Using Xtext and Xtend to Create Bmod

Randy Paredis, s0151613

Master in Computer Science, University of Antwerp

Abstract

In this document, we will try and describe the abilities and drawbacks of using Xtext (and Xtend) within the context of creating a building modelling language. The modularity of Xtext will be explored and combined with a C++ pedesitian simulator.

Keywords: Bmod, Breact, C++, DSL, Java, Modelling, Model Driven Engineering, PedSim, Pedestrian Simulator, Xtend, Xtext

1. Introduction

Within this document, we will explore the Java library of Xtext [1] and by extension Xtend. In order to describe all the possibilities and features Xtext offers, we will use it to create Bmod, a building modelling language and Breact, an additional file specification that allows for modelling the behaviour of people. The exact idea behind these languages will be respectively described in *Sections 3* and *4*, while *Section 2* will go deeper into detail on Xtext and Xtend itself.

Seeing as any file can be generated in Xtext, we will limit the scope of this project to generating a C++ project will be generated that will be used in conjunction with PedSim[2] to simulate the behaviour of people in emergency situations. More details will be discussed in *Section 5*.

1.1. Related Work

This project is based upon the same constructs as [3], but uses another angle on the topic (Building Modelling instead of RPGs).

Preprint submitted to MSBDesign *January 15, 2019*

1.2. Code Base

All the code for this project was made publicly available on GitHub.

2. What is Xtext and Xtend?

2.1. Xtext

Xtext [1] is a Java library/framework that allows software engineers to create custom DSLs (Domain Specific Languages). It is based at the Eclipse Org [4], but was not designed to be solemnly an Eclipse plug-in. Originally, the project was meant to also be available for JetBrains' IntelliJ and web applications, but only the Eclipse plug-in reaches its full potential.

> *For IntelliJ IDEA the situation is different. Neither the Xtext integration has been updated with the last release, nor has Jetbrains yet started to work on LSP support. The code for the IDEA integration is quite extensive and deep. So deep that we get regularly broken because we use non-public API. Since the demand for IDEA integration is not high, maintaining it doesnt make sense to us.* [5]

Needless to say, [5] clearly states the IntelliJ plug-ins for Xtext are discontinued.

When taking a look at the web-integration, we can see that an IDE can be easily obtained from the Eclipse plug-in (if you know what you're doing), but it still requires applets, or other Java entry points. This is mainly the reason we will not be focussing on this part of the plug-in.

2.2. Xtend

Xtend is, as its name implies, an extension for Xtext. It is a modular language that uses Java as an entry point and has a lot of practical (and less practical) features of common programming languages.

One of the handy features that will become available is a templatable string, which can prevent a lot of concatenation structures. The downside of this feature is that it uses the French guillemets, which by default don't come on a keyboard. So it's a good thing that Eclipse will automatically make them available within the correct context when using the auto-completion shortcuts[1].

Behind the scenes, it is translated to plain Java code, which is bundled with the project and compiled when necessary. This makes the two of them interchangeable within an Xtext project.

2.3. General Workflow

Generating a custom DSL with Xtext needs an accustomed workflow, that will be described here (apart from the obvious "Create a Project").

2.3.1. Create a Grammar

Using the power of Antlr4 [6], combined with a set of handy shortcuts (such as cross references, ranges, until-structures[2]...), any file specification can be defined.

Xtext is rather proud of one of those shortcuts, called "Cross References", which allows the user to give their language an automatic validation for existing objects. Beside its functionality being incredibly useful and powerful, I've noticed a massive drawback when making my project. When using such cross references, you can only use a single one in the same rule of the grammar. The workaround? Split up the rule in multiple rules.

2.3.2. Compile the Grammar

Making use of the power of MWE2, the project's workflow can be described in a JSON/YAML-like structure. Within this file, you can (among others) define the set of languages to compile and which file extensions to associate them with.

For Bmod and Breact, we have two different grammars, with dissimilar file extensions, but Bmod references structures from Breact. All this information can be described in this MWE2-file.

[1] `CTRL + SPACE` if no remapping was done, or `CTRL + LESS THAN` and `CTRL + GREATER THAN` for a specific type of guillemet.

[2] This structure allows you to match anything between two sets of tokens.

See the GenerateBmod.mwe2 file for an example on this.

2.3.3. Add Generation, Scoping and Validation

By default, Xtext will add an empty generator and an empty validator. The scoping will default to the Java-scoping. Luckily Xtext allows for overriding these classes.

The Generator is the class that will create a set of files based upon the grammar. Unfortunately, it is not possible to include a custom-made logic library to be linked in this grammar. Let's say (within the context of Bmod) we need some logic that is static, for instance to describe the logic of visually representing the simulation. A good feature would have been to be able to create a file that has this set of files as library. In this project, I've worked around this problem by explicitly setting the JAVA_ROOT in Eclipse to the workspace directory and by implementing a class that (using the Java file-handlers) loads a file into a string, which can be send through the generator.

The ScopeProvider was not wildly used in this project, but after doing some research, I've noticed it can be quite powerful.

The Validator class allows the creator of a DSL to describe when the semantics of the language becomes invalid. For Bmod and Breact, this has been used to check the constraints of both languages.

In addition to the Validator, it is possible to add Fixes for certain invalid constraints. For instance, in Bmod, this has been used for (among others) to allow for swapping the coordinates of an area if it was not defined from top-left to bottom-right.

3. Designing Bmod

Bmod: A Building Modelling Tool. Continuing on the assignments throughout the semester, this project makes use of Bmod, a fictional language that should allow architects to easily create floorplans and run simulations on them. The main focus of this year was a fire in the building, which is also what we will be discussing in this paper.

To create a Bmod file, you must create a file with the extension bmod. In this file, it is possible to add comments in the same way as most languages do that:

```
// This is a single-line comment
/* This is a multiline
        comment */
```

3.1. What is in Bmod?

Bmod is quite a simple language, but nevertheless it's important to state all concepts within this DSL and built them incrementally. All the information below is based upon the information given in [7].

Cells. A cell is the smallest unit of size in our model. Depending on the exact requirements/needs of the architect/user, these will be bigger or smaller. They are represented with a coordinate (x, y), which have its origin in the top-left corner. A cell itself should not be instantiated by the user to disallow for a bad floorplan creation.

Rooms. A collection of cells is what we call a room, or an area. Internally, in our system, we will use the terminology area to describe a group of connected cells, that can be joined and can have certain cells missing.

We're also making a simplification and say that a room will not have any internal walls.

A square room called cusom_room_name from $(0, 0)$ to $(5, 5)$ can be created in Bmod as follows:

```
Room custom_room_name
        from (0, 0) to (5, 5)
```

Another way of doing this is using joins (keyword and):

```
Room custom_room_name
        from (0, 0) to (4, 5) and
        from (5, 0) to (5, 5)
```

If you don't want the center in a 3 by 3 room anchored in $(0, 0)$? Simply make use of the `without` keyword:

```
Room custom_room_name
        from (0, 0) to (3, 3)
        without (1, 1)
```

Doors. When we have multiple rooms, we must also have something that connects them, which are doors. These doors are made up by two neighbouring coordinates of two different rooms.

A door can be created similarly to a room:

```
Door my_door_name from (0, 0) to (1, 0)
```

Please note that the `from` and `to` cells are interchangeable, yielding the above example exactly the same as:

```
Door my_door_name from (1, 0) to (0, 0)
```

Exits. To allow the building to be exitted, some cells can be marked as exit cells. Please be aware that it is required to have at least one exit in a floor.

When you want to mark cell $(5, 6)$ as an exit cell, you just write this:

```
Exit in (5, 6)
```

Fire. Similarly to describing an exit, a fire can be described, which denotes where a fire is created on the floor.

Igniting cell $(8, 8)$ can thus be done like this:

```
Fire in (8, 8)
```

Emergency Signs. Each door can have one or more signs that denote the emergency path. It points towards the next door to go to, when going through this one.

Describing an emergency sign that points you from door `door1` to `door2`, must be done as so (of course both doors must exist somewhere in the file):

```
EmergencySign from door1 to door2
```

Persons. Of course, in order to do some simulations, we must have some people that will undergo the fire in the floorplan. Each person belongs to a coordinate (which can change throughout the simulation), has a name and an action profile. The latter is a reference to a `Breact` action.

To create a person called `my_person` in $(3, 3)$ that follows action `my_action` from the `actions.breact` file in the same project, you must use this syntax:

```
import "actions.breact"
Person my_person in (3, 3): my_action
```

Please do note that the references to `Breact` (using the keyword `import`) must be defined in the beginning of the file, not just anywhere.

Dangerous Conditions. As per the assignments during this semester, there is only one single type of dangerous condition: the *occupancy condition.* This condition forces the simulation to halt, or show a warning (or both) that it has been reached.

Instead of implementing a dangerous condition similar to all other constructs, the decision was made to allow it inside of the creation of a room:

```
Room custom_room_name [3]
        from (0, 0) to (3, 3)
        without (1, 1)
```

As you can see, there is a `[3]` behind the name of the room, which implies an occupancy condition of 3.[3] When this constraint is reached, the user will be notified.

[3] At least, in this example. The value can be any positive real number, including 0. Why you would ever want to use 0, is beyond my understanding.

3.2. Constraints

All the above concepts still allow for misuse, bad design and undefined behaviour, which is why we will define some constraints on all the above objects. These are checked in `Xtext`, in the code validation.

1. Each room must be made up of a group of connected cells. To simplify, each cell in a room must be reachable from every other cell. This can easily be obtained by a floodfill-like algorithm.
2. Rooms cannot, may not and shall not have any cells in common, i.e. two different rooms must not have any overlapping areas.
3. Similar to the first constraint, all rooms must be connected to one another and thus must be reachable.
4. As said before, the two cells of a door must be neighbouring and must be part of two different rooms. This is to prevent a door from being placed in the middle of a room.
5. Each floorplan must have at least one exit.
6. An emergency path cannot be cyclic. That is, any path from an emergency sign cannot eventually point back to itself.

3.2.1. Implicit Constraints

Besides the above-mentioned constraints, there are others that are checked, but not necessarily correlate to the model.

1. Each area in a room must be defined from top-left to bottom-right.
2. Both the locations of a fire and an exit must exist within the floor.
3. An emergency sign may not cause the exit to be non-reachable.
4. An import must point to a valid file.
5. The action of a person must be defined in any imported file.
6. The action of a person must be unique.

4. Designing Breact

Breact, a combination of Bmod and the word react, is the language that comes bundled with the Bmod project and describes how people react in an emergency, by following a certain set of rules.

A Breact file has the extension of breact or br and, just as with Bmod, comments can be used in exactly the same way.

4.1. Actions

The main construct of a Breact file is an action, which has a name and a set of find statements, each ending in a semicolon. The name of an action must be defined, so it can be used in Bmod. But, note that there are no namespaces or similar constructs, yielding in undefined behaviour for actions that have the same name. This is automatically checked in the validator for Bmod, though.

An action can additionally be defined as shared, allowing for cooperation. All persons with a shared action profile within the same room will act according to a single profile[4].

The idea of shared actions came from [8], but can be expanded in a more psychological manner.

4.2. Find

Each action consists of a set of find statements. The most simple find statement is as follows, which does what it says on the tin (locate the nearest exit):

```
find nearest Exit;
```

Besides the use of nearest, it's also possible to find the farthest, the first or the last of any object (being one of Cell, Fire, Door, Person or Exit). For completion's sake, let's describe the meaning of all different selectors:

nearest Finds the closest object.

[4] Technically, they will react according to the first profile that could be found.

farthest Finds the farthest object, e.g. the object that is the farthest away from the current person.

first Finds the first object that suffices.

last Finds the last object from the list.

Both `first` and `last` can come in handy when looking for doors, seeing as the doors will be ordered according to the emergency path. Finding the last door thus means finding the door that is farthest along the emergency path (and logically closer to the exit).

Another way of modifying the search is by using `farthest from` instead of merely `farthest`, which will look for the non-burning cell that is farthest away from the described object.

```
find farthest from Fire;
```

By default, a find statement will look for these within the same room as the person is in. If you would rather it find on the entire floor, the find can be modified by a `global` keyword:

```
global find nearest Exit;
```

Another way to describe what to find is to modify the objects. Which can be done by adding either `burning` or `normal` to the objects. When using the former, the object that needs to be matched must be on fire, whilst, when using the latter, it may not be on fire. When using none of the above, it will mach any object.

```
find farthest from burning Door;
find nearest normal Exit;
```

4.3. Combined

Now that we know how find-statements can be constructed and we know that an action consists of a series of them (at least one), we can identify the semantics of an action.

```
// Original experienced action profile
action experienced:
        find nearest normal Exit;
        find nearest normal Door;
        global find nearest normal Exit;
```

The above Breact code segment will do the following (for each person with the action of experienced):

1. Find all the exits in the room that are not on fire. If there are exits, try moving to the nearest one.
2. If no exit was found, try looking for the nearest door in the room and go through that door.
3. If (somehow) there are no doors, move to the cell that is the closest to an exit on the floor.

It's also good to denote that the distance function uses maze-distance throughout the floor and not the simpler euclidean distance.

5. Connecting the Pedestrian Simulator

PedSim is a microscopic pedestrian crowd simulation library[5]. It allows for creating obstacles and agents which are simulated over an interval of time. [2]

We will be using the aptly named 2dvis from PedSim's ecosystem, listening on port 2222. Our generator will create a program that will send XML messages to the UDP port of 2222, which can be read by 2dvis.

5.1. Compilation

Seeing as PedSim was written in C++ and Xtext is a Java library, we have to create a simple mapping in order to simulate our Bmod and Breact files.

But, unfortunately, there is quite a set of additional logic and functionality that must be liked and compiled validly, which is why our generator creates a

[5] The words of the creator.

cmake-project that holds the name of our Bmod file. Simply execute the command `cmake .; make` in the project's root folder and you're good to go. Now you can start the simulation by executing simxxxx where xxxx stands for the project name.

For instance, let's say we have an example.bmod and an actions.br file. By default, there should be a project folder called example, which, when compiled, will create a simexample file.

6. Related Work

7. Conclusions and Future Work

I have shown in this paper that anyone with no notable experience with Xtext (but of course with programming experience) can create a simple set of DSLs, which can generate anything.

Unfortunately, there are still some issues with Xtext, like the inability to add custom libraries and the convoluted manner in which rules with multiple references must be created in grammars. Also, the documentation of the new system is lacking terribly in the necessary explanation on MWE2 and there were a lot of precise features that I had to scramble together from forums and outdated tutorials in order to arrive where this project is now.

Just as with any project, Bmod/Breact is still expandable to more features and functionality, like:

- The ability to interrupt going to a waypoint and changing direction. i.e. interruptable action.
- The ability to add urgencies to each find.
- Being able to "spawn" a random group of people.
- Ability to target any object.
- ...

References

[1] Xtext home page.
URL https://xtext.org

[2] PedSim home page.
URL http://pedsim.silmaril.org

[3] V. Hannu, Making domain specific language with xtext and xbase for creating role playing games (2014).

[4] Eclipse home page.
URL https://eclipse.org

[5] TypeFox, Xtext LSP vs. Xtext Web.
URL https://typefox.io/xtext-lsp-vs-xtext-web

[6] Antlr home page.
URL https://www.antlr.org/

[7] C. Gomes, Assignment 1 - building evacuation modelling in metaDepth.
URL http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/assignments/Assign1-metaDepth.pdf

[8] X. Zheng, Y. Cheng, Modeling cooperative and competitive behaviors in emergency evacuation: A game-theoretical approach (2011).