Assignment 6

Group 4

Igor Vassilevski
Marcus Thorström
Michal Musialik
Moses Msafiri
Peter Eliasson
Vidar Åsberg

Contents

Introduction

This report details the implementation of a simple configurator by using a model based approach. The structure of the report follows the various types of tasks performed, starting with defining the meta-models required and ending with a discussion of the results and general thought of the work performed.

Solution Overview

The solution that was chosen for the project was to have two meta-models. One for the configurator, presented in Figure 1, and one for the requirement specification, presented in FIgure 2. Of note here is that the configurator also holds the selected values for the features. As such it is essentially a combination of the meta model for a configurator, as well as the meta model for a values selected.

Running The Configurator

Run the /assignment6_editor/src/se/chalmers/mde2016/group4/editor/Main.java file as a Java application. Provide the filename of the .xmi representation of a configurator model instance. Options are selected using console input.

Meta-Models

Two meta-models were required for the project. The meta-model for the configurator, as used for most task in the assignment, and an extended requirements meta-model to be used for the final transformation.

Configurator Meta-Model (Task 1)

The configurator created (see Figure 1 below) was based on the feature model presented in the lecture slides, with a couple of differences. It was decided to have an explicit root node called Configurator, instead of a root feature, as to make things more clearly understandable. Additionally, enums were also favoured over separate classes, as it was noticed in previous assignments that this made creating the xtext grammar easier. Furthermore, the model was changed to allow groups to have mandatory features, as it might be useful for OR-groups.

Each feature in the model can be selected, meaning that the boolean features are essentially implemented by all types of features. An additional type of feature was also added, IntegerFeature. This feature allows for specifying an integer feature, between a min and a max value and with a specific step size. It was decided that the list feature could be realized using groups of features, instead of having it be a separate type of feature.

The meta-model was additionally extended with dependencies, so that each feature can express what other features it requires. Two types of dependencies were created, IsSelectedDependency and IntegerValueDependency. The first expressing that the requirement targeted must be selected, while the latter requires that the IntegerFeature pointed out has a specific value selected.

The Configuration meta-model also has placeholder fields for the selected values of the features. This was done to make implementation of the editor easier, as it could be created to execute directly on the meta-model, as opposed to work with yet another intermediate model.

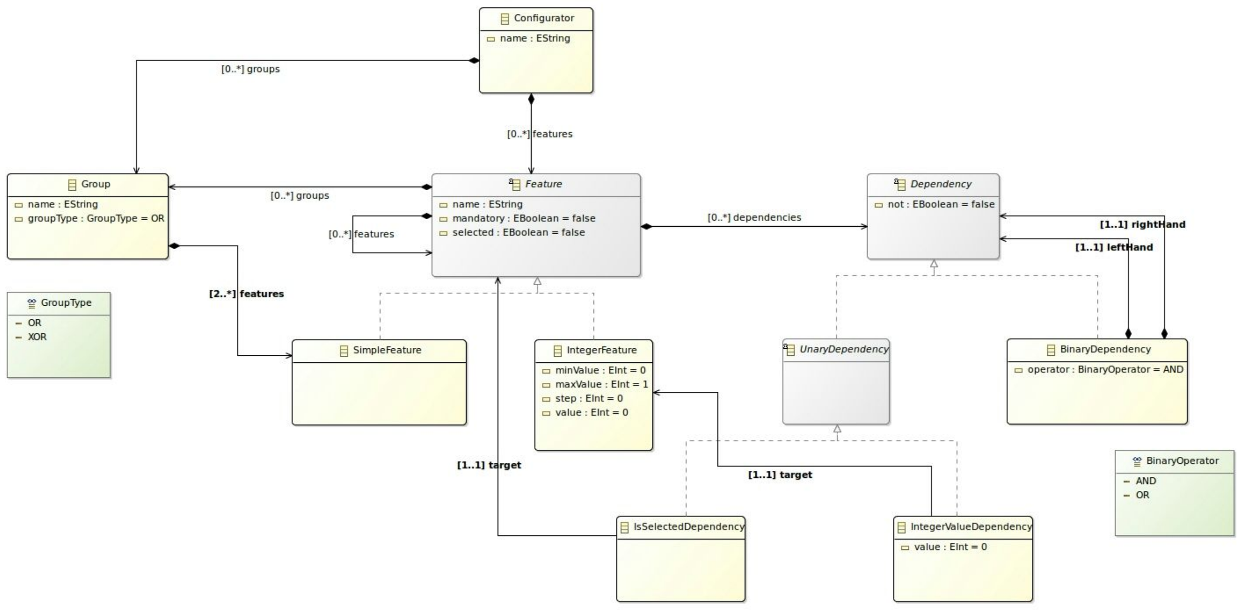

Figure 1. Configurator meta-model.

Extended Requirement Specification Meta-Model (Task 6)

To allow for mapping between requirements and features, the requirements meta-model was extended (see Figure 2 below). This was done simply by adding a new class to the meta-model for a feature, holding only a name. The mapping can then be performed by matching the name of the features in the requirement specification to the name of the features in the configurator model.

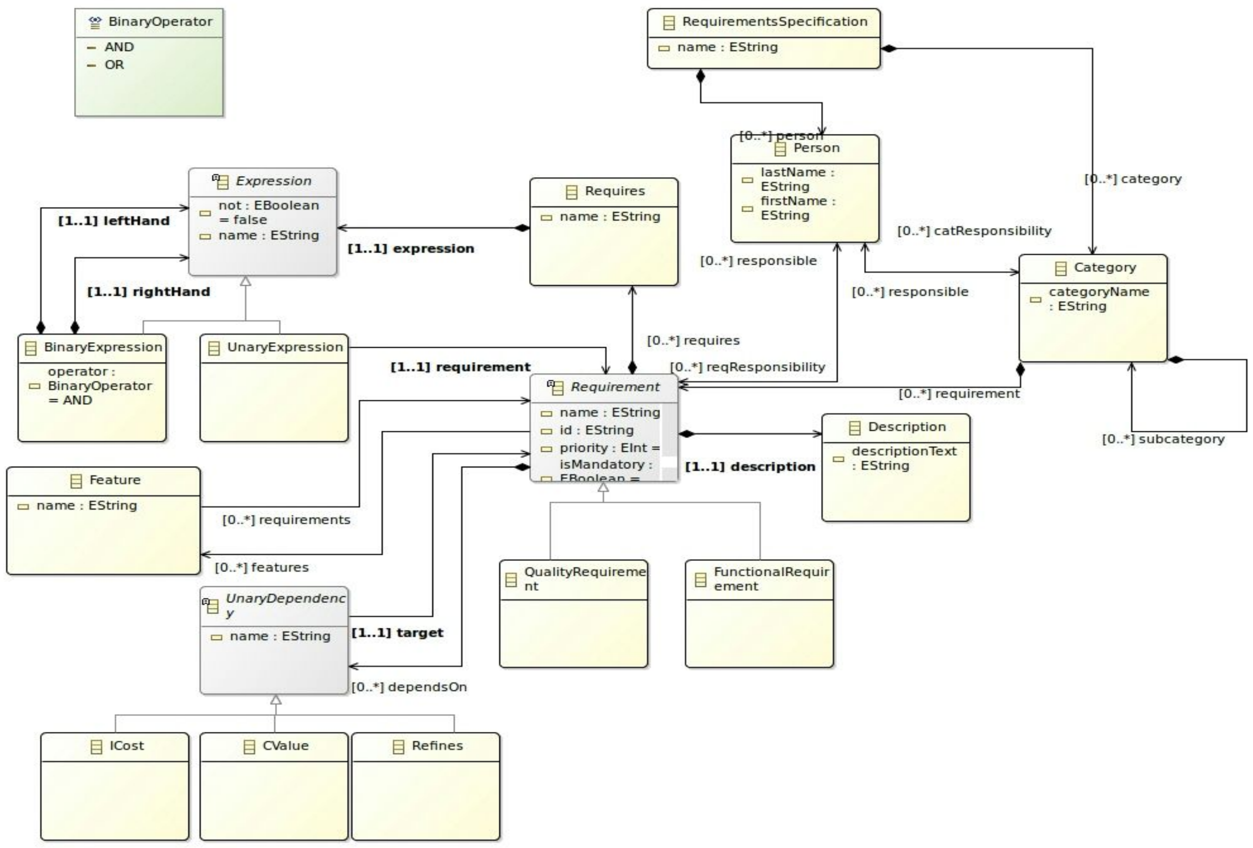

Figure 2. Extended Requirement Specification Meta-Model

Constraints

We have divided the constraints into two types; constraints on the meta-model (via OCL) and constraints on the allowed values for features (enforced by the “graphical” editor). The first type of constraints makes sure that the configurator instance that is created is valid. For example, a configuration can not have a mandatory feature as part of a non-mandatory feature. The second type of constraint makes sure that the selected features are valid, and should take into account the user-defined dependencies in the configurator model.

Meta-Model Constraints (OCL, Task 2)

The identified constraints for the meta-model are the following, grouped by context:

- Feature
    - **(dependsOnSelf) Can not have dependency to self:** This would make it impossible to select the dependency, as it would have to be selected to be selectable.
    - **(parentIsNotMandatory) Can not be mandatory if container is not mandatory:** If the container (i.e. “parent”, the owner of the requirement) is not mandatory it follows that none of its sub-features can be mandatory.
    - **(mandatoryWithDependencies) Can not be mandatory and have dependencies:** This is essentially a way of saying the dependent features are also mandatory, which should instead be done by marking them as such.

- IntegerFeature
    - **(minLessThanMax) MinValue must be less than MaxValue**
    - **(evenSteps) Even steps**: The IntegerFeature allows for setting a “step” property that determines the allowed increment (i.e. with a step of 2, one can only change the value in steps of 2). This constraint makes sure that the step allows the user to input both the minValue and the maxValue.
- Group
    - **(noMandatoryIfXOR) Can not be XOR and have mandatory features:** Since an XOR group can only have exactly one selected, it makes no sense to “fill” this by having one option being mandatory already.

These constraints are implemented as OCL in the “model/assignment6_model.ecore” file of the “assignment6_model” project.

Allowed Values Constraints (Editor)

The editor also has to enforce a couple of constraints, as identified:
- Feature
    - **Can be selected only if all dependencies are fulfilled**
    - **Containing feature must be selected/mandatory**
- Group
    - **Must have at least one selected/mandatory**
    - **Must have exactly one, if XOR-group**

Configurator DSL (Step 3)

To make creating a Configurator model easier, a DSL was created using Xtext. The DSL is fairly straightforward, with mostly the standard Xtext grammar as it was generated. The expressions were the primary point of change, where the grammar was changed to allow for creating expressions in a more natural way.

A simple transformation application from the dsl to a .xmi representation of the model was also created in this step. This was done so that the model could easier be read by the editor and the ATL transformation.

Model Editor (Step 4+5)

The editor should allow for editing a configuration model instance, setting the values of the features. Two types of editors was created, one written as a console based Java application using ecore to modify the model, another as an HTML-based web page. This was done as the HTML-based approached seemed risky, in that it might not be able to get it working. As such, the Java based application was created as a backup should the other attempt fail.

Java-based editor

The Java based editor reads an .xmi representation of a configurator instance and allows the user to edit the values of the features. It does this by using the generated java source files of the configurator model, and changing the instance values of the java objects. The objects are then again serialized via ecore, and saved as a new configurator instance. As the output is of the same type as the input (instance of configurator), it also has the added benefit of being able to save and load in progress edits.

As to simplify the implementation, validation is performed manually and only on the entire model at once. That is, the user choice is never limited, but is instead only presented with a list of errors when the validation is run.

HTML-based editor

The HTML based editor is created by a model-to-text transformation that takes a configurator instance as input and generates an HTML form as output. Furthermore, the HTML output is extended with javascript to also validate the dependencies of the model. Visually this is done by enabling and disabling the various inputs, depending on if their dependencies are fulfilled or not. As the selections are complete, the HTML editor generates a JSON document representing the selected values.

Unfortunately, a transformation that takes the JSON data and creates a model instance from it (i.e. a text-to-model transformation) was never created. This was as it seemed to be quite time consuming to complete, and that the group members would rather spend that time practising for the exam. Especially as the console based solution seemed to work (although not as visually impressive). This essentially leaves the HTML-based editor somewhat useless, but it was nevertheless seen as a good exercise in model-to-text transformations.

ATL transformation (Step 7)

The ATL transformation is used to create a new requirement specification from an existing requirement specification and a configurator instance. Each requirement is only added to the new requirement specification if it is selected, mandatory, or required by another requirement. In other words, one can see the configurator as a filter applied to the original requirement specification, and the elements that pass the filter is what makes up the new requirement specification.

The implementation follows the idea of the filter. Each requirement, category and person is checked to make sure it should be included before the element is mapped. The check is done by calculating the entire dependency graph for each requirement, which is obviously inefficient, but does on the other hand result in code that is easier to follow.

The task also says that the solution should take requirement dependencies into account. The presented transformation does this by copying each requirement also if any other requirement requires it. This is done by looking at all the unary expression targeting each requirement, for each of these expression the source is found and checked if it is included or not. If it is, the target requirement is also included.

Conclusion and Future Work

The project implements a very simple configurator. It is defined in using a simple DSL and features can be selected in an editor. The model driven approach made the model-code much easier to create, as all one had to do was to create the class diagram and it could generate a lot of boilerplate java-code automatically. Although this did make some parts easier, it also added dependencies on ecore and various other libraries and tools, which to us seemed like a pretty large drawback.

The above seemed true for most parts of the projects. If one managed to get the tool working it worked well and did save a lot of time. However, getting it working was unfortunately often hard, involving a lot of copy-pasting of already existing code.

An obvious missing part of the current project is the lack of any real evaluation of the feature dependencies. For real usage, one would have to ensure that these dependencies could be probably parsed and enforced.

Appendix A. Team Member Contributions

Configurator Meta-Model: Everyone
Requirements Meta-Model: Peter
Configurator OCL: Michal, Peter, Vidar
Xtext Grammar: Marcus, Peter
Java-Editor: Peter
HTML-Editor: Moses, Marcus
ATL: Igor, Vidar, Peter
Report: Igor, Moses, Peter, Michal