DSL for the simulation of evacuation plans with Xtext

Schoofs Ebert[1]

ebert.schoofs@student.uantwerpen.be

University of Antwerp[a]

[a] *Prinsstraat 13, B-2000 Antwerp, Belgium*

Abstract

In this paper we will discuss a new Domain Specific Language, Bmod, made for the modeling of a floorplan, and it's occupants behavior in case of an emergency. Bmod was made possible thanks to the Xtext framework, a framework made by the Eclipse Foundation.

Keywords: DSL, Bmod, Xtext, MDE

1. Introduction

The creation of models can best be left to domain experts, but we cannot assume that they have all the skills needed to develop a simulation tool from scratch. This is were Domain Specific Languages (DSL) come in handy. A DSL is a language offering expressive power focused on a particular problem domain. [1]. It thus makes it possible for domain experts to focus on their domain.

One of the frameworks for the development of DSL's is Xtext from the Eclipse Foundation. This frameworks allows us to easily create a parser for our DSL, validate it according to the domain specific restrictions and translate it to other languages. In this paper we will create a DSL, named Bmod, which allows modeling of a floor plan to simulate the behavior of occupants in case of a fire without proper programming knowledge.

In section 2 we will describe the design of Bmod. We will take a look at how

[1] 20161650

it's implemented using Xtext in section 3, and finally, in the last section we will compare Bmod with MetaDepth, a framework for deep meta-modelling.

2. Bmod

As stated in the introduction Bmod is a DSL for the modeling and simulation of a floor plan in case of a fire. It is a simple language meant to be understandable and easy to create for non-programmers. In combination with Eclipse, it provides auto-complete and rigorous validation and automatic translation to Java.

In this section we will go over the design of the Bmod language, how a model can be constructed and we will elaborate on some design decisions. If some things are still not completely clear after reading through it, don't worry. If you use Bmod in combination with Eclipse, the IDE will help you write the model through auto-complete and give warnings and errors if there are design errors.

2.1. Core Concepts

A Bmod file, ends with the extension `.bmod` and consists of one mandatory part, the model, and one optional part, the output options of the simulation.

The model defines the floor plan, the persons in it and the emergency signs. Every model needs it's own file.

2.2. Model

The model in it's most simple form looks as follows:

```
FloorPlan Name
{
    Feature Name
    {
        ...
    }
    ...
}
```

Inside the FloorPlan we will define how the FloorPlan looks like, and every person in it. A feature can be a `Cell`, `Room`, `Door`, `Person`, `EmergencySign` or `Fire`. `Fire` is the only feature which can only be once in the FloorPlan.

Cells are really the core of our FloorPlan, everything is build around them.

2.2.1. Cell

A cell is identified by it's name, and has an positive x- and y-coordinate.

```
Cell c00
{
    x : 0
    y : 0
}
```

2.2.2. Room

Cell's can be combined to form a room, which is also identified through it's name.

```
Room r1
{
    cells : [c00, c01, c10, c11]
    max occupancy : 2
}
```

`max occupancy` is an optional parameter of a room. During the simulation, each time cycle there will be a check how many persons there are in the same room. If a room has more persons than, or equal to it's maximum occupancy, this is reported in the output.

2.2.3. Door

In Bmod, there are two kinds of doors; normal doors and emergency doors. A normal door connects two cells of two different rooms, to connect the rooms. On the other hand an emergency door only has one cell. If a person reaches an emergency door, they can exit the building and are saved.

```
Door normal_door
```

```
{
    cells : [c00, c01]
}
Door emergency_door
{
    cell : c00
}
```

2.2.4. EmergencySign

Like in real buildings, there is a need for emergency signs to find our way through the rooms to an emergency exit.

```
EmergencySign es
{
    from : d0
    to : d1
}
```

`from` and `to` should be the identifier of a door. The validation of emergency signs is one of the places where the custom validation of Bmod shows its strengths. If a chain of emergency signs doesn't lead to an emergency exit, a warning will be thrown, the same happens if there are two conflicting signs; i.e. a sign with the same doors but in the opposite direction.

2.2.5. Person

The whole goal of Bmod is to study the movement of persons in case of a fire, so naturally we also need to define persons with different perception and action profiles.

```
Person Hans
{
    action : experienced
    perception : listener
    cell : c32
}
```

There are three different action profiles, and four perception profiles. Let's first

take a look at the action profiles, they dictate the movement of a person once it has detected a fire.

- `experienced`

  Will go to the door if there is only one door in the room he is in.

  If this person has crossed a door, and in the new room is an emergency sign following this door, he will go to the next door this sign dictates. Otherwise, he will choose a random sign to follow.

- `newcomer`

  Moves to the closest person in the room

  If there are no persons in the same room, he moves to the closest door to find someone in another room.

- `panic`

  Moves to a random cell.

Note: If a `newcomer` moves to another room, and there is also no one there, he will keep switching between both rooms till he finds someone.

A person only moves if he has detected a fire. How a person detects there is a fire is defined by the perception profile.

- `nervous`

  Detects a fire once the simulation starts.

- `listener`

  Detects a fire once someone else in the same room has detected a fire.

- `smeller`

  Detects a fire if it has spread to the same room the person is in.

- `observer`

  Detects a fire if it has spread to a cell neighboring its cell.

2.2.6. Fire

`Fire` is the only feature which can only be present once in the model and thus is the only feature which doesn't need a name. It describes the cells which are on fire at the start of the simulation. If we want the fire to start on different locations, we simply add more cells.

```
Fire
{
    cells : [c01,c44]
}
```

During the simulation, fire will spread to neighboring cells in the same room, or spread to another room through doors.

2.3. Options

The output options can be defined as follows:

```
Options
{
    File : outputfile.txt
    Output : [human|parse]
    Silent
}
```

Each option is optional, and can thus be omitted and the order is interchangeable. `File` writes the output of the simulation to `outputfile.txt`. If there is no output file defined, the simulation will be printed to the console.

The Output option defines the style of the output; `human` writes the output in easy to understand sentences whereas `parse` creates lines which are easy to parse:

`TIME| EVENT: EVENT DETAILS`

Finally, `Silent` disables the output of the simulation, except if dangerous conditions are met and the final results of the simulations.

2.4. Simulation

If there is a valid Bmod file in the project, two `.java` files will be generated in the `src-gen` folder of the project. One `FloorPlan.java` file per project, and

one `Simulate*.java` file per model (i.e. `.bmod` file).

The `FloorPlan.java` file defines all the classes and functions needed for the simulation, and the `Simulate*.java` file initializes the simulation for a specific model/configuration. The simulation can be run by running the `Simulate*.java` file.

How those files are generated, and how the Bmod models are validated is explained in the next section.

3. Development of Bmod with Xtext

Bmod was created through the creation of only three files, `Bmod.xtext`, `BmodValidator.java` and `BmodGenerator.xtend`. A fourth file tests the validator and parser; `BmodParsingTest.xtend`. Everything else does Xtext for you!

3.1. The Grammar Language

The first step to create our DSL is to describe the concrete syntax using Xtext's grammar language. The grammar language is, on its turn, also a DSL created using Xtext. In this case, our concrete syntax is defined in `Bmod.xtext`.

In the background, Xtext leverages the ANTLR parser, which is an LL(*) parser.

When Xtext parser our `.bmod` files, it uses the Eclipse Modeling Framework (EMF) models as the in-memory representation of the parsed text files. Our model is thus translated to an abstract syntax tree (AST). A meta-model of this AST is also defined in a language called Ecore. Ecore is defined in its self, and is thus its own meta model (meta-circularity).

3.2. Validator

Xtext automatically validates the syntax of our DSL. But it also makes it easy to write our own custom validation rules that are specific to our domain.

This makes it possible to, by example issue an error once the model is saved, but not all cells, or rooms are connected. This increases the user-friendliness of Bmod, because it makes it almost impossible to create an inaccurate model.

3.2.1. Custom Validation Tests

To test those custom validation rules, Xtext makes it possible to define an `.xtend` file. In this file, you can define unit tests for your validation rules. We can feed it a model, and assert a certain warning or error is given.
This allows us to be sure that our validation rules work as expected.

3.3. Generator

Now that we have defined our grammar, and our custom validation rules, it's time to put our DSL to action. Xtext provides us with an `.xtend` file which is run each time our DSL is saved. This makes it possible to, either run our simulation, or, as is the case with Bmod, translate it to another language, in our case Java.

Bmod is translated to Java, because, on the one hand, it makes it possible to inspect the code, and study how the simulation works.
On the other hand, this makes it possible to build a program around the generated code of Bmod, and extend upon it. By example, it could be possible to make an graphical user interface on top of it, if that's needed.

4. Comparison with metaDepth

metaDepth is a framework for deep meta-modeling created by Juan de Lara and Esther Guerra. [2] While metaDepth, and Xtext could both be used to define a meta-model or concrete syntax of a DSL, Xtext goes way further.

Xtext makes it possible to easily create a custom validator, code generator, linker and even include existing Java concepts.

While in the case of Bmod, implementation via metaDepth or Xtext has, at it's core the same result; a grammar to define a model, and the possibility

to run a simulation of this model, it isn't that difficult to come up with a case where metaDepth wouldn't suffice.

A LaTeX-like language model could be constructed through metaDepth, but the real implementation of the language would only be possible with Xtext, or similar frameworks.

5. Conclusion

In this paper, we introduced the concept of a DSL for the modeling of a floorplan, and simulate the behavior of occupants in the case of an emergency. Furthermore, we briefly discussed how Xtext made it possible to easily create a powerful DSL.

References

[1] K. Czarnecki, Overview of generative software development, UPP '04 (2004) 326–341.

[2] J. de Lara, E. Guerra, Deep meta-modelling with metadepth, TOOLS Europe (2010) 1–20.