Textual languages with Xtext[1]

Fedor Biryukov

University of Antwerp

Abstract

Keywords: Modeling, Xtext, RAMification[2]

1. Introduction

In his lections [1], Vangheluwe pays a lot of attention to visual concrete syntax and it is almost asserted that domain-specific modeling is done in either visual or hybrid manner. While I am convinced that visual modelling has a lot of advantages and far more expressive power [2] (shape, color etc.), I am eager to investigate textual languages as well.

In section 2 I give the requirements for the (family of) railway formalism(s). In section 3 I develop one such formalism with Xtext. In section 4 I model a formalism to specify railway patterns. In section 5 I build upon section 4 and introduce a notion of "action rule" that consists of a railway pattern and an executable action. In section 6 I show by example how action rules can be used to animate a railway system. In section 7 I describe the main contribution of this paper, namely, how the RAMification[2] is done. In section 8 I provide my conclusions and ideas for future work.

2. The family of railway formalisms

A railway system consists of a set of interconnected segments(straights, stations, turnouts and junctions), a set of trains and a set of schedules.

Straights and stations have one input and one output. Each station has a unique name and must be connected to at least one other segment. Turnouts have one input and two outputs. Junctions - vice versa. At the end of each segment there is a light which is either green or red. Each train has a unique identifier and a schedule(start station, end station and an instruction for each turnout). Turnouts and junctions have two modes: straight and diverging. While undesirable at run-time, the formalism should allow for more than one train to be present on a railway segment.

There are other restrictions that should be checked during the validation of a railway system. But these are less relevant to the work I present here. So, I take the liberty to leave them out of consideration.

[1] `https://eclipse.org/Xtext/`

[2] **RAM**ify is a shorthand for **R**educe, **A**ugment, **M**od**ify**.

Email address: `fedor.biryukov@student.uantwerpen.be` (Fedor Biryukov)

$$
\begin{aligned}
Railway &\equiv < Segments, Trains, Schedules > \\
Segments &= Stations \cup Straights \cup Turnouts \cup Junctions \\
Station &\subset AllowedNames \times Segments \times Segments \times Trains \times Lights \\
Lights &= \{RED, GREEN\}
\end{aligned}
$$

Figure 1: Formal specification of a railway system model (excerpt)

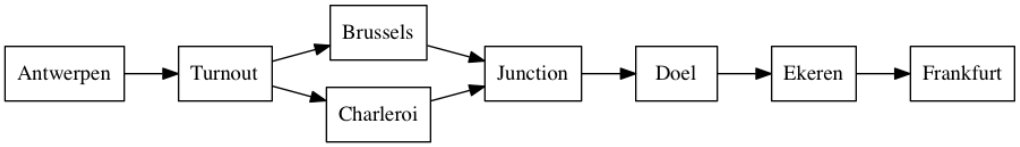

Figure 2: Sample railway system

3. "Railway model" language

My primary interest is to construct a textual language (or two). More specifically, a textual concrete syntax for a DSML from the family of railway formalisms. For example, I would like to write the following to specify the railway system from Figure 2.

```
railway RailwaySystem

// Segments
station Antwerp (; fromAtoBC)
turnout fromAtoBC (Antwerp; Brussels, Charleroi)
station Brussels (fromAtoBC; fromBCtoD)
station Charleroi (fromAtoBC; fromBCtoD)
junction fromBCtoD (Brussels, Charleroi; Doel)
station Doel (fromBCtoD; Ekeren)
station Ekeren (Doel; Frankfurt)
station Frankfurt (Ekeren; )

// Trains
train T0001
train T0002
train T0003

// Schedules
schedule S1 T0001 (Antwerp; Doel)
schedule S2 T0002 (Charleroi; Ekeren) STRAIGHT
schedule S3 T0003 (Frankfurt; Antwerp)
```

Listing 1: Sample railway system in RML

The Xtext grammar for this language includes enough information for both concrete and abstract syntax of the formalism. At the same time it is very concise. And if you

take a look at the formal specification in Figure 2 and at the grammar in Listing 2 you will notice the how similar they are.

```
grammar rml.RML with org.eclipse.xtext.common.Terminals

generate rml "http://www.RML.rml"

Model:
        'railway' name=ID (segments+=Segment | schedules+=Schedule |
            trains+=Train)*;

Segment:
        Station | Straight | Turnout | Junction;

Station:
        'station' name=ID '(' in=[Segment]? ';' out=[Segment]? ')'
        (trains+=[Train] ','?)* light=Light?;

Straight:
        'straight' name=ID '(' in=[Segment] ';' out=[Segment] ')'
        (trains+=[Train] ','?)* light=Light?;

Turnout:
        'turnout' name=ID '(' in=[Segment] ';' out1=[Segment] ','
            out2=[Segment] ')'
        (trains+=[Train] ','?)* light=Light? mode=Mode?;

Junction:
        'junction' name=ID '(' in1=[Segment] ',' in2=[Segment] ';'
            out=[Segment] ')'
        (trains+=[Train] ','?)* light=Light? mode=Mode?;

Train:
        'train' name=ID (segment=[Segment])?;

enum Light:
        RED | GREEN;

enum Mode:
        STRAIGHT | DIVERGING;

Schedule:
        'schedule' name=ID train=[Train] '(' start=[Station] ';' end
            =[Station] ')'
        (turnoutModes+=Mode ','?)*;
```

Listing 2: Xtext grammar of the RML formalism

Regarding the semantics of RML: from RML you can generate Xtend code(from which Java code is produced). Xtend code can output its state in RML, ensuring you can do the translation in both directions.

4. "Railway pattern" language

The first question here is: "What should a patter look like?"

If we want to match all stations, saying "station s" should be enough. However, you may want to find a station with a train on it(station has a reference to the train) or a station with a green light(value of light attribute is GREEN). In general, you should be able to restrict the selection based on the value of some attribute or on the references/links between different objects.

Consider the following example in which we look for a train that has a green light and may advance from the station it is currently at to the next one. We have a train(t), two stations(a and b) and references/links between them. We leave open what their names(and some other properties) are. However, we impose a restriction on the value of attribute light. It is implemented as a string which must contain valid Xtend code.

```
train t {
        name:
        segment: a
}
station a {
        name:
        trains: t
        light: 'it==Light.GREEN'
        in:
        out: b
}
station b {
        name:
        trains: t
        light:
        in: a
        out:
}
```

Listing 3: Sample railway system pattern in RPL

5. RPL action rules

Now that we can find things, let us also define the actions we want to execute. Consider the following example in which we move a train that has a green light from the segment it is currently on to the next one. Base class Segment does not contain any references to the other segments. So, we cannot say in the pattern that they are connected. But we can check it in the post-action that we introduce.

```
rule moveTrains
pre {
        train t {
                name:
                segment: a
        }
        segment a {
```

```
                matchSubclasses: true
                name:
                trains: t
                light: 'it==Light.GREEN'
        }
        segment b {
                matchSubclasses: true
                name:
                trains:
                light:
        }
}
post <<<
        if (b.train == null && a.isConnectedTo(b))
                t.moveTo(b)
>>>
```

Listing 4: Sample railway system action rule in (the second version of) RPL

Regarding the semantics of RPL. There is no automatic code generator yet. However, the above rule should produce code that is equivalent to the following.

```
def static RailwaySystem moveTrains(RailwaySystem it) {
  for (Train t : trains) {
    if (t.segment != null) {
      var a = t.segment
      if (a.train == t && [it==Light.GREEN].apply(a.light)) {
        for (Segment b : segments) {
          if (b.train == null && a.isConnectedTo(b))
            t.moveTo(b)
        }
      }
    }
  }
  return it
}
```

Listing 5: Sample railway system action rule, translated to Xtend

6. Experimentation

Imagine a model similar to that in Listing 1 with all lights set to green and only one train, T0001, which is in Antwerp. Xtend code automatically generated from such a (RML) model, augmented with the manually translated moveTrains rule from the previous section and the following main function would produce a trace with 7 snapshots in it, i.e. the train would move all the way from Antwerp to Frankfurt.

```
  def static void main(String[] args) {
    var rs = new RailwaySystem
    var prevSnapshot = ""
    var i = 0
```

```
    while (prevSnapshot != rs.toString) {
      prevSnapshot = rs.toString
      // output current state
      i++
      println('-----SNAPSHOT#'+i+'----------')
      println(prevSnapshot)
      // perform transformation rules
      rs.moveTrains
    }
  }
```

Listing 6: Sample railway system action rule, translated to Xtend

7. Automatic RAMification

I provide a RAMifier that takes an Ecore meta-model(abstract syntax) as input and produces a Xtext grammar and templates.xml(predefined code templates for the generated language), based on the principles outlined in sections 4 and 5. You can find the RAMifier in `rml/src/ram/RAM.xtend`, its main function shows how to use it. In this section I will explain how it works on a higher level of abstraction.

The start rule of the input grammar is removed and the following rules are added:

```
Model:
  rules+=Rule*;
Rule:
  'rule' name=ID pre=Pre post=Post;
Pre:
  {Pre} 'pre' '{' (segments+=Segment | trains+=Train | schedules+=
      Schedule)* '}';
Post:
  'post' body=TEXT;
terminal TEXT: '<<<' -> '>>>';
```

Rule `Model` is the start rule of the output grammar. Rule `Pre` here is the one generated for the Railway Model Language given in listing 2. It has three rule calls(Segment, Train and Schedule) because there are three "root classes" in the input Ecore meta-model, i.e. three classes that are not derived from any other class.

For each class in the input Ecore meta-model (except the one corresponding to the start rule) I create a separate rule in the output grammar. E.g., consider EClass Train that has an EAttribute name and an EReference segment. For this class the following rule is produced:

```
Train:
  'train' name=ID '{'
    'name' ':' nameExpr=STRING?
    'segment' ':' segment=[Segment]?
  '}';
```

As you can see, the rule has the same name as the EClass. For each attribute an "expression attribute" of type EString is created. You are supposed to write valid Xtend code there that returns a boolean value. This way you can filter matched objects based

on the value of each attribute. For each EReference a new EReference is created and the cardinalities are preserved. This way you can specify connections between objects. If the class is derived, it inherits all members(attributes and references) of the super class.

Super classes are handled in a slightly different way: I create two rules in the output grammar for each of them. One rule to represent the hierarchy. The other - to provide an implementation for the super class. Moreover, each super class gets an extra EAttribute to specify whether its subclasses are to be matched. Consider the following example:

```
Segment:
  SegmentImpl | Station | Straight | Turnout | Junction;
SegmentImpl:
  'segment' name=ID '{'
    'matchSubclasses' ':' matchSubclasses=('true'|'false')
    'name' ':' nameExpr=STRING?
    'trains' ':' (trains+=[Train] ','?)*
    'light' ':' lightExpr=STRING?
  '}';
```

All this is done by the `XtextGenerator`. You can find it in the `ram` package.

8. Conclusion and future work

Xtext is a fascinating framework. It makes engineering a DSL and developing an IDE for it easy, as never before, by combining a number of tools, such as:

(EMF) Ecore – a meta-model for describing models.

By default, the ecore model is auto-generated from the grammar. This makes inconsistencies between the abstract and the concrete syntax impossible and shortens development time.

A reason why using Ecore for the abstract syntax is wise is that this format is shared by the multitude of tools available in the Eclipse Modeling Framework. For example, it should be possible to develop a visual formalism based on it.

ANTLR – ANother Tool for Language Recognition.

ANTLR is used for the lexical analysis, parsing and code generation. Xtext builds upon ANTLR and introduces a lot of extra features that bring these three aspects of DSL engineering to a new level. Xtext grammars and code generators are very concise. I would even say that Xtend makes code generation a feast.

Google Guice – generic framework for dependency injection using annotations.

Guice allows you to customize almost every aspect of your language.

Regarding the future work, there is room for improvement in at least two directions:

1. Further improving the RAMifier
2. Using Xbase

Ideally, the RAMifier should generate not only the grammar and templates, but also the validator, code generator/type inferrer, and even tests.

I have not used Xbase in any of the formalisms I provide. However, I think it would be a very good idea to infer Java types for each model element and to use Xbase expressions (instead of opaque strings) at some strategic places, such as the conditional expressions and the action code in the pattern language. It would allow for code analysis, i.e. better feedback, instant error messages and "free" code generation.

After all, I do not claim that the pattern language provided here is perfect. I am even sure that a far better one can be conceived.

References

[1] Model driven engineering, `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/`, accessed: 2015-12-11.

[2] D. L. Moody, The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering, IEEE Transactions on Software Engineering 35 (6) (2009) 756–779. `doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2009.67`.