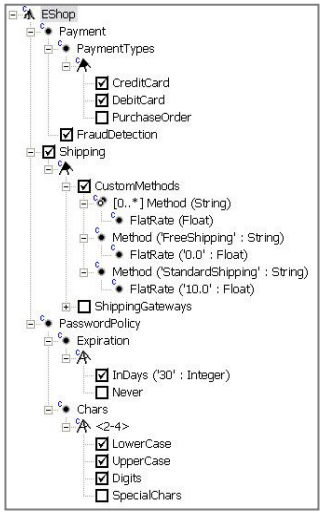


Figure 6. Feature Modelling plug-in for Eclipse [18].

2.2.1.3. Feature Modeling Plug-in for Eclipse

Feature Modeling Plug-in [17] [18] is an Eclipse [22] plug-in that represents feature models based on an indented list. The design of the tool is very similar with the above tool (Feature Modeling Tool [15] [16], *Chapter 2.2.1.1*). The nodes of the visualization represent the features; also the or-groups and alternative-groups are placed under a new node that informs about the type of the grouping.

When the user clicks on a feature, an auxiliary window shows the information about the node (attributes, description ... etc). In addition, the feature

dependencies are not presented in the model: there is another window that informs about their existence and their information.

As with the Feature Modeling Tool [15] [16] (see *section 2.2.1.1.*), the purpose of this tool is to configure different product lines from the representation of the model. Therefore, once more it is assumed that this representation probably is not the most appropriate solution for our problem.

2.2.1.4. XFeature

XFeature [21] is another plug-in for the Eclipse platform [22], tool which supports feature modeling of product families. Ondrej Rohlik and Alessandro Pasetti from *P&P Software GmbH* and the *Automatic Control Laboratory* of ETH-Zürich designed and developed the XFeature tool to demonstrate a concept for automating the modelling and configuration process of reusable software assets in a tool.

The visualization of the feature model in the tool is based on a tree-structure, where the nodes represent the features and the links the hierarchy composition (see *Figure 7*).

The modeller has to indicate the cardinality of the relations using the auxiliary points placed on the origin of the links. Therefore, depending on this cardinality the modeler can get the type of the feature; for example, the optional features are linked with the cardinality <0...1> and mandatory with <1…1>. In addition, these auxiliary points represent the grouping of features (alternative - or) when the origin point of the links are shared.

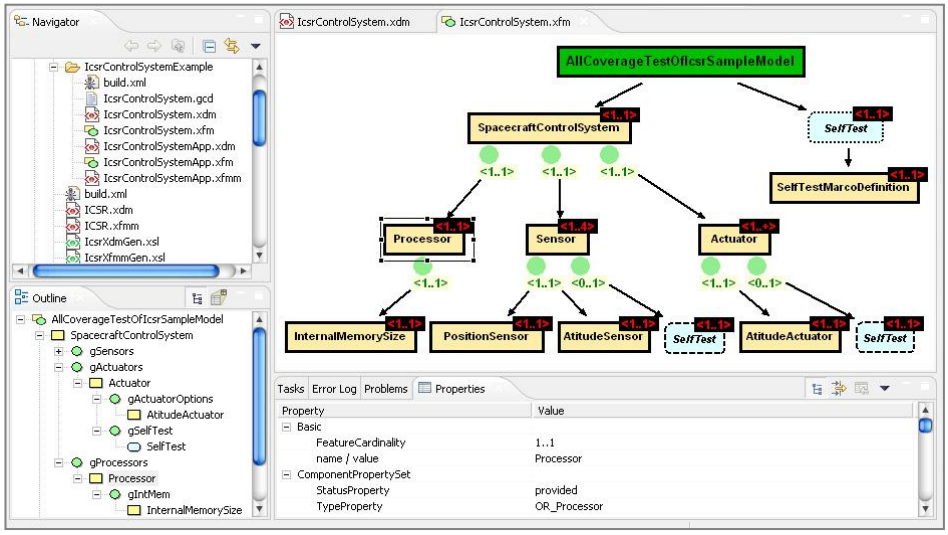


Figure 7. Screenshot of XFeature tool [21].

The notation based in cardinalities is a good solution for the representation because the user gets the type of the feature and its group from the same component (the ellipse). On the other hand, at first glance it is difficult to distinguish the different types from the representation; the modeler has to relate the cardinality with the type, imposing sometimes a big mental effort for the user.

To finalize, the existence of attributes are represented as a red-fill ellipses placed near the features. Thus, the modeler can recognize in the representation that a feature has related attributes from the situation of this ellipse.

2.2.1.5. FaMa Tool Suite

FaMa Tool Suite (FaMaTS) is a tool for the automated analysis of variability models [23]. The application provides an extensible framework for easily reading variability models, and automating (also used in a semi automated way) the configuration of a final product.

As the majority of the feature modelling applications, FaMa Tool Suite uses GUI tree (tree structure in indented list) as a representation of the model. The difference in this case lies in the process of modeling; the user has to develop the structure of the feature model writing it in an XML[1]. Then, the tools read the document and visualize the content of it like in the *Figure 8*, allowing the user to interact with the representation.

The nodes represent the features, relations, and cardinalities of relations in the model. In this case, the modeller obtains the type of the feature by the node cardinality that each feature contains as a description. In addition, all the features that form a group are contained inside the relation nodes. Furthermore, the hierarchy component is based on the indentions of the list.

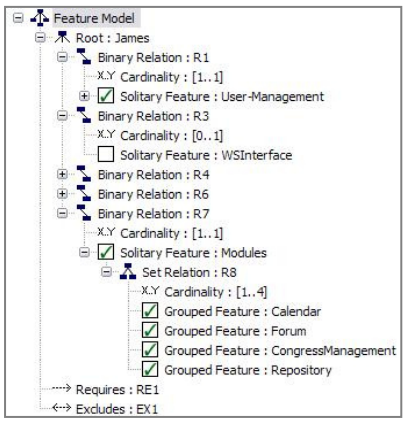


Figure 8. FaMa Tool Suite [24]

2.2.2. Existing tools for Ontology visualization

The motivation of researching ontology visualization tools is due to the existence of a big quantity of them and the many similarities between this type of tools and the feature modeling tools [25].

A commonly accepted definition done by Gruber [26] of ontology is the "explicit specification of a conceptualization". Ontology visualization tries to represent the semantics of concepts and the relationships between them using a descriptive notation. As commented previously, a basic feature model is also a concept description technique [25] where its representation is quite similar to that of ontologies (features linked by dependencies and parent-child like relations). Therefore, the study of the following tools is useful for our purpose, because ideas implemented in ontology visualization gives insight

[1] The Extensible Markup Language (XML) is a general-purpose *specification* for creating custom markup languages.

on how to represent rich complicated information structures which also apply to the case of representing feature models.

In this case, the study is done from the feature model point of view; i.e. the components of ontology can be used to represent the components of a feature model. Thus, instead of referring to classes we will refer to features. Also attributes and hierarchy are considered instead of properties and taxonomy classifications respectively.

As in the research of feature modeling tools, the study is based on different visualization categories. For example, the research in Protégé [27] is an overview of the indented list model representation characteristics. Other tools presented are GoSurfer [28], IsaViz [29] and OntoViz [30], among others.

2.2.2.1. Protégé

Protégé [27] is an open-source platform useful for the construction of domain models and knowledge-based applications with ontologies. Protégé implements a rich set of structures and actions that support the visualization of ontologies in various representation formats.

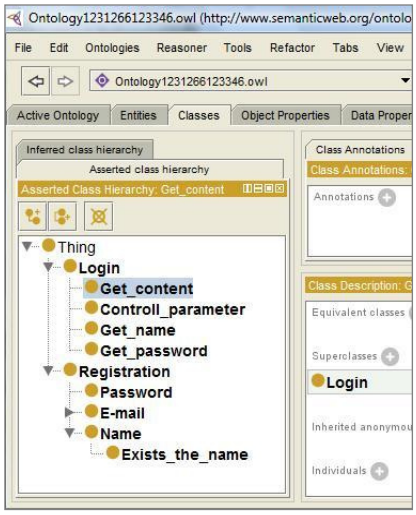


Figure 9. The Protégé feature/class browser

One of the structural techniques that Protégé uses for the visualization of ontologies is based on the indented list. Protégé window allows the user to explore a tree-view representation of the features in the model. Each node of the tree represents a feature which can be expandable showing the rest of the lower features under the hierarchy, or retractable hiding it (see *Figure 9*).

In this case, the comprehension to understand the structure of the hierarchy is easy for the user. The child features are placed under their parents and indented to the right; so if the user expands all the nodes from the model, the representation becomes a complete view of the feature hierarchy.

Corresponding to the other components in the model, the visualization of attributes is displayed in a separate window. For the user, this may be inconvenient due to the difficulty required to match each attribute with its feature. In addition, there exists no visualization of the dependencies; Protégé only supports the addition and modification of them through the properties window.

2.2.2.2. OntoViz

The second tool analyzed, OntoViz, [30] is a popular Protégé visualization plug-in with support of GraphViz [31] library. In this case, the ontology is presented as 2-D graph visualization where the nodes represent the features. In addition to the name of the feature, each node contains inside its attributes and its dependencies using labeled links.

The user is allowed to select which features will be displayed, as well as prune some parts of the ontology from the panel situated in the left side. When the user clicks the right button of the mouse on the view, OntoViz show an auxiliary window for zooming-in and zooming-out the content of the visualization.

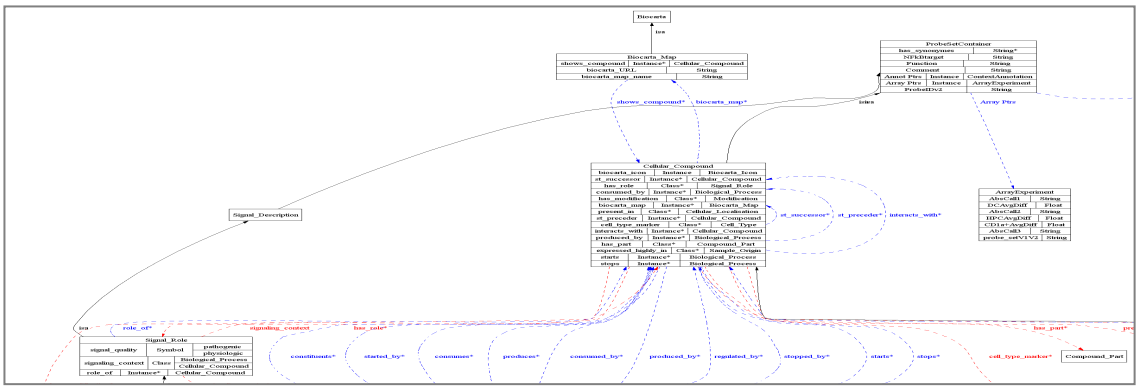


Figure 10. OntoViz plug-in for Protégé [30]

The inconvenience of this type of representation arrives from the amount of data that the tool displays. The visualization sometimes becomes hard to manipulate when the underlying visualized information exceeds a certain limit.

In addition, when one feature contains several attributes the size of its node becomes huge while other features without attributes maintain the same size. This sometimes generates problems to understand the model. It is easier to distinguish some features from others in the representation. Also, the model provides to a certain level of differentiation in number of attributes between the features, when it should not occur.

Besides, the use of the zoom is required to visualize the smaller nodes because of the irregularity in the size of the features. This means that from some perspectives the user is allowed to recognize some part of the nodes that compose the model and for the rest is necessary the use of the zoom.

2.2.2.3. GoSurfer

GoSurfer [28] is a data mining tool for visualizing GO [32] data, i.e. sequences of *genome-wide computations*. Although GoSurfer is not a common tool to represent ontologies, it has a characteristic way to show the content of the input data. In this case, the

particularity lies in the way showing the large number of nodes using a top down tree (see *Figure 11*).

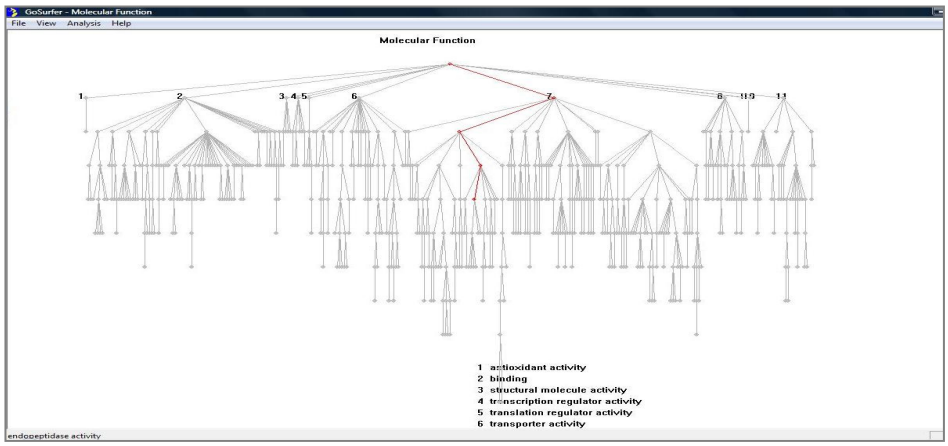


Figure 11. GoSurfer: Molecular Function representation [30]

The highlight of this representation is that the user can distinguish without problems the hierarchy of the model, even with a big amount of nodes. The key is to situate the nodes according with the level of each one in the hierarchy, presenting a precise structured tree model.

On the other hand, sometimes this representation can produce anti-aesthetic trees, e.g. if in a big hierarchy the majority of nodes belong to the same level the tree becomes small and very wide.

2.2.2.4. IsaViz

IsaViz [27] is a visual environment for browsing and authoring RDF ontologies. A common representation of ontology in IsaViz is a set of labeled ellipses representing the ontology concepts, a set of links representing the hierarchy, another set of labeled links representing the role relations, and a set of rectangles linked to the nodes (ellipses/features) representing the attributes.

In IsaViz there also exists the problem of the large amount of data shown on the visualization. If some features have several attributes the graph becomes huge containing many elements and connections (one rectangle and link for every new attribute). For example, if a feature has got four attributes, two hierarchical links and two feature dependencies, it means that from this node emerges a total of an eight links to eight new features (in case there are no features used twice for different relations).

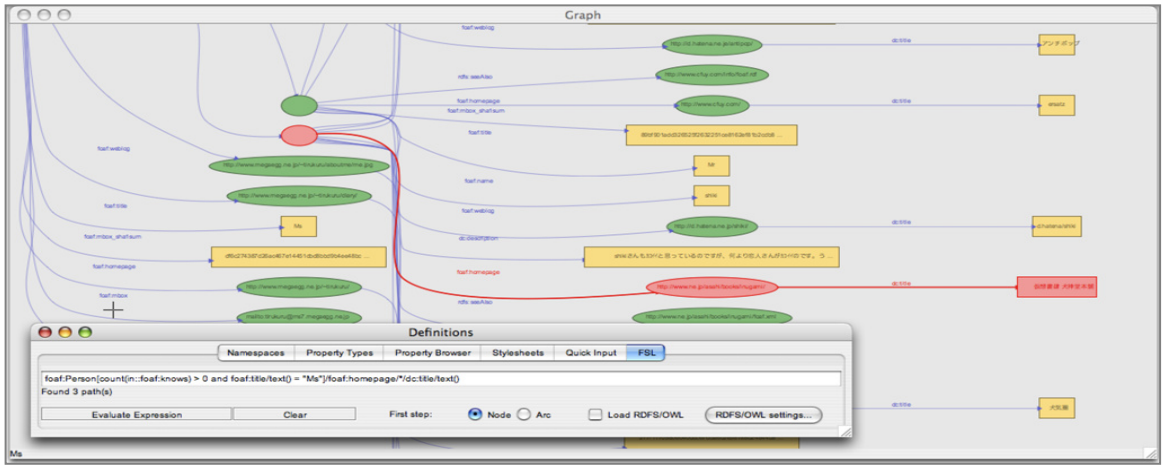


Figure 12. IsaViz environment [27]

2.2.3. Conclusion of the study

Studying related tools was the starting point of the design of our tool. From the conclusions (more details later) of this work, we already get some first ideas on how to design our tool. On the other hand, the information obtained in this way is quite limited, and a more theoretical study in the area of information visualization was considered useful. This will be given in *section 2.3* and *2.4*.

In this section, we have looked at different feature modeling visualization tools to understand the existing support given to software modelers developing variable software and thus help us identify the shortcomings of these tools from a HCI[2] prospective. *Table 3*

[2] Human–computer interaction (HCI) is the study of interaction between people (users) and computers.

summarizes the major characteristics of the existing feature modelling tools. From the study of this table, we can extract some notes regarding the representation of the feature models in existing feature modeling tools:

- It seems that the common visualization view to represent feature models is the indented list. Different components and information of the feature model can easily be represented within the list, although this situation sometimes causes large lists. This leads to the loss of the hierarchical structure of the information and makes it difficult for the user to follow.

- Most of the existing feature modeling tools are optimized for configuration of feature models rather than design of feature models. Thus the indented list view usually contains check boxes for configuring different product lines from the model. This is one of the reasons that this representation includes all the information. The modeler is allowed to select or deselect all the possible information for the good configuration of product lines.

- Some of the existing feature modeling tool tend to have more than one method for visualizing the model, while the indented list method is similar in all the tools. Some tools contain other visualizations which hold additional information that was not possible to represent in an indented list. So generally, each tool will require a different type of supporting representation depending on the needs of the users and the level of support it provides for visually modeling variability represented by feature models.

From our point of view, the above tools are optimized for configuration of feature models rather than feature modeling design. For example, is not essential to use an indented list view because a feature model design tool not provides the possibility to configure the product line (it maybe included on a different view for example a configuration view). Even, the inclusion of all the components in the indented list seems excessive and difficult to understand.

Tool	Method	Features				Hierarchical component				Dependencies				Attributes			
		T	L	G	Comments	T	L	G	Comments	T	L	G	Comments	T	L	G	Comments
 Feature Modeling Tool	Indented List + Tree representation.	✔	✔	✖	Represented as an item in the list, and as a node in the tree.	✔	✔	✖	List: The child nodes are placed indented to the right from the parent. While the grouping of features is represented as a new item indicating the group-type, which also includes the features indented to the right. Tree: Represented by links using the common notation.	✖	✔	✖	List: Represented as an item placed inside the feature representation. Tree: No representation.	✖	✔	✖	List: Represented as an item placed inside the feature representation. Tree: No representation.
 Pure variants	Indented List (main window) + Graph representation (auxiliary window).	✖	✔	✔	List: Represented as an item. Graph: Represented as a box. Icon inside it indicating the type.	✖	✔	✔	List: The child nodes are placed indented to the right from the parent. No representation of grouping. Graph: Black-stroke arrows linking the different features. No differentiation between different groups.	✖	✔	✔	List: Placed as an item inside the feature representation. Graph: The nodes can be expanded showing colored links representing the dependencies.	✖	✔	✖	List: Placed as an item inside the feature representation. Graph: No representation.
 Feature Modeling Plug-in for Eclipse	Indented List.	✖	✔	✖	Represented as an item in the list.	✖	✔	✖	The child nodes are placed indented to the right from the father. While the grouping of features is represented as a new item indicating the group-type, which also includes the features indented to the right.	✖	✔	✖	Represented as an item placed inside the feature-item related.	✖	✔	✖	Represented as an item placed inside the feature-item related.
 XFeature	Tree representation.	✔	✖	✖	Represented as a node in the tree.	✔	✖	✖	Black-stroke arrows to represent the hierarchical component. The grouping of features is represented from an origin point where the links of the group emerge. Uses cardinality to give the modeler the information about the existence of groups and the type of features.	✖	✖	✖	No representation.	✔	✖	✖	Red-fill ellipse situated near the box that represents the related feature.
 FAMA	Indented List.	✔	✖	✖	Represented as a node in the list.	✔	✖	✖	The child nodes are placed indented to the right from their parent.. Features form the same group is placed inside the same node relation.	✔	✖	✖	Represented as a node.	✖	✖	✖	No representation.

Table 3. Representation of the components in feature modeling tools
(T: tree representation, L: indented list, G: graph representation)

We also studied other visualization of data in different domains such as ontologies. Ontology modeling tools also have to deal with massive amount of data structured in a rich hierarchical manner. *Table 4* provides the representation of the components in of ontology modeling tools; the aim of this study was to gain some experiences in how other domains dealt with the scalability issue when visualizing knowledge.

Tool	Method	Features				Hierarchical component				Dependencies				Attributes			
		T	L	G	Comments	T	L	G	Comments	T	L	G	Comments	T	L	G	Comments
Protégé	Indented List.	✖	✔	✖	Presented as nodes in an indented, expandable and retractable tree.	✖	✔	✖	Child nodes are placed under their parent and indented to the right	✖	✖	✖	No representation.	✖	✖	✖	No representation. Displayed in a separate window.
OntoViz	Tree representation.	✔	✖	✖	Represented with rectangle nodes.	✔	✖	✖	Child nodes are placed under the parent ones and linked with an “isa” link.	✔	✖	✖	Represented with labeled links.	✔	✖	✖	Displayed on the node.
GoSurfer	Tree representation.	✔	✖	✖	Represented as tree nodes. Selected nodes are marked with numbers with their labels listed underneath the tree structure.	✔	✖	✖	Nodes are linked to their parents.	✔	✖	✖	Represented as an item placed inside the feature-item related.	✖	✖	✖	No representation. Displayed in a separate window.
IsaViz	Tree representation.	✔	✖	✖	Represented as labeled ellipses.	✔	✖	✖	Nodes are placed under their parent nodes.	✔	✖	✖	Represented with labeled links.	✔	✖	✖	Displayed as rectangle nodes linked to the instance with a link labeled, including in this link the name of the attribute.

Table 4. Representation of components in ontology modeling tools.
(T: tree representation, L: indented list, G: graph representation)

Lessons learned from these ontology modeling tools are:

- Firstly: Based in feature modeling quality, there does not exist one specific visualization method that seems to be the most viable solution for visualizing feature models. Rather a good tool is one that provides different views (visualization options) to allow the user to gain more flexibility when modeling. Furthermore switching between these modeling views should be allowed.

- Secondly: A good visualization tool should provide some efficient searching mechanism or querying for features with certain properties in the model and, thus, ease the modeling task in the case of very large feature models. The majority of ontology visualization tools provide some functionality that implements this function. Browsing sometimes is not powerful enough for searching for specific features in large visualizations.

- Thirdly: We found quite large symmetries between representing ontology concepts and features in feature models. Similarly feature dependencies resemble properties linking ontology concepts. The component that is more difficult to represent in feature model is the attribute.

Three types of representations are the most appropriate for representing rich knowledge representation models: the tree, the graph and the indented list.

Chapter 3

Visualization and interaction of feature models

3.1. Initial requirements analysis

Designing a tool for visualization and interaction of feature models should be done keeping usability in mind. To obtain a good usability, it is necessary to consider the target users of the tool as well as the tasks that the tool should support [33]. Therefore, we first have to study the future users of the tool and their requirements to be able to achieve an optimal and effective design for our visualization tool. In our case, the target users of the tool are ICT professionals, as well as other professionals with no or little knowledge in developing feature models. Also during the development of the visualization tool, we always have taken into account the fact that the tool will be used by these types of users.

As the type of visualization used will be critical for the usability of the tool, it was necessary to first perform a study of the different possible types of feature model visualizations. The final decision taken after this study is described in the next pages and looks for (1) the simplicity of the tool, (2) the intuitive understanding as well as (3) easy interaction and (4) a comfortable user experience.

Starting from the observation that the target users will be ICT professionals, as well as other professionals with no or little knowledge in developing feature models, we can derive the following basic requirements:

1. A non-experienced user has to be able to use the tool and understand what he can do with it with minimal training.
2. The visualization used in the tool has to show all the possible information contained in a feature model, without resulting in a visual overload.
3. The user should be able to manipulate easily the different components of the model. Move, edit or delete are basic actions that the tool must support.
4. Save and load feature models from a file are necessary actions of the tool, as the development of the models may be spread over time.
5. Good response times are necessary: the actions of the user must have an instantaneous response from the tool.
6. Interaction with the components of the model represents the power of the tool. The more actions take place inside the model representation instead of in auxiliary windows, the more intuitive the use of the tool will become.
7. To avoid the need for installing software, which may be difficult for non-experienced users, the tool has to run in a browser.

These requirements are only the basic ones. During the development of the design, more requirements were added, because of the programming environment used or because a new issues that emerged at that time.

3.2. First consideration: 2-D versus 3-D

The first question that arises in the design of a visualization tool is whether representation should be in 2-D or be in 3-D. All the tools studied in *chapter 2* are using a 2-D visualization, and the existence nowadays of 3-D tools for feature modeling is almost obsolete.

The idea of developing a 3-D visualization was discarded mainly because of the difficult manipulation of 3-D models. The reason of this discarding was based on the next points:

- Both the screen and the mouse are 2-D devices; via those ones it is difficult to achieve a real 3-D visualization.
- It is difficult to manipulate 3-D spaces with the current interaction techniques. Almost all the well-known types of manipulations are based on 2-D applications (e.g., scrolling, dragging).
- Navigation in a 3-D model is not easy and the user first needs to get acquainted to this. This would complicate his task of creating correct feature models significantly and would increase the learning time, which is in conflict with the first requirement.
- 3-D is not yet supported by default browsers. Therefore, special plug-ins would be needed to visualize 3-D in a browser environment. Currently, only little software is available that supports this. As requirement 7 states that the tool should run in a standard browser without the need to install specific software, this could be a problem. In addition, the software and the implementation needed for 3-D usually needs extra non-standard tools, and requires up-to-date hardware to have good response times.

3.3. Information visualization and interaction techniques

3.3.1. Information visualization methods

In our effort to gather all possible information for the design, we also present the following recompilation of other type of information visualization (InfoVis) methods useful for our tool. These techniques can be useful in the field of feature model visualization, and were taken in account in the design of the tool (see *Chapter 4*).

3.3.1.1. Indented List

The technique of the indented list is based on a vertical correlation of a set of items that can be indented to the right in different levels. From the feature modeling visualization point of view, the natural representation of the method allows the modeler to position the different items (features) according to the hierarchical structure. For instance, if one feature is the specification of another feature, the indented list represents this specification as a sub-item indented to the right from the position of its item father.

A list can contain a large number of different items, so the rest of the components in a feature model (attributes, dependencies, types) can be represented and placed inside the list. Nevertheless, when the feature model contains a large number of components, the inclusion of all these components can generate huge lists very inefficient to modeler's work.

In addition, this problem supposes a big effort for the reader to find and differentiate the types of items. For example, the inclusion of all possible feature model components supposes at least five different types of items (feature, type of feature, hierarchical grouping, attribute, feature dependency).

3.3.1.2. Venn diagrams

Designed by John Venn around 1880, the Venn diagram [34] is an illustration of the relationships between and among sets or groups of objects that share some or all characteristics. The principle of the diagram is the use of regions to represent classes or sets. These regions use to be represented with ellipses which could be overlapping with each other depending on the relationships between the classes or sets (see *Figure 13*).

This idea can be adapted to represent the structure of a feature model. The features can be displayed as regions in the diagram and the hierarchy component can use the structure of this method. For example, a father-child relationship can be represented in the model as a

large region (the father) that contains a smaller region (the child). The solution in case of the hierarchy could be the use of different colored regions (e.g. circles red-filled suggests that they include lower levels). To navigate in the hierarchical representation, the modeler only has to click in a region that will be magnified, with the use of animation, making its contents visible.

One of the disadvantages of this technique is the difficulty to distinguish the types of features. Although this technique allows playing with the sizes and the colors of the regions, these possibilities are already used to represent the hierarchy component. Thus, the method must use low-efficiency solutions for user's point of view, in order to achieve the representation of these types.

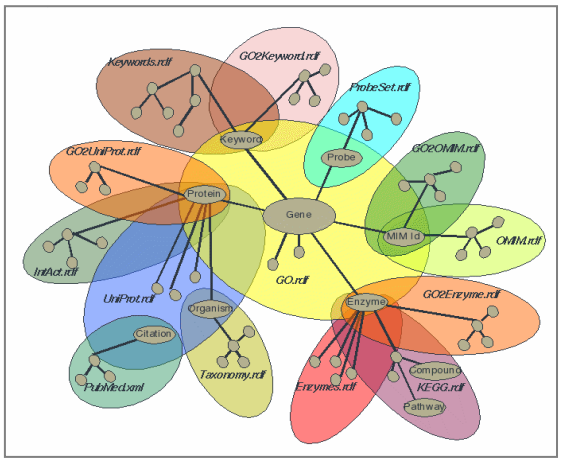


Figure 13. Venn Diagram [35]

3.3.1.3. Tree structure

A tree structure allows the modeler to represent hierarchical natures of a structure in a graphical form, which is formed by a series of nodes connected by links between them.

In the field of feature modeling tools, this method of visualization is most widely used. The hierarchical nature of the structure is valid for representing the hierarchy component of feature models, although it must make use of some notation applied to distinguish between the different types of relationships (normally Czarnecki-Eisenecker notation [11]).

On the other hand, the model can not include feature dependencies represented as links. The nature of the tree structure prohibits double connections between nodes or links that

create closed paths[3] in the structure. Hence, it is not possible to visualize the feature model as a natural tree, if the modeler wants to represent both feature dependencies and hierarchical component with links.

3.3.1.4. Treemap

Treemap [36] is a space-filling visualization that shows nodes as a rectangle area organized in a hierarchical structure by size and color-coding. The representation enables users from its composition to compare sizes of nodes and sub-trees (see *Figure 14*).

This technique was proposed by Baehrecke and Babaria as a tool for visualizing the GO ontology [32] (2004). In the application of this method, the features can be represented as colored squares of size proportional to a selected attribute. In addition, labels can be displayed up to a certain depth. For the hierarchy component, lower level nodes can be placed inside their parent nodes.

Figure 14. Treemap [36]

One of the disadvantages of using this technique for feature modeling visualization is the difficulty to represent the feature dependencies and the attributes. The structure of the model only provides the representation of components and its hierarchical relationships furthermore feature type groups will be difficult to represent.

[3] Jumping from node to node by the use of the relationships, and visiting the nodes only one time, it is possible to arrive to the starting node visited previously

Moreover, the use of coloration and proportional sizes are not powerful enough to represent the rest of the components. It means that the only option to display these components is by the use of auxiliary windows.

3.3.1.5. 2D hyperbolic tree

The hyperbolic tree technique defines a visualization method for a graph based on a hyperbolic geometric transformation [37].

Referring to features and their hierarchical structure, the root of the tree (representing the root of the feature model) is initially placed in the middle of a circular area with the child nodes placed around it, their child nodes placed around them and so on (see *Figure 15*).

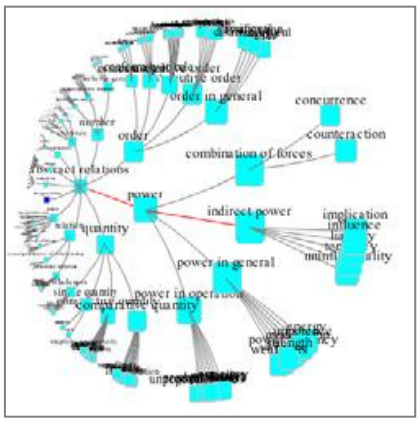


Figure 15. Hyperbolic tree [37]

This technique is not appropriate enough for our case. Compared with the classic tree representation method, the hyperbolic tree entails the loss of the natural tree structure. In other words, the representation seems more like a graph than like a tree for the user point of view, while in our case our tool is looking for the comprehension by the users of the tree hierarchical structure.

3.3.1.6. Graph representation

The graph representation is an illustration that can be understood as a set of connected or non connected nodes with links between them.

A feature model represented with this method means the possibility to visualize both hierarchical links and feature dependencies as relationships; as well we have to provide some notation on the edges to distinguish the two types of links.

Thus, the clearest way to adapt the feature model with the graph method is by the representation of the features as nodes and the hierarchical components and dependencies as relationships (commented previously). On the other hand, there is a wide range of possibilities for interpreting the rest of the model components. As an example, attributes can be represented as new nodes linked with the related feature or as descriptions placed inside the feature. So, the representation may vary according to the necessities of the feature modeling tool.

3.3.2. Human–computer interaction

Interaction between human and computer is at the heart of the modern information visualization [38]. One of the principles that were followed in the design of the tool (see *Chapter 3*) was to allow as much as possible interaction between the user and the representation of the feature model. The more interaction with the visualization is possible the easier it will be for the user to management the models. The next sections present an overview of some HCI techniques that were helpful for the design of the tool.

3.3.2.1. Explicit Representation

Explicit Representation refers to drawing methods which display the hierarchy as links between nodes [39]. The goal of this technique is represent the information as intuitive as possible. In the case of feature models, the best way to represent the hierarchical relations and dependencies is by the use of links between nodes representing the features.

In addition, the possibility to represent the links in different ways (e.g. curves, straight lines …) provides the possibility to differentiate between type of relations. For feature modeling, this possibility can be applied to represent different types of relations. The modeler would then see the difference between three types of relationships: hierarchical links, feature dependencies, and attributes related to features; each represented by means of a different link representation.

From my point of view, it is more intuitive to provide as the nodes of the model only the representation of features. Therefore, it obviates the use of links to represent the related attributes of features; otherwise these attributes should remain as nodes.

3.3.2.2. Color coding

The encoding based on colors adds another layer of information for visualizing information [39]. The colors can be used to provide information about the state of components or to differentiate between types of components.

This technique can be used at different moments during feature modeling, and could help the modeler in the development process of a feature models. For example, when the user clicks on a node to add a hierarchical link, nodes that can be linked (features not linked with the same link type) could change their fill-color. This could help the modeler to recognize from the visualization the set of nodes that can be related by e.g., the hierarchical link.

In addition, the coloring of features or dependencies can also be useful distinguishing their types (each type can be associated with a color). Therefore, this method can be applied in many different ways when visualizing feature modeling

3.3.2.3. Details on Demand

Sometimes it is hard to represent all information graphically in the visualization, because of the large size of its representation or because of the abstract meaning of the information (difficult to obtain a clear representation). In these cases, the use of the technique *Details on Demand* could be a good solution for the problem.

This technique refers to the facility whereby the stakeholder can choose to display additional detailed information at a point where this data would be useful [39]. In the case of feature modeling, for example, the representation of the attributes can create a problem when a feature contains a large list of attributes, this then by using this technique we show information about attributes only when the user is in need of it or requests it.

3.3.2.4. Techniques based on big representations

The modeler may have difficulty understanding the model's representation when it contains a large amount of components. In addition, depending on the number of components contained in the visualization, the displaying of all this information becomes a difficult task for the tool because of the small size of these components when visualizing the full of content.

The next sub-sections present two techniques that try to bring solutions to such problems: *Incremental Browsing*, and *Focus + Context*.

3.3.2.4.1. Incremental Browsing

Incremental browsing is a technique used to solve some aspects in these cases. The method is based on the filtering the content of the representation by limited sections of the

visualized structure displayed [39]. The rest of the representation is hidden and can be showed when modeler's desire to.

In feature modeling, the high-level nodes of the hierarchy component can be displayed as a starting point. The modeler can visualize the rest of the hierarchy as he explores the visualization by the use of mouse events applied on the components.

The representation sometimes can provide a level of distinction between the components in different levels, for making the distinction between of the hierarchy's components easier. On the other hand, sometimes this solution becomes a problem in the case of models with a high number of levels. For instance, if the representation of the feature model is based on a graph, the incremental browsing of the information space is based on visualizing only limited sections of the whole graph [40]; according to the user clicks, the rest of the graph is visualized.

3.3.2.4.2. Focus + Context

To quote Dürsteler [41]: '*The main problem of information visualization is the insufficient space, which restricts the user in showing detail and context contemporaneous, is called "presentation problem". The Focus + Context system allows the user to show detailed information linked with the context, by also having the possibility to focus on other information by interacting with the system.'*.

This technique tries to display contextual information without the loose of the modeler's relative orientation when zooming into a big representation. The use of the zoom and pan allow the modeler to scale the view and preserve the focus of the content. In addition, sometimes it is useful to provide the visualization tool with multiple windows or viewpoints for displaying the location in the overall representation and, in consequence, preserve the context of the content.

In feature modeling the use of this technique is applicable when the representation is based on graphs or trees. A feature model with a high number of features needs to use some mechanism for the modeler to focus on a subpart of the representation preserving the general context of it.

3.4. Selected visualization techniques for feature models

The difficulty when designing a visualization tool for feature models is how to achieve a good visualization where all the information is showed in a coherent manner. Features, feature dependencies, hierarchy and attributes are mainly too much information to show in one representation. Although it is possible to show all information at the same time, sometimes the visualization will be useless due to the information overload. For example, visualizations where all the attributes are shown linked to their features (see *Figure 12*) become huge. Showing all information at the same time is good for small models but don't scale for large ones.

Therefore, is not only important to show all the components, the visualization must also allow a human being to build a correct mental model of the model shown. Even when the model is very large, the visualization must support this.

One can say that the key to measuring the solvency of good feature model visualization is obtained by the impressions from a user when looking to the model. If the user can recognize all the components of the model as well as the global context (mainly understand the hierarchy and distinguish it from feature dependencies), the representation can be considered as suitable.

In the effort to find a single visualization that meets all the characteristics discussed above, we found that the best way to visualize all the content of a feature model was using the graph view. However, the problem with this type of visualization is that it will not be easy for the user to recognize the hierarchy of the model from the graph.

The solution of this problem is to eliminate the links that represent the feature dependencies. Then the graph becomes a tree and the hierarchy is easily recognized by the user. However this introduces a new inconvenience: the loss of visual information (the dependencies are not visual anymore).

The natural way to visualize feature dependencies is by means of links from feature to feature; any other way to represent this component would be unnatural. Representing a dependency in or attached to only one of its feature would require an unnecessary decision from the modeler to decide where to place it. Representing the dependency inside both features will introduce an unwanted redundancy. Therefore, our solution was to provide both types of representation in the tool: the graph view and the tree view.

The user has to be able to visualize the two types of representations. This means that when the user is interested in analyzing the hierarchy he/she can use the tree view. Otherwise, from the graph model the user can visualize all the possible information and interact with it. The more different visualizations are provided for a certain feature model, the easier it will be for the user to understand and manipulate the model.

Following this theory, we also decided to add another representation: the indented list. The idea is to provide the graph view with a complementary view and interaction mode representing the hierarchy of the feature model. In this way, the user can see the structure of the hierarchy (using the indented list) while is working with the graph model.

Thus, the final decision was the utilization of three types of visualizations for the feature models: the graph model, the indented list and the tree view. The following subsections show in detail the design of the different visualizations, and also how the user can interact with each one. Having these three different visualizations has added an additional requirement to the tool:

8. The three visualizations need to be kept consistent with each other while the user is manipulating one of them.

3.4.1. The graph model

A graph is a diagram that shows a set of relationships, often functional, between a group of points or numbers. Each of these points or numbers has coordinates determined by their relationships. This diagram represents a mathematical structure or a symbolic representation of a network [42].

The graph representation of our tool is based on this definition. This is the only representation that tries to show almost all the content of the model at once. The graph representation will be given in the main window of the tool. From there, the user will work most of the time when developing the feature model.

3.4.1.1. Graph-view design

To make the understanding of the graph-view design easier, *Figure 16* shows a representation of graph-view model in our tool. This model is the representation by our tool of the model presented as an example in the *Chapter 2*, *Figure 1*.

At a first glance (see *Figure 16*), the user can distinguish between boxes (nodes of the graph) and edges (relationships) in the representation. Each box represents a feature where its name is placed right in the middle. The user can differentiate between four types of boxes; each represented by a different the border (described further on). In addition, two buttons are situated on the bottom of the boxes; one contains the letter ‘A’ and the other one the letter ‘L’, these buttons are discussed later.

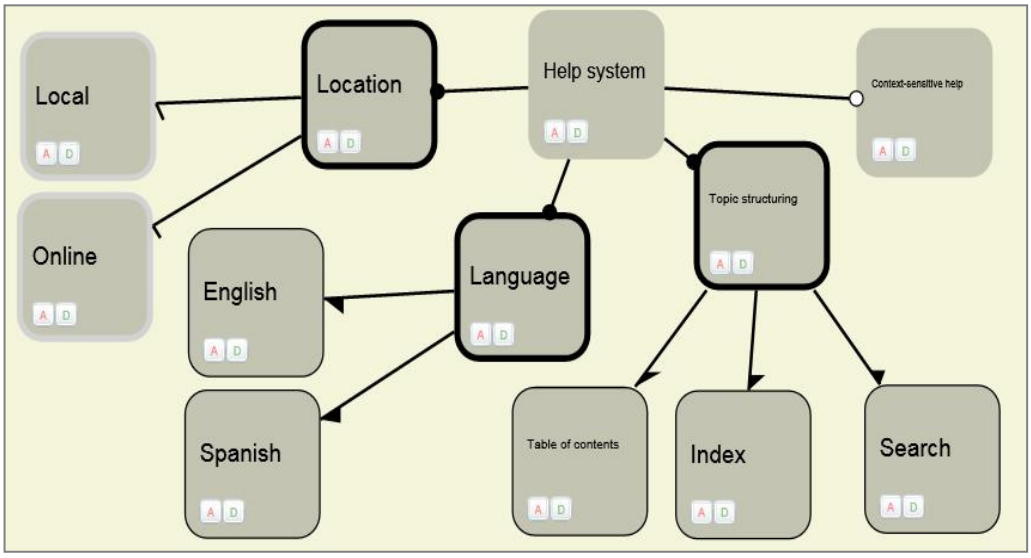


Figure 16. Example of graph model representation

Also there are two set of lines (relationships) linking features in pairs: the black lines and the colored lines. The black lines ends with a graphical annotation that depends on the type of the destination feature (mandatory-optional-or-alternative), while the colored lines represents feature dependencies and is annotated with a mark (ellipse situated on the central point of the edge) that indicates the type of dependency. The next sections describe in details the different graphic notations used.

3.4.1.1.1. Features and attributes inside graph view

As commented previously, the representation of features are based on boxes. The different border of the boxes depends on the type of the feature (see *Figure 17*). The reason for this differentiation is to support different ways of creating feature models. For example, if the modeler starts adding all the features, at the moment that he/she starts to create the hierarchy, there is some visual aid to distinguish the different types of features. Otherwise the user has to remember the types of all the features that were already created.

The two buttons situated at the bottom of the boxes gives to the user visual information about dependencies and attributes related to the features. The ‘A’ button reports on the existence of attributes while the ‘D’ button reports on the existence of feature dependencies.

The 'A' button is not enabled when the feature does not have attributes. The same happens for the 'D' button when the feature is not linked with other features by means of feature dependencies; then the 'D' button is not enabled. On the other hand, if the user clicks on the 'A' or 'D' buttons when they are enabled, the tool shows a new window with the list of attributes or feature dependencies respectively.

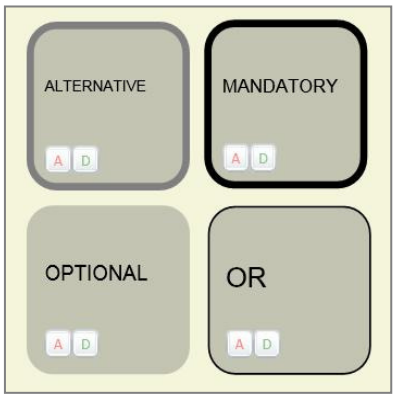


Figure 17. Different types of features

The attributes are not directly represented in the model due to the problem of the possible large amount of data that should be showed. But from the representation of the feature, the user can recognize the existence of attributes

3.4.1.1.2. Feature dependencies

As in the case of the features, the modeler also must distinguish between the different types of dependencies in the model. The use of colors is the first possibility to provide the representation for this differentiation; each type of relation corresponds with a different color (please refer to Color Encoding technique, *section 3.3.2.2*). The problem of this differentiation lies in the fact that the users needs to know the association between color and type of dependency. With the use of different colors the modeler can distinguish different feature dependencies, but he may not know or recall the meaning of the colors.

A color has not an implicit meaning to relation with the descriptions of dependency types (see *Figure 1*). Because of that, the tool also provides icons on each link that describes the type of relation (see *Table 5*).

After some study on different possible icons, we decided to associate a character to each kind of link. For example, because of the meaning of the dependency *same*, the most obvious is to associate this type with the character '='.

The representation of these icons is an ellipse that contains the correspondent character. In the representation, they are situated on the central point of the link.

Feature dependency name	Symbol	Color
Excludes		
Extends		
Includes		
Incompatible		
Requires		
Uses		
Same		

Table 5. Representation of dependencies marks

As a result, it may be easier for the modeler to recognize in the visualization the type of each dependency by mean of the icons. In addition, it is also easy to distinguish between the different types of the links by means of the colors.

3.4.1.1.3. Composition hierarchy

Representing the composition hierarchy of a feature model in the graph view is more complicated than it looks. Some compositions originate from group relations, such as alternative and or feature relations. As mentioned previously, the set of black-fill links are the components responsible for displaying the power structure. The destination of each edge depicts the hierarchy's type (see *Figure 18*). In the case of optional and mandatory links, the representation uses the Czarnecki-Eisenecker [11] (ending with an empty-fill circle for optional links and ending with a black-fill circle for mandatory links).

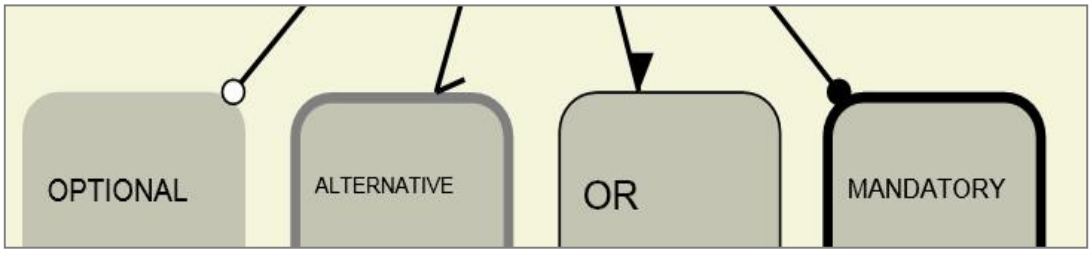


Figure 18. Representation of the hierarchy (no using groups)

On the other hand, the Czarnecki-Eisenecker notation [11] is not a good solution to represent the alternative and OR links in our case. That is, if the representation would use this notation the model would lose the graph structure due to the grouping of the links in the composition hierarchy. The solution of this problem was solved by introducing a new notation to represent groups of features. Each group has a name representing the name of the feature or functionality provided by the group, this name is used as an identifier to distinguish different groups. The group name is included with each link to one of the group members; *Figure 19* shows an example alternative group.

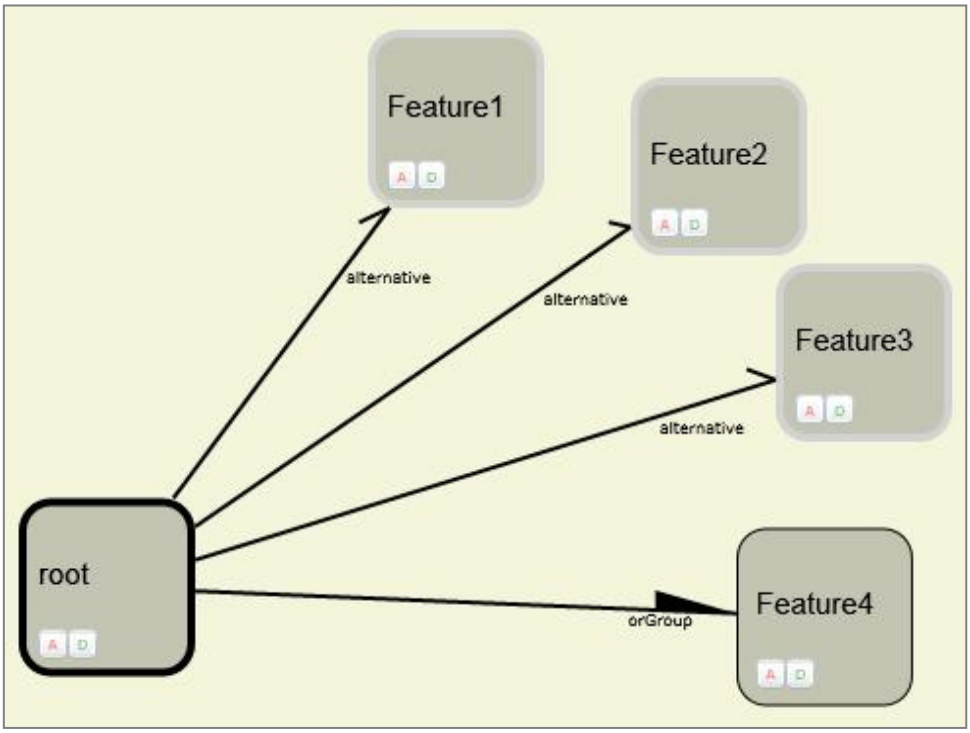


Figure 19. Grouping representation

3.4.1.2. Graph-view interaction

One of the requirements when designing our tool was the possibility for the modelers to create, manipulate and delete components of the model through the visualization. In this section we present the different user scenarios to interact with the components placed in the graph-view.

For the ease of user interaction a toolbar that contains all the possible modeling actions is provided. In our case, we call this toolbar the *Drawing Toolbar* (see *Figure 20*). It is composed of ten buttons, and the function of each one is described in the next paragraphs (starting from the button situated on the top of the bar to the one situated on the bottom):

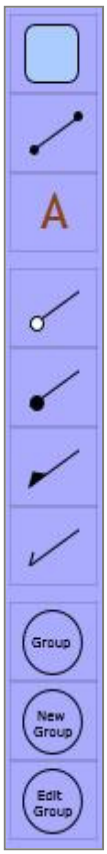


Figure 20. Drawing Toolbar

The first button gives the modeler the possibility to add new features to the graph-view. When clicking this button, the representation of the new feature appears as a red box situated at the position of the cursor. The user only has to drag this box to the position where the feature needs to be situated in the visualization workspace (user work space), and release the mouse button. The last step is to complete the detailed information of the new feature. This information contains: the feature name which will be placed in the middle of the representation, a brief description of the feature (in case it is necessary for the modeler) and the type of it (alternative, mandatory, optional and or-feature). In this case, it is obligated for the modeler to provide the name and type of the feature.

The decision to use the dragging in the creation of features was taken because of the easy way to position the new feature in the user work space. Otherwise the user would have had to indicate the coordinates when adding a new feature. In addition, it is easier for the modeler to control the collisions between the new feature and the existing ones than doing this automatically.

The addition of feature dependencies is implemented by means of the second button. When the user clicks on it, the cursor becomes a point and the tool changes the status of operation, which in this case adds a blank feature link. Then, the modeler has to select step by step the two features in the representation that he wants to link (it an auxiliary line that goes from the origin feature to the mouse release position is placed, only one feature selected). To finish, the user has to select the type of feature dependency between the lists of possible (see *Table 1*.) on a new window that emerges when the second feature is selected.

The third button is used to add an attribute to a certain feature. As with the feature dependency addition button, the cursor and the status of operation changes when this button is clicked. The user only has to click on the related feature in the graph-view and complete the information of the new attribute in an auxiliary window.

The following four buttons are used to develop the composition hierarchy in the model. Each button represents a feature relation type (subsequently: optional, mandatory, or and alternative). The way to add these components is the same as the addition of feature dependencies, so the behavior of the user when adding a hierarchical link is the same than when adding a feature dependency.

To finish, the last three buttons control the grouping in the composition hierarchy. The last two are used to create and edit the definition of a group, while the antepenultimate button is used to relate (a) feature(s) to a group. In that case, the user only has to select the feature (or list of features by pressing the keyboard *Ctrl* button), and then click the *group* button. The last step is the selection of the group from an auxiliary window. The user has the option to select from already existing groups or select to create a new group. If he selects to create a new group the new group window is opened.

3.4.1.2.1. Editing and deleting concepts

Our tool also provides the possibility to modify, both the visual properties of the components (position, size) and the actual content of the model (features, feature types,

dependencies and attributes) of the components inside the graph-view. All these actions take place in the workspace by means of easy drag and drop operations.

With regard to the global representation, the tool contains a scroll panel to modify the perspective of the visualization (*PAN*). In addition, at the top of the window a slider is located to increase or decrease the size of the visualized components when the modeler changes its position (*ZOOM*).

The existence of these functionalities in the tool is tremendously important as it allows the modeler to visualize the full model or a reduced the perspective of the model when the representation is full of features and links. For example, the user can zoom in on a node when he wants to drag it more exactly in the visualization using the mouse.

On the other hand, one of the problems that arise from the use of two types of links (feature dependencies and composition hierarchy links) is the overlapping of edges. From one feature to one other feature a maximum of one hierarchy link and seven feature dependencies can be added.

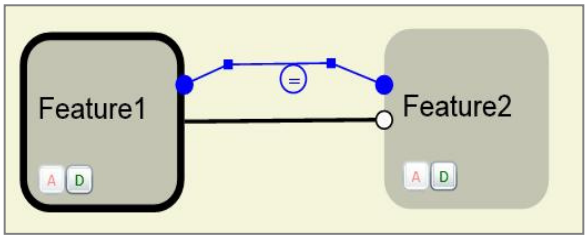


Figure 21. Representation of a dependency

The use of curves to represent the links was discarded, because of the hard calculations for positioning these curves. The solution presented (see *Figure 21*) is the possibility to modify the relative position of the dependencies using auxiliary points. The two squares on the link can be clicked and dragged by the user changing the shape of link.

To finalize, the modeler is allowed to deleted components of the representation via the selection of them (for features clicking on the box, for dependencies clicking on the line segment representing the relation or on the mark and for hierarchical links clicking on the

edge). Once the selection is made, the deletion is done simply by pressing the *Delete* button on the keyboard. The user is first asked to confirm the delete operation and then after confirmation, the model is modified with the removal of the selected items (which includes also all the components connected or associated with it

3.4.2. Indented list

The indented list is the second type of visualization in our tool. The reason for the inclusion of this representation is due to the difficulty for the users to recognize the composition hierarchy from the graph-view. So, the indented list can be considered a visualization support for the graph-view.

A list is a series of objects organized in a logical order. In our case the objects are the features of the model, and the organization is the composition hierarchy of this elements. Sometimes a list representation is difficult to read, especially when the organization of the list elements is more than a simple succession of objects.

The indented list is a solution to represent the hierarchical organization of the feature model as a serial. The structure is very similar to the one used by Protégé (see *section 2.2.1.1.*), the child elements in the list are placed under their parents and indented to the right.

3.4.2.1. Interaction with the indented list

Like in Protégé, the nodes included in the list are expandable showing the rest of the lower features under the hierarchy, or retractable hiding it. The user only must click on the arrow situated on the right of the parent feature (see *Figure 22*). When a node is expanded, this arrow changes shape and becomes a black-fill. Arrows of non-expanded nodes are represented as an empty-fill arrow.

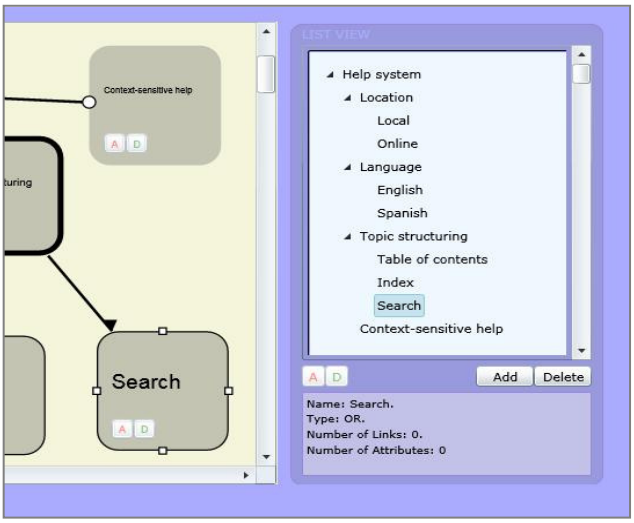


Figure 22. List-view representation

In the area below the list visualization, the user is allowed to visualize the information about links and attributes of a feature using the buttons situated at the left side (see *Figure 22*). As in the graph-view, disabled buttons means that there are no attributes and/or dependencies related to the feature selected. The info window showed when the modeler clicks on these buttons is the same than when clicking the correspondent buttons on the graph-view.

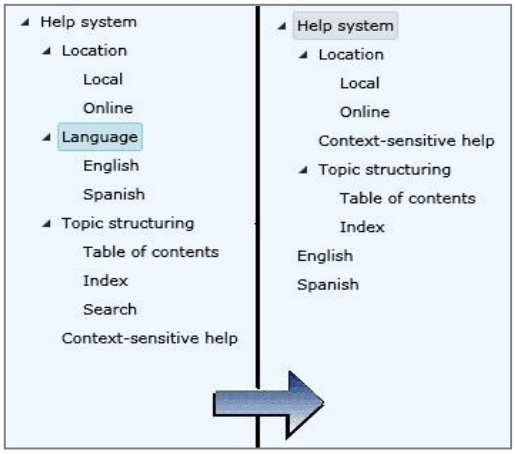


Figure 23. Deletion in indented list-view

In addition to this, the representation also includes a text panel situated on the bottom that reports the essential information of the feature selected: its name, its type, the number of feature dependencies linked to it and the number of attributes that it contains.

Another functionality of the list-view is the possibility to add a new child feature to the selected node. The user only has to click on the button 'Add' in the representation and, after it, introduce the information about the new feature. Also it is possible to delete the

selected node using the keyboard 'Delete' button. The child features of the deleted node and their sub-hierarchies become free root nodes so they are moved to level 1 of the component hierarchy in the list-view.

Figure 23 shows in detail an example of a deletion, according to the model analyzed throughout the document. The figure shows the state of the list-view before and after deleting the nodes *Language* and *Search* in the model. In the case of the *Language-feature* deletion, all the nodes placed under it (*Spanish* - *English*) are repositioned as free root nodes at the end of the list.

In addition, the tool provides synchronization between the graph-view and the list-view to achieve better user interaction and facilitate model manipulation. So when the modeler selects a node in the indented list, this feature also becomes selected in the graph representation, and vice versa. The goal of this synchronization is to help the user comprehending the model by easily switching from one view to the other.

3.4.3. Tree-view

The last type of visualization that our tool provides is the tree-view. The design of this representation tries to resemble as good as possible as the classic representations of feature models [7]. This is a good option for users who are used to work with this type of representations and wish to view the model in this way.

The features are represented as boxes in the diagram (see *Figure 24*), and the hierarchy component is represented as a black-fill links connecting the different nodes. The Czarnecki-Eisenecker [11] is the notation chosen to distinguish the different links (mandatory, optional, or, alternative).

From the *Drawing Toolbar*, the modeler is allowed to add new features. To give more possibilities for interaction, the user can also add features clicking the mouse's right button anywhere in the workspace. In this case, a drop-down menu is showed with the options of adding features or links, among others. The links can also be added from the *Manipulation Bar*, the user only has to click on the correspondent button and select initially the origin feature, and then the destination feature.

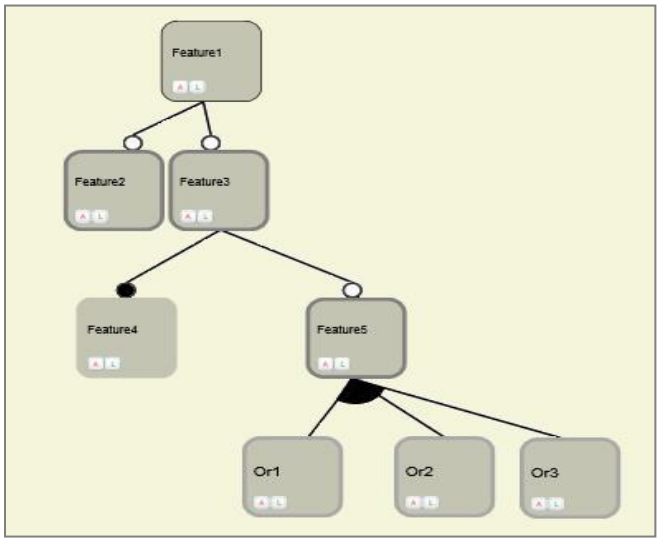


Figure 24. Tree-view example

As in the graph-view, the boxes representing features contain more information than only the name of the feature. It is indispensable for the user to recognize if a feature contains attributes and dependencies in a brief and easy manner. For this reason, the view uses the notation based only on annotating the feature components with letters (same case than graph-view), called AD-notation. Thus, inside the boxes representing features two buttons containing the capital letters (A and D) are placed. The have the same meaning as in the graph-view:

When the A-button is enabled, it informs about the existence of attributes in this feature. As in the graph-view, the tool opens a new window with the information of the attributes related with the feature, when the user clicks on this letter. To increase the user interaction, this window is a non modal window and stays open until the users closes it; in fact, if the user clicks on another feature, the window will be reloaded with the information about the attributes of the newly selected feature. In addition, the window is moveable on the main layer, so it can be move when it obstructs in the construction of the model.

When the D-button is enabled, it informs that this feature has dependencies. It is important to show that there exists links in the model because these are not placed in the model. In

this case, the function of this button is different than in the graph-view: when the user clicks on it, the main window will be reloaded with the graph-view.

As commented previously, this notation is very important because it permits the user to distinguish the characteristics of each feature in the model. Furthermore, the feature dependencies of the model are not placed as links because this would result in the graph-view. In conclusion, by providing the tree view we want to achieve a visualization of feature models according to the classic representation of feature models.

3.4.3.1. Editing of the components

The manipulation of the components in the tree-view is very similar to the graph-view. This is quite important because we want to ensure the consistency in our tool with respect to the user interaction. As an example, the user would make a double click in the feature representation for editing its information, both in tree-view and graph-view). Otherwise, the tool will become more complex and difficult to learn and use if the behavior of it differs greatly with respect to the graph and the tree visualization. The user would need to adapt to the two modes of use each time when changing the type of visualization.

In this case, the modeler is only allowed to interact with the features and the hierarchical links (the only components represented in the model). The features can be selected, dragged, deleted and modified like in the graph-view. On the other hand, the hierarchical links can also be removed from the model; when this happens, the tree-view has to remove all the components of the structure placed hierarchically under the feature that was linked from the deleted edge (included itself).

Therefore, the representation of the feature model is always consistent. The user can consider the tree-view as the correct visualization part of the feature model created.

3.5. Save and restore the feature models

With regard to the requirement number 4 (see *Chapter 3.1, Initial requirements analysis*), in the design of the tool we also take into account the possibility of saving and restoring consistent feature models from files.

Figure 25. Toolbar Save/restore model

Figure 25 shows the toolbar used by the modelers to save and restore the feature model. The first button is the restore model button. When the user clicks it a new file selection window opens, when the user selects the representation model file that he wants to load into the tool by clicking this button, the representations and content of the tool are updated with the content of the selected file, if and only if the file contains a consistent and correct feature model representation.

Meantime in the case of saving the model (second button), the tool checks that there is only one root node, incase this constraint is violated the application reports an inconsistent feature model error to the user.

The third button is used to create a new model (all the content and the representations are cleared). Finally, the fourth button displays in a new window the content of the current model file, in case it will be saved in a file.

3.5.1. XML as the model file format

We use XML to save the contents of the visualized model in a serializable manner. This decision was taken because of the easy understanding for the users of this language and its widespread usage by industry as exchange format.

Figure 26 presents the content of a feature model file based on the model presented as example in the *Chapter 2*, *Figure 1* (adding some dependencies and grouping of features, for a better understanding). The file contains different tags represented as follows:

- <ListFeatures>: List all the features (*Feature* tag) of the model. Each feature contains its identification, its name, description, its type and its relative position in the graph-view and in the tree-view.
- <ListHierarchies>: List all the hierarchical links (*Hierarchy* tags) of the model. Each hierarchy contains the identification, the type, and the destination feature and the source feature of the relationship.
- <ListLinks>: List all the dependencies (*Link* tags) of the model. Each link contains the identification and the destination feature and the source feature of the relationship.
- <ListGroup>: List all the groupings related to alternative-or (*Group* tag) in the model. Each group contains the identification, the name, the type of the grouping, the brief description and the list of features (*FeatureGroup* tag) related to the group.
- <ListAttribute>: List all the attributes related to features (*Attribute* tag) in the model. Each attribute contains the identification, the name, the type, the comparator, the result of the comparison and feature related to the attribute.

```
<!--
   Created 11:07:57
-->
<Root numLink="22" numGroup="0" numFeature="17">

  <ListFeatures>
    <Feature Id="5" Name="Help system" Description="" Type="2" xPos="663" yPos="125"
        xTPos="663" yTPos="125"/>
    <Feature Id="6" Name="Location" Description="" Type="1" xPos="513" yPos="131"
        xTPos="513" yTPos="131" />
    [...]
    <Feature Id="14" Name="Table of contents" Description="" Type="3" xPos="652"
        yPos="352" xTPos="652" yTPos="352" />
    <Feature Id="15" Name="Index" Description="" Type="3" xPos="760" yPos="354"
        xTPos="760" yTPos="354" />
    <Feature Id="16" Name="Search" Description="" Type="3" xPos="885" yPos="351"
        xTPos="885" yTPos="351"/>
  </ListFeatures>

  <ListHierarchies>
    <Hierarchy Id="6" Type="1" fDestiny="Feature6" fOrigin="Feature5" />
    [...]
    <Hierarchy Id="21" Type="2" fDestiny="Feature9" fOrigin="Feature5" />
  </ListHierarchies>

  <ListLinks>
    <Link Id="0" Type="1" fDestiny="Feature2" fOrigin="Feature1" xOrgPos="108"
        yOrgPos="148" xDestPos="108" yDestPos="194" />
  </ListLinks>

  <ListGroups>
    <Group Id="0" Name="Alternative Group" Type="0" Description="">
      <ListIds>
        <FeatureGroup Id="6" />
        <FeatureGroup Id="16" />
      </ListIds>
    </Group>
  </ListGroups>
  <ListAttributes />
</Root>
```

Figure 26. File content of a feature representation

Chapter 4

Feature Modeling Visualization Tool Design

4.1. Interaction between views

The modeler's key guide to understanding the tool is to realize he is working with three types of visualization. We support this understanding by making the design of the interaction between the modeler and the three views as simple as possible; this is indicated in the following points:

- *Understanding* the three visualizations means depicting the commonalities between the content of the three views. To achieve that, our tool works with the synchronization of the three views whenever a modification in model fires (e.g. when a representation of a feature is removed in the list-view, also its representation in the tree-view and graph-view has to be removed). This gives consistency to the software and helps the user understand what is happening by switching simultaneously between three views.

- *Working* with three different views supposes easier feature modeling. Thus, the tool has to be designed to facilitate the modeling process and not complicate more the modeler's job. The introduction of three views is not the introduction of an obstacle to the modeler; rather it introduces more possibilities for the user to understand the

model. Complex application design sometimes implies a hard effort from the user to know how to use the tool.

To facilitate the understanding and the working task for the modelers when using our tool, we decided to maintain the list-view as a static part of the application. Then the desktop application design of our tool is divided in two parts (see *Figure 27*). In the first part (left side), is placed the *Drawing Toolbar* for manipulating two possible views: the graph and the tree. The user can decide which of them he wants to visualize. Therefore, the right side of the tool represents the list-view.

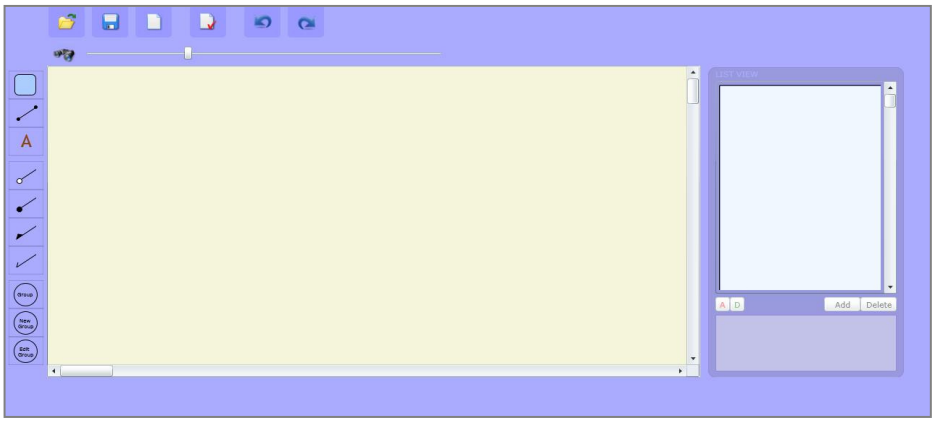


Figure 27. Screenshot of the application

The decision to show the list-view as a static part of the application was taken because the list is the best representation to work as a complement to the other two views. Firstly, the list-view occupies a small space in the desktop application. And secondly, and the more important, from this small space the user is allowed to distinguish the hierarchy component of the model (important when modeling with the graph-view, where it is hard to understand the hierarchy).

4.2. UML Model for the feature modeling visualization tool

In order to explain the design of the tool, this chapter provides the basic UML component diagram. The specification of the application is divided in three artifacts responsible for the visualization: the graph-view artifact, list-view artifact and tree-view artifact.

This division is done to provide the design independence between the main artifacts which form together the global specification. Although these three share some components (the kernel of the application), the differences between them are significant because of the different representation of the content.

Thus, from the designer point of view, the application is based in a common part that contains consistently and maintained all the relevant information, in addition of three different parts which make use of the information contained in this common part.

4.2.1. Graph-view artifact

Figure 29 presents the UML of the graph-view artifact. This time the design varies dramatically compared to the list-view artifact. For example, the feature representation (idem name for the class) is composed by a set of four classes instead of only one element in the list-view artifact (*Rectangle* representing the box, *TextBlock* representing the name of the feature, *Button Attribute* and *Button Dependencies*).

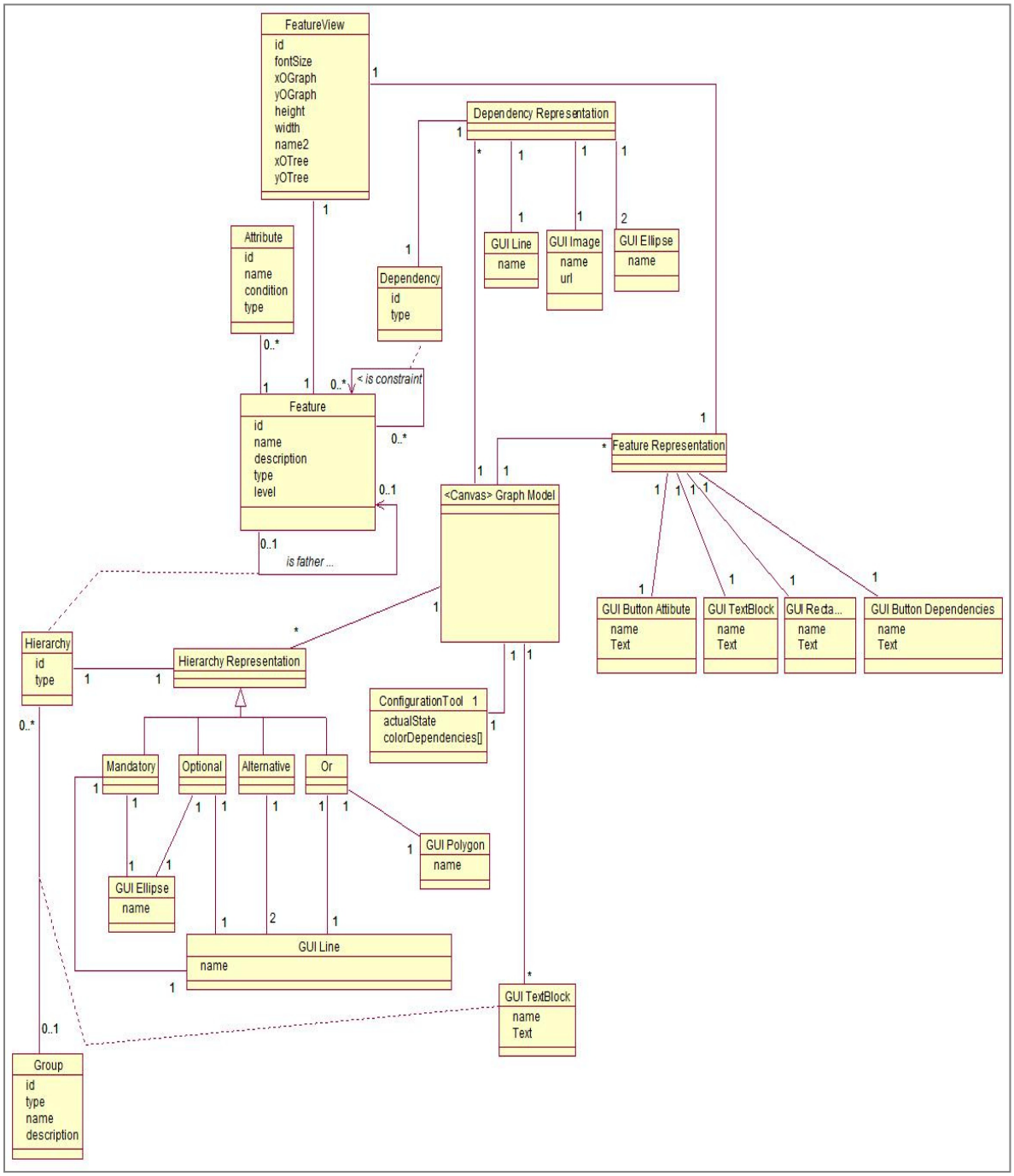


Figure 28. UML Diagram graph-view artifact

In addition, each dependency, hierarchy and related group has a representation in the *Canvas Graph Model*. The *TextBlock* class is responsible for relating groups with hierarchies' portrayal, while a set of components (*Image*, *Line* and two *Ellipses*) provides the tree-view the dependency representation.

On the other hand, the class *Hierarchy Representation* has to be classified in four subclasses, depending on the type, and based on the different link representations according to *Figure 18* (see *Chapter 3*). The following table shows the composition of each type of hierarchy link, in the design, and how it relates to the GUI shape classes.

Hierarchy type	GUI Ellipse	GUI Line	GUI Polygon
Alternative	0	2	0
Mandatory	1	1	0
Optional	1	1	0
OR	0	1	1

Table 6. Relation between classes and hierarchies

4.2.2. List-view artifact

Figure 28 presents the UML of the list-view artifact. The domain of the model is represented by the classes *Attribute*, *Dependency*, *Feature*, *FeatureView*, *Hierarchy* and *Group*.

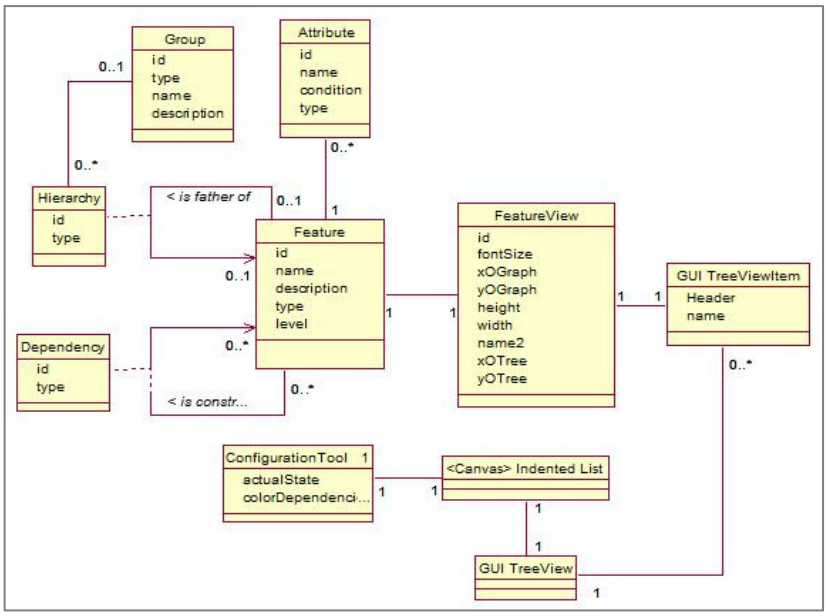


Figure 29. UML Diagram list-view artifact

These classes are common for all three artifacts, and contain all the relevant information about the state of the feature model (we can consider the group of these as an internal representation of the model). In addition, the singleton *ConfigurationTool* handles the configuration and state of the tool during its running process.

The *Canvas* class *Indented List* is the representation of the list-view in the tool. It contains a *TreeView* (the indented list representation) which, at the same time, contains a group of *TreeViewItems* (the nodes of the indented list). Each of them is related to its *FeatureView*.

4.2.3. Tree-view artifact

Figure 30 presents the UML of the tree-view artifact. This design is quite similar to the graph-view, but with the difference that in this case the dependency representation is not included in the UML.

In addition, the representation of the hierarchical links varies in the design; while the *mandatory* and *optional* maintain their status, the *or* and *alternative* hierarchies become a part of a *Group Representation* class. From the point of view of the tree representation, each group of hierarchical links represents only one element because of the arc that encompasses all the edges.

Thus, the definition of four types of hierarchical links does not work in the tree-view artifact. In this case the design shows two types of hierarchical relationship to a feature (optional - mandatory), and two type of hierarchical grouping (or - alternative).

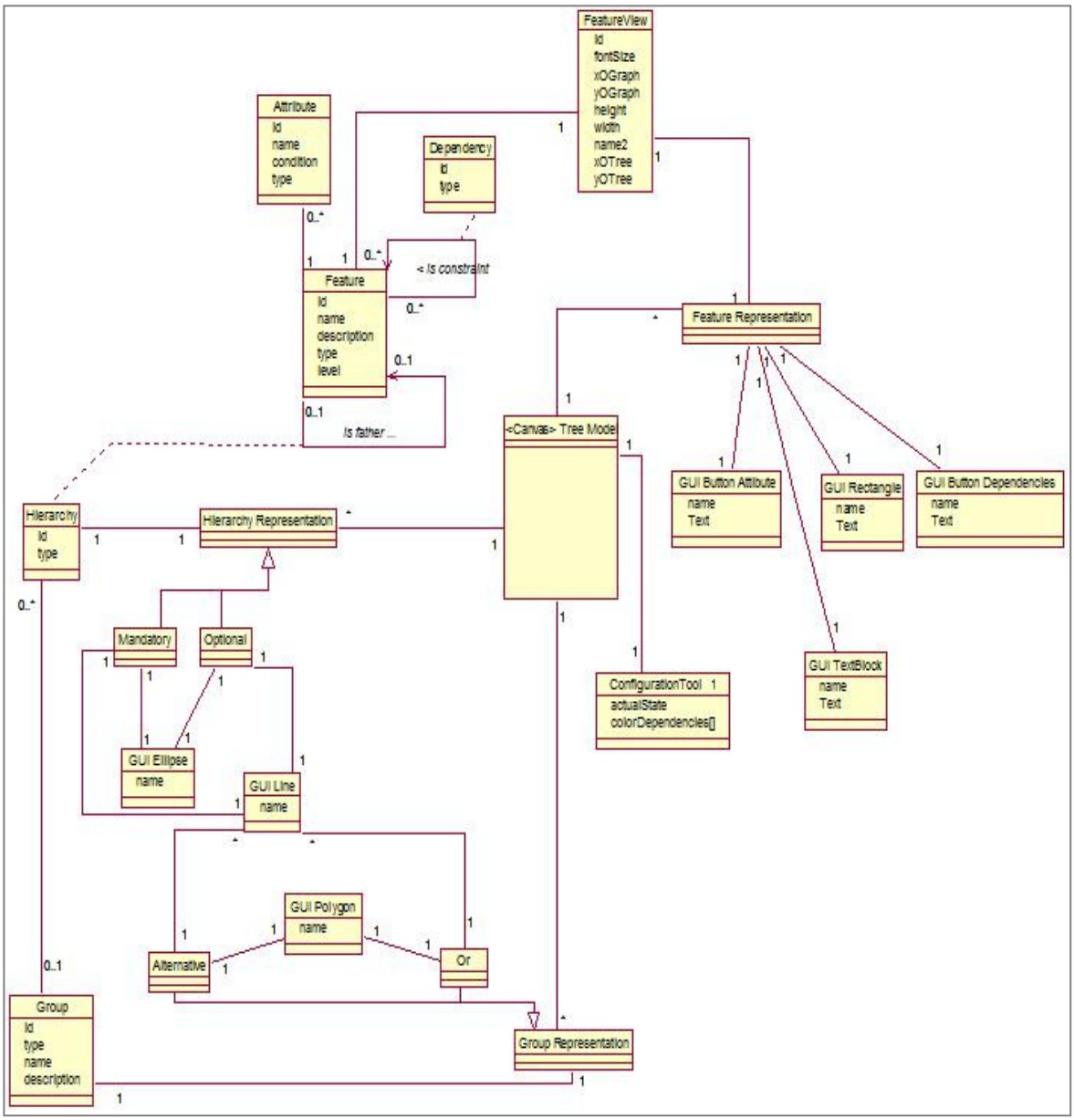


Figure 30. UML Diagram tree-view artifact

4.3. Use Cases

The Use Cases describe what the system has to do from the point of view of the user. That is, the description of the tool's behavior regarding the use of it and its interaction with the user.

The Use Cases are divided in four groups depending on the origin of the actions and the users preferred visualization view. For example, it is not the same to add a feature from the graph-view than from the indented list-view. In this case, the division is as follows: graph-view, tree-view, list-view and common Use Cases.

4.3.1. Graph-view relevant Use Cases

The set of next Use Cases are related to the relevant actions that the modeler is allowed to realize interacting with the graph-view. The list of these actions are: *Add feature, Add a hierarchical relation, Add attribute, Add a feature dependency, Delete component, Delete attribute, Delete hierarchical group* and *Grouping hierarchical group with feature/s.* In addition, each description of Use Case contains a screenshot as auxiliary information for the understanding of the flow.

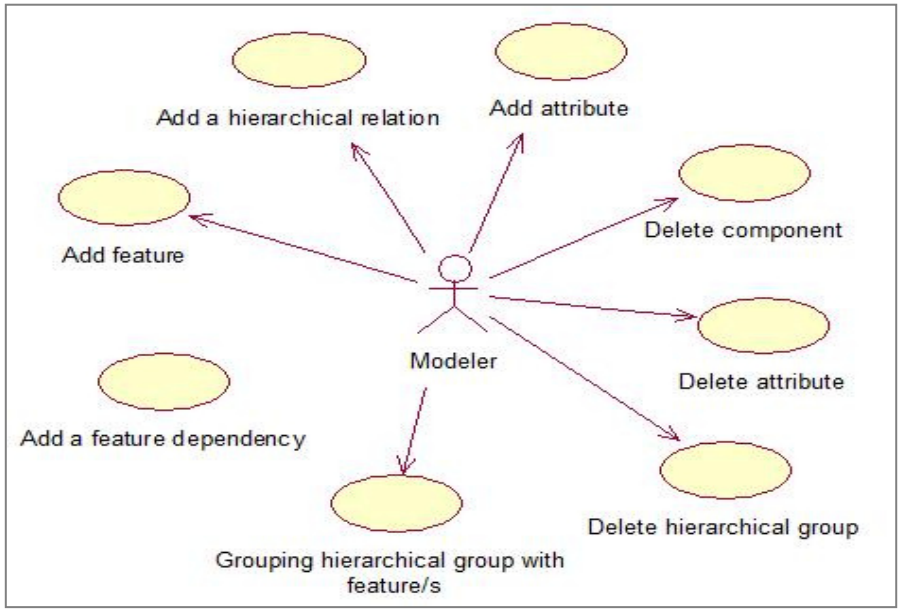


Figure 31. Relevant Use Cases Graph-view

Name: *Add feature*

Description: Add a new representation of a feature in the graph-view.

Actors: Modeler

Preconditions: -

Flow:

1. Modeler clicks on the button 'Add feature'
2. The system shows a representation of the new feature centered in the position of the cursor.
3. The modeler drags the feature representation at the position where he wants to place the final representation, and releases the mouse.
    3.1. If the new feature is situated on another feature, the system recovers the normal state.
    3.2. If the new feature is not situated in the visualized model workspace (cursor outside the representation), the system recovers the normal state.
    3.3. If the new feature is situated in a free space in the visualized model workspace, a new window will appear to insert the properties of the new feature.

Post conditions: The feature is inserted in the model on the same position of the new feature.

Name: *Add a feature dependency*

Description: Add a new representation of dependency between two features in the model

Actors: Modeler

Preconditions: As a minimum, the model has to contain two features.

Flow:

1. User clicks on the button of adding a new dependency.
2. The system changes the cursor that becomes from an arrow to a point.
3. The user selects the feature origin of the link in the representation.
4. The tool shows an auxiliary line with its source as the feature clicked and the destination the position of the cursor.
5. The user selects the destination feature in the model.

5.1. If the destination feature is the same as the source feature, the system recovers the normal state and a new error window will appear to inform the user about this situation.

5.2. If the destination feature is different from the source feature, a new window will appear to select the type of feature dependency.

Post conditions: A new representation of dependency linking the feature origin and the feature destiny is placed on the representation.

Name: *Add attribute*

Description: Add a new feature attribute.

Actors: Modeler.

Preconditions: The model has to contain as a minimum one feature.

Flow:

1. User clicks on the button of adding a new attribute.
2. The system changes the cursor that becomes from an arrow to a point.
3. The user clicks on the feature that he wants to add the attribute to.
4. A new window will appear for completing the addition of the attribute.
5. The user completes the information in the window by filling the name of the attribute, the type, a comparator, and the result of the comparison.

5.1. If exists another attribute related with the same name, the system recovers the normal state and a new error window will appear to inform the user about this situation.

5.2. If the feature has got no attributes related, the system recovers the normal state, the attribute is related to the feature and the *A-button* placed in the box representation of the feature becomes enabled.

Post conditions: The attribute is added in the model related with the selected feature.

Name: *Add a hierarchical relation*

Description: Add a representation of a hierarchical relation (alternative, mandatory, or, optional) in the model.

Actors: Modeler

Preconditions: The model contains features not related in the hierarchy component.

Flow:

1. User clicks on one of the buttons of hierarchically linking two features.
2. The system changes the cursor that becomes from an arrow to a point.
3. The user selects the feature origin of the link.
4. The features from the type related to the link (alternative, mandatory, or, optional) change the fill color. Also the tool shows an auxiliary line which origin is the feature clicked and the destination the position of the cursor.
5. The user clicks on the destination feature of the hierarchical link.
    - 5.1. If the destination feature is already linked by another hierarchical component, the system recovers the normal state and a new error window will appear to inform about this situation.
    - 5.2. If the linking generates a graph representation instead of a tree, the system recovers the normal state and a new error window will appear to inform about this situation.
    - 5.3. If the destination feature belongs to a different type of hierarchy, the system recovers the normal state and a new error window will appear to inform about this situation.
    - 5.4. If the linking and the feature does not meet the requirements set out in points 5.1, 5.2, 5.3; the system recovers the normal state and the link is added in the graph-view.

Post conditions: The link representation is added in the model.

Name: *Delete hierarchical group*

Description: Remove a group related to a hierarchical link from the model and thus the visualization.

Actors: Modeler.

Preconditions: The model has to contain as a minimum one group related to a hierarchical link.

Flow:

1. The user clicks on the name of the group placed in the representation.

2. A new tooltip will appear with the information of the group selected.
3. The user presses the *Delete* button from the new window.
4. A new window will appear for the confirmation of the deletion made by the user.

Post conditions: The group is deleted.

Name: *Delete component*

Description: Remove the representation of a feature constraint or a hierarchical link or a feature in the model.

Actors: Modeler.

Preconditions: The model has to contain as a minimum one link representation or one feature.

Flow:

1. The user clicks on the link/feature representation.
2. The fill color of the link representation becomes red or the feature is selected by the use of four small white-fill auxiliary boxes (as a representation of the selection, placed on the top, bottom, right-side and left-side of the feature representation).
3. The user presses the Delete key from the keyboard.
4. A new window will appear for the confirmation of the deletion made by the user.

Post conditions: The representation of the link or the feature is deleted in the model.

Name: *Delete attribute*

Description: Remove an attribute related to a feature.

Actors: Modeler

Preconditions: The model has to contain as a minimum one attribute related to a feature.

Flow:

1. The user clicks on the A-button enabled from a feature.
2. A new window will appear with the list of attributes related to the feature.

3. The user selects one attribute from the list and clicks on the button delete.
4. A new window will appear for the confirmation of the deletion by the user.
    4.1. If the user confirm the deletion and the feature only contains this attribute related, the system recovers the normal state, the attribute is deleted from the list of attributes and the *A-button* placed on the box becomes disabled.
    4.2. If the user confirms the deletion and the feature contains more than one attribute related, the system recovers the normal state and the attribute is deleted from the list of attributes.

Post conditions: The attribute related to the feature selected is deleted from the model.

Name: *Grouping hierarchical group with features*
Description: Addition of a group relation in a list of features.
Actors: Modeler.
Preconditions: The model has to contain as a minimum one feature with 'or' or 'alternative' as a type and a group created of the correspondent type.
Flow:

1. User selects a feature or a list of features (pressing the Ctrl key).
2. User clicks on the button 'Grouping' from the *Manipulation Bar*.
    2.1. If the features selected are from different types, a new alert window will appear to inform about this situation.
    2.2. If there are selected other components that are not features, a new alert window will appear to inform about this situation.
    2.3.1. If the features selected do not meet the requirements set out in points *2.1* and *2.2*; a new window will appear for selecting the group to relate these features to.
    2.3.2. The user selects a group from the window and click on the 'Accept' button.

2.3.2.1. If the type of the group is not the same as the type of the feature/s selected, a new alert window will appear to inform the user about this situation.

2.3.2.2. Otherwise, the system adds on the endings of the hierarchical links of the feature/s selected.

Post conditions: The representation of the group is added in the model.

4.3.2. List-view relevant Use Cases

The set of next Use Cases are related to the relevant actions that the modeler is allowed to realize interacting with the list-view. The list of these actions is: *Add feature, Delete feature, Delete attribute*.

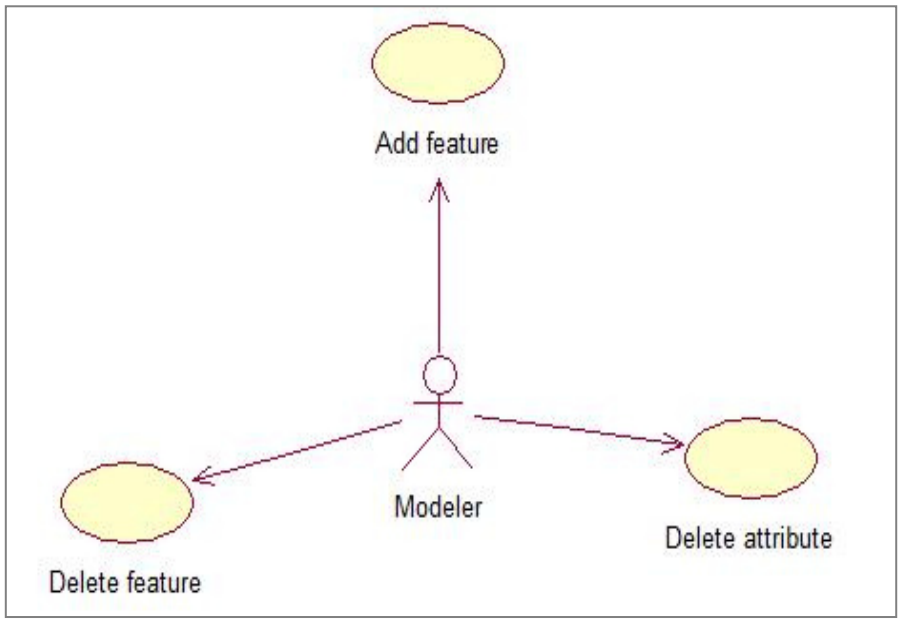


Figure 32. Relevant Use Cases List-view

Name: *Add feature*

Description: Add a new representation of a feature in the list-view.

Actors: Modeler.

Preconditions: None

Flow:

1. User clicks on one node in the indented list that represents a feature.
2. The background of the node changes its color.
3. The user clicks on the *Add Feature* button in the list representation.
4. A new window will appear to insert the properties of the new feature.
5. The user completes the properties of the feature and confirms the information.

Post conditions: The feature is inserted in the model, and the representation of it is added in the list as a new node placed under the node selected previously, indented to the right and containing the name indicated in the properties.

Name: *Delete feature*

Description: Delete the representation of a feature in the list-view.

Actors: Modeler.

Preconditions: The list has to contain as a minimum one node.

Flow:

1. User clicks on one node in the indented list that represents a feature.
2. The background of the node changes the color.
3. The user presses the *Delete* button from the new window.
4. A new alert window will appear to confirm the deletion.
5. The user accepts this confirmation.

Post conditions: The feature is deleted in the model, and the node is removed from the list where its child nodes and their sub-hierarchies become free root nodes moved to level 1 of the component hierarchy in the list-view.

Name: *Delete attribute*

Description: Delete an attribute related to a feature in the list-view.

Actors: Modeler.

Preconditions: The model has to contain as a minimum one attribute related to a feature.

Flow:

1. User clicks on one node in the indented list that represents a feature.
2. The background of the node changes the color.
3. The user clicks on the A-button in the list representation.
4. A new window will appear with the list of attributes related to the feature.
5. The user selects one attribute from the list and clicks on the keyboards delete button.
6. A new window will appear for the confirmation of the deletion by the user.
    - 6.1. If the user confirms the deletion and the feature only contains this attribute, the attribute is deleted from the list and the A-button of the list-view representation becomes disabled.
    - 6.2. If the user confirms the deletion and the feature contains more than one attribute related, the attribute is deleted from the list.

Post conditions: The attribute related to the feature selected is deleted from the visualization and the model.

4.3.3. Tree-view relevant Use Cases

The set of next Use Cases are related to the relevant actions that the modeler is allowed to realize interacting with the tree-view. These actions are: *Add feature, Delete components, Edit feature* and *Visualize feature dependencies*.

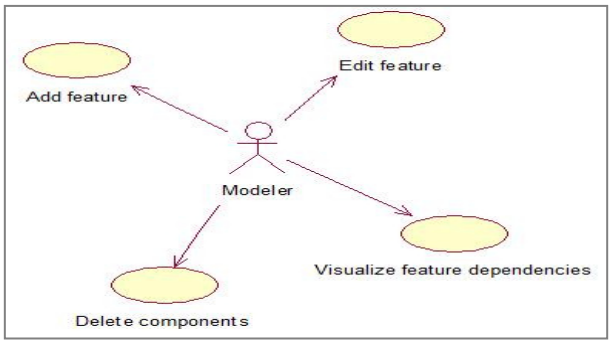


Figure 33. Relevant Use Cases Tree-view

Name: *Add feature*

Description: Add a new representation of a feature in the tree-view.

Actors: Modeler

Preconditions: -

Flow:

1. Modeler clicks on the button 'Add feature'
2. The system shows a representation of the new feature centered in the position of the cursor.
3. The modeler drags the feature representation at the position where he wants to place the final representation, and releases the mouse.
    - 3.1. If the new feature is situated on another feature, the system recovers the normal state.
    - 3.2. If the new feature is not situated in the visualized model workspace (cursor outside the representation), the system recovers the normal state.
    - 3.3. If the new feature is situated in a free space in the visualized model workspace, the user completes the properties of the new feature (name, description, type and father feature) from a new window will appear.
        - 3.3.1. If the tree representation has more than one root node or some features are not connected or dead features exist in the model, a new error window will appear to inform the user about this situation.
        - 3.3.2. If the addition of the feature does not meet the requirement set out in point 3.3.1, the system recovers the normal state and the feature is added in the model.

Post conditions: The feature is inserted in the model on the same position of the new feature and linked hierarchically as a child of the parent indicated in the description.

Name: *Delete component*

Description: Remove the representation of a hierarchical link or a feature in the model.

Actors: Modeler.

Preconditions: The model has to contain as a minimum one hierarchical link representation or one feature.

Flow:

1. The user clicks on the link/feature representation.
2. The fill color of the link representation becomes red or the feature is selected by the use of four auxiliary boxes (placed on the top, bottom, right-side and left-side of the feature representation).
3. The user presses the Delete key from the keyboard.
4. A new window will appear for the confirmation of the deletion by the user.

Post conditions: The representation of the link or the feature is deleted in the model, and all the components related to these representation. In the case the deletion of a hierarchical link creates two different and isolated trees, all the components from the tree that no contains the node are remove from the representation.

Name: *Edit feature*

Description: Change the properties of a feature in the model.

Actors: Modeler.

Preconditions: The model has to contain as a minimum one feature representation.

Flow:

1. The user makes double click on the feature representation.
2. A new window will appear with the properties of the feature (name, description and type).
3. The user modifies the properties and saves the modification.

Post conditions: The properties of the feature are modified in the model. In case the type of the feature changes, the hierarchical link is reloaded with the representation of the new type of relation.

Name: *Visualize feature dependencies*

Description: Navigates to the graph-view to visualize the dependencies of a feature.

Actors: Modeler.

Preconditions: The model has to contain as a minimum two features.

Flow:

1. The user clicks on the D-button of a feature.
2. The tree-view is replaced by the graph-view.

Post conditions: The tree-view window is reloaded by the graph-view, positioned the center of the screen on the position of the feature that the user clicked the D-button.

Chapter 5

Implementation

5.1. RIA as the environment of the tool

With regard to the requirement number 7 (see *Chapter 3.1, Initial requirements analysis*) to avoid that users need to installing software, our tool should preferably operate in a browser. Due to the fact that our tool is ideated as a desktop application, the type of environment used for its implementation was easily chosen to be RIA[4].

There are many advantages in the use of RIA that our tool requires for implementing its functionality. A RIA provides a very viable technology, capable of addressing the problems that we have to deal with [43]. For example, the most important advantage is the existence of a wide range of useful interfaces in the field of visualization.

This is a basic functionality that our tool requires. RIA technologies generally allow constructing graphics on the fly, and some of them can even provide full-motion animations in response to data changes [44].

Although the existence of a large number of technologies for rich internet applications, the final choose for implementing the tool was between: Adobe Flex [45] and Microsoft

[4] Rich Internet applications (RIA) are web applications used for implementing desktop applications running in browsers.

Silverlight [46]. After experimenting and testing some tools from the two environments (basically to check the graphic power), the development environment and language chosen for the development of the tool was Microsoft Silverlight, because of the rich set of graphical functionalities and the big extension of tools that is currently using Silverlight as an environment.

5.1.1. Silverlight

Microsoft Silverlight is a programmable web application plug-in that enables functionalities such as animations, graphics and audio-video, among others [47]. The support of .NET languages provides the programmer the possibility to work with different program languages (e.g. Visual Basic, C#, JavaScript, IronPython, IronRuby, etc).

Silverlight applications are delivered to a browser in a text-based markup language called XAML, the Extensible Application Markup Language, which offers markup capabilities that target user interface creation and programmable object creation. The XAML contains a Canvas, an object type that represents the presentation layer (i.e. the visual information that the application will show).

The user calls the functions from the kernel already implemented by the .NET languages, which modifies the properties of the XAML and the content of the Canvas. In addition, and as a good property, the Canvas provides the developers with a number of rich graphics effects like rotate, scale, skew, translate and matrix transforms.

5.2. Implementation Environment

Our feature modeling tool is defined as a web based application. *Figure 34* shows the architecture, which consists of these elements:

- A web server where the files are hosted. It includes the scripting engine that processes these files or *scripts* and carries out the specific action that they indicate. This server will receive the HTTP requests from the clients and will send by the TCP/IP protocol, and in this order, the following HTTP content to their browsers:

    - HTML + JavaScript: The HTML contains the definition of the Silverlight application, and the JavaScript functions (e.g. function to check if the navigator of the browser has installed Silverlight plug-in, functions to begin and control the installation of Silverlight, creators of object/events …).
    - XAML + .NET Assembly: XAML represents the starting content of the application (the fist image that the application will show in the browser); this content will be modified according to the actions that the user will take during the live of the application. These actions are fired from the interaction between the elements in the scene and the user. The Assembly provides the user with the definition and implementation of all the actions that the user is allowed to call from the tool.

- Different computer terminals used by Internet Users that will send HTTP requests to the web server by typing our URL in the browser. One of these users will be the administrator that will be able to manage the information of the application with a password.

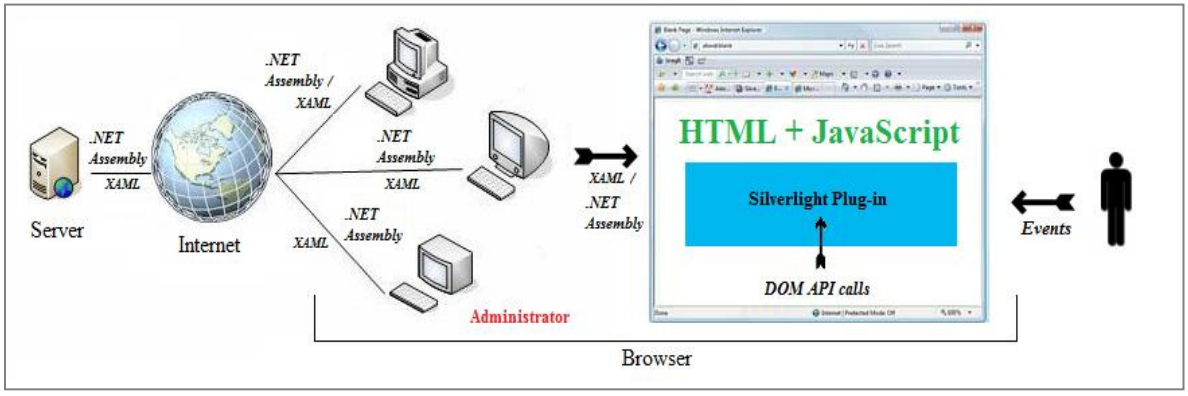


Figure 34. Silverlight Architecture Model

5.3. Software architecture

Figure 35 presents the software architecture of our tool. As already explained, the application is developed as a Silverlight application which runs in a browser or in a virtual machine. The kernel of the application contains all the relevant information that the tool needs for the development or construction of the three different types of views (list-view, graph-view and tree-view).

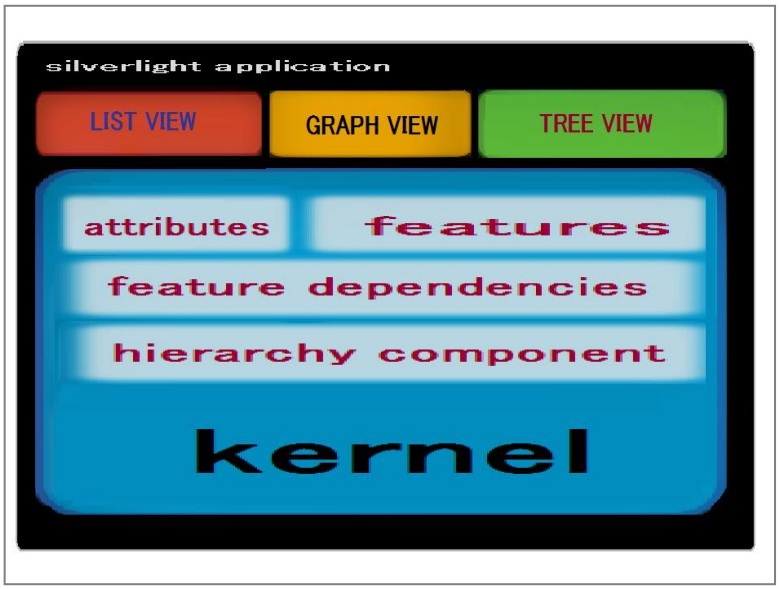


Figure 35. Software Architecture Model

The kernel (domain layer) contains information about the features, its dependencies, its attributes and its hierarchy component; all this information organized and serialized by the use of different data types. From here, each view artifact gets this information for its own convenience, and models this information to create the desired environment displaying.

The modifications of the user are transferred directly from the different views to the kernel, which in turn modifies its content and triggers the events to reload the content of the three artifacts.

Chapter 6
Conclusions

6.1. Summery

The main objective of this thesis was to design a user-friendly application visual interactive application for feature modelling.

Current feature models do not scale very well in real industrial cases were the number of features becomes very large. As the number of features grows, along with the increasing number of relations between features, the need rises to have good visualization tools that allow modelers to quickly and efficiently create scalable feature models that clearly show features and their relations.

The main challenge in this project was to identify from an HCI perspective the best method(s) to visualize feature models such that not only inspect and understand the model but also efficiently create and interact with the model. In addition, the existence of feature modeling tools is not very large and the majority of these tools are designed with regard to support the configure product lines from the feature models. Therefore, the design was more difficult because of the lack of information and examples.

This thesis tried to provide the state of the art, complemented with some own reflections, about the visualization and interaction of feature models. From this research and the design of the tool, we can now answer the following questions:

- Why feature model visualization?
- What are the different possible presentations of feature models?
- How can we benefit from the different type of relations available in the feature model to have multi types of representation?
- Is it possible to have more than one presentation of a feature model available at the same time in a tool and being efficient for feature modeling?
- Applying an HCI perspective in visualizing feature models, how can we make the user experience better?

6.2. Lessons learnt

The design and development of our feature modeling tool and the research done for this has resulted in the following lessons learnt (among others) in the field of feature modeling visualization:

- The design of a feature modeling tool requires a big effort in the study of the requirements and the desires of the stakeholders. A small modification in the requirements represents a variation in the way to visualize the model.
- There exist many techniques adaptable to represent feature models, the common ones are the indented list, the tree and the graph and are the more appropriate for this situation.
- The representation of a feature model needs to take the use of HCI techniques for large big representations into consideration. Zoom and pan are basic techniques that the tool has to provide.
- Diverse types of information require adding to the representation some encoding to distinguish between the different types of information. Some components of a feature models (e.g. dependencies, hierarchical links, features) need the use of such an encoding.

- The inclusion of different visualizations is good for the modeler. From only one view, it is quite difficult to represent all the components of a feature model.
- Different visualizations require synchronization between all of them in the actions taken during the modeling (modification, deletion, addition, selection …etc of components).
- The more interaction between the visualization and the modeler is supported, the easier the modeling will be.
- Rich Internet Applications technology is a good choice to develop a feature modeling tool that should run in a browser, also due to the rich set of graphical functionalities that are already available.

6.3. Future work

Actually the tool is still in the implementation process. At this point, the graph view and the list view are implemented and they operate synchronously. The next step is the implementation of the tree view. Further testing and application of the use tool to a test case is required..

The future work with respect to our feature modeling tool, after finishing the implementation, is the enhancing of the prototype via testing the tool by a set of stakeholders and modelers. From there, we also will get more ideas from the real user experience to enhance the tool and, thus, more conclusions and relevant information that can be taken into account in the field of feature modeling visualization.

In addition, two possible functionalities that our tool could adapt are:

- In the case of saving the model the tool, trigger the checking the consistency of the model (e.g. all features are connected, no dead features exist in the model, the graph model is connected without cycles …).

- Provide a search mechanism by the use of complex queries. Sometimes it is difficult for the modeler to find visually a component in the representation, so it is a good idea to provide our tool with this possibility.
- Study the possibility to add a mechanism for merging different feature models.

References

[1] Kwanwoo Lee, K. C. Kang, and Jaejoon Lee (2002). *Concepts and Guidelines of Feature Modeling for Product Line Software Engineering* (pp 790-784). Korea. Department of Computer Science and Engineering - Pohang University of Science and Technology.

[2] P. Clements, L. Northrop, and L. M. Northrop (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.

[3] H. Gomaa (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley Professional. Part of the Addison-Wesley Object Technology Series.

[4] M. Šípka. *Exploring the Commonality in Feature Modeling Notations*, Bratislava. Slovak University of Technology - Faculty of Informatics and Information Technologies.

[5] D. S. Batory (2005). *Feature Models, Grammars, and Propositional Formulas*. Austin: University of Texas at Austin.

[6] J. Santos (2006). *Software and Factories, and making sense of it for you. FAQ - What is it, and when to use it?*. http://blogs.msdn.com/jezzsa/default.aspx.

[7] M. Riebisch, J. O. Coplien, D. Streitferdt (2003). *Modelling Variability for Object-Oriented Product Lines*. BookOnDemand Publ. Co., Norderstedt.

[8] IEEE Standard for Software Test Documentation. IEEE Std 829-1998

[9] D. S. Batory (2008): *Using modern mathematics as an FOSD modeling language*. Nashville. Generative Programming and Component Engineering.

[10] L. Etxeberria, G. S. Mendieta, L. Belategi (2007). *Modelling Variation in Quality Attributes*. VaMoS 2007:51-59.

[11] K. Czarnecki and U. Eisenecker (2000): *Generative Programming Methods, Tools, and Applications*. Boston: Addison-Wesley.

[12] D. Benavides, P. Trinidad, and A. R. Cortés, *Automated Reasoning on Feature Models*. University of Seville - Seville 2005.

[13] L. Etxeberria, G. Sagardui and L. Belategi (2007). *Modelling variation in quality attributes*. Faculty of Engineering – University of Mondragon.

[14] Lamia Abo Zaid, Geert-Jan Houben, Olga De Troyer and Frederic Kleinermann (2008), *An OWL- Based Approach for Integration in Collaborative Feature Modelling*. 4th Workshop on Semantic Web Enabled Software Engineering - SWESE2008.

[15] Grupo de Investigación en Reutilización y Orientación a Objeto (GIRO) – *Feature Modeling Tool*. Avalaible from: http://www.giro.infor.uva.es/FeatureTool.html

[16] Rubén Fernández, Miguel A. Laguna, Jesús Requejo, Nuria Serrano (2009). *Development of a Feature Modeling Tool using Microsoft DSL Tools*. Department of Computer Science, University of Valladolid.

[17] M. Antkiewicz, K. Czarnecki (2004). *FeaturePlug-in: Feature Modeling Plug-in for Eclipse*. In Eclipse '04: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange, OOPSLA, Vancouver, British Columbia, Canada.

[18] Feature Modeling Plug-in. Generative Software Development Lab – University of Waterloo: http://gsd.uwaterloo.ca/projects/fmp-plug-in/.

[19] Pure-systems GmbH (2008). *pure::variants User's Guide: Version 3.0 for pure::variants 3.0*. Available from: http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf.

[20] Pure-systems GmbH: http://www.pure-systems.com/.

[21] XFeature (P&P Software): http://www.pnp-software.com/XFeature/Home.html.

[22] Eclipse: http://www.eclipse.org/.

[23] Fama Tool Suite (FaMaTS): http://www.isa.us.es/fama/.

[24] David Benavides, Sergio Segura, Pablo Trinidad and Antonio Ruiz-Cortes. *FAMA: Tooling a Framework for the Automated Analysis of Feature Models*. Department of Computer Languages and Systems University of Seville. Seville, Spain

[25] Krzysztof Czarnecki, Chang Hwan Peter Kim: *Feature Models are Views on Ontologies*. Canada: University of Waterloo, Generative Software Development Group

[26] T. R. Gruber (1993): *Towards principles for the design of ontologies used for knowledge sharing*. Stanford University.

[27] Protégé: http://protege.stanford.edu/.

[28] GoSurfer: http://bioinformatics.bioen.uiuc.edu/gosurfer/.

[29] IsaViz: http://www.w3.org/2001/11/IsaViz/.

[30] OntoViz: http://protegewiki.stanford.edu/index.php/OntoViz.

[31] GraphViz: http://www.graphviz.org/.

[32] Gene Ontology Consortium: http://www.geneontology.org/.

[33] GandrKB - A knowledgebase for integrative modeling and access to Microarray annotation data: www.bioinf.mdc-berlin.de/~schober/GandrIntro/.

[34] Joris Klerkx, Erik Duval and Michael Meire. *Using Information Visualization for Accessing Learning Object Repositories*. Computer Science Department, K.U.Leuven. Leuven, Belgium.

[35] World Wide Web Consortium (W3C): http://www.w3.org/.

[36] Ketan Babaria. *Using Treemaps to Visualize Gene Ontologies*. Human Computer Interaction Lab and Institute for Systems Research. University of Maryland, 2001.

[37] Jason L. Baumgartner, Timothy A. Waugh. Roget2000. *A 2D Hyperbolic Tree Visualization of Roget's Thesaurus*. School of Library and Information Science, Indiana University, Bloomington, IN 47405.

[38] Robert Spence. *Information Visualization. Design for Interaction*. Pearson Prentice Hall, Edinburgh, 2007.

[38] Stone, R. B. & Chakrabarti, A. (2005). *Special Issues: Engineering applications of representations of function*. AI EDAM, 19.

[39] Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciar´an Cawley, Patrick Healy. Applying *Visualisation Techniques in Software Product Lines*. Lero, the Irish Software Engineering Research Centre - University of Limerick, Ireland.

[40] Mária Bieliková and Michal Jemala. *Adaptive Incremental Browsing of Ontology Structure*. Institute of Informatics and Software Engineering - Faculty of Informatics and Information Technologies, Slovak University of Technology. Bratislava, Slovakia

[41] Juan C. Dürsteler (2002, Retrieved at 2004). *Focus+Context, Inf@Vis!*. http://www.infovis.net/E-zine/2002/num_85.htm

[42] Thesaurus dictionary: http://www.thefreedictionary.com/.

[43] Joshua Duhl (2003). *WHITE PAPER. Rich Internet Applications* Global Headquarters. Speen Street Framingham, MA 01701 USA. Sponsored by: Macromedia and Intel.

[44] Cameron O'Rourke (2004). *A Look at Rich Internet Applications*. Oracle Technology Network. http://www.oracle.com/technology/oramag/oracle/04-jul/o44dev_trends.html

[45] Adobe Flex: www.adobe.com/es/products/flex/

[46] Microsoft Silverlight: silverlight.net/

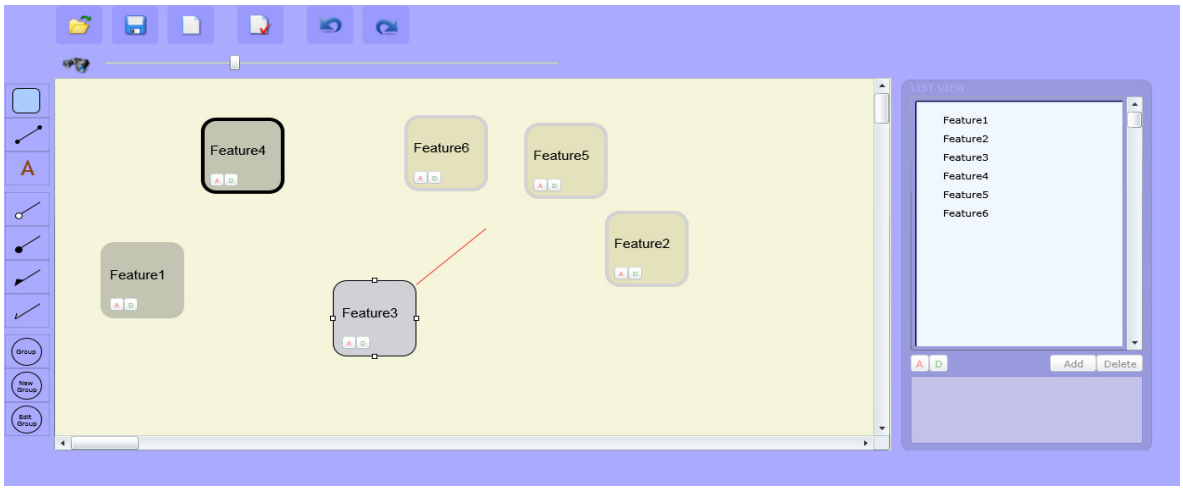
[47] Wikipedia: http://en.wikipedia.org/wiki/Microsoft_Silverlight

[

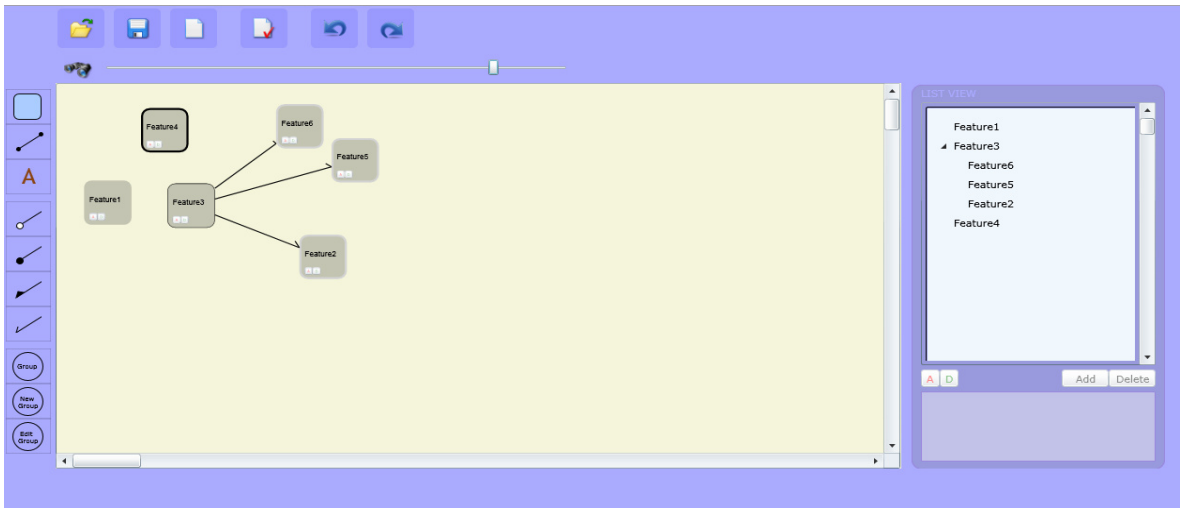
Appendix: Tool Screenshots

Screenshot 1. Addition of a feature

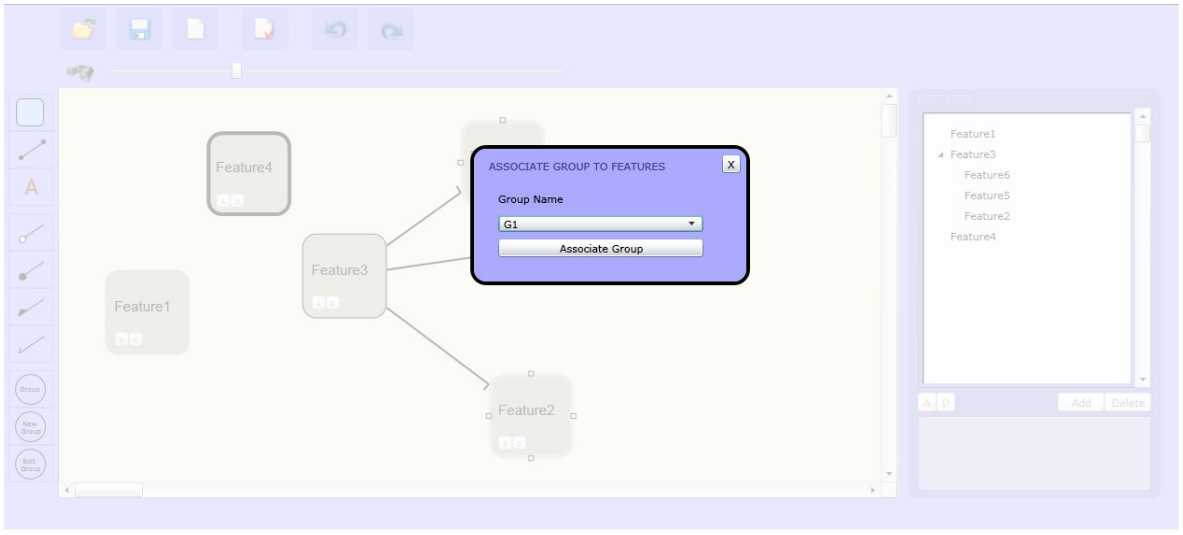
Screenshot 2. Addition of feature information



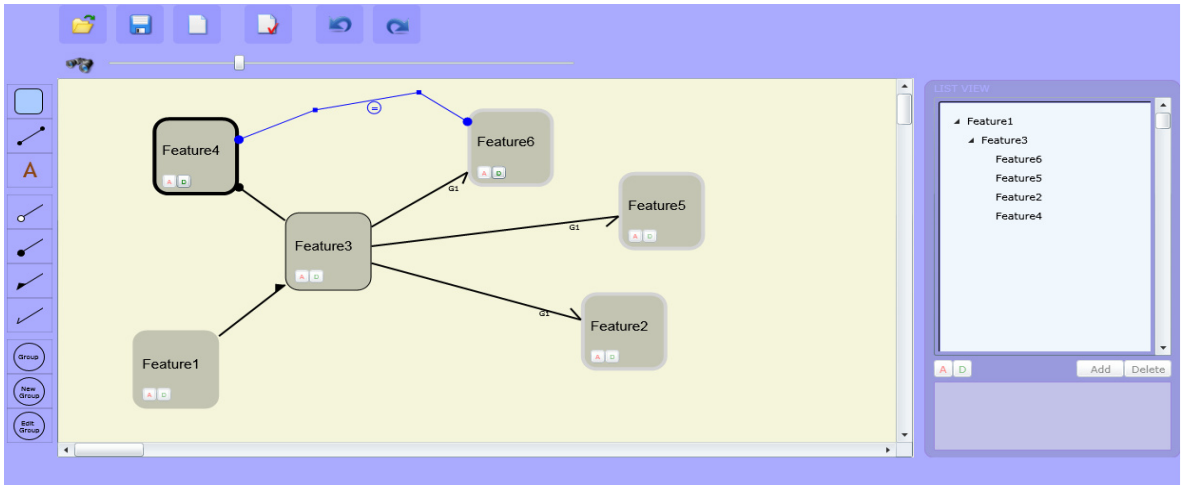
Screenshot 3. Linking alternative hierarchy (3 features possible)



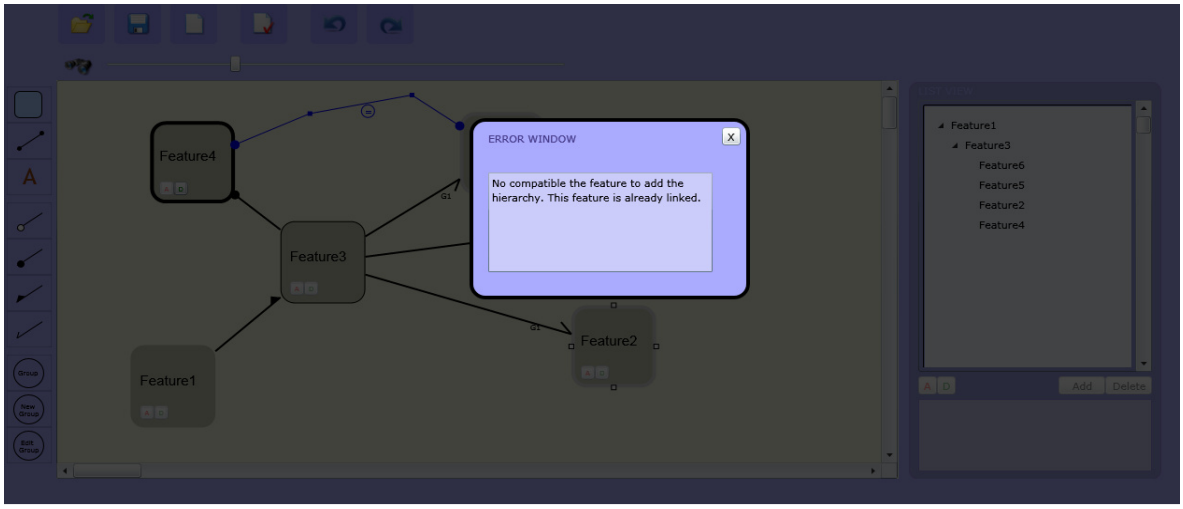
Screenshot 4. Zooming out



Screenshot 5. Associate list of features with a group



Screenshot 6. Addition of a feature dependency



Screenshot 7. Error window example

Screenshot 8. XML representation of the visually created feature model