Domain-specific languages with JetBrains MPS: A comparison with AToM^3

Kevin Buyl

University of Antwerp, Belgium

Abstract

Since model-driven engineering is becoming more and more popular these days, it is good to spend some time analyzing the current tools for this job. In this paper we are going to build a domain-specific language for a traffic network in JetBrains MPS. This relatively new tool delivers a brand new concept of software development environment implementing the Language Oriented Programming paradigm, which is a style of computer programming in which, rather than solving problems in general-purpose programming languages, the programmer creates one or more domain-specific languages for the problem first and then solves the problem in those languages. Since we already implemented the traffic formalism in another modeling tool, AToM^3, we can use this to compare the two tools. Besides summing up the general equalities/differences between them, we can also give some advantages/disadvantages based on this traffic example.

Keywords:
Language Oriented Programming, Generative programming, Meta Programming, Model-Driven Engineering, Model Transformation, Domain-Specific Languages, JetBrains MPS, AToM^3

1. Introduction

These days a lot of computer programs are still being developed by a mainstream programming approach. With a general-purpose language such

Email address: `kevin.buyl@student.ua.ac.be` (Kevin Buyl)

as Java or C++, it is possible to implement almost every solution to a problem. However some solutions will take years to implement due to the nature of a general-purpose language. This is why mainstream programming has come to a dead-end. It is very unproductive and has several issues. It forces the programmer to think like a computer rather than have the computer think more like the programmer, like it is stated in Dmitriev (2004).

A domain-specific language (DSL) on the other hand lets us think more in terms of concepts. This way, it is much easier for the programmer to build his solution. But the strength of DSLs, domain specificity, is also their weakness. What we really want are different languages for every specific part of the program that can work together. To achieve this kind of freedom we need to create, reuse and modify languages. This is where language oriented programming comes in.

Terms like model-driven architecture, generative programming and intentional programming all specify a specific part of the model-driven engineering domain but in this paper we are going to unit them all under one name, language oriented programming.

In section 2 a brief introduction to language oriented programming is given, based on Dmitriev (2004) and Ward (1994). We will also see why this new paradigm is needed and we give some advantages over mainstream programming. In section 3 we will show you how to build the Traffic language in JetBrains MPS. In section 4 a comparison is made between JetBrains MPS and AToM^3. Finally we present our conclusions and future work in section 5.

2. Language Oriented Programming with JetBrains MPS

2.1. What is Language Oriented Programming?

This paradigm focuses on building a domain-specific language for the problem instead of developing the whole application in a general-purpose language, like C++ or Java. Like it is stated in Dmitriev (2004), the three steps of development of an application in a general-purpose language are:

1. Think: You have a conceptual model in your head.
2. Choose: You choose a general-purpose language.
3. Program: You write the solution by performing a difficult mapping from the conceptual model to the programming language.

The Program step is the bottleneck in the process because the mapping is far from easy. In section 2.2 we will see more on that and also look at some other issues with mainstream programming.

For language oriented programming on the other hand the process contains four steps:

1. Think: You have a conceptual model in your head.
2. Choose: You choose some specialized DSLs to write the solution.
3. Create: If there are no appropriate DSLs for your problem, then you create one.
4. Program: You write the solution by performing a straightforward mapping from the conceptual model to DSL.

The problem is now shifted to the Create step, but with the support of an IDE, this problem is not so challenging as the previous. The DSL is also transformed into a language that the computer understands, but this mapping is far more easy and usually only written once when the DSL is created.

In other words: everything starts by developing a high-level, domain-oriented, language. The development process then splits into two independent stages: (1) Implement the system using this 'middle level' language, and (2) Implement a compiler or translator or interpreter for the language, using existing technology.

2.2. Why do we need Language Oriented Programming?

There are several reasons for doing language oriented programming rather than doing mainstream programming. The main reason for this is due to the nature of a general-purpose language. We will now give some issues, proposed in Dmitriev (2004), with this kind of programming. The first one is the time delay to implement ideas. There is always a long gap between the idea of a solution and the solution itself in the form of a program. There always has to be a object-oriented design (OOD) step in order to convert the ideas to classes, methods and functions. With language oriented programming this is not needed anymore. A second issue is understanding and maintaining existing code. Even if the code is written by yourself, the problem stays the same. Since for general-purpose language the high-level idea is converted to low-level features of the language, the big picture is lost after the implementation step. Trying to reconstruct the main idea from the

low-level code requires a lot of effort and time. The third issue is due to the domain learning curve. With the OOD, extending the language is done by using class libraries. The problem is that those libraries are not expressed in terms of domain concepts, but in low-level general-purpose features. For example with a GUI or database library, which is not easy to learn, we have a steep learning curve because the mapping of the domain to this library is not obvious. If you are an expert in the domain and you are familiar with the library, there's still a the problem developing the application. It takes very long to actually build the application in such a library. This is because of the complicated constructions you have to make with simple components in the library.

In Ward (1994), they list several advantages of language oriented programming. The first one is the separation of concerns between design issues, which are addresses in a domain-specific language and implementation issues, which are addressed in the implementation of the language and are separated from the design of the system. Another advantage is the high development productivity. This is due to the fact that with a problem-specific very high level language, a few lines of code are sufficient to implement highly complex functions. The very high level language means that a small amount of code in this language can achieve a great deal of work. The third advantage is related to the previous one. Language oriented programming improves the maintainability of the design. With traditional programming it becomes very difficult for maintainers to determine all the impacts of a particular design decision, or conversely, to determine which design decisions led to this particular piece of code being written in this way. With language oriented development, the effects of a design decision will usually be localized to one part of the system. Another advantage of language oriented programming is a highly portable design. Porting to a new operating system or programming language becomes greatly simplified: only the middle language needs to be reimplemented on the new machine, the implementation of the system (written in that language) can then be copied across without change. A final advantage we're gonna discuss is the opportunity for reuse. There is a great potential for reuse of the middle level languages for similar development projects. The languages encapsulate a great deal of domain knowledge: including knowledge of which data types, operations and execution methods are important in this domain, and what are the best ways to implement them.

2.3. What is a language in Language Oriented Programming?

Before we can introduce JetBrains MPS, we first have to answer a very important question: "What is a language in language oriented programming?". To answer this question we refer to Dmitriev (2004), which states that a language is defined by 3 main components: structure, editor and semantics. The structure defines its abstract syntax (what concepts are defined and how they are arranged). The editor defines its concrete syntax (how it should be presented). Finally the semantics describe how it should be interpreted and how it should be transformed into executable code.

2.4. JetBrains MPS

The Meta Programming System (MPS) of JetBrains implements this new paradigm of language oriented programming. To start we will explain why MPS is not just another text editor. Normally programs are all stored as text and edited by a text editor. But why should we do this if the most important part of a language is its grammar. When we compile a program, the code written as text is first parsed into a abstract syntax tree (AST) during compilation. The major drawback of storing text like this is the loss of extensibility. Since we cannot easily make changes to a language's grammar, the language cannot be extended by programmers itself. Also adding new features can make the language ambiguous. For this reason JetBrains MPS separates the representation and the storage of the program from the program itself. To make creating languages easy, the program and all language concepts are directly stored in a structured graph and not as plain text. So MPS differentiates itself from many other language workbenches by avoiding the text form. Your programs are always represented by an AST. You edit the code as an AST, you save it as an AST and you compile it as an AST. Due to this feature of MPS it is possible to easily extend languages. It is also possible to mix languages. When one wants to use a concept of an already existing language, you can just import this concept without making the existing or the new language ambiguous.

The basic notions of JetBrains MPS are nodes, concepts and languages. Nodes form a tree. Each node has a parent node (except for root nodes), child nodes, properties, and references to other nodes.

Nodes can be very different from one another. Each node stores a reference to its declaration, its concept. A concept sets a "type" of connected nodes. It defines the class of nodes and coins the structure of nodes in that class. It specifies which children, properties, and references an instance of a

node can have. Concept declarations form an inheritance hierarchy. If one concept extends another, it inherits all children, properties, and references from its parent.

A language in MPS is a set of concepts with some additional information. The additional information includes details on editors, completion menu, intentions, typesystem, generator, etc. associated with the language. This information forms several language aspects. Obviously, a language can extend another language. An extending language can use any concepts defined in the extended language as types for its children or references, and its concepts can inherit from any concept of the extended language.

A project is the main organizational unit in MPS. Projects consist of one or more modules, which themselves consist of models. A model is the smallest unit for generation/compilation. To give your code some structure, programs in MPS are organized into models. Think of models as somewhat similar to compilation units in text based languages. Models typically consist of root nodes, which represent top level declarations, and non-root nodes. Models themselves are the most fine-grained grouping elements.

Modules organize models into higher level entities. A module typically consists of several models accompanied with meta information describing module's properties and dependencies. MPS distinguishes several types of modules: solutions, languages, devkits, and generators.

A solution is the simplest possible kind of module in MPS. It is just a set of models unified under a common name.

A language is a module that is more complex than a solution and represents a reusable language. It consists of several models, each defining a certain aspect of the language: structure, editor, actions, typesystem, etc.

The structure aspect of the language defines the 'structure' of a new language. To define a language's abstract syntax you should enumerate all the types in the language. The types simply represent the features, or concepts, that the language supports. Each concept should be defined by its name, the internal properties of its instances, and the relationships (basically links) its instances can have with other nodes.

The editor language aspect helps you define the layout of cells for each concept in the language. You can define which parts are constant, like braces or other decorations, and which parts are variable and need the user to define them. The editor language also helps you add powerful features to your own editors, like auto-complete, refactoring, browsing, syntax highlighting, error highlighting, and anything else you can think of. Since it completely

works with an AST and the editor aspect of the language lets us specify its presentation in a very detailed way, we can conclude that JetBrains MPS is not a purely textual tool, it is in the middle of textual and graphical.

Generators define possible transformations of a language into something else, typically into another languages. Generators may depend on other generators. Since the order in which generators are applied to code is important, ordering constraints can be set on generators.

We have already mentioned that a basic notion of JetBrains MPS is a node. A concept, specified in the structure aspect of the language, is itself a node and its instances are all nodes to. Even the components of the language aspects (structure, editor, actions, typesystem, ...) are nodes. This is because the language aspects are languages them selfs. The structure language, that is part of MPS, for example, specifies every structure language aspect of a language you create. In terms of model-driven engineering, you can see an instance of your language as a model, the language itself as an meta-model and the languages of the structure, editor, actions, ... aspects of the your language as meta-meta-models.

We conclude that MPS has all features that describe a language, like we discussed in section 2.3. An abstract syntax presented by the structure language aspect, a concrete syntax presented by the editor aspect of the language and semantics by using a generator.

3. JetBrains MPS: Traffic language

It is now time to build an example language in JetBrains MPS. We're going to build a Traffic language, in which it is possible to connect roads to form a traffic network. Since we have already created this formalism in AToM^3, we are going to use the same features. By doing this, we get a more clear comparison between the 2 implementations.

The concepts we need for this language are:

1. Input: an input port
2. Output: an output port
3. Generator: generates a number of cars (1 output)
4. Road: a road (1 input, 1 output)
5. Merger: merges two roads together (2 inputs, 1 output)
6. Splitter: splits a road in 2 roads (1 input, 2 outputs)
7. Sink: the end of a road where cars can enter (1 input)

8. Car: can be connected to a road
9. Constraint: can be connected to several road segments (Road, Merger and Splitter) and has a value that determines the maximum number of cars that can be on those road segments
10. TrafficLight: can be connected to a road segment (Road, Merger and Splitter) and has different modes/lights (pass/not pass) for the cars

By using input and output ports the road segments (Road, Merger and Splitter), generators and sinks can be connected to each other more easily, with less constraints and redundant connections.

To show the power of extending/mixing languages in MPS, we're going to build 2 languages. The Traffic language contains all the concepts above, except for the TrafficLight. In this language it will be possible to connect road segments together, specify constraints and connect cars. To implement the TrafficLight we are going to build another language. This language consists of a Light concept and a TrafficLight concept. The idea is to specify the order of the lights. We will do this by connecting them in a loop. After some specified time the light will change to the next light. This language can be seen as a state machine. After building the TrafficLight language, the Traffic language can use this language to specify a traffic light for a road segment. This way we showed that extending/mixing languages is possible.

To start we create a new MPS project, named Traffic, and create 2 languages, be.ac.ua.traffic and be.ac.ua.trafficlight. Both languages are contained in 1 MPS project in order to manage them more easily. We also create a solution, with a sandbox in it. Here you can insert instances of your language, by creating instances of your root node concept(s). In our case, TrafficNet will be the root node concept for our Traffic language. Please see Figure 1 for the creation of the two languages and the solution.

Let us first develop the TrafficLight language (be.ac.ua.trafficlight). At the left of Figure 2 you see the two concepts of this language, TrafficLight and Light, describing the abstract syntax of the language. TrafficLight has 1 child of concept Light, with multiplicity 0..n. This means it can have zero or multiple children of concept Light. It also has a reference startLight that specifies which light is used as start light. On the right side of Figure 2 you can see the Light concept of the structure language aspect. The Light concept has a reference nextLight that defines the next light in the chain. It also three properties: a color, which is just an integer in this case but can be implemented in Java once we use a generator, a time, that specifies how long

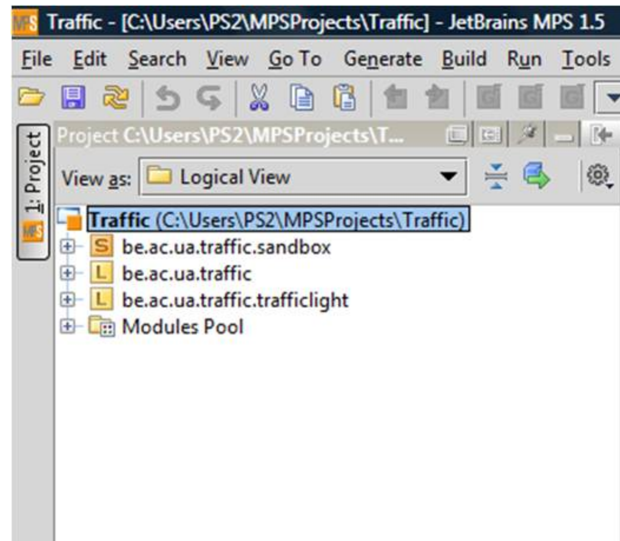

Figure 1: Creation of the 2 languages be.ac.ua.traffic and be.ac.ua.trafficlight and a solution be.ac.ua.sandbox

it takes before changing to the next light and finally a boolean pass, which specifies if the cars can pass or not if this light is turned on.

Now we're going to specify the concrete syntax of the TrafficLight language. We can do this by using the editor aspect of the language. On the left side of Figure 3 you can see that for both concepts a corresponding editor node is created. On the right side the editor node for the TrafficLight concept is expanded. It consists of several cells. To represent this concept we used a vertical collection cell with two horizontal collection cells in it. The first contains a constant cell "Lights" followed by a vertical children collection cell for the lights. A children collection cell lets you create and remove children in the solution sandbox. When you create a child the editor of that child's concept will be shown. In this case, where Light is child of TrafficLight, it will show the editor of the Light concept. The second contains a constant cell "Start light" followed by a reference cell for the editor of the reference itself. The editor of the Light concept has cells for its name, time, pass and color.

We don't create a generator for the TrafficLight language because we're not going to execute a traffic light on its own. It is possible to do this, but since this is not the main goal we will focus on simulating a traffic network. We will use a generator for the Traffic language to transform it to Java code.

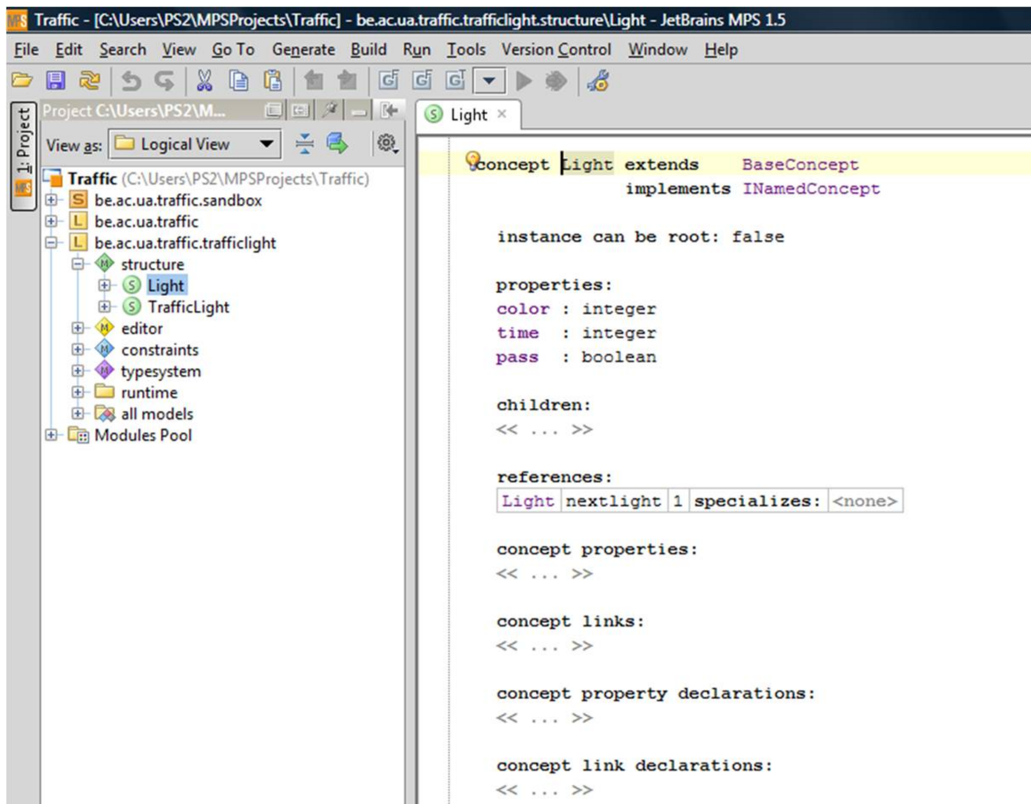

Figure 2: Structure aspect of the TrafficLight language

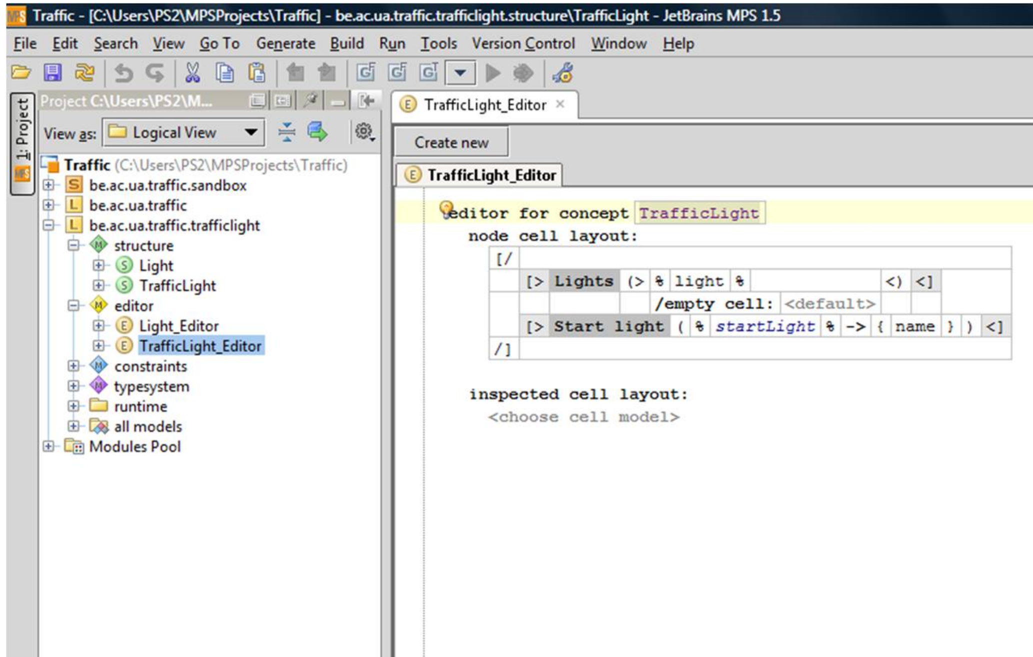

Figure 3: Editor aspect of the TrafficLight language

We will now develop the Traffic language (be.ac.ua.traffic). In Figure 4 we see all the concepts needed for the language. At the right we see the TrafficNet concept. It can be a root node and it has several children: roads, mergers, splitters, generators, sinks, cars and constraints. They all have multiplicity 0..n. At the left side you see 2 interface concepts RoadObject and RoadSegment. Like in Java, an interface concept cannot be instantiated. Generator, RoadSegment and Sink all inherit from RoadObject. We did this do be able to use the generator to generate Java simulation code for the traffic network, but more on this later. RoadObject also implements the build-in INamedConcept, that lets you define a name for an instance of that concept. In our case, a name is necessary for connecting the roads, generators and sinks, since referring to a node's address is not very handy. Concepts Road, Merger and Sink inherit from the interface concept RoadSegment, which has a child of concept TrafficLight (from the TrafficLight language we discussed above) with multiplicity 0..1. This way we need to specify the TrafficLight only once, since it is also possible for a merger and splitter to have a traffic light. To be able to choose this TrafficLight concept as a

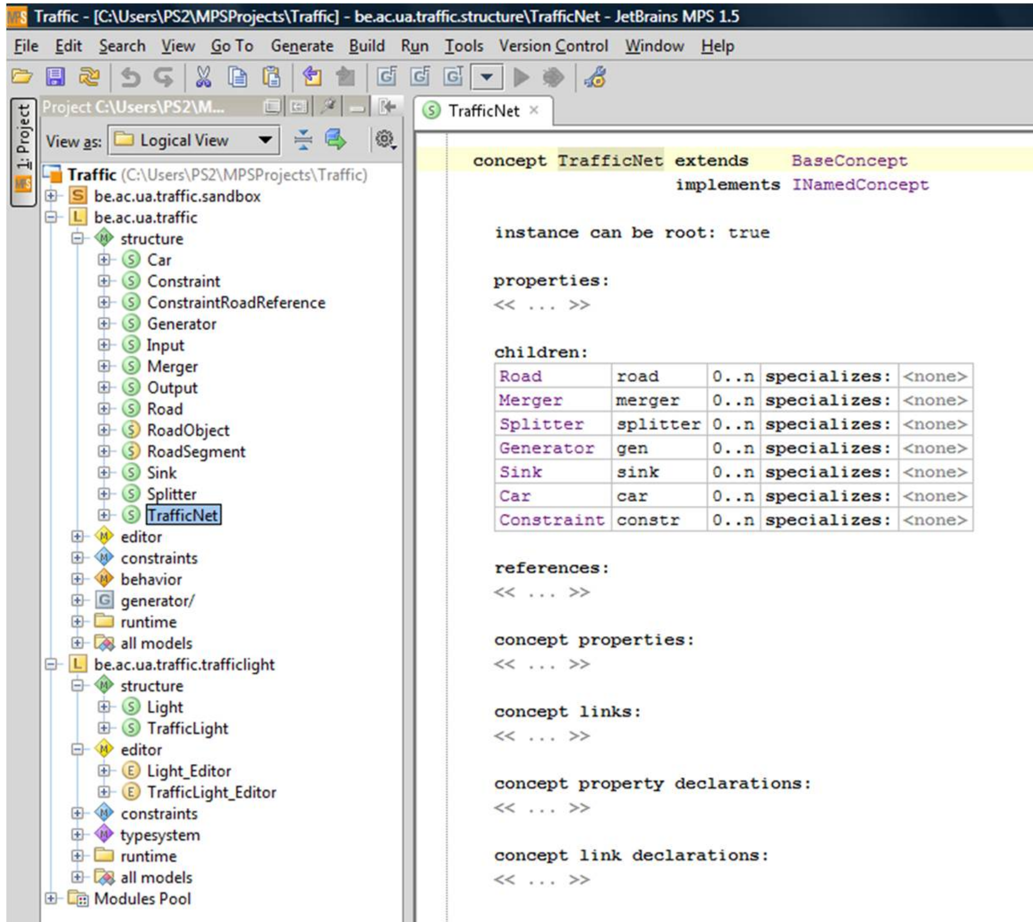

Figure 4: Structure aspect of the Traffic language

child, you have to add the TrafficLight language (be.ac.ua.trafficlight) to the Extended languages in Language properties.

A Road has two children, one Input concept and one Output concept. A Merger has three children, two Input concepts and one Output concept. A Splitter also has three children, one Input concept and two Output concepts. A Generator only has one child, an Output concept. A Sink only has one child, an Input concept. An Input concept has a reference to the Output concept and the other way around. The Car concept has a reference to the RoadSegment concept. Since a concept can only have references with multiplicity 0..1 or 1..1, we have created the ConstraintRoadReference concept. The Constraint concept has a child of the ConstraintRoadReference concept

Figure 5: Editor aspect of the Traffic language

with multiplicity 0..n and the ConstraintRoadReference has a reference of 1..1 to the RoadSegment concept.

Now we can develop the editor for all the concepts. At the left of Figure 5 you see the editor of every concept of the Traffic language. At the right of the figure you see the expanded editor of the Road concept. In Figure 6 you can see the editor of the root concept TrafficNet.

It is also possible to add some constraints to the language. For every concept it is possible to add constraints regarding its properties, children or references. At the left of Figure 7 you can see several constraints, for example a constraint for the Constraint concept that says that the value of it needs to be larger or equal to 1. At the right of Figure 7 you can see the constraints for the Generator concept. It has a constraint that automatically sets the name of the output (child) port when you change the name of the generator itself. For example, if you change the name of the generator to "gen", the name of the output port will change to "gen.out". So now we can connect

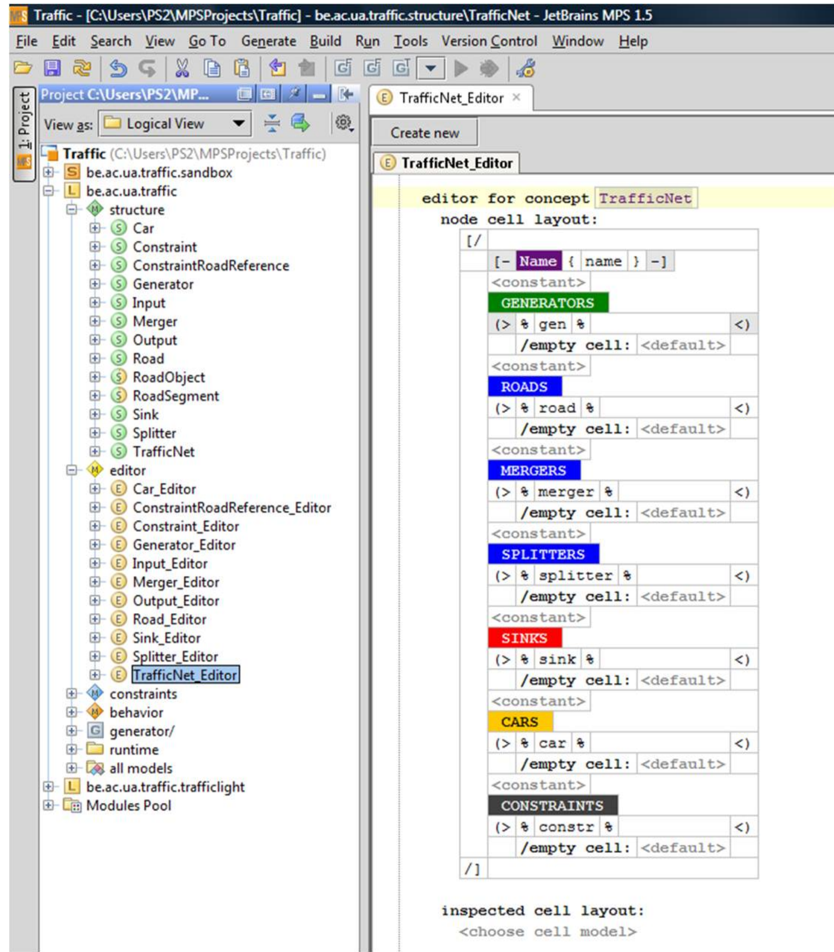

Figure 6: Editor aspect of the Traffic language

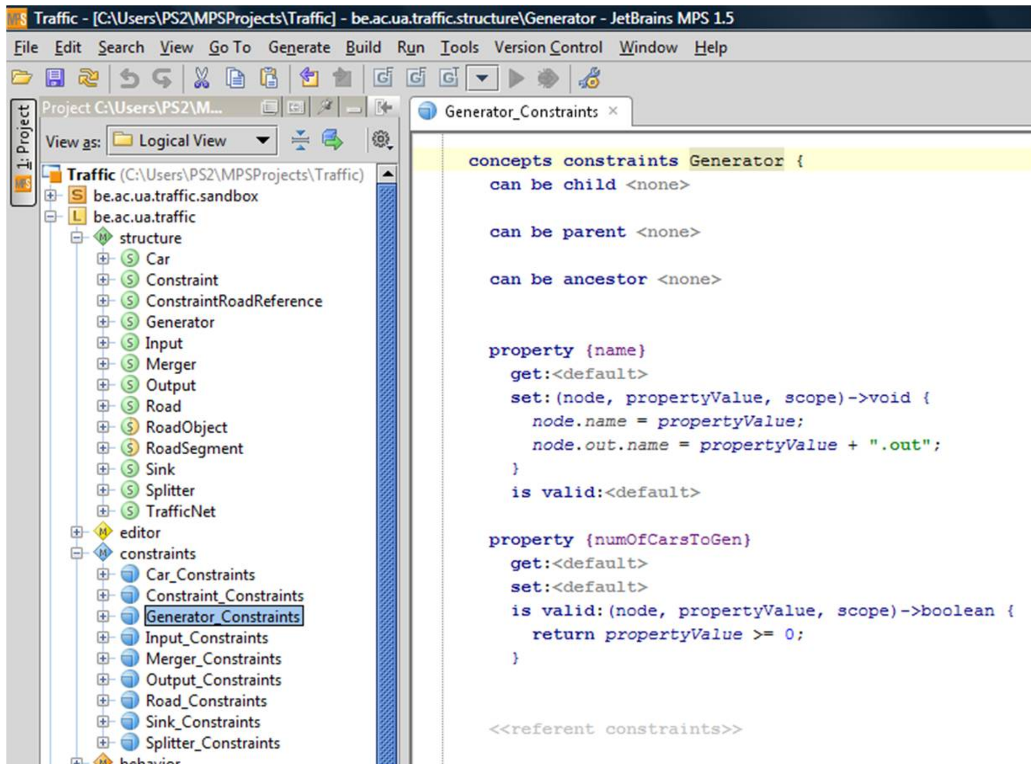

Figure 7: Constraints of the Traffic language

an input port to this output port by using its name. The second constraint of the Constraint concept says that the property number of cars to generate needs to be larger or equal to 0.

We also used the behavior aspect of the language to specify a constructor for the Generator, Road, Merger, Splitter and Sink concepts, like you can see in Figure 8. For example, the Road constructor creates new instances of the Input and Output concepts and sets them as its children. Then the roadobject reference of the input and output ports will be set to this road. We need this in order to get from a port to the corresponding road in the generator language.

Since we have the abstract and concrete syntax of the language, we can start specifying the semantics of the language. We're going to use the generator language to build a transformation to Java. With the Java implementation we can simulate the traffic network step by step. We call this kind of semantics, operational semantics.

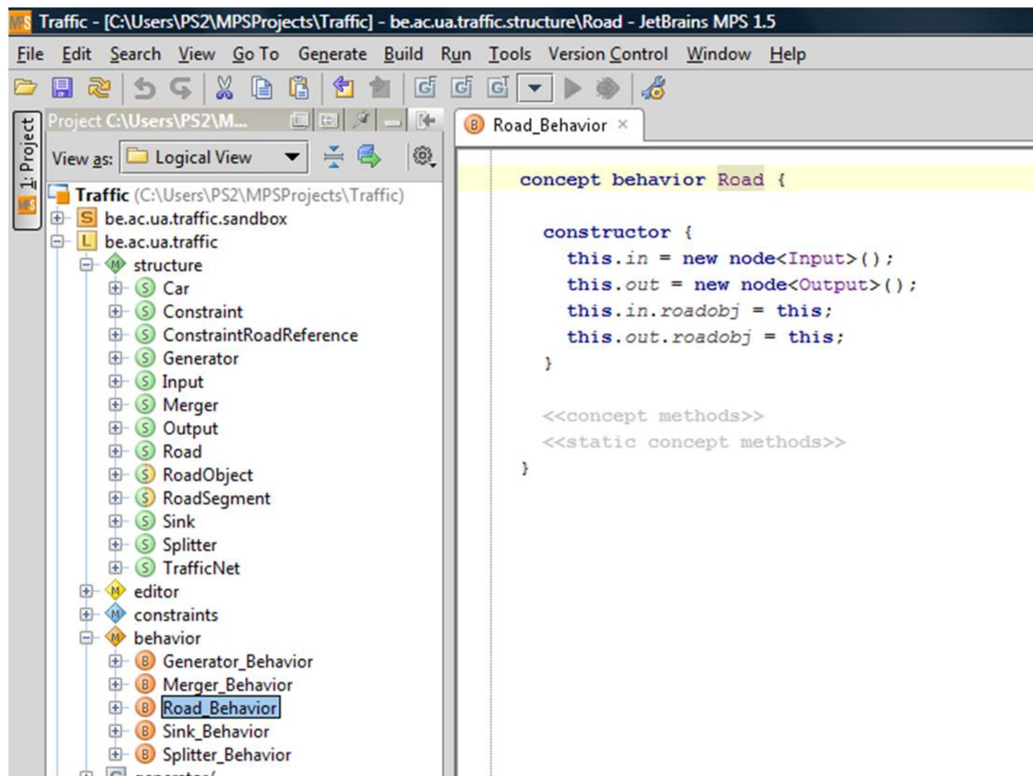

Figure 8: Behavior aspect of the Traffic language

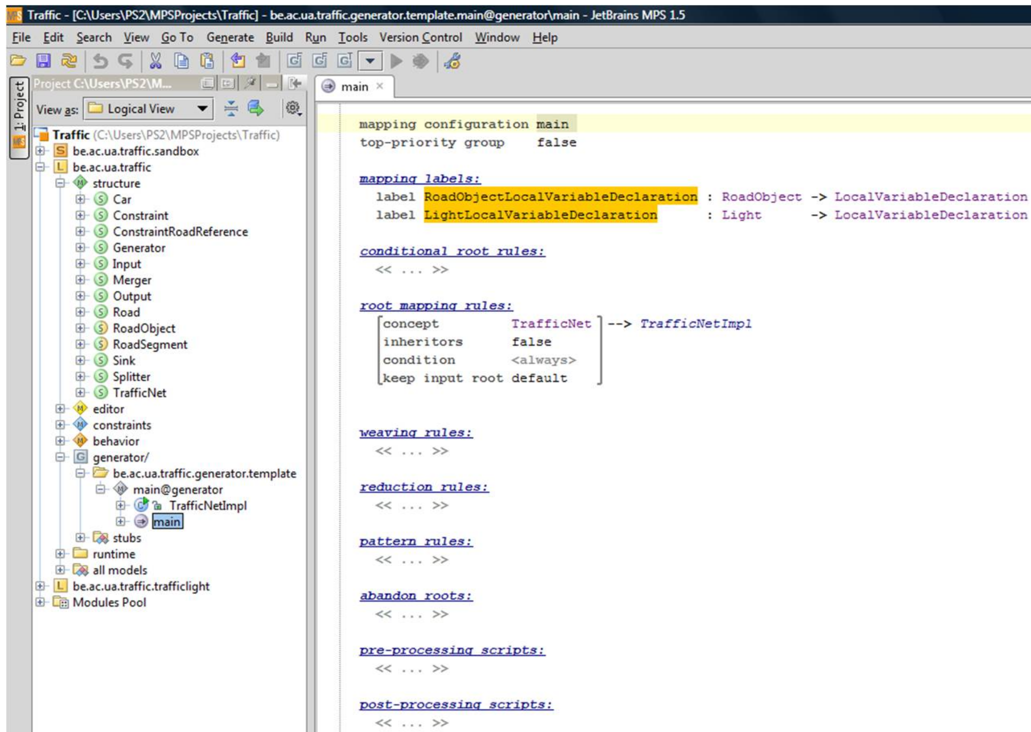

Figure 9: Mapping from the generator of the Traffic language

To make the transformation to Java, we first create a generator in the language and we specify a mapping. In Figure 9 you can see the mapping. The root mapping rule "TrafficNet to TrafficNetImpl" specifies a mapping from the TrafficNet concept, that has all the information of the traffic network, to a Java class TrafficNetImpl. We're going to use this class to store all the roads, generators, sinks, etc and to start the simulation. At the top of Figure 9 you can also see the mapping labels. These labels are used to keep track of the names of local variables if we create one for a concept. It just maps the concept onto the local variable declaration. We only create a TrafficNetImpl class with the generator, all the other classes (Road, Constraint, Car, ...) we need to obtain a OO design are written in Java and provided to the MPS language as a stub (at the left of Figure 9).

In Figure 10 you can see a part of the Java class that will be generated. At the top we specify the input node, this is the TrafficNet concept. The class is named TrafficNetImpl but this name will be changed to the name

of the TrafficNet concept instance because it is encapsulated by a property macro. A property macro lets us replace a piece of code by a property of the concept. As fields we have lists of generators, roads, sinks, constraints, traffic lights and cars. The classes Merger and Splitter in the runtime stub code both inherit from the class Road. That's why we only need a list of roads. We use this lists to iterate over all the elements once we run the simulation. For example when every car needs to move to the next road, we iterate over the list of cars. In the constructor of the class we create local variables for all the generators, roads, mergers, splitters and sinks in the TrafficNet instance. We do this by using a loop macro. It inserts the encapsulated code for every input node. In case of the generator local variable, this statement is inserted for every generator in the TrafficNet instance. The only thing we need to be careful of is the name of the local variable. We have to give them unique names. This is done by the property macro around "gen" (in case of the generator statement). The mapping labels, we discussed earlier, keep track of the names. What follows next is connecting the roads together, building the traffic lights, creating cars and constraints and adding them to the lists. Since road segments have input and output ports that link them together in the high-level language, we can use this information to find their next and previous road segments. We now have a class that can construct itself with all the information of the traffic network defined in the high-level language. It also has a main function and some (step by step) simulation code, but more on that later.

We are now ready to create an instance of the language. In the solution module, we created a sandbox be.ac.ua.traffic to add traffic networks. We add a new root node TrafficNet to the sandbox. You can see in Figure 11 that the instance has the layout specified in the editor of the TrafficNet concept. We named the traffic network "MyTrafficNet". We can now add some generators, roads, sinks, etc. We can also give them names, set their properties and connect them using the names of their ports. For a road, merger or splitter we can build a traffic light using the TrafficLight language editor we constructed. The result of the example traffic network is shown in Figure 12. In the sandbox of the project you will also find an example of a roundabout, named "MyRoundabout". In this traffic network there can only be one car on the roundabout at once. Figure 13 shows the "MyRoundabout" traffic network. For more details on the examples you should look at the solution module of the project.

Once you constructed your traffic network you can generate your solution

```
root template
input TrafficNet

public class $[TrafficNetImpl] extends <none> implements <none> {
  <<static fields>>

  <<static initializer>>
  private list<Generator> fGenerators = new linkedlist<Generator>;
  private list<Road> fRoads = new linkedlist<Road>;
  private list<Sink> fSinks = new linkedlist<Sink>;
  private list<Constraint> fConstraints = new linkedlist<Constraint>;
  private list<TrafficLight> fTrafficLights = new linkedlist<TrafficLight>;
  private list<Car> fCars = new linkedlist<Car>;
  <<properties>>
  <<initializer>>
  public TrafficNetImpl() {
    $LOOP$[$MAP_SRC$RoadObjectLocalVariableDeclaration[ Generator $[gen] = new Generator("$[name]", $[0]); ]
    $LOOP$[$MAP_SRC$RoadObjectLocalVariableDeclaration[ Road $[road] = new Road("$[name]"); ]
    $LOOP$[$MAP_SRC$RoadObjectLocalVariableDeclaration[ Merger $[merger] = new Merger("$[name]"); ]
    $LOOP$[$MAP_SRC$RoadObjectLocalVariableDeclaration[ Splitter $[splitter] = new Splitter("$[name]"); ]
    $LOOP$[$MAP_SRC$RoadObjectLocalVariableDeclaration[ Sink $[sink] = new Sink("$[name]"); ]
    $LOOP$[{
        ->$[gen].setNextRoadObj(->$[road]);
        fGenerators.add(->$[gen]);
      }]
    $LOOP$[{
        ->$[road].setPreviousRoadObj(->$[road]);
        ->$[road].setNextRoadObj(->$[road]);
        fRoads.add(->$[road]);
        $IF$[{
            TrafficLight tl = new TrafficLight(->$[road]);
            $LOOP$[$MAP_SRC$LightLocalVariableDeclaration[ Light $[l] = new Light($[0], $[0], $[true]); ]
            $LOOP$[{
                ->$[l].setNextLight(->$[l]);
                tl.addLight(->$[l]);
              }]
            tl.setCurrentLight(->$[l]);
            ->$[road].setTrafficLight(tl);
            fTrafficLights.add(tl);
          }]
      }]
    $LOOP$[{
        ->$[merger].setPreviousRoadObj(->$[merger]);
```

Figure 10: Some generation code for the TrafficNetImpl class

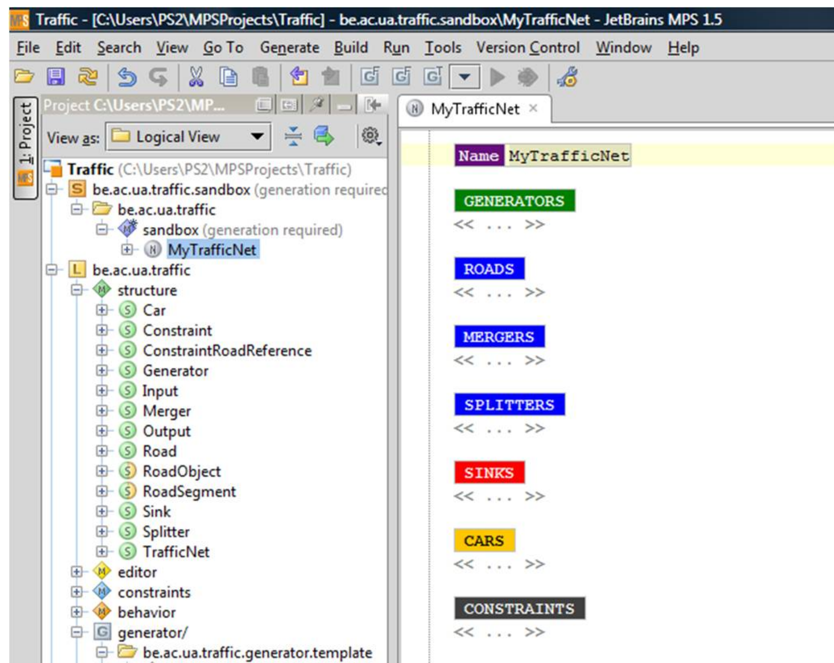

Figure 11: A traffic network example, named MyTrafficNet, created in the solution sandbox

```
Name MyTrafficNet

GENERATORS
Generator Name g1
          Number of cars to generate 15
          Output g1.out -> m1.in1
Generator Name g2
          Number of cars to generate 10
          Output g2.out -> m1.in2

ROADS
Road Name r1
     Traffic light Lights Name light1 Color 2 Pass true Time 3 Next light light2
                          Name light2 Color 1 Pass false Time 2 Next light light1
                   Start light light1
     Input r1.in <- m1.out
     Output r1.out -> s1.in

MERGERS
Merger Name m1
       Traffic light Lights Name red1 Color 1 Pass false Time 5 Next light green1
                            Name orange1 Color 3 Pass false Time 1 Next light red1
                            Name green1 Color 2 Pass true Time 3 Next light orange1
                     Start light red1
       Input1 m1.in1 <- g1.out
       Input2 m1.in2 <- g2.out
       Output m1.out -> r1.in

SPLITTERS
<< ... >>

SINKS
Sink Name s1
     Input s1.in <- r1.out

CARS
Car -> r1

CONSTRAINTS
Constraint Name constr
           Roads m1 r1
           Value 1
```

Figure 12: A traffic network example, named MyTrafficNet, created in the solution sandbox

Name MyRoundabout

GENERATORS
Generator Name gen
Number of cars to generate 10
Output gen.out -> merger1.in1

ROADS
Road Name road1
Traffic light <no trafficlight>
Input road1.in <- merger1.out
Output road1.out -> splitter1.in

MERGERS
Merger Name merger1
Traffic light <no trafficlight>
Input1 merger1.in1 <- gen.out
Input2 merger1.in2 <- splitter2.out1
Output merger1.out -> road1.in

SPLITTERS
Splitter Name splitter1
Traffic light <no trafficlight>
Input splitter1.in <- road1.out
Output1 splitter1.out1 -> splitter2.in
Output2 splitter1.out2 -> sink1.in
Splitter Name splitter2
Traffic light <no trafficlight>
Input splitter2.in <- splitter1.out1
Output1 splitter2.out1 -> merger1.in2
Output2 splitter2.out2 -> sink2.in

SINKS
Sink Name sink1
Input sink1.in <- splitter1.out2
Sink Name sink2
Input sink2.in <- splitter2.out2

CARS
<< ... >>

CONSTRAINTS
Constraint Name constraint
Roads merger1 road1 splitter1 splitter2
Value 1

Figure 13: A traffic network example, named MyRoundabout, created in the solution sandbox

using the "Generate Solution" button. It will generate a Java class with all the roads, constraints, etc. The name of the Java class is the name of the TrafficNet concept instance, because we used a macro for this. Now you can copy the generated file to your favorite Java IDE and run it. Remember that you need all the classes in the Traffic runtime stub in order to run it. The first step in the execution prints all the information about the traffic network. It prints the details of every generator, road, merger, splitter, constraint and traffic light. A road segment also has a counter that represents the number of cars on it. Next we perform a step by step execution. At every step (time slice) every generator, car and traffic light can perform one move. First every traffic light can do one step, this means, increasing its timer and changing its light when necessary. Second every car can move from one road segment to another, if there are no constraints violated and it has clearance of the traffic light (if present). Last every generator can generate a car. It will decrease its remaining cars to generate and will put a new car on the next road segment (if possible). The execution stops when there is no car anymore and every generator is finished. For more details you can look at the TrafficNetImpl class in the generator or at the Traffic runtime stub. Figure 14 and Figure 15 show 2 steps of the execution of the example traffic networks, "MyTrafficNet" and "MyRoundabout".

4. Comparison with AToM^3

4.1. What is AToM^3?

Like JetBrains MPS, AToM^3 is also a model-driven engineering tool, more precisely, it is a tool for multi-formalism and meta-modeling. In AToM^3 it is possible to build your own formalism (domain-specific language) by using an existing formalism, for example Class Diagrams. It lets you create all the classes you need for your new formalism in a graphical way. For example, the meta-model of the Petri Net formalism in AToM^3 is an instance model of the Class Diagram formalism. In the Class Diagram model a place, a transition, their attributes, their associations and the constraints of the language are specified. An association specifies how to connect a class to another class. Besides graphical modeling it is also possible to write some constraint code in Python for the formalism. For every class in the Class Diagram formalism one can specify the graphical appearance of the class in an instance model by drawing it or selecting an image/icon. To create an instance model from the formalism you need to restart AToM^3, so it can be compiled, and load this

```
========================= Step 21 =========================
------------------------ Constraint ----------------------
Name: constr
Value: 1
Roads: m1 r1
----------------------------------------------------------
---------------------- Traffic light ---------------------
From road: r1
Current light: light1
Current light color: 2
Current light pass: true
Counter: 0
Switch after: 3
----------------------------------------------------------
---------------------- Traffic light ---------------------
From road: m1
Current light: green1
Current light color: 2
Current light pass: true
Counter: 3
Switch after: 3
----------------------------------------------------------
------------------------ Generator -----------------------
Name: g1
Number of cars to generate: 11/15
Next: m1
----------------------------------------------------------
------------------------ Generator -----------------------
Name: g2
Number of cars to generate: 10/10
Next: m1
----------------------------------------------------------
-------------------------- Road --------------------------
Name: r1
Number of cars: 0
Previous: m1
Next: s1
----------------------------------------------------------
------------------------- Merger -------------------------
Name: m1
Number of cars: 1
Previous: g1
Previous2: g2
Next: r1
----------------------------------------------------------
-------------------------- Sink --------------------------
Name: s1
Number of cars: 4
Previous: r1
----------------------------------------------------------
```

Figure 14: One step of the execution of the example traffic network MyTrafficNet

```
========================= Step 34 =========================
------------------------ Constraint ----------------------
Name: constraint
Value: 1
Roads: merger1 road1 splitter1 splitter2
----------------------------------------------------------
------------------------ Generator -----------------------
Name: gen
Number of cars to generate: 2/10
Next: merger1
----------------------------------------------------------
-------------------------- Road --------------------------
Name: road1
Number of cars: 0
Previous: merger1
Next: splitter1
----------------------------------------------------------
------------------------- Merger -------------------------
Name: merger1
Number of cars: 1
Previous: gen
Previous2: splitter2
Next: road1
----------------------------------------------------------
------------------------ Splitter ------------------------
Name: splitter1
Number of cars: 0
Previous: road1
Next: splitter2
Next2: sink1
----------------------------------------------------------
------------------------ Splitter ------------------------
Name: splitter2
Number of cars: 0
Previous: splitter1
Next: merger1
Next2: sink2
----------------------------------------------------------
-------------------------- Sink --------------------------
Name: sink1
Number of cars: 7
Previous: splitter1
----------------------------------------------------------
-------------------------- Sink --------------------------
Name: sink2
Number of cars: 0
Previous: splitter2
----------------------------------------------------------
==========================================================
```

Figure 15: One step of the execution of the example traffic network MyRoundabout

formalism into the program. Then one can create instances of the classes by using the buttons and clicking into the drawing window. It is also possible to add buttons for other jobs, like simulation or transformation. In this case you write Python code for the button yourself.

4.2. Comparison of both tools

In this section we're going to compare the two tools based on the Traffic language. For the Traffic and TrafficLight languages in JetBrains MPS we used every concept of the AToM^3 implementation to be able to compare the two implementations more easily. The constraints for both language implementations are quite the same. Table 1 gives an overview.

The most obvious difference between the two model-driven engineering tools is of course the representation. With AToM^3 models are presented graphically (graph structure with associations). Even the creation of a language in the Class Diagram formalism is graphical. As we said before, everything in JetBrains MPS is a node in an AST. In the editor aspect of a language it is also possible to add cells with colors and special borders. Therefore JetBrains MPS is not completely textual.

The construction of the abstract and concrete syntax of a language in both tools is quite the same. In AToM^3 the classes in the Class Diagram model define the abstract syntax of the language. The associations, you draw between the classes, define the possible connections. In Figure 16 you can see a part of the Class Diagram model for the Traffic formalism. In JetBrains MPS the abstract syntax is specified by all the created concepts in the structure aspect of the language. Here, the children and references of a concept define the possible connections. In both tools the associations have multiplicities. For every class in the Class Diagram model of the language in AToM^3 and for every concept of the language in JetBrains MPS the concrete syntax can be specified.

On the other hand, specifying the (operational) semantics of a language is quite different. JetBrains MPS has a generator language that can transform the language to a Java implementation. AToM^3 doesn't have this feature, but it is still possible to give the language some semantics by making a button for the formalism that performs a transformation or by making rewrite rules. These rules can be defined with the built-in Transformation formalism. The rules can be executed on the current graph and every step can be made visible. So in order to do some simulation in AToM^3 you can make a button that holds code to transform the graph or you can use the built-in Transformation

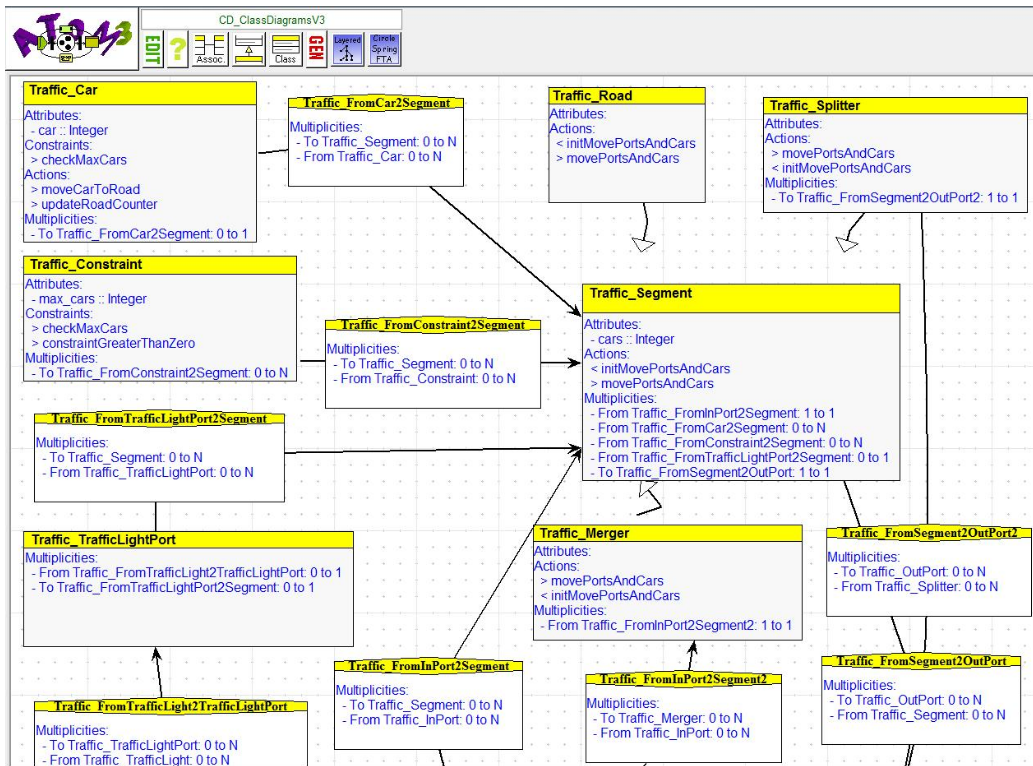

Figure 16: Part of the Class Diagram model for the Traffic formalism

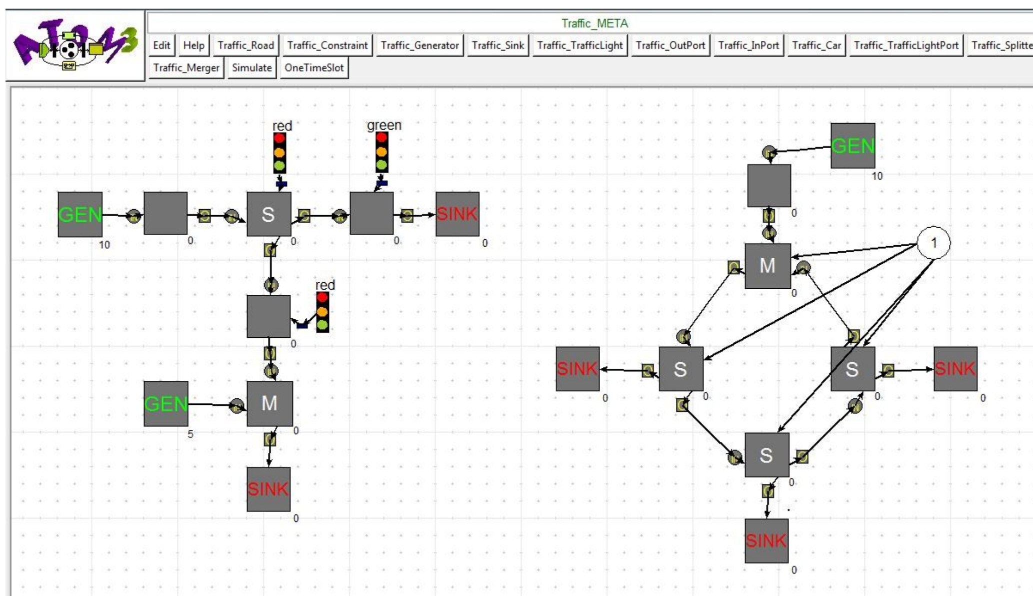

Figure 17: Loaded Traffic formalism and creation of a traffic network

formalism to do a rule based execution. In JetBrains MPS, simulation is not possible, you first have to transform the language to code with the generator. For many languages the visual simulation of AToM^3 can be more appropriate. But writing the generator is easier than writing code for a button because a lot of code for traversing the graph structure in AToM^3 is required. In Figure 17 you can see the loaded Traffic formalism with its buttons to create instances of the classes and an example traffic network. On the right you can see a roundabout, like the one we build in the sandbox of MPS. In Figure 18 you can see the same traffic network but now after pressing the 'One Step' button a couple of times. The 'One Step' button does one step of the execution. The 'Simulate' button does a complete step by step execution.

Defining constraints for the language is easier with AToM^3 because you can add global constraints concerning several classes. In JetBrains MPS it is sometimes difficult to add constraints that concern several concepts because a constraint only belongs to one concept. It is not always possible to get to a node in the tree by only walking through the children, the references and the parents, because at some point you don't know the node's type. It is possible to do a type check, but there is no way to cast the node to the specific concept. That's why writing constraint code in AToM^3 is sometimes

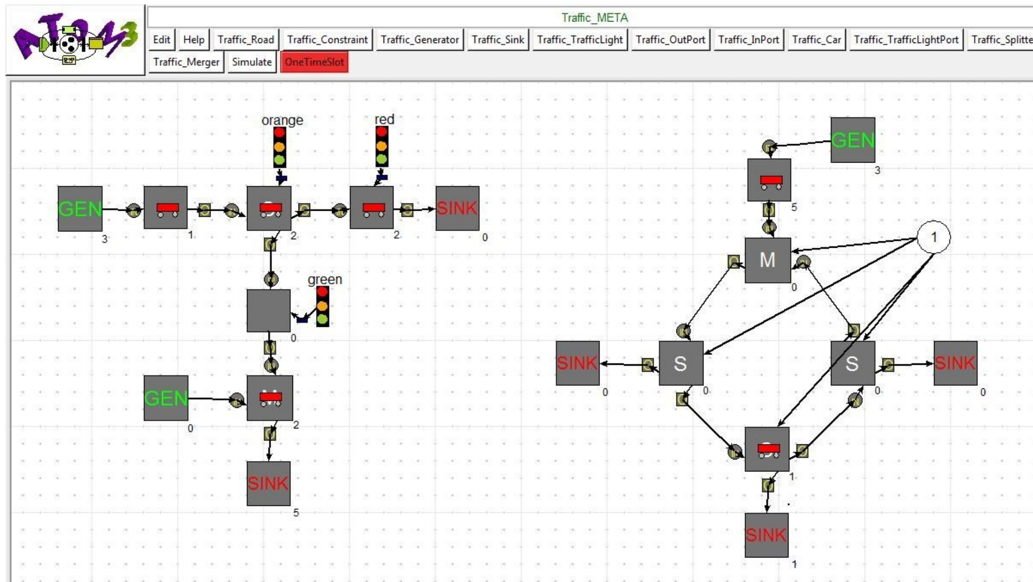

Figure 18: Execution of the traffic network in AToM^3

easier, despite all the logic for traversing the graph.

In both tools it is possible to open several formalisms/languages at the same time. In AToM^3 you can draw instances of classes from every loaded formalism. In JetBrains MPS a project can contain multiple languages and languages can use other languages. In the sandbox you can add root nodes from all languages in the project. Unlike JetBrains MPS, AToM^3 is not able to extend languages with other languages. It is also impossible to use classes of other languages in your language.

For model-driven engineering there are still a lot of things to do before it really can be a breakthrough in the software engineering community. For example: if a meta-model changes, all the models also need to change, but this requires some complicated transformation rules. It would also be great if the changes are immediately made in the model and not after compilation. This is still a hot topic in model-driven engineering. Therefore we are going to examine how well changes are performed on the instances of the classes/concepts after a change of the language/formalism is made. In AToM^3 the instance models are stored as Python files. A change in the meta-model (formalism) will result in a model (or some class instances) that can't be loaded anymore. In JetBrains MPS there is more support for changes

by automatic refactoring mechanisms. After a name change in the language structure aspect, the references (and children) are immediately changed by refactoring. The node instances in the sandbox model will be changed to. If you delete a concept or a property, the nodes in the sandbox model and in functions of the constraint and action aspects of the language are not aware of this and will be seen as errors.

Finally we compare the two tools based on user-friendliness. Here JetBrains MPS is the indisputable winner. AToM^3 still has a lot of bugs that makes developing hard. JetBrains MPS has powerful refactoring mechanisms and there are almost no bugs in it.

The conclusion is simple, building a small traffic network can easily be done by the MPS implementation, but when building larger traffic networks the graphical features of AToM^3 come in handy. With MPS it is easy to get lost in all the roads, mergers and splitters that are summed up in the traffic network.

5. Conclusions & future work

We saw that traditional programming is still very famous, but if we want to continue to evolve we need to think further than an ordinary object-oriented design. With model-driven engineering a new era of software engineering has begun. JetBrains MPS implements language oriented programming, a specific part of the model-driven engineering domain. We have shown with an example, the language Traffic, that extending and mixing languages is quite easy in this new paradigm. We then used the generator language to transform our higher-level language to a Java implementation, used to simulate the traffic network. In other words, we gave the language some operational semantics.

We came to the conclusion that some languages, like Traffic, are better designed in a graphical tool like AToM^3, but other languages, that don't need graphical simulation and features, can easily be designed in MPS too. In terms of IDE quality, MPS is somewhat better than AToM^3. AToM^3 still has some bugs in it that makes developing hard. MPS on the other hand works great, especially with the refactoring tools, and didn't seem to have any bugs.

	AToM^3	**JetBrains MPS**
Representation	Visual	Textual/Visual
Abstract syntax	Classes in Class Diagram model + associations	Concepts in structure aspect (children, references)
Concrete syntax	Icons/images for class instances	Editor aspect (cell layout)
Code generation	Button in formalism	Generator language
Simulation	Button in formalism Rewrite rules	Only after generation (in Java)
Constraints	Multiplicities Constraints in code	Multiplicities Constraint aspect
Multiple formalisms	Yes	Yes
Extending languages **Weaving languages**	No	Yes
Change in meta-model results in change in model	No	Names (after refactoring)
User-friendliness (+/++/+++)	+	+++

Table 1: A comparison of AToM^3 and JetBrains MPS

For the moment the Traffic language implementation in JetBrains MPS (or $ATоM^3$) is not really useful for real analysis, but it must be seen as a proof of concept. To make the language and the transformation to Java more useful we can add extra features like queuing for road segments, a notion of fairness or a more realistic execution (rather than a stepwise execution) and a more extensive TrafficLight language. For the Traffic language in JetBrains MPS we also lack the graphical features of $ATоM^3$. Probably the graphical simulation in $ATоM^3$ is more appropriate for this kind of language. But if we use MPS' generator to transform the program in the high-level language to an implementation in Java with a GUI, then this can give the same or perhaps better results then the $ATоM^3$ implementation.

Dmitriev, S., November 2004. Language oriented programming: The next programming paradigm.

Ward, M. P., October 1994. Language oriented programming. Tech. rep., Computer Science Department, Science Labs, South Rd, Durham, DH1 3LE.