Spoofax VS Xtext
a language workbench comparative case study

Leonard Elezi[a]

[a] *University of Antwerp, Antwerp, Belgium*

Abstract

Language workbenches are tools that offer mechanisms for creating domain-specific languages. Spoofax and Xtext are two such workbenches. In this paper a small DSL for a role playing game is designed and implemented with each tool.
This implementation serves as the comparison basis in order to reach a conclusions and find out the strengths and weaknesses of each tool.

Keywords: Spoofax, DSL, Stratego/XT, Xtext, Xtend, rpg

1. Introduction

Language workbenches are tools that offer efficient support for defining, composing and reusing languages and their IDEs. These workbenches make the whole development process of a new language affordable and straightforward. There have been many tools for creating DSLs through the years. For today's standards though, this is not enough. We have grown to expect more than just a compiler. We expect a full blown IDE with code completion, error detection and refactoring.
This is where language workbenches shine. They offer the possibility to create DSLs and use them directly on the same environment without deploying. They assist with the creation not only of the language but also of the IDE where the language can be used. In this paper we will compare two such tools that follow different approaches in language design.

Email address: Leonard.Elezi@student.uantwerpen.be (Leonard Elezi)

1.1. Comparison Criteria

To compare the tools some basic criteria have been laid out.

- Collaboration with other tools
  Development tools nowadays are rarely used in isolation. A language workbench should also be integrated with other tools such as version control systems and build/testing environment.

- IDE features Modern IDEs like Eclipse have many features like highlighting, error checking, debugging refactoring etc. Also it should provide means for exploring the language and documentation. This is especially useful when creating a DSL. Also it should assist the developer when creating and designing a DSL and offer tools to also implement IDE support for the DSL.

- Testing and Debugging This can be considered part of the IDEs features but since it's such a crucial step when developing we have given it it's own spot. What does the workbench in terms of debugging and testing the DSL?

- Concrete and Abstract Syntax How do the workbenches define language structure and syntax?

- Constraints Constraints are used to ensure the static semantics of a language. How are they implemented in our two tools?

- Transformation and generation When transformation or generation is performed, another artifact is created from a program, usually another program in a less abstract language.

In the sections to come we will explain how the tools behave when measured to the criteria above.

1.2. Spoofax

Spoofax is a platform for developing textual domain-specific languages with full-featured Eclipse editor plugins. With the Spoofax/IMP language workbench, you can write the grammar of your language using the high-level SDF grammar formalism. Based on this grammar, basic editor services such as syntax highlighting and code folding are automatically provided. Using high-level descriptor languages, these services can be customized. More sophisticated services such as error marking and content completion can be specified using rewrite rules in the Stratego language.[ref01]

1.3. Xtext

With Xtext you can create your very own languages in a snap no matter if you want to create a small textual domain-specific language or a full-blown general purpose programming language. Also if you already have an existing language but it lacks decent tool support, you can use Xtext to create a sophisticated Eclipse-based development environment providing editing experience known from modern Java IDEs in a surprisingly short amount of time. We call Xtext a language development framework. [ref02]

2. RPG DSL

We chose to design a DSL that will allow us to model role playing games (RPGs) after. We first design the DSL and implement the grammar, then add constraints and in the end performa code generation. The code that results we execute it in a python game framework which will show an animation of the RPG. Our RPG language has the following requirements:

- An RPGGame consists of exactly one scene (or "level"). The scene has a name, such as Forest, Desert, Castle, etc.
- The scene has a number of connected tiles
- Tiles can be connected to each other from the left, right, top or bottom. This way, a map is created for the scene.
- In the game, there is one character: the hero. The hero is always on exactly one tile.
- A tile can be an empty tile, or an obstacle, on which no character can stand.
- On an "standard" tile (not an obstacle), there can be a goal. There must be one goal.

Note that our implementation does not satisfy each and every one of these requirements but mostly serves as a proof of concept. The language we came up with was:

```
rpg FirstGame {
        scene HelloWorld{
                width=2;
```

```
                height=2;
                tile tile_0 {
                        x=0;
                        y=0;
                }
                tile tile_1 {
                        x=0;
                        y=1;
                }
                tile tile_2 {
                        x=1;
                        y=0;
                }
                tile tile_3 {
                        x=1;
                        y=1;
                }
                hero perseus{
                        tile = tile_0;
                }
                goal win {
                        tile = tile_3;
                }
        }
}
```

We have a rpg game container which can contain only one scene. The scene on the other hand has the size, a number of tiles, one hero and one goal. The tiles have their coordinates and the hero with the goal have the tile on which they stand defined. Right of the bat we can come up with many constraints we need to implement in order for the language to be as complete as possible. For example check that the hero and the goal don't stand on the same tile, or even that the tile they are standing actually exists. But since this is mostly a proof of concept we have only implemented constraints regarding the number of heroes or goals a scene should have, or the number of scenes a world/game can have.

3. Concrete and abstract syntax

The concrete syntax of a language is what the user interacts with to create programs. Basically the language we came up with in the above section. In our case it is textual for both of our tools. The abstract syntax is what it's called a data structure that holds the fundamental information in a program, but without notations of the concrete syntax like keywords, symbols, comments etc. Basically the abstract syntax is a tree data structure. Both Spoofax and Xtext use what it is considered a parser based approach in generating the abstract syntax tree (AST). Basically the concrete syntax gets parsed, non-core information gets "discarded" and the abstract syntax tree gets created. The way the AST is respresented in Spoofax is through ATerms (a structure similiar to XML or JSON) while Xtext is heavily dependent on the Ecore models. Basically in Xtext the AST is represented as an instance of Ecore. The syntax of the language in Spoofax is described by using SDF while in Xtext by using an EBNF-like notation which also specifies the mapping to the abstract syntax.

3.1. Xtext

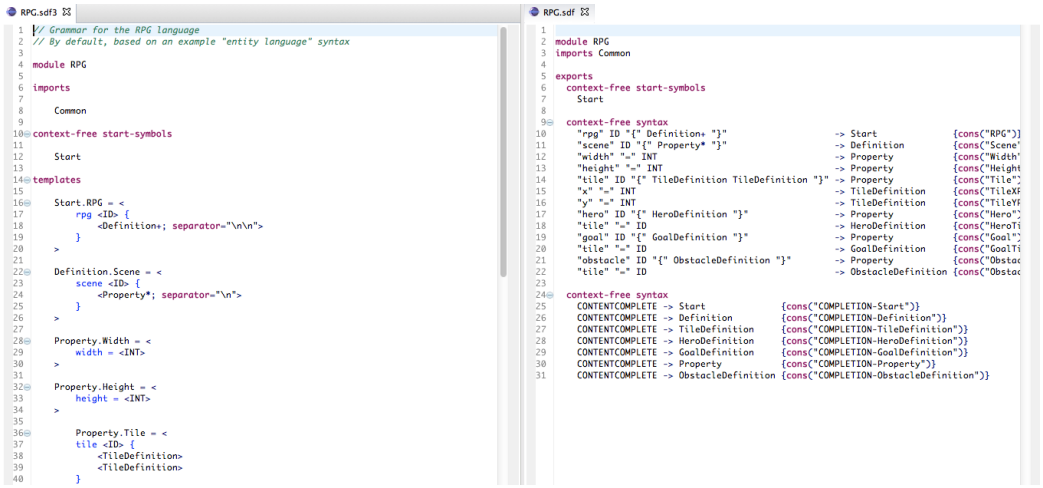
Part of the grammar for the language defined in Xtext:

```
Model: rpg = RPG;
RPG:
    'rpg' name=ID '{'
        scenes = Scene
    '}'
;

Scene:
    'scene' name=ID '{'
        'width' '=' width=INT ';'
        'height' '=' height=INT ';'
        tiles+=Tile+
        hero=Hero
        goal=Goal
        obstacles+=Obstacle*
    '}'
;
```

The code is self explanatory. We first create a model, rpg in this case which is comprised of the RPG rule. Then we define the RPG rule to be of the rpg keyword followed by a name followed by an opening bracket. Inside we want to have a scene which is given to us by the Scene rule. After the scene is declared we close the bracket. The Scene rule follows the same pattern.

3.2. Spoofax



Defining the grammar in Spoofax is a tad complicated and it requires a while to get the head around it. As we learned the best thing to do is work side by side with an example file to test the grammar while you are building it. Also if you actually fill the RPG.rdf3 file it will automatically update the RPG.rdf file which has the real grammar. The rdf3 file is a nice and intuitive way to edit the grammar.
As can be seen from the figure above our grammar specifies a module RPG. The good thing about Spoofax is that it is modular, which means that we can modularize our program and important only the parts that we actually need at a specific time. The next line does just that, imports a module called Common. Then we declare where the parsing should start and after we define the rules which will take the form of ATerms after.
All this is done through the help of the SDF (Syntax Definition Formalism). The rule in a nutshell is that the pattern on the left-handed side of the arrow is matched by the symbol of the right handed side. After the right handed side annotations maybe specified using curly brackets.

```
"rpg" ID "{" Definition+ "}"  -> Start  {cons("RPG")}
```

4. Constraints

Constraints are Boolean expressions that must be true for every program expressed in a specific language. Since not all the programs that conform to the grammar, are valid programs we need constraints to enforce and ensure the static semantics of a language.

4.1. Xtext

Constraints in Xtext are implemented in Java or Xtend. They are added to a validator class generated by the Xtext project wizard. We did not implement any constraint in our project, except the ones implemented in the grammar itself.

4.2. Spoofax

Spoofax uses rewrite rules to specify all the constraints. This is done through the Stratego/XT framework. They are easy to recognize because they exist in files with extension .str. There are two types of constraint rules in Spoofax: Basic and Index-Based. Basic are constraint such as constraint-error, constraint-warning which indicate constraints that trigger specific actions when an error or a warning occurs. Usually these constraints are overwritten by developers. Index based constraint rules are rules that need to interact with the Spoofax index.

5. Transformation and generation

In both transformation and generation another artifact is created from a program, usually with a less abstract language. With transformation the created artifact is an AST while code generation a textual concrete syntax is created.

5.1. Xtext

Since Xtext is based on EMF, any tool that can generate code from EMF models can be used. In our example we used Xtend, since it's the recommended one. Our generator is an Xtend class that implements IGenerator which requires doGenerate method to be implemented. The code below shows the implementation:

```
class MyDslGenerator implements IGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess fsa) {
        for (e : resource.allContents.toIterable.filter(typeof(Scene))){
            fsa.generateFile('game.py', e.compile);
        }
    }

    def CharSequence compile(Scene scene){
        '''
        from RPGGame import *
        worldMap = WorldMap()
        «scene.name»Scene = Scene(«scene.width»,«scene.height», "dungeon.jpg", "«scene.name»", True);
        worldMap.addScene(«scene.name»Scene)
        «FOR tile : scene.tiles»
            «tile.name»«scene.name» = Tile(«tile.position_x»,«tile.position_y»)
            «scene.name»Scene.addTile(«tile.name»«scene.name»)
        «ENDFOR»

        «scene.hero.name» = Player("hero.png", "«scene.hero.name»")
        «scene.hero.onTile.standingTile»«scene.name».setOccupant(«scene.hero.name»)
        worldMap.setHero(«scene.hero.name»)

        «scene.goal.name»«scene.name» = Gold("goal.jpg")
        «scene.goal.onTile.standingTile»«scene.name».setOccupant(«scene.goal.name»«scene.name»)
        worldMap.addGoal()

        «FOR obstacle : scene.obstacles»
            «obstacle.name»«scene.name» = Mountain("obstacle.jpg")
            «obstacle.onTile.standingTile»«scene.name».setOccupant(«obstacle.name»«scene.name»)
        «ENDFOR»

        runGame(worldMap)

        '''
    }
}
```

First we iterate over each scene and for each scene we begin the code generation process. Since we don't need fine grain structure we decided to go for a template based approach with the code generation. We first generate the code for the scene then for the tiles and then for the hero and the goal. Keep in mind that the code generated is Python and not Java.

```
14              result   := <to-java> selected
15
16  rules // Transformation to java strings.
17
18      set-global = ?(sceneName,value) ; rules(get-global: name -> value)
19
20    to-java:
21      //[_] -> "Hello world 1"
22     [_] -> <concat-strings> <map(to-java)>
23
24    to-java:
25      () -> "Hello2"
26
27    to-java:
28      RPG(x, d*) ->
29      $[from RPGGame import *
30        worldMap = WorldMap()
31        [d'*]
32        runGame(worldMap)
33       ]
34       with
35         d'* := <to-java> d*
36
37    to-java:
38      Scene(x, p*) ->
39      $[[x]Scene = Scene([p'*]
40      ]
41      with
42        rules(sceneName := x);
43        p'* := <to-java> p*
44
45    to-java:
46      Hero(x, HeroTile(t)) -> $[
47      [x] = Player("hero.png", "[x]")
48      [t][<sceneName>].setOccupant([x])
49      worldMap.setHero([x])
50      ]
51
```

As said previously in Spoofax code generations is specified by rewrite rules. This has the advantage of being the same for model to model transformation also. We use string interpolation, which in the code it happens inside $[...] brackets and allows us to combine fixed text with variables bounded to strings. One thing worth noting is that the string interpolation preserves indentation and since we were generating python code it was not good. We had to manually adjust the indentation at the end of the generation in order for the code to run.

6. IDE features

IDE features are the features that the workbench is able to give to the IDE which will be used to develop the DSL. We didn't do any experiment here but used the ones out of the box. There is no doubt here that Xtext has better support out of the box then Spoofax. Code completion, error detection and syntax coloring were particularly better. Spoofax also supports them but you also have to dig around alot to know how to activate them.

7. DSL testing

Both Spoofax and Xtext offer support for testing and debugging the DSL and also unit testing during development. There is no particular favorite in here since we used very little tests. Also the debugging process is way easier in Xtext. With Spoofax you have to download the nightly build to enable debugging and go from there.

8. Conclusions

Both Spoofax and Xtext represent state of the art language workbenches. From the development of our RPG DSL it was our experience that Xtext is more popular and has many resources available in case you need help. Also the functionality and user experience provided out of the box with Xtext is richer. With Spoofax there is only the official site. Spoofax seems to be mostly used in academic environment. Though we would argue that if you get the hang of rewrite rules and SDF Spoofax is a very powerful tool.

9. Bibliography

(a) `http://strategoxt.org/Spoofax` [ref01]

(b) `http://www.eclipse.org/Xtext/` [ref02]

(c) DSL Engineering - Designing, Implementing and Using Domain-Specific Languages (2013), pp. 1-558 by Markus Voelter, Sebastian Benz, Christian Dietrich, et al. [ref03]

(d) Implementing Domain-Specific Languages with Xtext and Xtend, L.Bettini [ref04]

(e) Lennart C.L. Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10).