IT University of Copenhagen

March 2012

Building a product modeller/configurator tool

Abstract:

The OpenConfigurator tool is a modelling and configuration tool that attempts to combine the world of Feature Modeling with that of highly usable and interactive web-based GUI's. The tool is fully browser-based, providing a visual modelling experience through a native SVG implementation. It also provides a configurator part with an interactive GUI where users are given real-time feedback by an engine based on Microsoft's Z3 SMT solver. To date, the rich visual modelling interface implemented as a browser application, as well as the feedback provided by the integration to an SMT solver are both novel.

Student:	Radu Mitache	sign.
Supervisor:	Joseph Roland Kiniry	

Table of Contents

Table of Figures

1. Introduction

Software Product Lines and Product configuration have been gaining more and more attention as companies strive to balance the flexibility in terms of products offered, with the efficiency of their production. This is valid both for software related companies, as well as for any other domains, from physical industrial products such as cars, to more intangible ones such as services or various consultancy jobs.

In order to allow for complex products to be configured, complex rules and constraints must be respected and adhered to. Such rules and constraints, put together, represent a model that describes a product's variability. A Feature Model is a type of model, with every feature describing a distinctive aspect of a product and with relations between the features describing the available product variants.

Though a variety of tools for creating models and creating configurations exist - either as part of the research/software oriented Feature Modeling world, or as part of the industrial/business world in the form of complex product configuration tools – few seem to address issues such as usability or ease-of-use.

In this paper we present the construction of a prototype tool that attempts to address the aforementioned issues, while also building on previous work done in the area of Feature Modeling. The paper is structured as follows: Section 1 gives an overview of related theory and frames the problem. In Section 2 the requirements are presented from high level goals all the way to low level specifications. Section 3 deals with the design process and decisions involved. Section 4 provides some additional details about the development and implementation. Section 5 concludes the paper and section 6 discusses future work.

1.1. Background

Software Product Line Engineering

Software Product Line Engineering (SPLE) is a discipline which allows organizations to develop their products from reusable components, rather than from scratch [1]. SPLE leverages commonalities among the products while managing the differences among them in a systematic way, thus providing a flexible variety of components which can be easily assembled into different a multitude of actual products.

Feature Modeling

Feature modeling is a key approach to capturing and managing commonalities and variabilities in a product line[13]. It was first mentioned in 1990, as part of the Feature-Oriented Domain Analysis method (FODA) [14]. Feature models define features and their usage constraints in product-lines.

A feature is defined as being an increment in product functionality[4]. Current methodologies organize features into a feature diagram (FD), which is used to declaratively specify product-line members [4] as well as other information (constraints and dependency rules) [2]. A feature diagram provides a graphical tree-like notation that shows the hierarchical organization of features.

A particular product-line member is defined by a unique combination of features, with each feature being an increment in program functionality. The set of all legal feature combinations defines the set of valid product-line members [4].

Types of Feature Models

Although there are quite a lot of papers talking about Feature Modeling "standards", there is no single universal standard to which tools and academic terminology adhere to, in terms of how the different relations should be visually represented, or what kind of rules and implications are used precisely.[2]

Initially, in the paper about Feature-Oriented Domain Analysis (FODA), Kang gave the first description of a basic feature model. Features were classified as mandatory, optional and alternative features. An AND-OR graph based notation for representing feature models was also introduced[14].

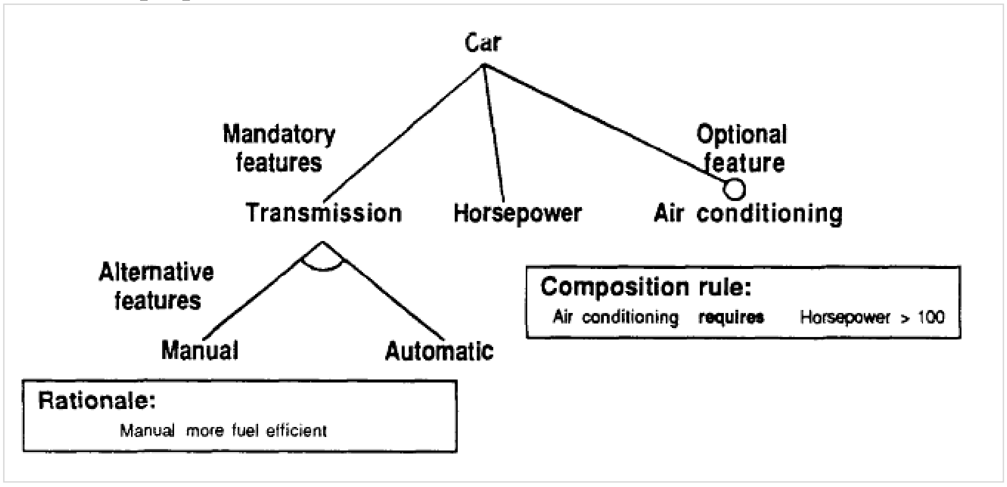
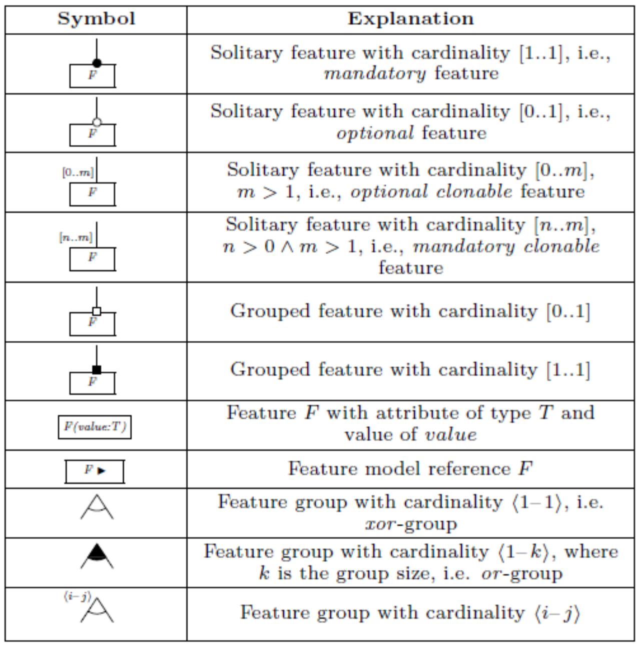


Figure 1 – Example of FODA notation

Czarnecki and Eisenecker later extended the original FODA feature models by adding support for feature and group cardinalities, feature attributes, feature diagram references and user-defined annotations [15]. The table below gives a brief overview of the different notations defined [5].



Symbol	Explanation
F	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
F	Solitary feature with cardinality [0..1], i.e., *optional* feature
[0..m] F	Solitary feature with cardinality [0..m], $m > 1$, i.e., *optional clonable* feature
[n..m] F	Solitary feature with cardinality [n..m], $n > 0 \wedge m > 1$, i.e., *mandatory clonable* feature
F	Grouped feature with cardinality [0..1]
F	Grouped feature with cardinality [1..1]
F(value:T)	Feature F with attribute of type T and value of *value*
F ▸	Feature model reference F
	Feature group with cardinality $\langle 1\text{–}1 \rangle$, i.e. *xor*-group
	Feature group with cardinality $\langle 1\text{–}k \rangle$, where k is the group size, i.e. *or*-group
⟨i–j⟩	Feature group with cardinality $\langle i\text{–}j \rangle$

Figure 2 – Cardinality based feature modeling notation (Czarnecki)

Besides the notations mentioned in the table above, Czarnecki also mentions the need for additional constraints. These can be constraints between cross tree features, such as requires and excludes. They can also be constraints over attributes, or special constraints over cloneable features. [5]

1.2. Problem Statement

When thinking of manually designing a Feature Model with the aid of a software tool, one assumes that the user should be able to project his mental picture of a feature tree representation into the tool. It should be a straightforward and intuitive process, comparable drawing a feature diagram on a piece of paper.

Visual modeling has been gaining ground in other fields of computer science, like for example the emergence of automatic code generation tools based on the visual design of UML diagrams, allowing users to focus on their design from a visionary level, rather than that of a low level programmer writing an endless stream of code.

This project aims to build a tool which supports standard feature modeling functionality, but with focus on the visual interaction concept, all within the context of a modern web 2.0 application in order to make it easier to use and more accessible for non-research oriented users.

1.3. Research related goals

To the best of our knowledge, the development of a feature modeling tool which provides a fully interactive and visual modeling user interface is novel. There are no other tools which offer such a degree of freedom for manipulating and interacting with feature models. Having it built as a web 2.0 application further strengthens the novelty factor.

At the same time, the fact that the core modeler/configurator functionality supports attributes and works by communicating with an underlying SMT solver is likewise novel.

1.4. Demonstration scenario

In this subsection, a demo scenario will be showcased using the OpenConfigurator tool. We consider the following example of a feature model diagram, taken from [16]:

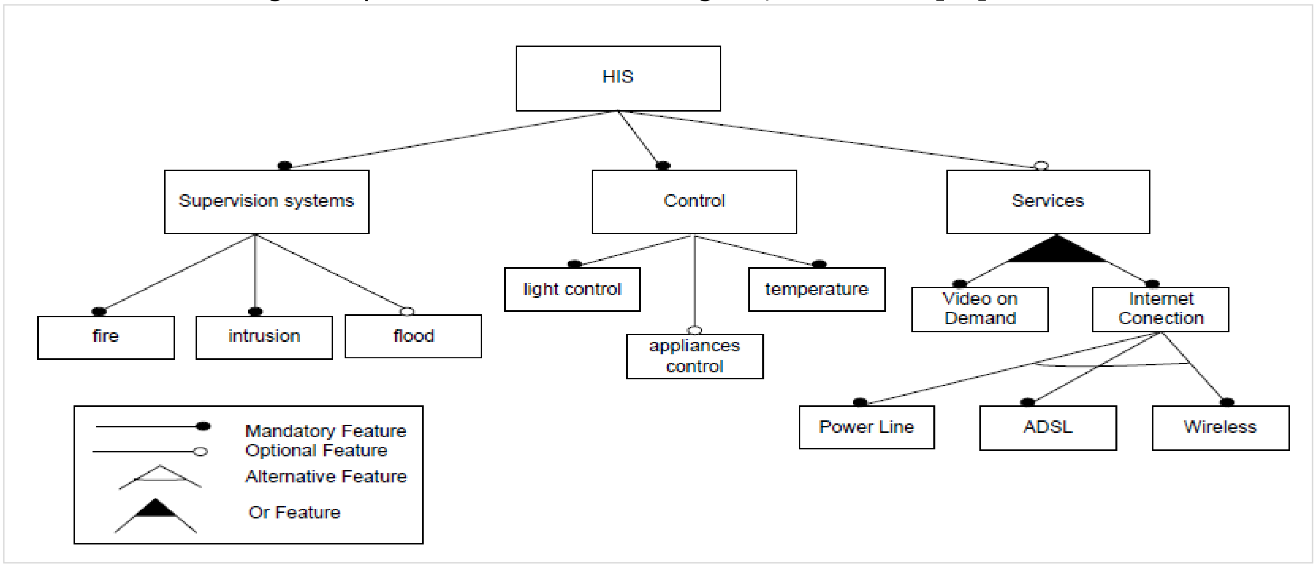
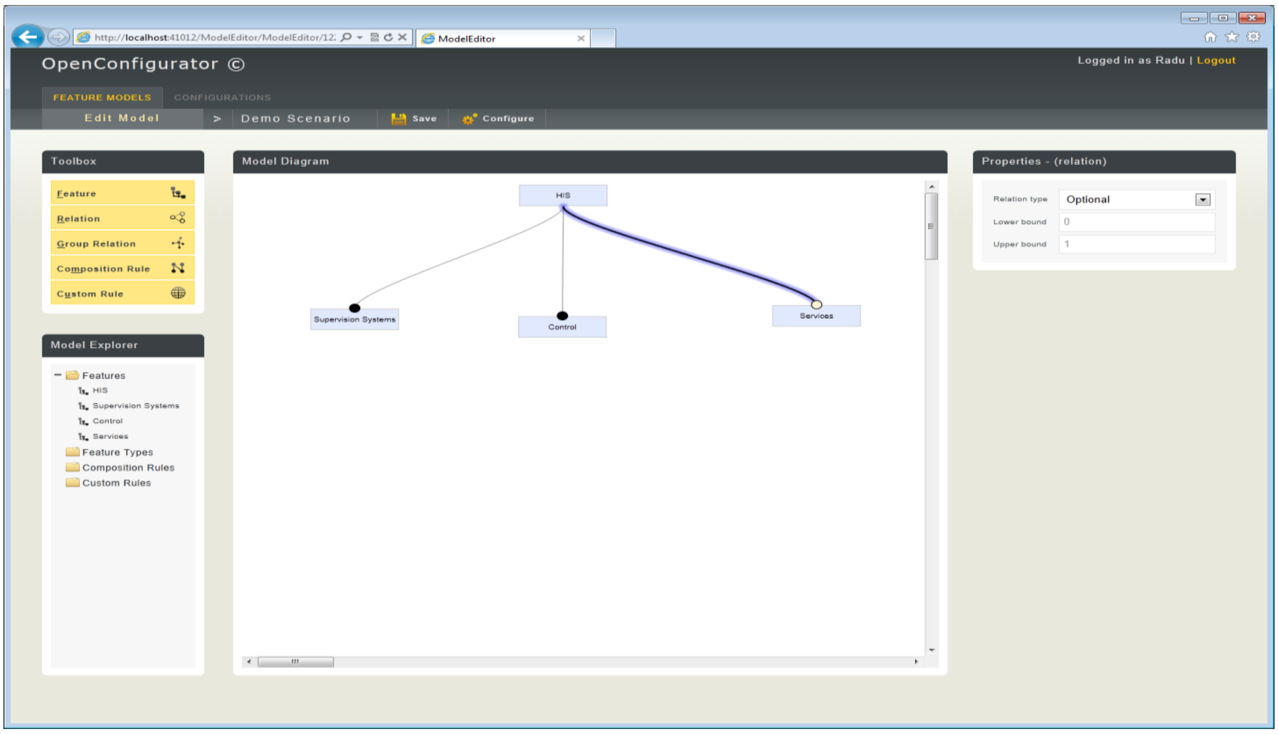


Figure 3 – Sample feature model diagram

The usage scenario will show step-by-step, how the above model is set up using the tool. Followingly, a configuration instance will be configured.

Step 1 – Start model design

The demo begins by creating the root Feature "HIS" and its 3 subfeatures : "Supervision systems", "Control" and "Services". The relations between them are set up, with mandatory relations for the first two and optional for the last.

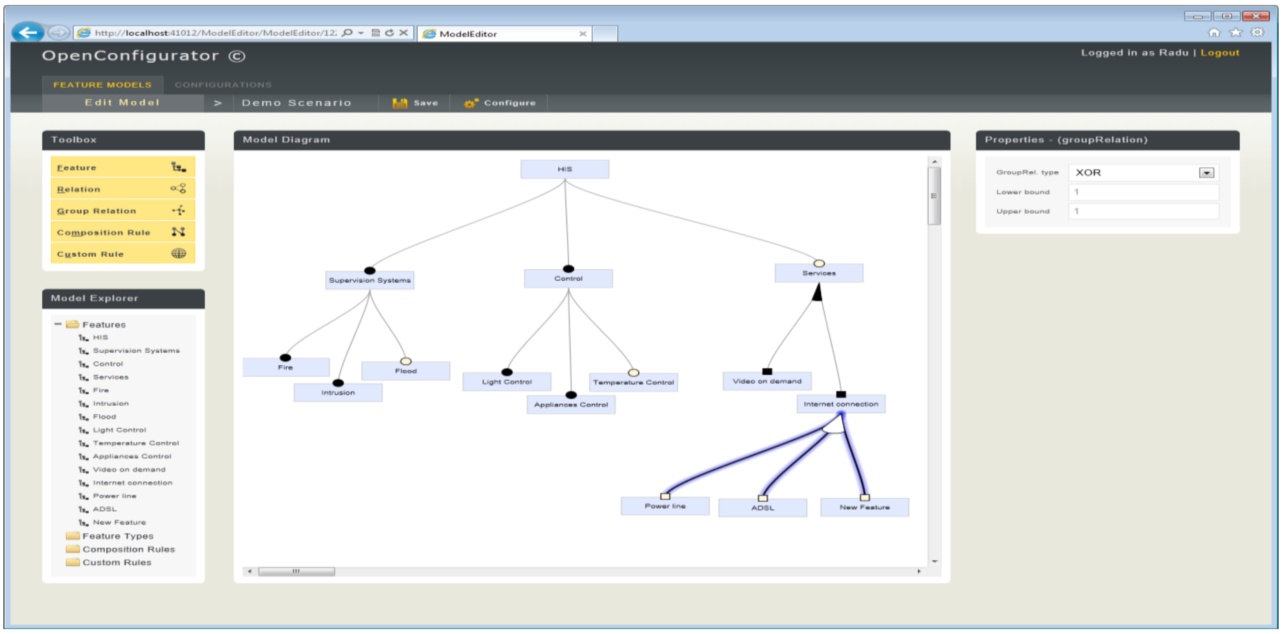


Relation type: Optional
Lower bound: 0
Upper bound: 1

Figure 4 – Initial model setup

Step 2 – Finish basic design

The demo continues with the creation of subfeatures for "Supervision systems" and "Control" , each with their corresponding relations and structure. Then, thesubfeatures for Services are created, with the use of special Group Relations to denote the presence of grouped features. At this point, the model in the original diagram is complete. The demonstration will continue by adding some additional elements and constraints.

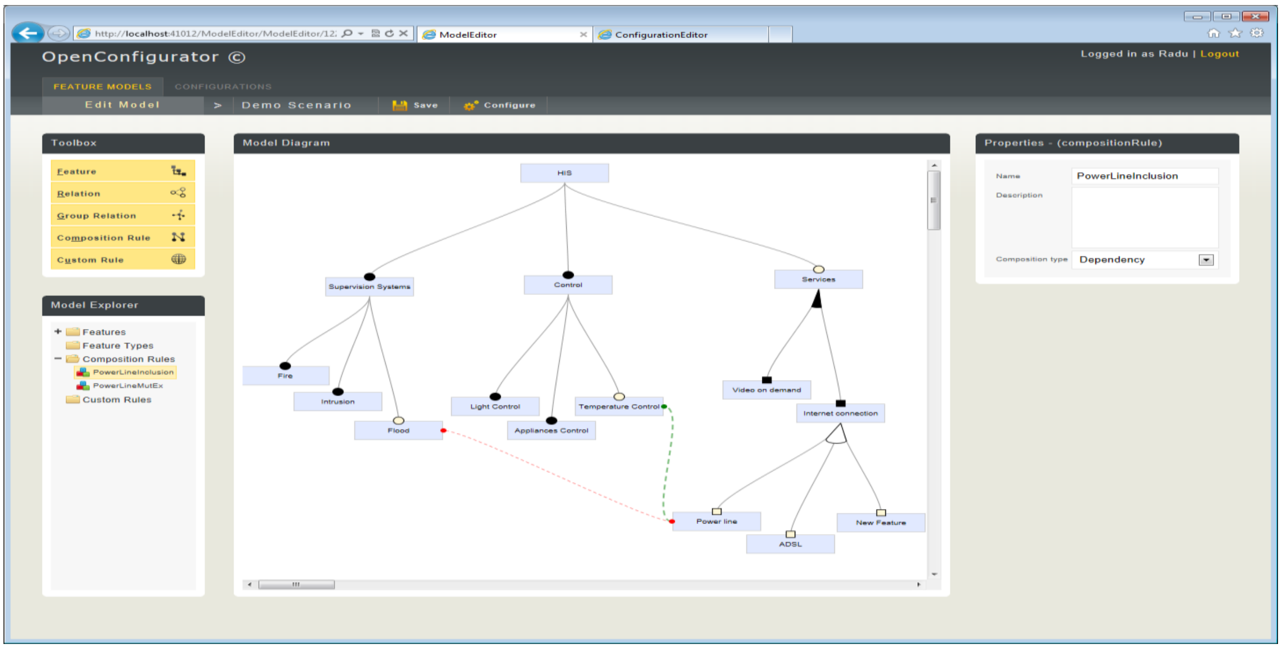


GroupRel. type: XOR
Lower bound: 1
Upper bound: 1

Figure 5 – Complete model

Step 3 – Adding extra flavor (Composition rules)

Two Composition Rules (cross-tree constraints) are added. They signify that the Feature "Power Line" requires "Temperature Control" and also that it is mutually exclusive with "Flood".



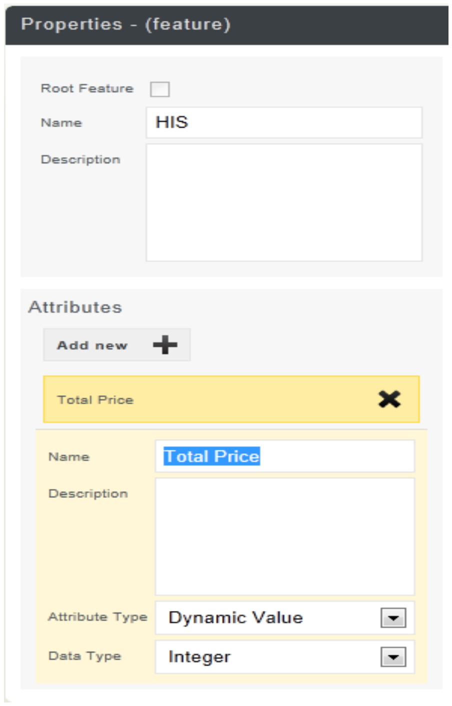
Name: PowerLineInclusion
Description:
Composition type: Dependency

Figure 6 – Composition rules

Step 4 – Adding extra flavor (Attributes and Custom rules)

The complexity of the model is further increased by adding attributes and custom rules.

An attribute "TotalPrice" is added on the root feature "HIS". Another attribute, "IndividualPrice" is added three times, on each of the root feature's direct children features. "TotalPrice" is set as a dynamic Attribute, while "IndividualPrice" is set as a constant. The values for the three "IndividualPrice" constant attributes are set to 130, 135 and 120 respectively.


Properties - (feature)

Root Feature ☐
Name: HIS
Description:

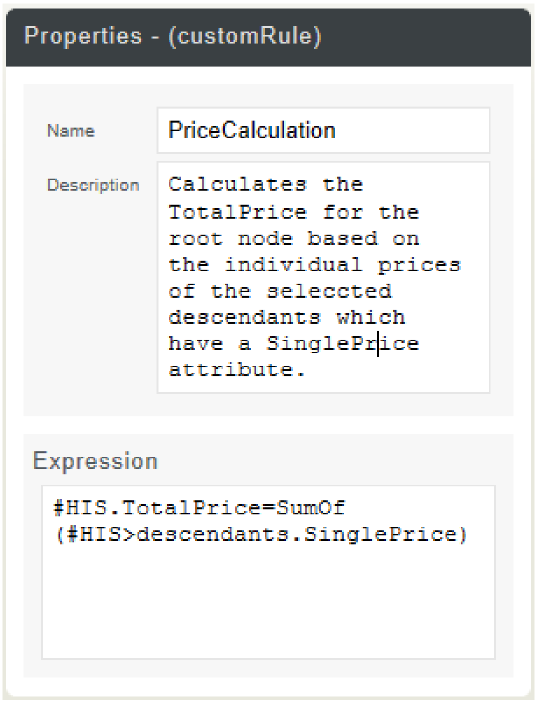
Attributes

Add new

Total Price

Name: Total Price
Description:
Attribute Type: Dynamic Value
Data Type: Integer

Figure 7 – Attribute creation


Properties - (customRule)

Name: PriceCalculation
Description: Calculates the TotalPrice for the root node based on the individual prices of the selected descendants which have a SinglePrice attribute.

Expression

```
#HIS.TotalPrice=SumOf
(#HIS>descendants.SinglePrice)
```

Figure 8 – Complex rule creation

A complex rule is created. The syntax can be seen in Figure 8. It has the role of setting the "TotalPrice" attribute's value to the sum of its selected descendants' "IndividualPrice" attribute value.

Step 5 – Interactive configuration

The model is now set up with both simple and more complex constraints and rules. A configuration instance can now be configured.

The application renders the model on which the configuration is based, in the form of features and groups, with selection states displayed for each feature. Attributes and their respective values are also displayed. The values for attributes which are editable by the user (of type user input) are displayed as editable html textboxes.

Features which are unselected are displayed as gray with an unchecked checkbox, while features which are selected have the checked checkbox as well as a blue colour. They are displayed with an additional colour to signify whether they were toggled by the user or by the configurator. Darker blue if toggled by the user or lighter blue if they were selected by the configurator. Features which were deselected by the configurator appear as light red.

In Figure 9, the configuration is displayed as it looks when it is first loaded. The root feature is automatically selected and implicitly so are a number of other features in order to have a valid configuration (those happen to be the mandatory features). As can be noticed, the value of the attribute "TotalPrice" on the root feature is equal to 265. This is equivalent to the sum of all its selected descendants' "IndividualPrice" attribute value. The selected descendants are "Supervision Systems" and "Control".

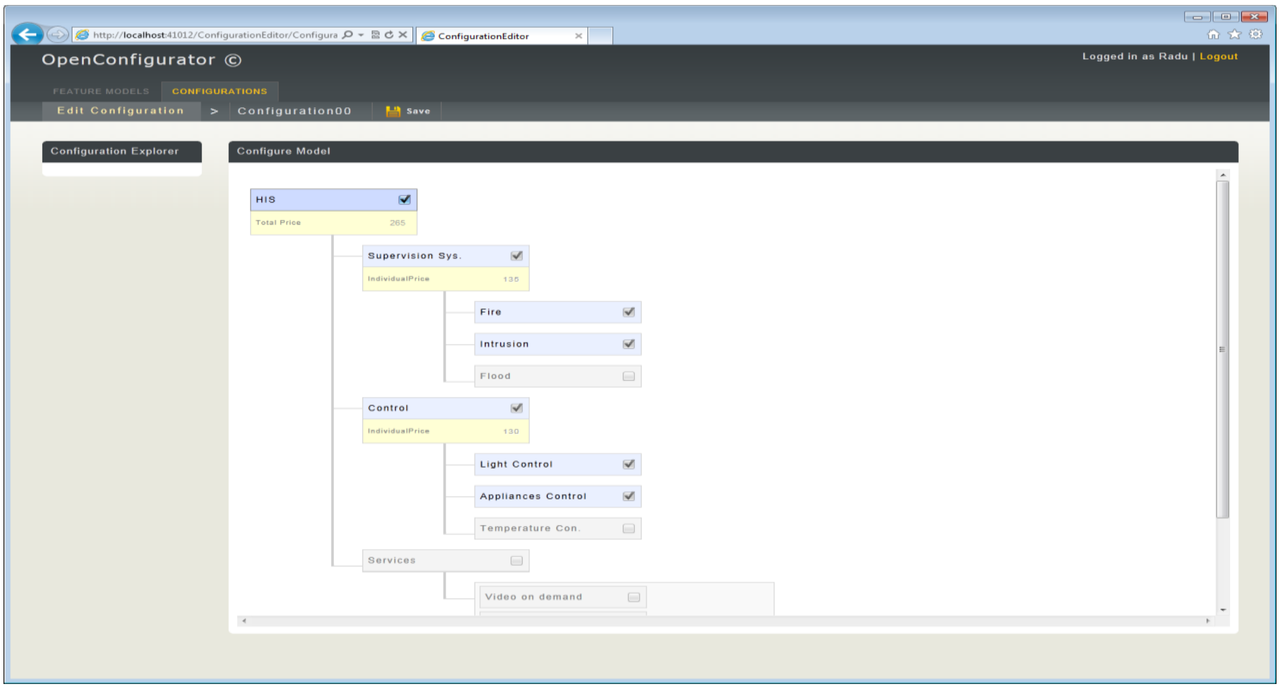


Figure 9 – Interactive configuration 1

"Services" is then selected in Figure 10. It can be noticed that the configurator updates the "TotalPrice" on the root to 395.

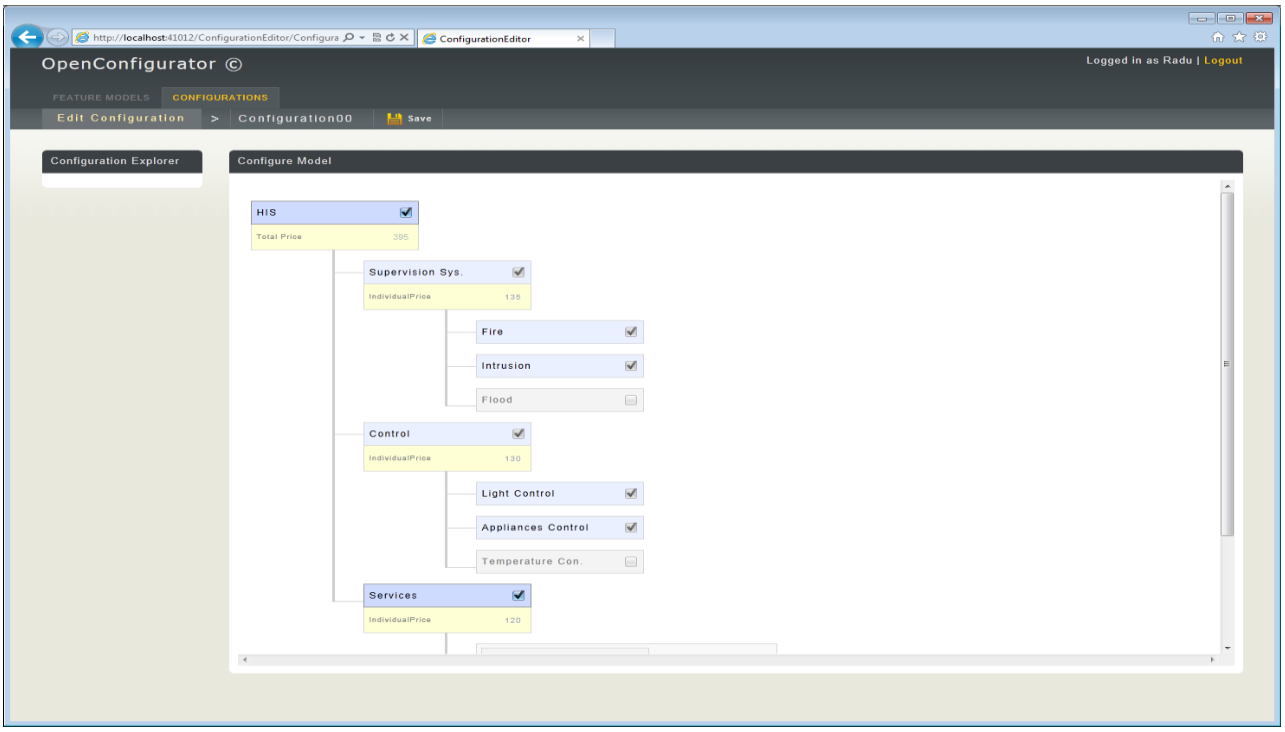


Figure 10 – Interactive configuration 2

By selecting "Power Line" (Figure 11), the configurator enforces the mutual exclusion between it and the feature "Flood". "Flood" is set to unselected and disabled. The configurator also selects "Internet Connection" and disables it. This is done in order for the selection of "Power Line" to be valid.

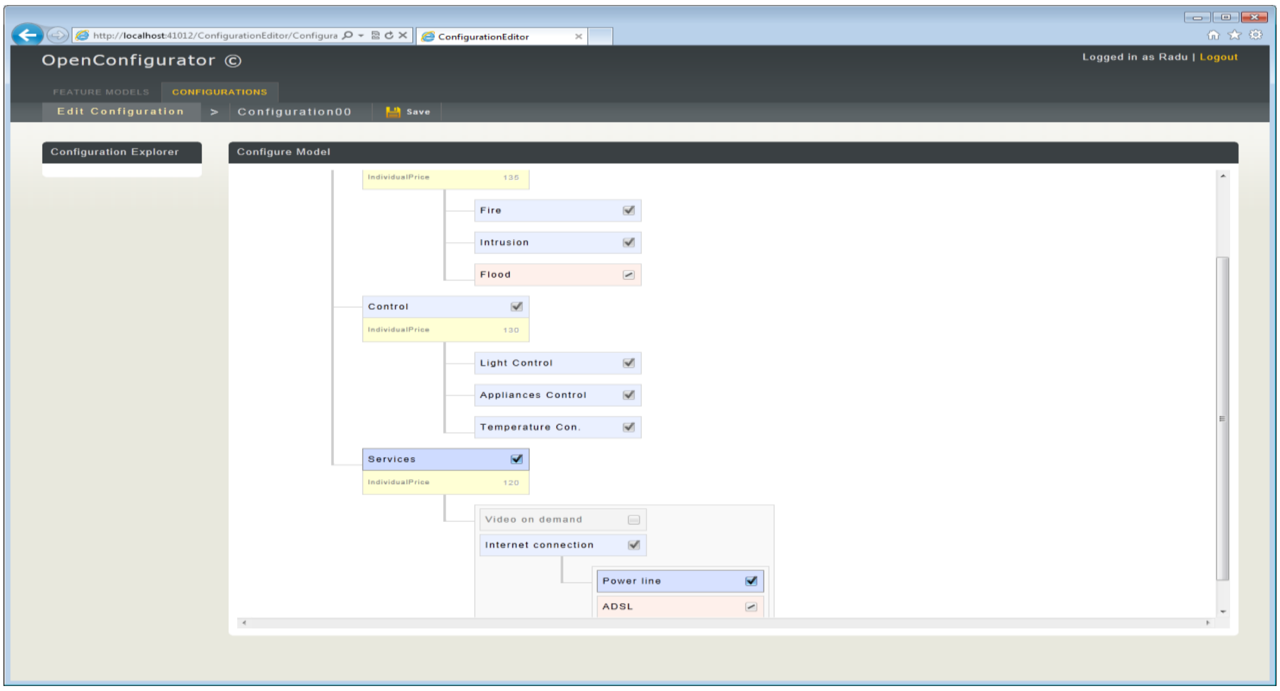


Figure 11 – Interactive configuration 3

At this point, the demonstration concludes. The user could keep selecting features or entering attribute values as desired. Since the tool is only a prototype there is no indication for when a configuration is "complete".

The rest of the report continues in the next section.

2. Requirements

In this section a detailed description of the tool's requirements are presented.

2.1. High level goals

This subsection presents the reader with the key goals of the project, describing them and arguing for their importance and relevance. The goals are presented either in the context of the current project or as part of the wider world of feature modeling.

The high level goals of the project are to successfully create a software tool that provides users with:

- standard feature modeling functionality,
- configurator functionality supported by an underlying SMT solver,
- support for advanced feature modeling primitives such as attributes and complex rules,
- support for visual modeling in a web 2.0 context

Standard feature modeling functionality

The purpose of the project is to build a feature modeling tool, hence an implied requirement being that it should support (at a minimum) standard feature modeling functionality. A definition of the term "standard feature modeling functionality" is therefore necessary, which we discuss below.

As mentioned in subsection 1.1, there are a number of different definitions and types of notations for feature models and their relevant elements. For the scope of this project, after consulting some of the relevant literature [4,13,16,17], the following definitions are presented :

A *feature* is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family [13].

Feature modeling is the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model [17].

A *feature model* represents the common and the variable features of concept instances and the dependencies between the variable features [17].

A *feature diagram* is a tree of features with the root representing a concept (e.g., a software system).

A *standard feature model* can present four types of direct relations between features. Direct refers to their position in the feature diagram, namely that they must be adjacent:

- <u>Mandatory:</u>
  Mandatory features depict a feature which must be part of the configuration if it's parent is selected.

- <u>Optional:</u>
  Optional features signify that a feature can be left out of a configuration regardless of its parent Feature, but it can only be included if it's parent is also included.

- Alternative:
  A child feature in an alternative relation may be present in a configuration if its parent feature is included. In this case, only one feature of the set of children is present. Is also called an *alternative group* in related literature [13].

- Or:
  The child feature in an OR relation may be present in a configuration if its parent is included. Then, at least one feature in the set of children may be present. Called an *or group* in related literature [13].

Furthermore, a *standard feature model* must support two additional types of relations between non-adjacent features. These come in the form of relations which are needed when specifying constraints on features that are part of different branches in the tree:

- Requires:
  Relation which implies that if a certain feature is included, then another corresponding feature must also be included.

- Excludes:
  Relation which means that two features cannot be both included in the configuration.

Configurator functionality with SMT solver

Most tools which include a configurator part that provides real-time feedback (in order to aid users and maintain the validity of the underlying model) have implemented it by either using Binary Decision Diagrams (BDDs), or more recently SAT solvers. The point of the BDDs/solvers is to provide feedback to the user when a feature is selected. The purpose of the feedback is to maintain the validity of the underlying feature model.

The tool presented in this paper (OpenConfigurator) proposes to use an SMT solver. Besides the fact that it is novel, it is interesting since it might open up new paths in interactive configuration due to the capabilities of modern SMT solvers. Additionally, using an SMT solver allows for encoding of attributes and more advanced constraints involving integers (as well as other types of variables) into the solver's mathematical model.

Support for attributes and complex rules

The notion of attributes, used together with feature models, was first introduced by Czarnecki and associates in [18] . According to the respective paper, attributes are considered as a way to further enrich the description of a model by representing product/feature characteristics through the use of values from a large domain, such as integer or string[13]. Other papers chose to simply call them a measurable characteristic belonging to a feature[16].

Looking through the relevant literature, a number of slightly different specifications for how attributes should be handled can be noticed. Czarnecki suggests allowing for features to have an associated attribute type. A collection of attributes can then be modeled as a number of subfeatures, each associated with its corresponding type.[13]

With Czaernecki's stance on attributes as a reference, the following definition of attributes is presented: Attributes are considered to be fields on the features of a feature model, the same way one has a field on a class in object oriented programming (OOP). There is no limit as to how many can be placed on a feature and each of them has a certain datatype and an attribute type.

The datatype refers to the pure data domain of the attribute's value, also called its attribute domain [16]. We convene that it can be of the following three types:

- Integer
- String
- Boolean

The attribute type depicts how the attribute will behave when part of a configuration instance. Again, we convene it is of three types:

- Constant
- Dynamic
- User input

Constant attributes are attributes which have a value set during model creation, but it remains constant throughout any configuration process. Dynamic attributes are those which have a value that can be modified by the configurator, as a result of enforcing various constraints and conditions, while the user is configuring an instance. User input attributes have values which can be manually set by the user during the configuration process.

Having attributes in a feature model gives rise to the need of manipulating or constraining them. This comes in the form of - what we in this paper refer to as - *complex rules*. In [16] they are mentioned as extra-functional features and defined as "a relation between one or more attributes of a Feature".

The interest for attributes and complex rules lies in the fact that they are necessary when working with products which have non-functional specifications.

Support for visual modeling in a highly usable, web 2.0 context

Models are an abstraction which allow human beings to deal with complexity [24]. Their purpose is to portray the essential parts of a complex problem by filtering out non-essential details, thus making the problem easier to understand. While models do not have to be visual, visualization further adds to the minimization of complexity.

Feature models adhere to the same principle. They describe a product line's commonalities and variability while abstracting from other details. They also embrace the notion of visual modeling by providing a visual representation in the form of feature diagrams [4]. One of the main concepts behind feature diagrams is that any feature model can be visually represented regardless of its complexity.

Visual models are represented through modeling languages using different types of notations, symbols and rules. As human beings, whenever dealing with external representations or user interfaces, we make use of an internal mental model. According to [25], the usability of a modeling language is strongly influenced by its ability to support appropriate translations between the user's mental model and the actual visual model [25]. Likewise, the usability of a software tool is influenced by its ability to match the user's mental model. If there are strong discrepancies between the two, users have to use additional cognitive resources to translate back and forth, thus distracting from their original modeling goals [25].

Ideally, an interface design must be consistent with people's natural mental models. For example, it makes sense to design a calculator program that has similar functionality and appearance to the physical hand-held calculators that everyone is familiar with [26].

In the case of feature models, it therefore makes sense to have a software tool which can match the user's mental model of a feature diagram . This can be done by a user interface with visual modeling capabilities. The capabilities should be both in a static sense, with the graphics rendered accurately resembling the feature diagram notations, as well as in an interactive sense by providing users with a lot of freedom in terms of manipulation.

The web 2.0 context refers to having the tool developed as a web-based application with focus on a rich, responsive user interface [27].
The idea behind the web 2.0 context is that it helps a lot in ease-of-use and deployment of the project. This can be noticed as a general trend, with a lot of high profile software companies, who have been moving some of their traditional desktop applications over to the web.

2.2. Requirements specifications

The requirements specifications are detailed requirements derived from the higher level goals. Before going into the actual requirementsl specifications, it is important to present the tool's conceptual class diagram, also called the domain model.

Domain model

The domain model represents the conceptual entities to which the tool is bound to. Some of the important concepts presented in the introduction section and discussed in the high level goals subsection can be noticed in the domain model diagram. These are the feature modeling notions and those related to interactive configuration, which are separated into two different areas in the diagram (Modeler, Configurator).

The conceptual representation of the Modeler area contains the following classes:

- Model:
  Represents a Feature Model. It consists of a number of child elements found in a feature model, as per-defined in previous sections in the paper.

- Feature:
  Represents a Feature in a Feature Model.

- Attribute:
  Represents an Attribute associated with a Feature. Has a DataType (Boolean, Integer or String) and a core AttributeType (Constant, Dynamic or UserInput).

- Relation:
  Represents a parent-to-child relation between two Features. It depicts that the child feature is Mandatory or Optional.

- Group Relation:
  Represents a parent-to-children relation between one Feature and a set of other Features. It can depict that the children are part of an OR or Alternative Group.

- Composition Rule:
  Represents a side constraint in a Feature Model. Can signify dependency, mutual dependency or mutual exclusion.

- Complex Rule:
  Represents a rule which can constrain or manipulate Attributes or Features.

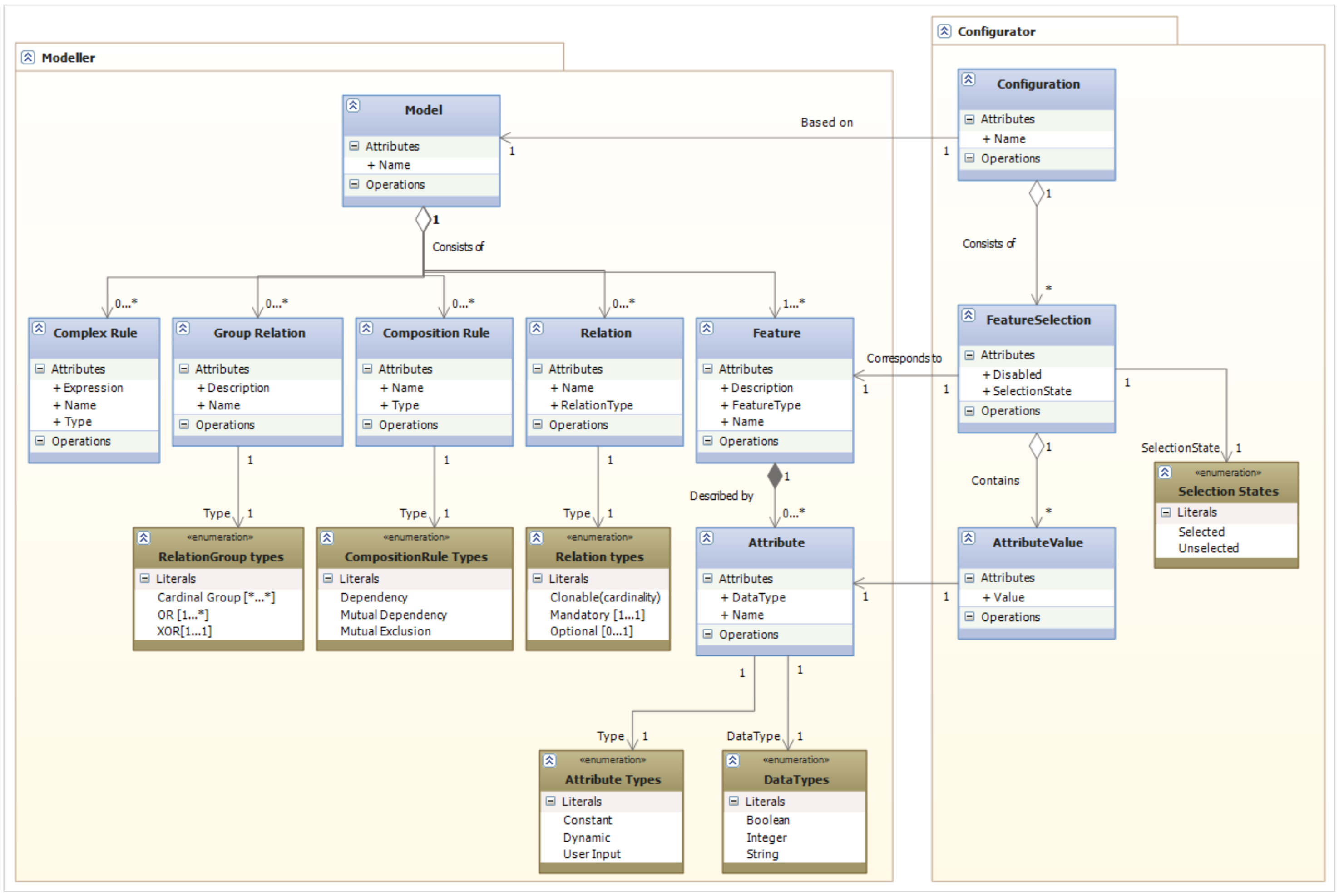


Figure 12 – Domain model diagram

The conceptual representation of the Configurator area contains the following classes:

- Configuration:
  Represents an instance of a Configuration based upon a specific Feature Model. Is made up of a set of FeatureSelections, which together represent the state of the configuration process.

- FeatureSelection:
  Represents the state of a specific Feature part of a Configuration process. Can be either Selected or Unselected. Additionally it can be disabled by the configurator.

- AttributeValue:
  Holds the value associated with a specific Attribute during the configuration process.

Use case diagram

While the domain model displayed a static depiction of the related concepts, the use case diagram displays a dynamic one, with emphasis on it is that the actors should be able to accomplish using the tool. Hence, the tool's functional requirements are sketched.

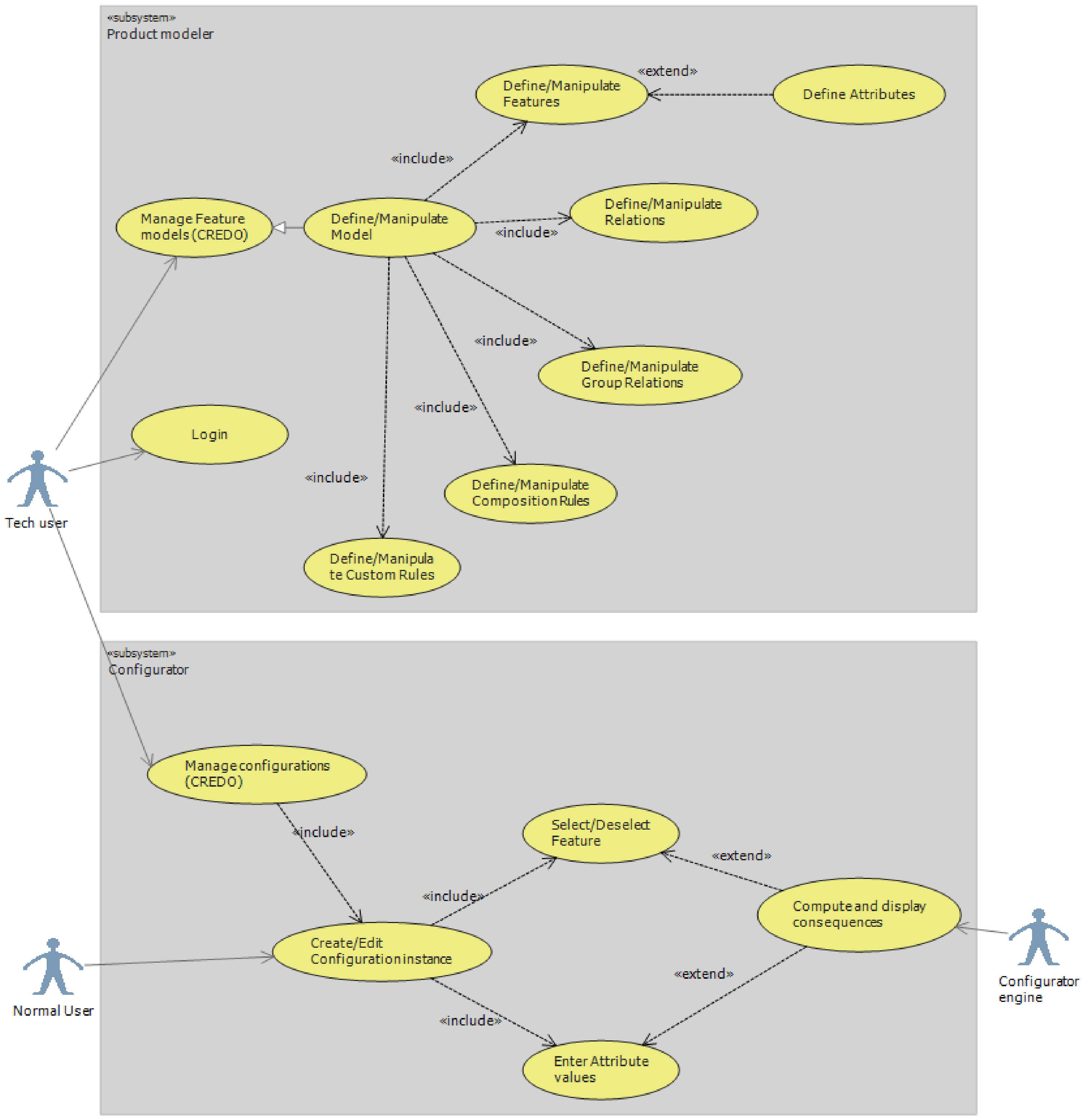

«subsystem»
Product modeler
Define/Manipulate Features
«extend»
Define Attributes
«include»
Manage Feature models (CREDO)
Define/Manipulate Model
Define/Manipulate Relations
«include»
«include»
Define/Manipulate Group Relations
«include»
Login
«include»
Define/Manipulate Composition Rules
Tech user
Define/Manipula te Custom Rules
«subsystem»
Configurator
Manage configurations (CREDO)
«include»
Select/Deselect Feature
«extend»
«include»
Compute and display consequences
Create/Edit Configuration instance
Normal User
«extend»
«include»
Enter Attribute values
Configurator engine

Figure 13 – Use case diagram

As in the domain model, we have a separation into two subsystems, one for the modeler and another for the configurator.

Actors

There are only three actors:

- The tech user, representing an advanced user who is responsible for creating models.

- The normal user, representing a user who does not necessarily need to know about feature models, but who knows what kind of a configuration he wants.
- The Configurator engine, a technical subsystem of its own, which is responsible for providing users with dynamic feedback and consequences as they manipulate configuration instances.

Use case descriptions

Modeller area:
This is where users can create and manipulate Feature Models. It includes a lot of lower level use cases which deal with creating/modifying/deleting Features, creating Relations, GroupRelations, CompositionRules, CustomRules. Below are a few descriptions for some of the more important use cases:

- Login:
  Allows users to access their Models and Configurations.

- Manage Feature models:
  Simple use case where users can get an overview of existing models, delete or open them, or create new ones (CREDO – create, read, edit, delete, overview).

- Create/Manipulate Model:
  Abstract use case made up of a lot of smaller use cases which allow users to manipulate (Create, Read, Update, Delete) different elements in a Feature Model.

Configurator area:
This is where users create and configure Configurations based on Models. Users makes selections of which Features they want, enter values for Attributes and the SolverEngine provides consequences as to what should be disabled and toggled on/off for the semantic model to be valid (satisfiable).

- Manage Configurations:
  Simple use case where users can get an overview of existing configurations, delete or open them, or create new ones.

- Create/Edit Configuration:
  Allows users to interactively configure a Configuration instance. Users can toggle Features on or off and enter values for Attributes.

- Provide Consequences:
  This is where the Solver engine comes into play. Whenever the User toggles a Feature or modifies an Attribute Value, the Solver provides the necessary consequences (toggles Features automatically on or off) in order to keep the Model valid.

Non-functional requirements

- Should be able to work in any modern browser
- Visual and interactive UI for creating models and configuring

Secondary requirements

In the original problem formulation, there were also a number of requirements/goals which unfortunately had to be downprioritized, therefore not making it into the final release. These are listed below and briefly explained.

- Online collaboration
- User defined visual representations for Configurations
- Feature model abstractions, through several layers of models mapped to each other

3. Design

This section describes how the tool was designed.

3.1. Requirements analysis

This subsection presents a brief analysis of some of the requirements. These are requirements which influenced major parts of the overall design or which necessitated thorough consideration due to their complex nature.

Browser based application

This is a non-functional requirement which was derived from one of the higher goals of the project, namely the fact that the application should provide a web 2.0 interface. Having it as a web 2.0 application will make it easier to use and more accessible to people outside of the research domain.

In terms of technical details, a browser based application needs to be developed using some sort of web development framework. There are quite a few such frameworks, each with its own strengths and weaknesses. However, since I had extensive experience from previous projects with Microsoft's ASP.net, the decision was straightforward and did not involve too much consideration.

As for the general architecture, asp.net offers a number of different solutions, from web forms with aspx pages rich in events and widgets, to MVC with customizable Views and innerviews. The choice which was found to be most suitable for this project was MVC since it offers a lot of freedom in terms of controlling exactly what the different Views render to the client's browser.

Highly interactive GUI with visual modeling capabilities

Considering the GUI had to give the user a high degree of freedom as well as a very visual and interactive experience, a powerful and flexible implementation was necessary.

For a standard rich web application project, there are a number of libraries which make development easier and provide such functionality. Among these JQuery stands out as being one of the most lightweight and easy to integrate with any development framework, especially with ASP.net.

For the part regarding a visual modeling interface, the only way this could be achieved inside a browser was either through the use of a 3rd party based implementation (such as Adobe Flash or Flex), or by making use of modern browsers' own rendering capabilities. The latter was chosen due to compatibility issues as well as being a native browser solution, which offered a high degree of control, flexibility and integration with the rest of the application.

Making use of a browser's rendering capabilities means using a combination of Javascript and Canvas or SVG to render graphics directly in the client. Canvas is a pixel based drawing platform supported by all browsers (and recently greatly enhanced in HTML5). Its drawbacks are however that it is purely bitmap based, having no notion of objects or support for manipulating them as such. On the other hand, SVG (scalable vector graphics) is a vector based drawing system supported by most browsers and intended for manipulating scalable vector representations. It has built-in support for a lot of the functionality necessary, such as event handlers for user interaction (click, mouse over events, etc) or vector state handling.

Thus, analyzing the pros and cons of Canvas and SVG, SVG was chosen since it would be more suitable considering it could support a visual modeling interface without too much hassle. As for the specific SVG implementation, this could be done either with pure SVG or with another library. Since the pure SVG implementation would run into browser compatibility issues, a library was chosen (RaphaelJS) which had an advantage over other SVG libraries that it also works on IE 7 and IE 8 (it uses an alternative VML implementation for this).

Standard feature modeling functionality

In order to implement this, users need to be able to create Feature Models involving Features, Relations, GroupRelations and CompositionRules. This is pretty straightforward as the SVG based design should support creating visual representations for each of these objects, while the data itself can be stored in a database.

Advanced feature modeling supporting Attributes and CustomRules

Attributes and CustomRules extend a standard feature model by providing a way to further describe Features as well as a way to create complex syntax based rules. Attributes can be easily added to Features, represented in the UI and stored in the database just like all the other elements.

These rules are either constraints, calculation formula or some other type of rule-based logic. For this a parser is needed which can interpret the text-based syntax and convert it to some sort of execution flow or constraint in the SMT solver.

Interactive Configurator functionality using an SMT solver

This solution for this requirement implies using an SMT solver to provide the feedback. For this there are two issues. Which SMT solver to use ? and how to integrate it with the web application ?

A bit of research into the world of SMT solver tools came with one very strong candidate, Microsoft's Z3. Since it is .net based and provides the required functionality (and more), it was easy to take a quick decision. As for the integration, since it is available as .net it would be straightforward to call it from the webapplication's business layer.

3.2. User interface design

The user interface is an important part of the OpenConfigurator tool. This is directly reflected in one of the aforementioned high level goals, namely that the tool must provide users with a highly usable user interface which exhibits visual modeling capabilities.

This subsection presents some general principles related to Usability and UI design, together with details as to how they were applied when designing the tool's user interface.

3.2.1. Background

Usability

Since one of the high level goals mentions "Usability", we will present the definition of usability according to the International Standardization Organization (ISO). According to ISO 9241-11:1998 usability is formally defined as "The *effectiveness*, *efficiency* and *satisfaction* with which specified users achieve specified goals in particular environments" [20]. Further the Effectiveness points out to "the accuracy and completeness with which specified users can achieve specified goals in particular environments" [20]. Efficiency refers to "the resources expended in relation to the accuracy and completeness of goals achieved" [20]. Satisfaction refers to "the comfort and acceptability of the work system to its users and other people affected by its use" [20].

The three terms presented above can be - according to Søren Lauesen[19] – interpreted into six factors when dealing with software systems. Together, they determine a product's usability [19]:

- Fit for use:
  The system can support the tasks that the user needs to accomplish.

- Ease of learning:
  Depicts how easy it is to learn to use the system by various groups of users.

- Task efficiency:
  Represents how fast or how easy users can accomplish tasks.

- Ease of remembering:
  Depicts how easy is it for occasional users to remember how to use the system.

- Subjective satisfaction:
  Represents how satisfied users are with usage of the system.

- Understandability:
  Represents how easy it is for users to understand the system. Is especially valid for scenarios when errors occur in the system.

General application design

The general application design is relatively simple, adhering to simple best-practices regarding web user interfaces. Based on the functional requirement, we identified the need for a number of application screens (or pages):

- Login:
  Simple screen where users can log on to the core of the application.

- Models overview:
  Screen where users can get an overview of all their models and perform operations such as delete, edit, create.

- Configurations overview:
  Screen where users can get an overview of all their configuration instances and perform operations such as delete, edit, create.

- ModelEditor:
  This is where users manipulate feature models. It is described in detail in the next subsection.

- ConfigurationEditor:
  Screen where users configure instances.

All of the screens in the application (excluding the Login page) inherit from a template screen, which can be seen in the sketch below and to the left.

The template, on the left side, contains a navigation menu and an indicator for a subtitle. On the right side it contains an indicator with the logged in user's name and a button for logging out. In the center it contains an area in which each child screen has its own UI elements. The template provides users with a simple way of navigation through the different pages, as well as a general look and feel identifiable with all parts of the application.

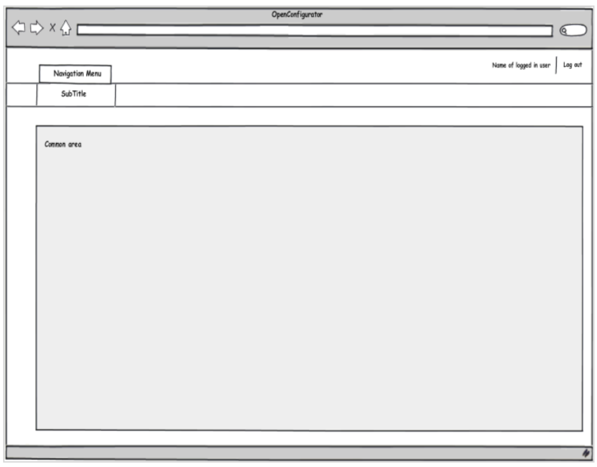


Figure 14 – Screen template

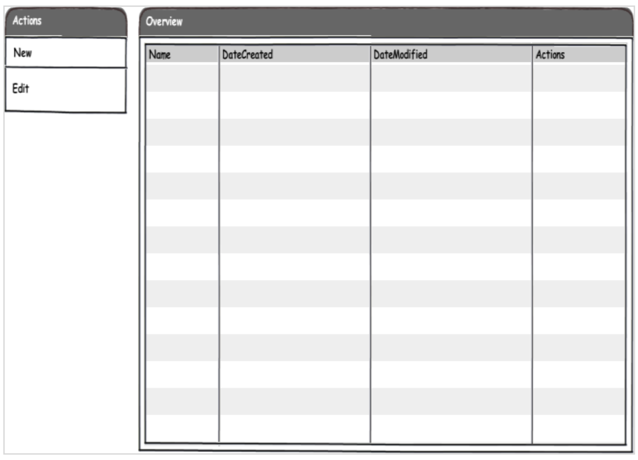


Figure 15 – Controls for overview screens

The sketch to the right shows the design for the Models and Configurations overview screens. They both use a simple two column layout with a datagrid component listing the user's existing Models or Configurations.

3.2.2. Design for ModelEditor screen

Ideally, when designing a user interface which aims for a high level of usability, the entire UI design process must revolve around the target users and the feedback given by them. A lot of user tests have to be performed, resulting in a lot of different UI prototypes being created during the development process [19].

In the case of the OpenConfigurator tool, this wasn’t possible due to lack of time and resources. So instead, we shifted our attention towards high profile applications known for their usability and intuitiveness.

One of the applications which caught our attention was Visual Studio, especially since the 2010 version comes with a flexible UML editor. The UML editor has visual modeling capabilities similar to those we are aiming for in our own tool. It allows for easy creation of visual elements (classes) and relations between them, with users being in full control of how they choose to set up their diagrams.

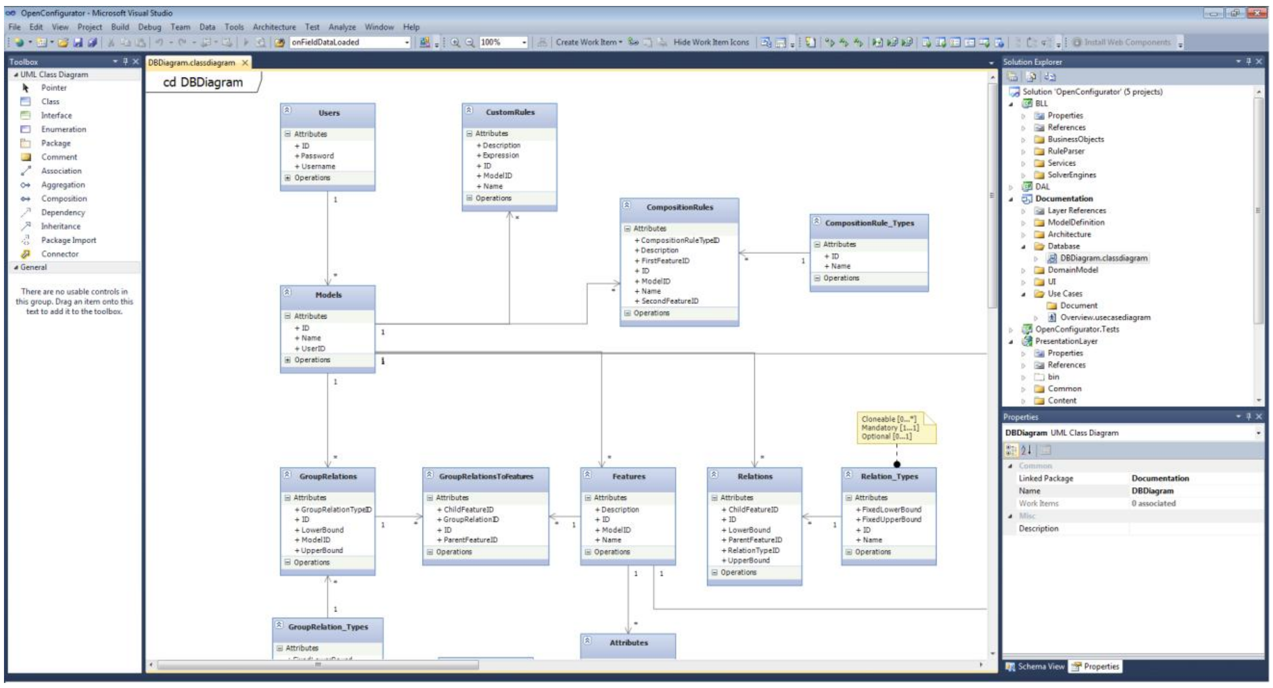


Figure 16 – Visual Studio 2010 UML editor

After a brief analysis, we used decided upon using the following concepts from Visual Studio for our own tool:

- Full screen width, taking advantage of displays with high resolutions by giving the user a very large main view of the visual model.
- Use of toolbox component.
- Explorer component offering a tree-based view for quick navigation.
- Properties component synchronized with the main view for editing elements.
- High level of interactivity in the main view, with easy selection and manipulation (delete, edit, drag move) of visual elements (classes, relations).

The sketch below represents the design for our ModelEditor screen, based on the mentioned concepts from Visual Studio. We split the design up into a number of subcomponents:

- A toolbox component presenting the different elements available (Features, Relations, etc).
- A main View where users can manipulate a visual representation of the feature model diagram.
- An explorer component giving a tree-based structure of the feature model for easier navigation.

- A properties component for displaying/editing details of elements.

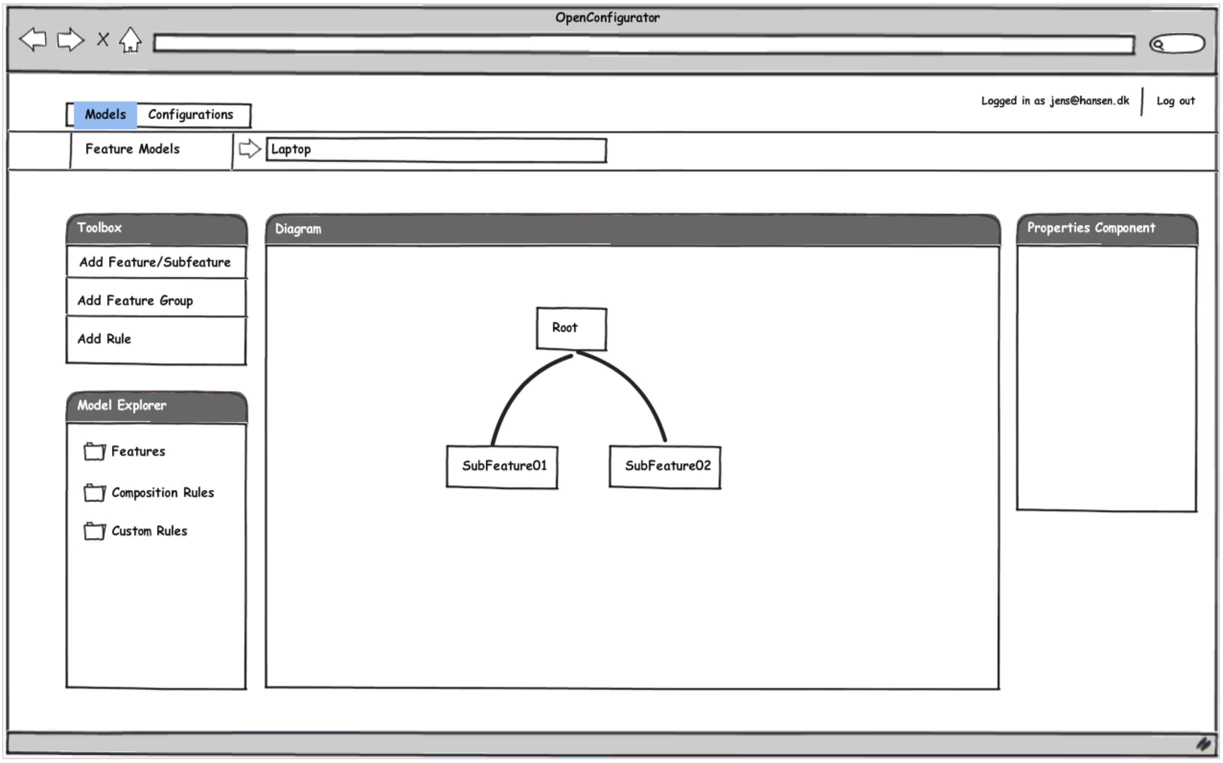


Figure 17 – UI sketch for ModelEditor

The main view is where users can interact with a graphical representation of a feature model. The selectability rules for the elements in the main view were inspired from Visual Studio, allowing for all elements to be selectable (features, relations, groupRelations, compositionRules). The selectability rules allow for operations such as select, deselect, shift + select.

3.3. Architecture

This section presents a detailed description of the underlying architecture, principles and design patterns used and covered. It also presents a low level using UML class diagrams and descriptions.

3.3.1. High level architecture

When designing the architecture, two key principles were considered. These were *modularity* and *extendibility*. In order to achieve a high level of modularity a hybrid between N-Tier and MV was chosen. This would provide a flexible general architecture, with low coupling between the layers as well as being easy to extend by adding additional Views, while also giving full control over what is rendered to the client through the MVC's Controllers.

N-Tier is a design pattern which allows for distributing an application throughout several layers. This enhances the application's modularity, making easier to understand as well as easier to scale in a real-world context.

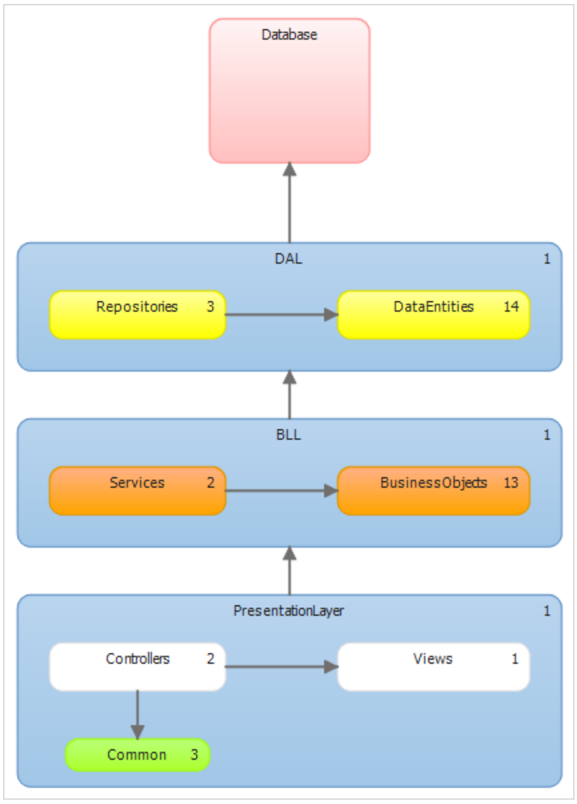


Figure 18 – High level layer diagram

The MVC pattern has been found to be very suitable especially for web applications as it allows for a logical separation of concerns between Model, View and Controllers [21]. This gives a great deal of control over the content that is rendered in the Views as well as making it easier to Test and isolate bugs and problems.

The diagram to the left shows the layered structure of the application. It is split into three logical layers as in a typical N-tier app, however in the Presentation Layer we have Controllers and Views, with the Business Logic Layer acting as the Model.

Having an N-tier MVC hybrid provides the benefits of both worlds, without causing any conflicts as there is no direct coupling between the two implementations.

The representation and handling of data is done by having a set of objects which are passed down from layer to layer.

Each layer has its own set of objects. Thus, as can be noticed in the diagram, there are DataEntities representing pure data objects on the DAL, BusinessObjects representing DataEntities with business logic on the BLL and ClientObjects representing BusinessObjects with UI logic on the PresentationLayer.

3.3.2. Server side

Database

The database design is based on the domain model and use case diagrams, allowing the storage of data necessary to accomplish all of the requirements. Basically the database diagram is an implementation of the domain model into a MSSQL server database, with the implied technicalities of the database.

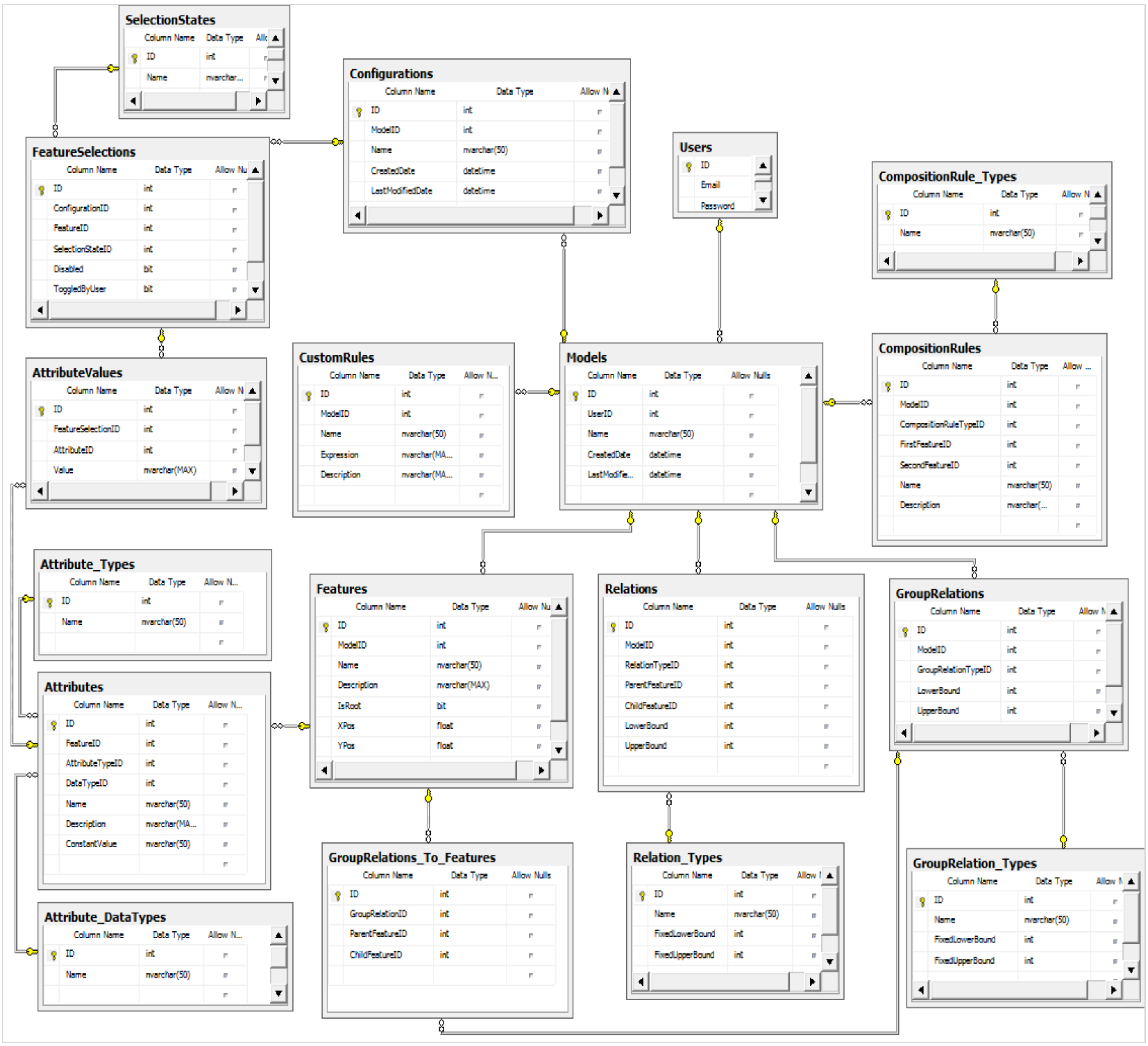


Figure 19 – Database diagram

Data access layer

The DAL is built using an ORM tool, namely Microsoft's Entity Framework. The ORM allows for the mapping of entities to different tables and fields in the database. Based on a diagram, it autogenerates the code for each of the different entities.

The data access layer consists of a number of data objects which are mapped to database tables, but contain no logic of their own. The logic to handle CRUD operations resides in a generic Repository class which is responsible for persistence.

Whenever a DataObject needs to be updated or modified, it is passed to an instance of the corresponding Repository type which then proceeds to interact with the Database and synchronize the changes.

Business Logic Layer

The BLL very closely resembles the initial Domain Model presented in the Requirements section. It basically encompasses all of the Business Logic in the application further splitting it into Business Objects and Services.

The Business Objects are wrappers for the DataObjects in the DAL, with additional business logic added to fields and properties.

The services are responsible for manipulating and retrieving collections of BusinessObjects,. They are also responsible for performing sensitive operations, such as providing feedback to users while interacting with a Configuration instance (subsection 3.2.2).

Additionally the BLL also contains the logic for the application's CustomRule parser, which is responsible for parsing rule text-based syntax and executing the logic.

Presentation Layer

The server side part of the Presentation Layer consists of a number of Controllers handling different Views. The client side consists of the Views and their related Javascript code (section 3.2.3). There is a Controller for each view, each Controller containing methods which are called from the client side code as well as methods called when first loading a view.

3.3.3. Client side

The client side part of the application is where the bulk of the application's functional code resides. It is made up of high level views through which users interact with the application's core business logic. Each view corresponds to an html page accessible through HTTP and each View consists of HTML and Javascript code (written using Jquery syntax).

A brief overview of all the views:

- Login:
  Page where users can log in to the application.
- Models:
  Page where users can see an overview of existing Models, edit/delete them or create new ones.
- ModelEditor:
  Where Models are visually manipulated
- Configurations:
  Page where users can see an overview of existing Configurations, edit/delete them or create new ones.
- ConfigurationEditor:
  Page where users can create interactive Configuration instances.

3.3.4. Model Editor and visual modelling

The Model Editor is the part of the application which provides the user with the ability to visually design feature model diagrams.

Its architecture contains code both on the client as well as on the server, though the bulk of the code resides on the client. The server side code basically consists of a Controller which has methods through which the client-side code can retrieve or persist data to the Database by going through the application layers (BLL and DAL).

The design for the client side part of the Model Editor consists of a number of components set up in an MVC pattern. This means that there is a data Model, a Controller and a number of Views, all present solely on the client and implemented in Javascript. This design allows us to limit the amount of data that would have to be fetched and sent back and forth from the client to the server every time the client interacts with the model.

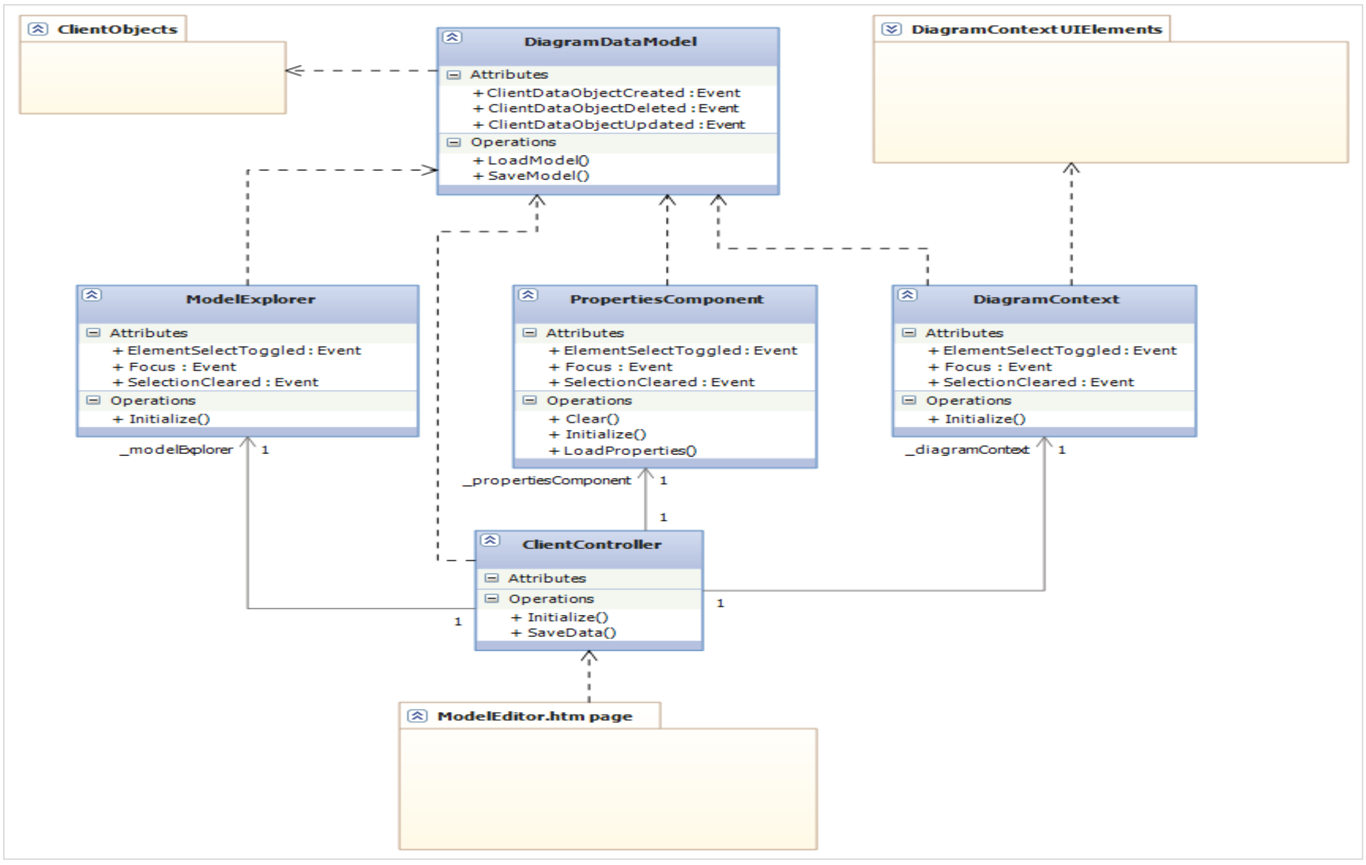


Figure 20 – ModelEditor client side class diagram

The client-side Model in the MVC is responsible for handling the data as well as interacting with the server side Controller to retrieve or persist data to the Database. It consists of a set of ClientObjects, which are basically wrappers for BusinessObjects with additional UI logic. This UI logic includes an object identification system, enabling the Client to use its own ID's instead of those coming from the database. This is necessary in order to be able to identify temporary objects which do not have database ID's, thus abstracting persistence details from the client-side.

The diagram shows the detailed structure of the inner MVC design, highlighting the dependencies between the components. The Views call the Model when the user modifies data, the Model indirectly refreshing the Views by notifying them that its data has changed, with the Controller being responsible for setting up synchronization events between the Views and between the Views and Model.

The components in the design are listed below, with brief descriptions:

- DiagramDataModel:
  This is the client-side Model. It contains the raw client data, consisting of ClientObjects, as well as all the methods to manipulate this data and sync it with the data on the server. It is responsible for keeping track of ClientObjects and their unique ID's.

- ClientObjects:
  These are the objects which wrap BusinessObjects with additional UI level logic. They correspond to Features, Attributes, Relations, GroupRelations, CompositionRules and CustomRules.

- ModelExplorer:
  This is one of the Views. It provides a tree-based representation of the Model allowing for easy navigation to Features and other objects.

- PropertiesComponent:
  This is also a View, but one that displays objects in detail. It allows users to modify specific properties on objects.

- DiagramContext:
  This is the most advanced View which offers a visually interactive graphical representation of the model. It has its own set of objects, called UIElements, which wrap graphical logic onto the existing ClientObjects.

- UIElements:
  This is a special collection of objects used by the DiagramContext view. They each encapsulate an element in a feature model diagram, providing their behavior and graphical representation in the visual context.

- ClientController:
  The controller is responsible for handling inputs from the user and calling the appropriate Views or Model methods. It is also responsible for setting up the Event/Eventhandler connections between Views and Model.

- ModelEditor.htm page:
  This is the simple htm.page, consisting of static html markup as well as a few lightweight scripts.

The design relies on a publisher/subscriber pattern for the communication between the different components. It is implemented as an Event/Eventhandler pattern in Javascript and it allows for low coupling between the components while adding a layer of indirection.

Using this event/eventhandler pattern, the View's are synchronized with each other, so whenever something is selected in one of them, the other two have the same selection. They also receive updates from the Model, who raises the corresponding events, of which the Views have set their EventHandlers on.

PropertiesComponent

The PropertiesComponent is a View which allows users to modify the individual properties of UIElement. Using it, users can for example modifying the properties of a Feature or its Attributes.

It contains a set of different controls which encapsulate HTML rendering logic as well as simple events. The control types are:

- Textbox
- Textarea
- Checkbox
- Dropdown
- Composite

These can be used to specify what UI the PropertiesComponent should display when a certain type of ClientObject is selected. The controls are mapped to the actual data fields of a ClientObject and will take care of loading data or notifying the PropertiesComponent that the user has updated it.

For each different type of ClientObject that can be edited, the PropertiesComponent contains a set of details of which controls are mapped to which fields. The UI specifications are made up of a list of mappings, between data fields and control types.

DiagramContext

The DiagramContext is the most important of the views, being responsible for visually rendering a feature model as well as allowing users to interact with it. The design principle behind it is that it should behave like a vector based diagramming tool (such as Visual Studio's UML editor for), where elements can be selected, moved and manipulated.

It contains its own set of special objects (UIElements), which are correspond to the ClientObjects that are visually represented .It keeps track of its UIElements and takes care of operations such as creating/deleting/updating existing instances, as well as sending updates to the Model or receiving them from the it.

This graphical representation is rendered with the help of the RaphaelJS library built on top of SVG.

UIElements

The UIElements are objects used by the DiagramContext view to provide an interactive visual modeling interface. They are split into a number of subclasses, which can be seen in the diagram below.

Even though SVG with RaphaelJS provides quite a lot in terms of manipulating graphical objects and handling their state, it still has some limitations. These limitations include issues with handling composite SVG elements. RaphaelJS and SVG cannot handle grouped elements which consist of multiple child elements.

Thus, simple operations such as moving a UIFeature, need some sort of handler which knows how to move all of its children and keep their position synchronized. For example it doesn't handle composite objects well enough, like having a text label inside a box and then moving them both around. Therefore, the UIElement objects encapsulate the logic that can keep track of their position and structure into their super class.

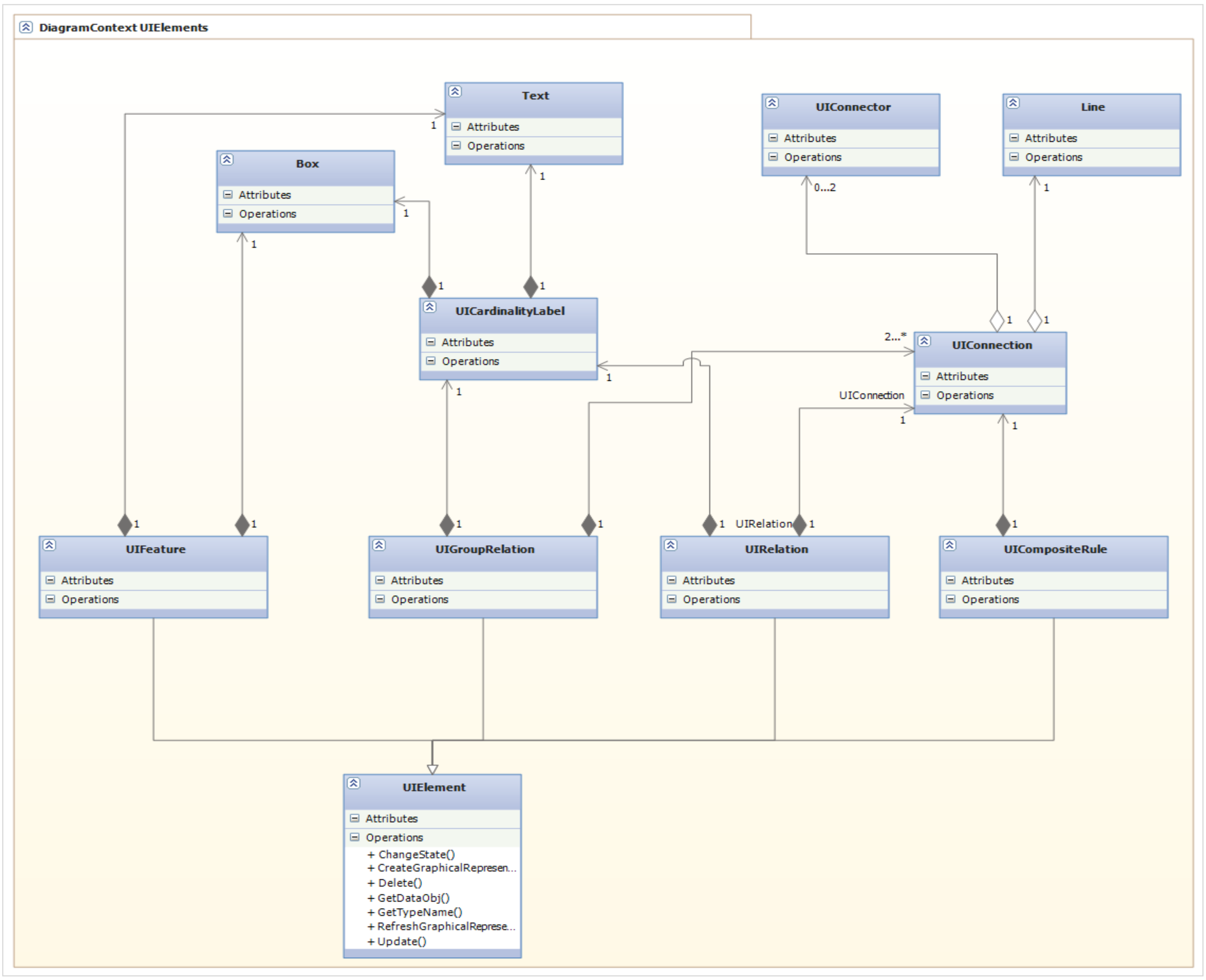


Figure 21 – Class diagram for DiagramContext's inner UIElements classes

The UIElements present in the diagram above can be split into three categories. The first category is the composite objects which represent a whole element in a feature model diagram:

- UIElement:
  Super class from which the other elements inherit their base behavior. Since Javascript inheritance isn't very much in tone with OOP principles, it is basically a guideline sort of inheritance. This means that the elements that "inherit" from it actually just duplicate the code.

- UIFeature:
  Represents a Feature. Keeps track of which Relations or GroupRelations it is part of. Built using a Box and Text as its inner elements.

- UIRelation:
  Represents a relation between two Features, can be either mandatory or optional. Built using a UIConnection and one or two UIConnectors. Can also have a UICardinalityLabel.

- UIGroupRelation:

Represents a relation which denotes a feature group, between one parent Feature and multiple child Features. Built using multiple UIConnections and UIConnectors, as well as a

- UICompositeRule:
  Represents a cross tree constraint. Is represented as a dashed line and is built using a Line and one or two UIConnectors.

The second category is made up of objects which are used as subelements for the composite UIElements:

- UICardinalityLabel:
  Used for displaying numeric cardinality by UIRelations and UIGroupRelations.
- UIConnection:
  Used for displaying a connection between two features. The connection is a line which is drawn using an algorithm which calculates the closest distance between the two Features. Each Feature has 4 connection points (N, E, S, W) at the middle of its four margins.
- UIConnector:
  Simple element which can depict a connector for the end of a line. Example, the empty/full circle depicting an optional/mandatory feature.

The last category are simple elements made up of pure SVG objects. These are Box, Text and Line.

3.3.5. Configuration Editor and integration with Z3 SMT solver

The ConfigurationEditor is a page which is responsible for allowing the user to perform an interactive configuration based upon a model of choice. It displays a visual overview of a Feature Model with selections and values for Features and Attributes (in the form of FeatureSelections and AttributeValues). Users can manipulate this representation by toggling Features on/off or entering values for Attributes. The underlying configurator responds by providing dynamic consequences through the use of a backend SMT solver.

Client side

On the client side we have a design similar to that of the ModelEditor page. We have an MVC pattern whose components reside entirely on the client.

All of the code related to the UI lies on the client side. It's architecture is very similar to that of the Model Editor, as it also has a client based MVC. It differs from the ModelEditor in terms of how the UIElements are rendered and what kind of ClientObjects it uses. The graphical representation of Features and their selection state (taken as a whole represent a Configuration) is realized through simple HTML rendering rather than using SVG.

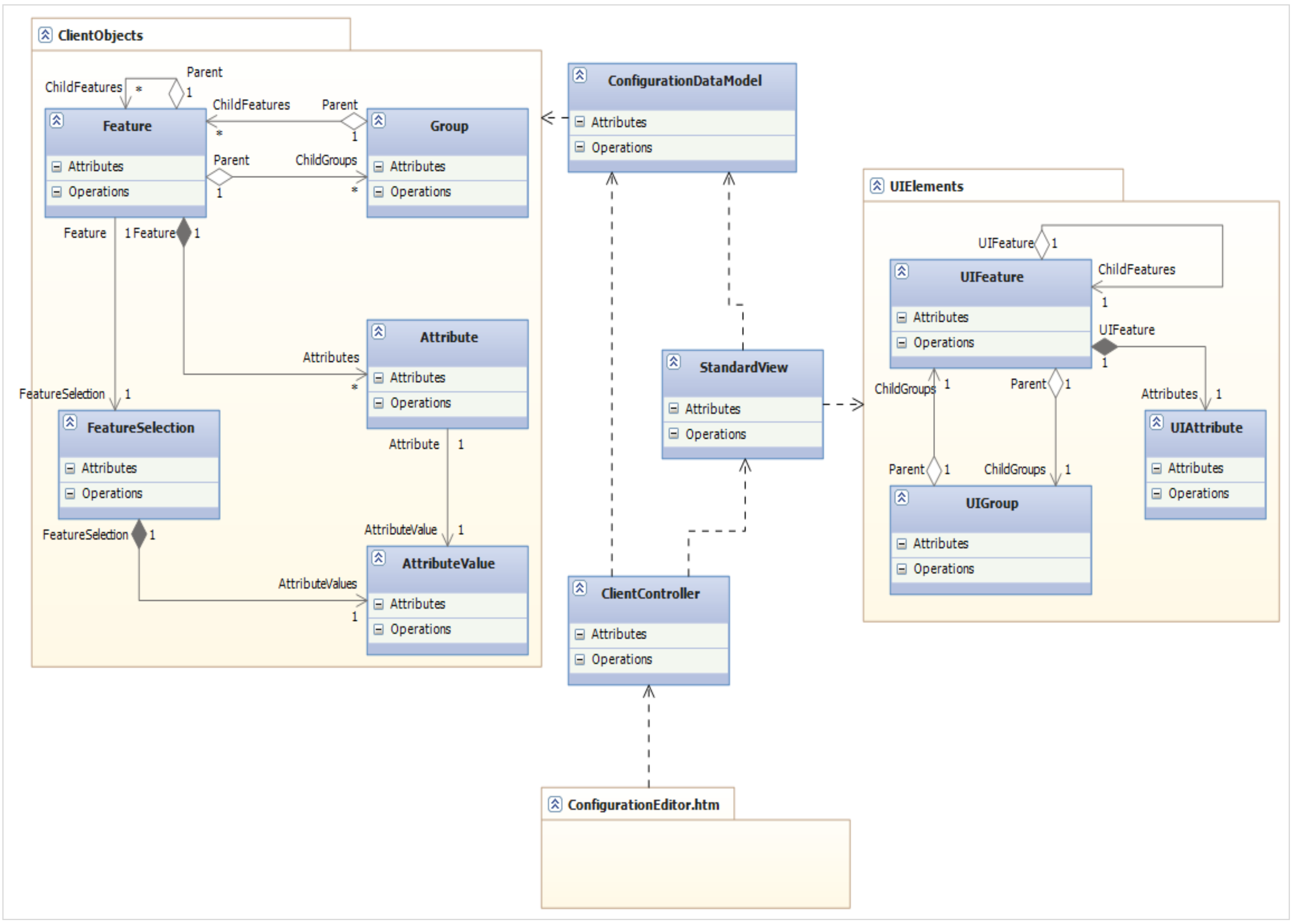


Figure 22 – Class diagram for ConfigurationEditor client side code

The classes are presented and briefly described:

- <u>ConfigurationDataModel:</u>
  Client side Model, similar to the one in the ModelEditor page, but with different ClientObjects suited for dealing with Configuration instances.

- <u>ClientObjects:</u>
  Correspond to Features, FeatureSelections, Groups, Attributes and AttributeValues.

- <u>StandardView:</u>
  Main view used for displaying an interactive Configuration instance.

- <u>ClientController:</u>
  Client side controller.

- <u>UIElements:</u>
  UIElements used by the StandardView component.

- <u>ConfigurationEditor.htm</u>

Server side

On the server we have the ConfigurationEditor controller which contains methods to provide a communication bridge between the client and the services in the Business Logic Layer.

The controller also interacts with a special service in the BLL, responsible for providing consequences to the user whenever a Feature is toggled or the value of an Attribute modified. This special service is the SolverService.

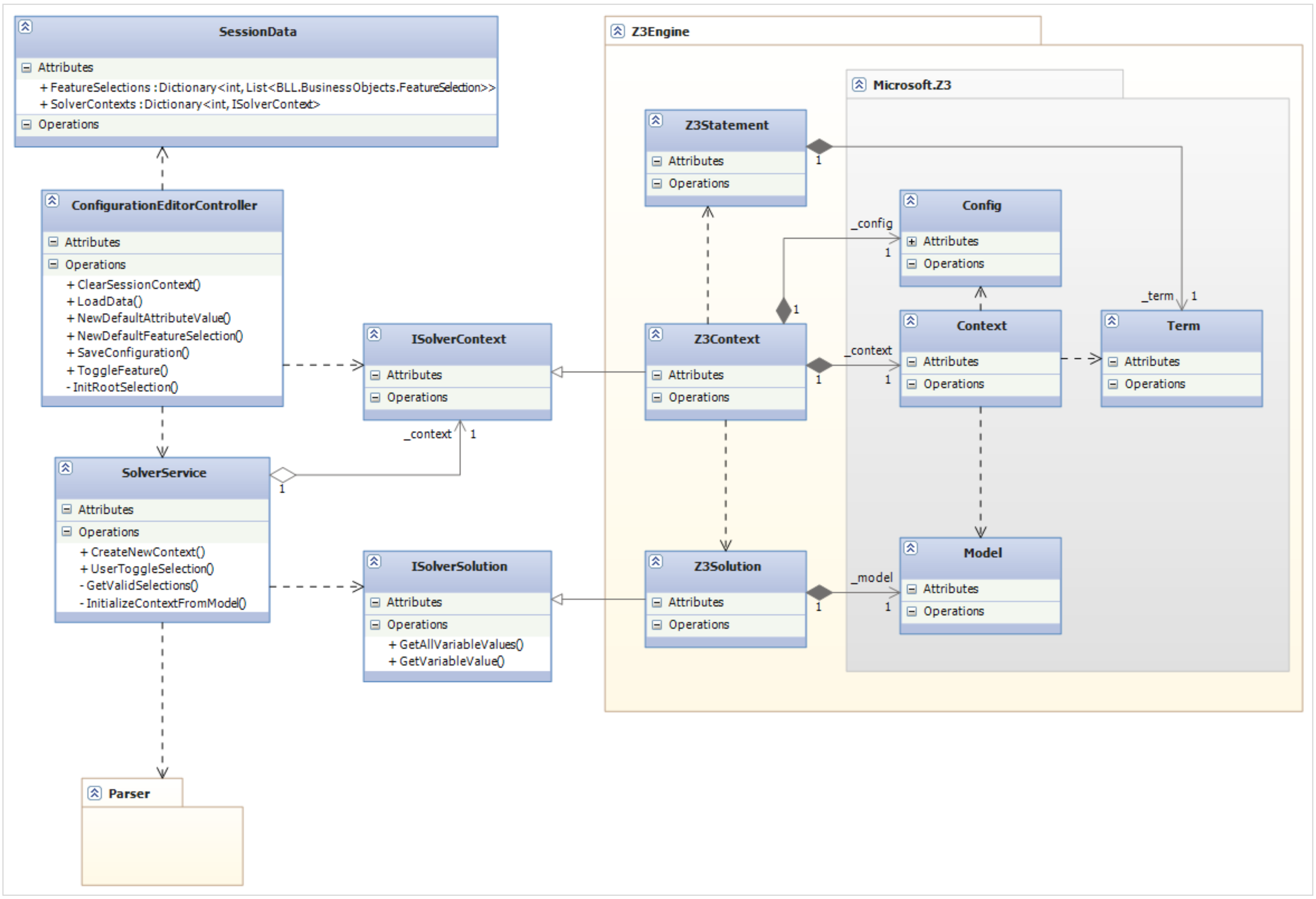


Figure 23 – Class diagram for ConfigurationEditor server side code

Descriptions for the classes above:

- SessionData:
  Special class which provides strongtyped access to data which needs to be stored in the application's SessionState. The data in this case is a set of FeatureSelections and a SolverContext, which are stored for every different Configuration that is being edited.

  It is stored for different Configurations in order to allow the user to open multiple tabs with different Configuration instances at the same time in their browser.

  The data itself is necessary because HTTP and implicitly the asp.net MVC framework are stateless. So if one wants to maintain any sort of state across different requests (calls to the Controller) SessionState must be used. There are also other ways of doing it.

- ConfigurationEditorController:
  This is the controller responsible for handling requests from the client page.

- SolverService:
  Special service which communicates with a Solver and returns consequences and calculations in the form of FeatureSelections and AttributeValues.

- ISolverContext:
  This represents a mathematical context, abstracted through an interface. It has methods which can be used to manipulate a context as well as attempt to return a solution.

- ISolverSolution:
  Represents an abstracted solution returned from a Context. Allows for asking for variable values.

- Z3Engine:
  This is an actual implementation of a custom wrapper for Microsoft's Z3 SMT solver. It contains a Z3Context which can give Z3Solutions when asked for. It is based on a direct .net wrapper from Z3.

- Z3Statement:
  A wrapper for Z3's own Terms.

- Z3Context:
  Wrapper for Z3's inner mathematical context representation.

- Z3Solution:
  Wrapper for Z3's solution class.

- Microsoft.Z3:
  A .Net wrapper from Microsoft for Z3.

SolverService and integration with Z3

The SolverService is a service just like all the others, but is responsible for communicating with the Microsoft Z3 solver. When given a set of FeatureSelections (representing the state of an interactive Configuration), together with a mathematical Context and information about a new feature which needs to be toggled, it communicates with the Z3 solver and returns an updated set of FeatureSelections.

Whenever a new Configuration is loaded, the SolverService is called to initialize a mathematical context which is based on the Feature Model which corresponds to the respective Configuration. The context is setup by looping through all the different elements comprising the Model and converting them to equivalent representations in the mathematical Z3 context.

The initial Context initialization is done in the following steps:

1. All Features are instantiated as Boolean variables in the Z3 Context, using their FeatureID as the variable identifier.
2. Relations are instantiated in Z3 as:
   - Optional: ChildFeature → ParentFeature
   - Mandatory: ChildFeature ↔ ParentFeature
3. GroupRelations are instantiated in Z3 as:

- OR: (ChildFeature1 ∨ … ∨ ChildFeatureN) ↔ ParentFeature
- XOR: ((ChildFeature1 ∨ … ∨ ChildFeatureN) ↔ ParentFeature) ∧ $\bigwedge_{i<j} \neg(\text{ChildFeature}_i \wedge \text{ChildFeature}_j)$

4. CompositionRules are instantiated in Z3 as:
    - Dependency: FeatureA → FeatureB
    - Mutual Dependency: FeatureA ↔FeatureB
    - Mutual Exclusion: ¬(FeatureA ∧ FeatureB)

The translations from feature model elements to logical formula (above) were taken from Mikolas Janota's phd paper (p30) [7].

After the context is initialized, the SolverService is requested to toggle the root feature to true, as a preliminary assumption. The SolverService responds by providing the initial feedbackin the form of a first set of valid FeatureSelections. At this point, the Configuration initialization is considered to be complete and the context and initial FeatureSelections are stored into the application's SessionState.

When the user takes further action by toggling a feature in the UI, a request is sent to the SolverService with the feature's new state (SelectionState can be selected or unselected). The solver communicates to Z3 that the value of the variable corresponding to the feature must be updated, after which it provides feedback as to what other changes must be made to the other FeatureSelections.

The feedback given by the Solver has the purpose of maintaining the validity of the underlying Model while the user is configuring a Configuration instance. Basically. whenever the user toggles a feature, the SolverService communicates with Z3 and determines which FeatureSelections must be on or off in order for the Model to still be satisfiable. Thus, the user is never allowed to make a decision that breaks the Models (backtrack-freeness [6]). The algorithm for providing feedback is inspired from [6]:

```
GetValidSelections()
  1. foreach FeatureSelection that was not toggled by the user
  2.     do CanBeTrue  := CheckSolutionExists(FeatureSelection, true)
  3.        CanBeFalse  := CheckSolutionExists (FeatureSelection, false)
  4.        if ¬(CanBeTrue ^ CanBeFalse)
  5.          then error "Unsatisfiable constraint!"
  6.        if ¬CanBeTrue then SET(FeatureSelection, unselected)
  7.        if ¬CanBeFalse then SET(FeatureSelection, selected)
  8.        if CanBeTrue ^ CanBeFalse
  9.          then RESET(FeatureSelection)
 10.              ENABLE(FeatureSelection)
 11.          else DISABLE(FeatureSelection)
```

What happens when GetValidSelections () is executed, is that for each FeatureSelection which wasn't made by the user, the algorithm attempts to see what happens to the satisfiability of the model if the variable is either true or false. If one of the two values makes the model unsatisfiable, then the variable is assigned the opposite value and disabled so the user cannot toggle it anymore. If neither false nor true break the satisfiability of the model then the FeatureSelection is enabled and reset.

The GetValidSelections () method is invoked whenever the user makes a decision, meaning whenever UserToggleSelection () is called. The pseudocode for UserToggleSelection ():

```
UserToggleSelection(context, featureSelections, featureID, newState)
   1. featureSelection := FindFeatureSelection(featureID)
   2. featureSelection.SelectionState = newState;
   3. if newState == selected
   4.   then AssumeBoolValue(featureID, true)
   5.        featureSelection.ToggledBy = user
   6.   else AssumeBoolValue(featureID, false)
   7.        featureSelection.ToggledBy = solver
   8. GetValidSelections(context, featureSelections)
```

The UserToggleSelection method receives an instance of the mathematical context (state of the Configuration used by Z3), the list of FeatureSelections (state of the Configuration used by the UI) together with the FeatureID and the new selection state of the Feature the user wishes to toggle.

It finds the FeatureSelection corresponding to the given FeatureID, after which it determines a Boolean value (true/false) based on the new state of the feature (selected/unselected). It then notifies Z3 that it should set the value of the inner variable corresponding to the given Feature to this Boolean value. We make an assumption that if the new state is to be unselected, it should be specified as not being set by the user. This is in order to allow the GetValidSelections() method to toggle it if necessary.

After the value was set in Z3 (or updated), GetValidSelections() is called to find out what consequences the assumption had.

Parser

The Parser is responsible for parsing and executing text based rules. It uses regular expressions encapsulated in the form of different statements to parse and execute the syntax. Its design is shown and explained following.

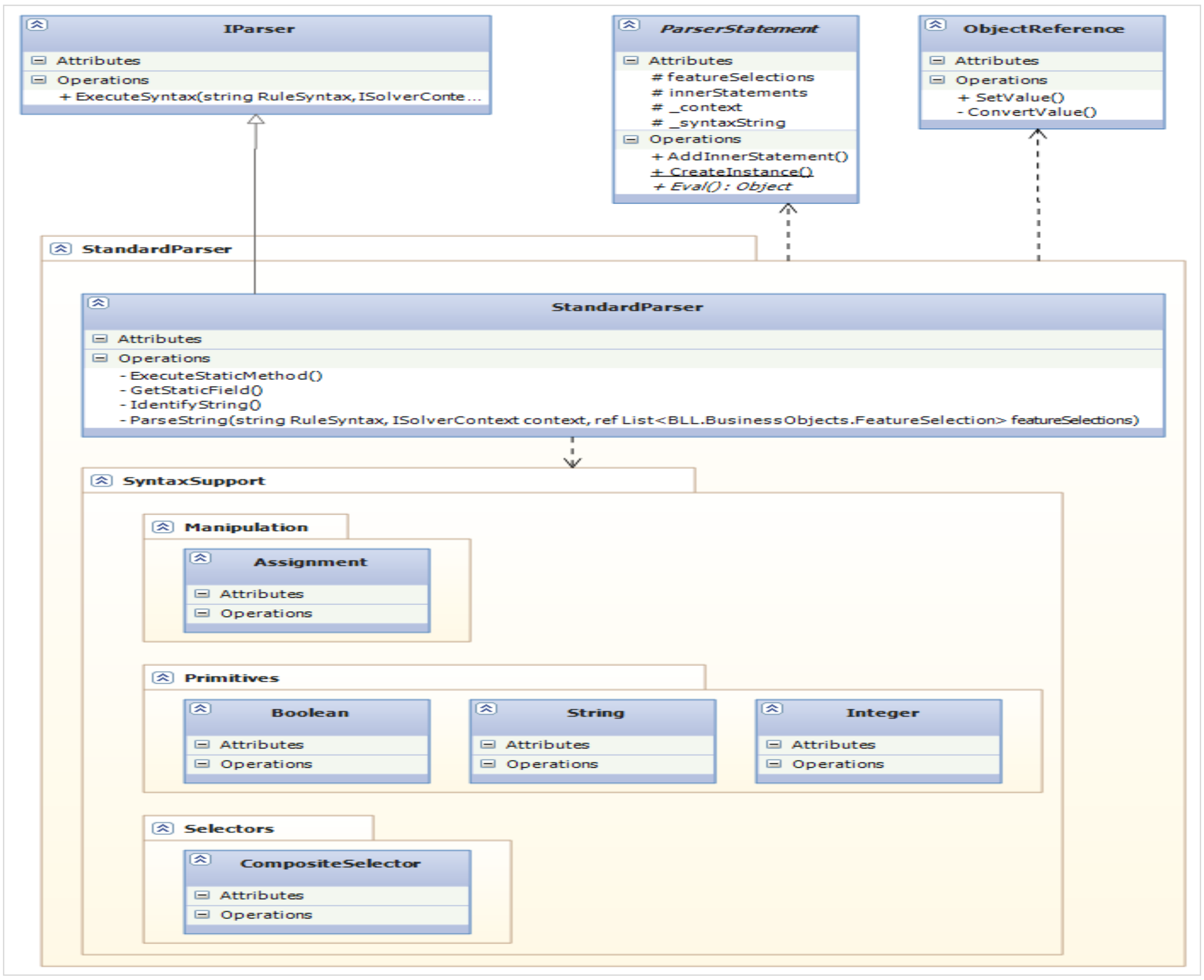


Figure 24 – Class diagram for parser

The different classes in the diagram are explained below:

- IParser:
  Interface used to abstract the details of the actual parser used. Allows for easy replacement with a different Parser implementation.

- ParserStatement:
  Abstract class from which all types of parser statements inherit basic behavior.

- ObjectReference:
  Helper class used to pass references pointing at primitive values, since C# doesn't support pointers.

- StandarParser:
  Actual implementation of a parser. Contains a collection of Statements it supports (SyntaxSupport) as well as an algorithm used for parsing the input.

- SyntaxSupport:
  Static class which contains all the different types of statements that the parser supports. The statements are split into different categories to make it easier to maintain them. The categories can be seen as Manipulation, Primivites and Selectors. Each statement type contains its own regular expression syntax, both for identifying itself, as well as for being able to identify any inner statements it may contain. It also contains the logic to evaluate

itself (execute, done through Eval() method).

When parsing a rule, the parser uses the recursive algorithm below:

```
Parse(str, context, featureSelections)
  1. statementType := IdentifyString(str)
  2. statementInstance := CreateInstance(statementType, context, featureSelections)
  3. innerStatementStrings := REGEX.Split(statementType.SplitRegex)
  4. foreach innerstatementString
  5.    innerInstance := Parse(innerstatementString, context, featureSelections)
  6.    statementInstance.AddInnerStatement(innerInstance)
  7. Return statementInstance
```

IdentifyString is a method which loops through all the different Categories defined inside SyntaxSupport and then for each different StatementType, tests its REGEX expression for identification. When a match is found, it returns the appropriate StatementType.

Parsing has the effect of creating something that could look like a tree of ParserStatements. Each ParserStatement can contain other nested ParserStatements. When Evaluating, each type of ParserStatement has its own implementation for it. For example, when evaluating an Assignment statement, it will first evaluate its inner statements and only then will it start the assignment.

The actual execution of a custom rule implies evaluating its outermost statement, which in turn will trigger a recursive evaluation process for all existing statements in the rule.

4. Implementation and other details

4.1. Development process

The project underwent an agile development process, similar to SCRUM with short iterations. The reason for an agile approach was due to several of the high level requirements involving novel functionality. Agile is good at such projects where it is impossible to predict and plan for everything.

Thus, even though the high level goals were clear from the beginning, detailed specifications were mostly discovered during the development iterations. As for managing the workload and agile planning, AgileZen an online tool for managing development was used.

No standardized and formal testing was performed, since the project isn't for a commercial purpose. However, ad-hoc manual testing was performed in order to ensure operability of the tool's main functions to be demoed.

4.2. Code metrics

The code metrics are presented in order to give an overall notion of the complexity and size of the project's code and implicit implementation process. The metrics have been split in two - for server and client - because code analysis tools encountered problems when analysing javascript.

Server Side

The c# code which was analysed consisted of the three layers, DAL, BLL and Presentation Layer (server side code). Code was analyzed with VS 2010.

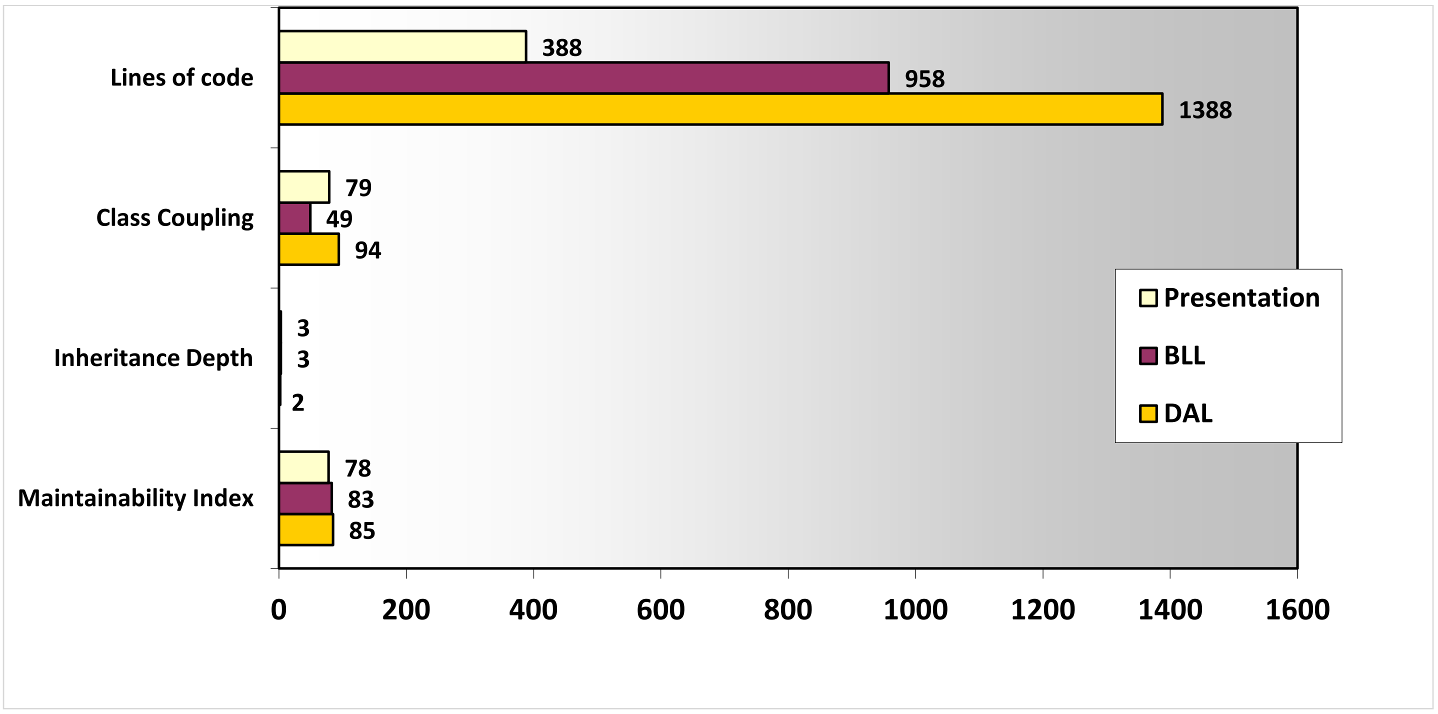


Figure 25 – C# code metrics

Client Side

The Javascript code analyzed consisted of two major files (ModelEditor.js and ConfigurationEditor.js) as well as a number of smaller scripts used on the other pages for UI based logic.

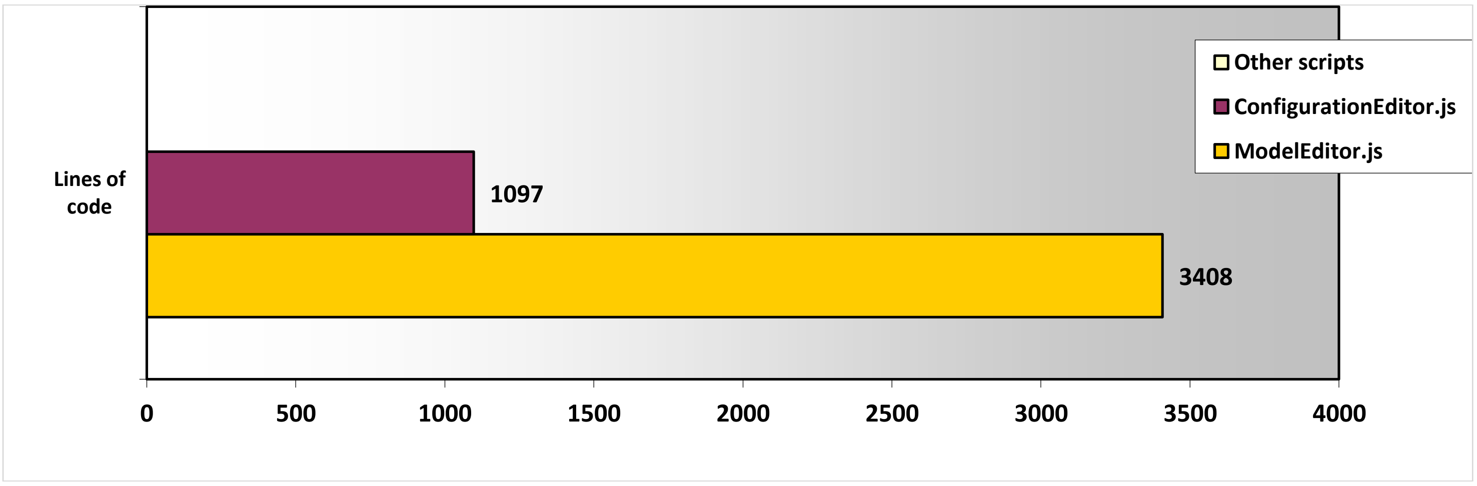


Figure 26 – Javascript code metrics

Metric interpretation

In order to better understand the chart, descriptions of the metrics can be found below [8]:

- Maintainability Index:
  Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability.

- Inheritance Depth:
  Indicates the number of class definitions that extend to the root of the class hierarchy. The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or/and redefined.

- Class coupling:
  Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration. Good software design dictates that types and methods should have high cohesion and low coupling.

- Lines of code:
  Indicates the approximate number of lines in the code. The count is based on the IL code and is therefore not the exact number of lines in the source code file. A very high count might indicate that a type or method is trying to do too much work and should be split up. It might also indicate that the type or method might be hard to maintain.

5. Conclusion

The tool developed and described in this paper, although far from being anything but a prototype, fulfills the main project goals as outlined in section 2. It provides users with the functionality necessary to define standard feature models (as per-definition in subsection 2.1). It also allows for more advanced models through support for user-defined attributes and complex rules. Manipulation of feature models is done through a rich user interface with visual modeling capabilities. The tool allows for interactive configurations to be configured, with consequences being computed and displayed, through the use of an SMT solver. All of the aforementioned functionality is served through a native browser based, web 2.0 implementation - independent of any 3rd party components.

We hope that the tool and this paper can serve as inspiration for any related future work in the development of similar tools. The next section presents ideas and potential solutions for improvements on the tool.

6. Further work

Although it was mentioned in the previous section that the tool accomplished all the initial goals we had in mind, there are still a lot of areas where the tool can be improved. This section presents a number of suggestions and ideas.

6.1. Z3 based implementation for complex rules

The handling for complex based rules is currently based on a custom .net implementation, meaning the parser described in subsection 3.3.5 uses C# to execute the logic in the rules. This can become a problem if users have a lot of rules that enforce constraints or if rules become very complex, since it can result in unpredictable behaviour which cannot be catered for with a custom implementation.

Since the capabilities of a mathematical SMT solver such as Z3 are very big, it could be a good idea to transform logic in rules into constraints and assumptions directly in the Z3 mathematical context.

6.2. Inheritance and Feature types

Often when dealing with large Feature Models, one runs into the situation where a lot of the design time is used on redundant work such as updating a lot of Features at once, or adding the same type of Attribute to a large number of Features. A solution for this is to provide inheritance for Features. Users would be allowed to define their own abstract Feature *types*, from which individual Features would inherit the Attributes. Additionally it would also provide a simple way of identifying Features when writing complex rules, which make for easier maintenance of Models.

For example, if a Feature Model has 20 Features spread around the tree which deal with a product's "physical endurance" and we wish to select them in a complex rule. Instead of specifying them one by one, we could create a new *type* called "Physical endurance" and have all of these 20 Feature inherit from it. Then, in our complex rule, we'd just selected the Features by selecting all Features which inherit from the *type* "Physical endurance".

There are however some considerations when considering inheritance, such as issues with multiple inheritance, or conflicting Attributes. But, even a simple implementation with a lot of limitations on the level of inheritance could prove to have significant benefits.

6.3. User-defined UI for configuration instances

Currently, the part of the application responsible for configuration instances (ConfigurationEditor) is hard coded to generate a fixed type of UI with standard controls and representations. This is useable by most users, but were the implementation based on a free xml style syntax, then users would have the freedom to create their own specific UI to use with their configurations.

The motivation for this improvement is that different types of models have different target audiences configuring them. For example someone configuring a car would expect to see a UI representation depicting an actual car matching the characteristics and features they've selected. Someone configuring a holiday package would expect to see a representation with pictures of the chosen hotel and country.

6.4. Multi-user collaborative configurator

This would allow multiple users to view and configure the same instance simultaneously. The idea behind this is that sometimes models are so complex that multiple people need to configure different parts of them at the same time, since different parts of the model are understandable only by certain types of people.

The implementation for this would imply setting up a timer service in the client code which polls the server for updates, while storing the configuration instance data somewhere on the server. Whenever a user toggles something in the Configuration, the configuration kept on the server is changed, which in turn notifies all the clients polling for updates that the data was modified. Thus, each of the individual clients would receive their updates and their views of the configuration would be synchronized.

6.5. Cloneable Features

Cloneable features were mentioned in [5]. They allow users defining models to state that certain Features are *cloneable* with a given accepted interval [m...n]. This in turn means that when configuring instances, those Features can be cloned at least m times and at most n times.

Cloning features isn't a simple task to accomplish due to the fact that it introduces *instances* of clones into the picture. [5] suggests a way to handle this by having different contexts.

6.6. User-defined cardinality Groups

Feature groups with user-defined cardinality were also mentioned in [5]. They simply depict a new type of Group, besides OR and Alternative, where one can define a precise interval [m...n] representing a minimum and maximum number of of features that can be selected from the respective group.

This is not so difficult to implement. Work would need to be done in order to generate the logical formula for it, which would then need to be mapped over to Z3 as a constraint.

6.7. Performance of SVG-based modeler implementation

The visual modeling is implemented using SVG, which has some performance limitations when dealing with a lot of elements, as it has to keep track of the state of each individual element. This can be noticed in the case of very large feature models with lots of Features and GroupRelations. A solution for this is unknown at this stage, but some articles mention the possibility of implementing a hybrid SVG/Canvas engine, making use of the strong sides of both – SVG performance isnt affected by screen resolution, Canvas performance isnt affected by number of different elements.

Besides the performance issues of SVG, there are some higher level algorithms which could be improved. These are the algorithms for updating real-time connections between Features as they are being dragged, or updating the position for Cardinality Labels.

6.8. Performance of configurator

The algorithm used for providing feedback (GetValidSelections), described in subsection 3.3.5 could run into problems when handling bigger models. Therefore, it's performance should be improved. This could be done as described in the original paper by Mikolas Janota [6].

6.9. Explanations provided by configurator

An enhancement which would help users understand the underlying model and constraints when making configurations, is to have the configurator provide explanations as to why it toggled or disabled certain features. An algorithm for providing feedback is mentioned in [6] and described in [22].

6.10. Improved usability and UI

The UI and usability of the application could be improved in a number of ways. These are presented below, with most of them refering to the ModelEditor user interface. The first mentions the need of a usability study to uncover flaws, while the rest describe some improvements which we believe will directly increase the tool's usability.

Usability analysis study

We believe it to be crucial that the current UI design is evaluated. This is important because the tool, in its current state has not undergone any user tests, so there might be areas with serious usability defects.

We propose the following activities for the usability analysis:

- <u>Heuristic analysis:</u>
  An evaluator analyses the system with regards to a set of heuristics meant to determine different subaspects of it's usability. Jakob nielsens heuristics could be used [23].

- User trials:
  Involves performing user tests meant to determine the system's usability through collecting subjective and objective data from actual users instructed to perform certain tasks or answer sets of questions.

Zoom functionality

The necessity for a zoom function in the ModelEditor arises when dealing large feature models. Currently, when a feature tree consists of many features spread out over a large area, users cannot see the whole model, having to scroll around a lot. A simple zoom in/out function would solve this.

A solution for an implementation would have to be done by scaling down the SVG elements. This is somewhat straightforward as SVG natively supports scaling, but proportions and distances between inner elements would have to be maintained manually. Additionally, the relations drawn using SVG path would have to be manually scaled so they are properly maintained.

Multi select manipulation

This functionality is present in most vector editing tools (including the Visual Studio uml editor). It implies that users should be able to select and incur operations on sets of multiple elements. Like for example, selecting a whole branch of a tree and then moving it around, or selecting and editing multiple Features at the same time.

The architecture supports an implementation like this, but some small changes would have to be made.

Additional standard editor functionality

Functionality which is found in standard editor tools such as undo/redo or copy/cut/paste.

7. References

1. Kyo C. Kang, Jaejoon Lee and Patrick Donohoe - "Feature-Oriented Product Line Engineering"
2. Jing Sunm, Hongyu Zhang, Yuan Fang Li, Hai Wang - "Formal Semantics and Verification for Feature Modeling"
3. Goetz Botterweck, Mikolas Janota and Denny Schneeweiss - "A Design of a Configurable Feature Model Configurator"
4. Don Batory - "Feature Models, Grammars, and Propositional Formulas"
5. Krzysztof Czarnecki and Chang Hwan Peter Kim - "Cardinality-Based Feature Modeling and Constraints: A progress report"
6. Mikolas Janota - "Do SAT solvers make good configurators?"
7. Mikolas Janota - "SAT Solving in Interactive Configuration"
8. Danilo Beuche and Holger Papajewski - " Variability management with feature models"
9. Mikolas Janota and Joseph Kiniry – "Reasoning about Feature Models in Higher-Order Logic"
10. MDSN – Code metric values http://msdn.microsoft.com/en-us/library/bb385914.aspx
11. http://en.wikipedia.org/wiki/Feature_model - Feature Model
12. Miloslav ŠÍPKA – "Exploring the Commonality in Feature Modeling Notations"

13. Krzysztof Czarnecki, Simon Helsen and Ulrich Eisenecker – "Staged Configuration using Feature Models"
14. Kang, K., Cohen, S., Hess, J., Nowak, W. and Peterson, S. – "Feature-oriented domain analysis (FODA) feasibility study"
15. Krzysztof Czarnecki, Simon Helsen and Ulrich Eisenecker – "Formalizing Cardinality-based Feature Models and their Specialization"
16. David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortes – "Automated Reasoning on Feature Models"
17. Krzysztof Czarnecki – "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models"
18. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report
19. Søren Lauesen – "User Interface Design: A Software Engineering Perspective"
20. ISO Standard – part 11: Guidance on usability (ISO 9241-11:1998).
21. MSDN- ASP.net Overview - http://msdn.microsoft.com/en-us/library/dd381412.aspx
22. L. Zhang and S. Malik. – "Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications"
23. Jakob Nielsen's 10 usability heuristics - http://www.useit.com/papers/heuristic/heuristic_list.html
24. Terry Quatrani – "Visual Modeling with Rational Rose 2000 and UML"
25. Kathrin Figl, Jan Mendling and Mark Strembeck – "Towards a Usability Assessment of Process Modeling Languages"
26. Mary Jo Davidson, Laura Dove, Julie Weltz – "Mental Models and Usability"
27. San Murugesan – "Understanding Web 2.0" (http://91-592-22.wiki.uml.edu/file/view/understanding_web_20.pdf)