A Traffic DSL in Scala

Jelle Slowack

Abstract

In this report we will build a internal domain specific langauge (DSL) using Scala[1] [2] [3]. More precisily it will be a Traffic DSL, where we will model a Traffic Network containing cars, trafficlights, roads and so on. Afterwards we will compare this with a traffic formalism we created in AToM3.

Keywords: Scala, AToM3, DSL, Traffic, Domain Specific Language, Car, Road, Split, Merge, Implicit Conversion, Model Transformation

1. Introduction

In Section 2 we will give a short tour of Scala. If you know what Scala is, you can skip that section. Section 3 describes our internal Traffic DSL in Scala. We will discuss all the classes (concrete and abstract syntax). Then we will compare this to a Traffic formalism in AToM3 in Section 4. Section 5 contains the conclusion and future work.

2. Scala

What is Scala? On the website of Scala[4], there is a page that describes Scala better than I could describe:

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.

2.1. Scala is object-oriented

Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits. Classes are extended by subclassing and a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

2.2. Scala is functional

Scala is also a functional language in the sense that every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying. Scala's case classes and its built-in support for pattern matching model algebraic types used in many functional programming languages. Furthermore, Scala's notion of pattern matching naturally extends to the processing of XML data with the help of right-ignoring sequence patterns. In this context, sequence comprehensions are useful for formulating queries. These features make Scala ideal for developing applications like web services.

2.3. Scala is statically typed

Scala is equipped with an expressive type system that enforces statically that abstractions are used in a safe and coherent manner. In particular, the type system supports: generic classes, variance annotations, upper and lower type bounds, inner classes and abstract types as object members, compound types, explicitly typed self references, views, and polymorphic methods. A local type inference mechanism takes care that the user is not required to annotate the program with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

2.4. Scala is extensible

In practice, the development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries: any method may be used as an infix or postfix operator, and closures are constructed automatically depending on the expected type (target typing). A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

2.5. Scala interoperates with Java and .NET

Scala is designed to interoperate well with the popular Java 2 Runtime Environment (JRE). In particular, the interaction with the mainstream object-oriented Java programming language is as smooth as possible. Scala has the same compilation model (separate compilation, dynamic class loading)

like Java and allows access to thousands of existing high-quality libraries. Support for the .NET Framework (CLR) is also available.

3. Traffic DSL

In this section we will describe the traffic DSL. We will not treat everything in detail, if someone wants to

3.1. Road

3.1.1. Concrete syntax

A road can have a length and/or a name. We assume that a road is a one-way road, i.e. traffic is in one direction (from startpoint to endpoint). We can connect different roads to eachother. In the example below we connect four roads using the '>' operator. Road r1 has a length of 5 m, road r2 his name is 'e17', road r3 has a length of 10 km and his name is 'a12' and finally r4 is a road without a name or length.

```
1          val r1 = new Road(5 m)
2          val r2 = new Road("e17")
3          val r3 = new Road(10 km, "a12")
4          val r4 = new Road()
5          r1 > r2 > r3 > r4
```

3.1.2. Abstract syntax

We assume the reader has a basic knowledge of Scala and we are not going to discuss all of the implemented code. Most of the time we will focus on the support of Scala for internal DSLs.

The Road Class

The `Road` class extends `InOutSegment`, which is a trait (traits in Scala are like interfaces in Java). The `InOutSegment` defines that the `Road` class should have an in and out `Segment`. This is used to connect different roads to eachother and later on cars will enter a road at his in `Segment` and exit roads at the out `Segment`.

```
1  class Road
2  extends InOutSegment {
3    var in: Segment = null
4    var out: Segment = null
5    [..]
6    def >(r: Road): Road = {
7      TrafficBuilder.connect(this, r)
8      r
9    }
10 }
```

The road append method `>(r: Road)` is used to append a road to another road. For instance: `road1.>(road2)`, the out variable will point to road2 and the in variable of road2 will point to road1.

Multiple constructors

In the previous listing we could use different constructors of the Road class. In Java, you sometimes give classes multiple constructors with overloaded parameter lists. You can do that in Scala as well, however you must pick one of them to be the primary constructor, and place those constructor parameters directly after the class name. You then place any additional auxiliary constructors in the body of the class as methods named `this`. In our `Road` class we have no arguments in the primary constructor.

```
1  class Road
2  extends InOutSegment {
3    [..]
4    def this (l : Int, n : String = NextName.road) = {
5      this()
6      length = l
7      name = n
8    }
9    def this(n: String) = {
10     this()
11     name = n
12   }
13   [..]
14 }
```

The primary constructor of our Road class doesn't have any arguments (i.e. `new Road()`). In the body we have defined two other constructors. The first one has the length as a parameter and the name of the road as an optional parameter. So if we only create a road with a length this constructor is used (see r1 in previous example). The second constructor only has a name as a parameter. Also notice the explicit call to the primary constructor, `this()`. All constructors except the primary constructor must always call another constructor in the beginning of its body.

Infix Operator Notation

Dots, parentheses and semicolons are optional in Scala,

```
1        r1 > r2 > r3
```

is equivalent to

```
r1.>(r2).>(r3)
```

and this is equivalent to

```
r1.>(r2)
r2.>(r3)
```

In other words, roads are connected using `>(r: Road)` method in the `Road` class.

Implicit conversions

In this paragraph we will explain how it is possible to write for example "5 km" as a parameter.

```
val r1 = new Road(5 km)
```

In the previous section we already talked about optional dots, with that knowledge we could rewrite the parameter as `5.km()`. But an Integer doesn't have a `km()` method, so how did we do that? The answer to this question is implicit conversions. First of all we create a class `Measure`, that takes an Integer as parameter:

```
protected case class Measure(val amount: Int){
        //Length
        def km: Int  = amount*1000
        def m: Int  = amount
        [..]
}
```

Using this class we could write something like this `Measure(5).km()` returning 5000 or `Measure(5).m()` returning 5. In other words, this class transforms an amount in the appropriate amount of meters. It is also easy to add support for `cm` or `mm`, you just need to add another method.

Finally we can use implicit conversions to convert all `Int` instances to a `Measure` instance.

```
implicit def int2Measure(i: Int) = new Measure(i)
```

By declaring this method to be implicit, you tell the compiler to automatically use this conversion method in situations where a value of type A is called for but a value of type B was passed.

3.2. Merge and Split

3.2.1. Concrete syntax

Merging and splitting roads is very easy! Suppose you want to merge two roads (`r1` and `r2`) into one road (`bigRoad`), then we will write the following code:

```
1 // create roads
2 val r1 = new Road()
3 val r2 = new Road()
4 val bigRoad = new Road()
5
6 //merge
7 (r1,r2) > bigRoad
```

If we want to split bigRoad instead of merging the roads:

```
1 bigRoad > (r1,r2)
```

3.2.2. Abstract syntax

Merging roads using implicit conversions

It is obvious that `(r1,r2) > bigRoad` is equivalent to `(r1,r2).>(bigRoad)` and we also know that `(r1,r2)` is the notation for a tuple in Scala (`Tuple2[Road, Road]`). But offcourse tuples don't have the road append method: `>(r: Road)`. We solve this problem by using an implicit conversion, just like the `Measure` class used in Road. Only this time we will transform a `Tuple2[Road, Road]` to a `DualRoad` class. This class will contain the two roads.

```
1 implicit def tuple2DualRoad(t: Tuple2[Road, Road]) = new DualRoad(t._1, t._2)
```

To complete our task we add the road append method `>(r: Road)` to the the `DualRoad` class.

Splitting roads

Splitting roads is not difficult, we only need to add a method in the `Road` class that can append a `DualRoad` (i.e. `>(dr: DualRoad)`).

3.3. Car

3.3.1. Concrete syntax

It is possible to define the speed of the car or you can give a name to the car. It is also very easy to add a car to a road, the following example will explain the different possiblities.

```
1  val e313 = new Road()
2
3  // add Tractor to the road e313
4  new Car("Tractor") at e313
5
6  // add Ferrari to the road e313 with speed = 180kmh
7  new Car(180 kmh,"Ferrari") at e313
8
9  // just create a car with speed = 140 ms
10 val vw = new Car(140 ms)
11 // add car vw to a new road
12 vw at new Road()
13
14 // add car with default parameters to e313
15 new Car() at e313
```

It is also possible to make your own car class, a simple example is shown below. If this car reaches a split in the simulation, he will always choose the left road.

```
1 class LeftCar(s: Int, name : String) extends Car(s,name){
2         override def getDirection(left : Road, right : Road): Direction = Left
3 }
```

3.3.2. Abstract syntax

For the `Car` class we also use multiple constructors, only this time our primary constructor has two arguments (speed and name). The `Measure` class is also extended to support the methods `ms` and `kmh` (see 3.1.2: Implicit conversions). Method `getDirection(left: Road, right: Road)` is used in the simulator to choose which road it should take.

```
1  class Car(val speed: Int = Default.speed, val name: String = NextName.car) {
2    def this(n: String) = this(Default.speed, n)
3    [..]
4    def at(r: Road) { .. }
5    def getDirection(left: Road, right: Road): Direction = {
6      if (Random.nextBoolean) {
7        return Left
8      } else {
9        return Right
10     }
11   }
12   [..]
13   def at(): Segment = { attached }
14   def getPos(): Int = { positionOnRoad }
```

```
15 }
```

3.4. Generator and Sink

3.4.1. Concrete syntax

A generator will generate cars and a sink is the place where a car ends. It is possible to define a configuration for a generator. The following parameters could be set:

- amount: the amount of cars to be generated
- tracker: whether or not the generated cars should have a tracker (see 3.7)
- speed: the speed of the cars
- name: the name of the generator A sink can have a name as parameter.

Example:

```
1 val e313 = new Road()
2
3 //generate 6 cars and each car has a tracker
4 new Generator(6,true) > e313 > new Sink()
5
6 //generate 11 cars, speed of car = 6kmh and name = jos
7 new Generator(11,false,6 kmh, "jos") > new Road() > new Sink("end")
8 [..]
```

3.4.2. Abstract syntax

The `Generator` class extends `OutSegment`, which is a trait (traits in Scala are like interfaces in Java). The `OutSegment` defines that the `Generator` class should have an outcoming `Segment`. This is used to append a road to the generator.

```
1  class Generator(val amount: Int, var tracker: Boolean = false,
2          var speed: Int = Default.speed, var name: String = NextName.generator)
3          extends OutSegment {
4    TrafficSim.+(this)
5    var out: Segment = null
6
7    def >(r: Road): Road = {
8      TrafficBuilder.connect(this, r)
9      r
10   }
11 }
```

The `Sink` class extends `InSegment`, which is a trait. The `InSegment` defines that the `Sink` class should have an incoming `Segment`. This is used to prepend a road to the sink. We also added the method `>(s:Sink)` to the `Road` class.

```
1  class Sink(var name: String = NextName.sink) extends InSegment {
2    TrafficSim.+(this)
3    var in: Segment = null
4  }
```

3.5. Capacity

3.5.1. Concrete syntax

It is possible to impose a maximum capacity (of cars) to a number of roads. In the following example `r1` and `r2` have a total capacity of 4 cars and `r3` has a capacity of 1.

```
1  val c = new Capacity(4)
2  val r1 = new Road() + c
3  val r2 = new Road()
4  r2+c
5  val r3 = new Road() + Capacity(1)
```

3.5.2. Abstract syntax

First of all we added the following method in the `Road` class.

```
1    def +(c: Capacity) = {
2      if (cap != null) throw new Exception("Multiple capacities at " + this)
3      cap = c
4      c.addRoad(this)
5      this
6    }
```

Now it is possible to add a capacity to the road. For the `Capacity` class we use a case class, because it makes it possible to construct instances of these classes without using the `new` keyword. The `available()` method is being used for simulation.

```
1  case class Capacity(val c: Int) {
2    var roads: List[Road] = List()
3
4    def addRoad(r: Road) {
5      roads = r :: roads
6    }
7    def available(): Int = {
8      var i = 0
```

```
9       for (r <- roads) {
10        i += TrafficSim.carsOn(r)
11      }
12      val a = c - i
13      if (a < 0) { throw new Exception("Capacity exceeded! ("+i+">"+c+")") }
14      a
15    }
16  }
```

3.6. TrafficLight and TrafficLightControl

3.6.1. Concrete syntax

It is possible to add a trafficlight to a road and those trafficlights are controlled by a `TrafficLightControl` object. `TrafficLightControl` decides which colour the trafficlight should take. It is also possible to take the opposite of the colour, by this we mean if the TrafficLightControl indicate that the trafficlight should be red, the trafficlight will be green. Otherwise if the trafficlight should be green or yellow, it will be red. This is useful for modeling crossroads.

```
1 // define a trafficlightcontrol that starts on Green
2 val tlc = new TrafficLightControl(Green)
3 new Road() + TrafficLight(tlc)
4 // trafficlight with opposite = true, this means the trafficlight will start red
5 new Road() + TrafficLight(tlc,true)
```

3.6.2. Abstract syntax

```
1  case class TrafficLightControl(private var light: Light = Green){
2          TrafficSim.+(this)
3          private var t = 0
4
5          def tick{
6                  t+=1
7                  if(light.duration == t){
8                          light = light.next
9                          t = 0
10                 }
11         }
12         def getLight(): Light = light
13 }
14
15 case class TrafficLight(val tlc : TrafficLightControl = TrafficLightControl(), val o
16         def light: Light = { if(opposite) tlc.getLight.opposite else tlc.getLight }
```

```
17          override def toString() = light.toString
18  }
```

3.7. Tracker

3.7.1. Concrete syntax

A road or a car can have tracker, this will save all events of that car or road during a simulation in a xml file. An event is for instance when a car enters a new road. The generated XML could be loaded in the DEVS Trace Plotter[5], this will plot all the events and will give you a good overview.

```
1  new Car + Tracker()
2  new Road + Tracker()
```

3.7.2. Abstract syntax

Because the `Tracker` class just generates an XML file, we will not discuss this. For more info, see `Tracker.scala`

3.8. Simulate

If you want to simulate your traffic, it is possible by using the `simulate()` function. It is possible to set a maximum time for the simulation, this is useful if someone is just interested in the first 10 seconds of the simulation.

Example `Test.scala`:

```
1  import traffic._
2
3  class LeftCar(s: Int, name : String) extends Car(s,name){
4          override def getDirection(left : Road, right : Road): Direction = Left
5  }
6
7  object Test extends TrafficBase{
8          val c = new Capacity(4)
9          val tlc = new TrafficLightControl(Green)
10         val r2 = new Road("lr21") + TrafficLight(tlc,true) + c
11         val l1 = new Road("left") + TrafficLight(tlc)
12         val r3 = new Road("r3")
13         val r4 = new Road("r4")
14         val l2 = new Road("left2")
15         val e313 = new Road(5 m,"e313") + c + TrafficLight(tlc,true) + new Tracker()
16
17         new Car(1 ms) + Tracker() at e313
18
19         new LeftCar(2 ms,"testcar") + Tracker() at e313
```

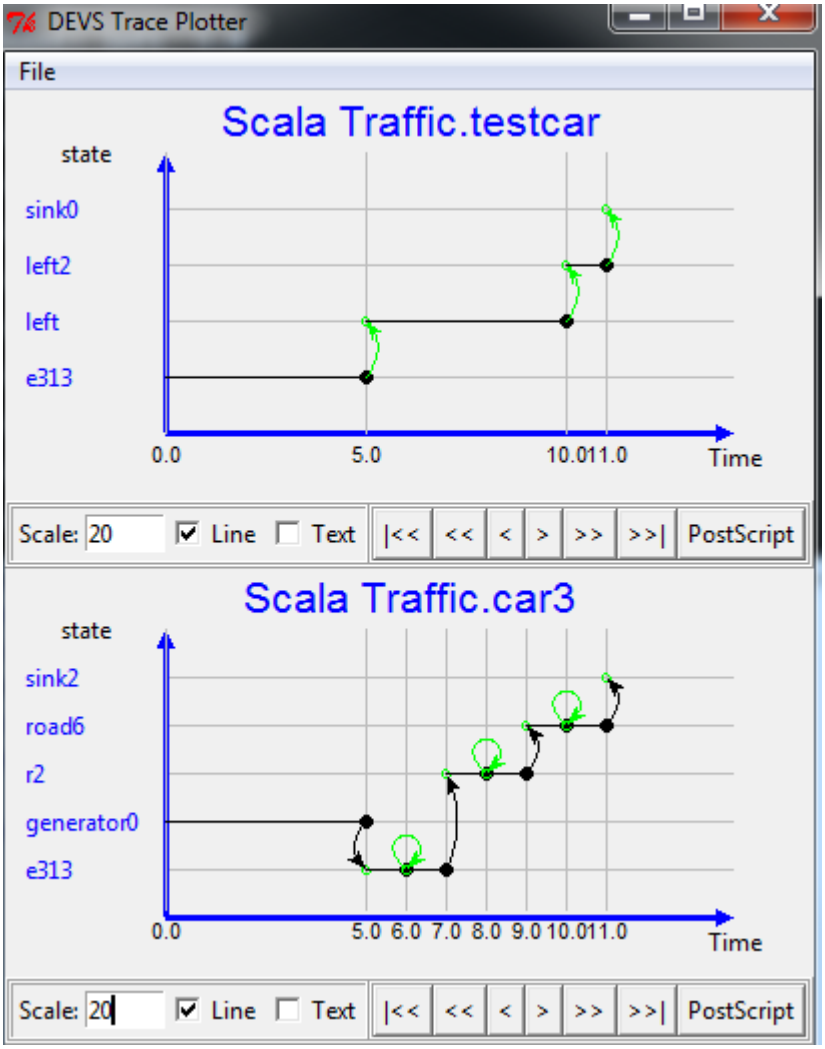

Figure 1: Screenshot of DEVS Trace Plotter

```
20
21          new Generator(6,true) > e313 > (l1,r2)
22          l1 > (l2,r4)
23          l2 > new Sink()
24          r4 > new Sink()
25          (r2,r3) > new Road() > new Sink()
26
27          simulate()
28  }
```

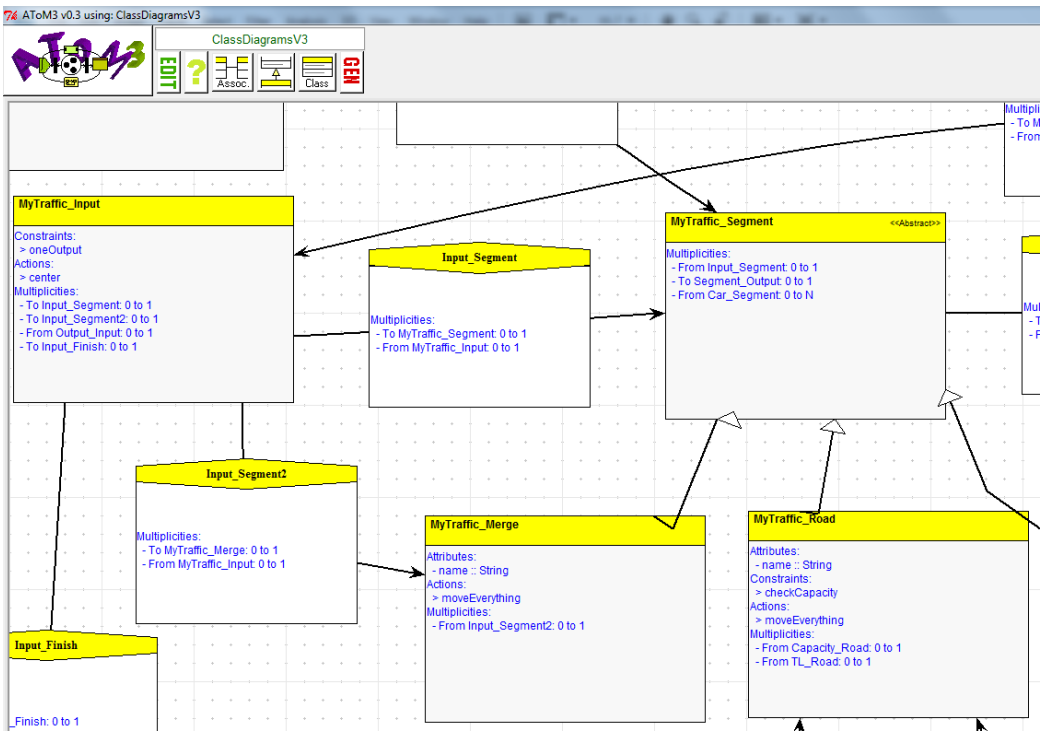

Figure 2: Screenshot of AToM3: Class Diagram formalism (abstract syntax)

4. Comparison with AToM3

Here we are going to compare Scala with AToM3[6]. We should note that the (internal) Traffic language in Scala is more sophisticated than the language in AToM3. For example we didn't implement Trackers and TrafficLightControl in AToM3.

First of all the representation is different. Scala is a textual language and AToM3 is graphical, where formalisms and models are described as graphs.

4.1. Abstract Syntax

The abstract syntax of the traffic language in AToM3 is defined using the Class Diagram formalism. In the Class Diagram formalism we can create associations, create classes and inherit from other classes. Every class can have attributes, constraints (written in python), actions (written in python).

In Scala we use the Scala language as abstract syntax, to be more precise classes and traits (and also implicit conversions). The actions in the CD formalism are in some way similar with the methods of Scala classes, except that actions are triggered by an event (SAVE, CONNECT, ..) and methods are triggered when you call them in your code. The constraints in AToM3 are easier to use, because all the constraints are grouped together (Figure 3). In Scala you have to put them in the function bodies.

Another difference is the multiplicities, using the Class Diagram formalism in AToM3, it is possible to give the associations multiplicities. In Scala you cannot really make associations, you use attributes of a class to define an association and you need to check the multiplicities programmatically (eg. check if the attribute is null).

4.2. Concrete Syntax

On the the other hand the concrete syntax in AToM3 is defined using our Traffic formalism. We can connect roads to eachother, add cars to a road, ... and all of this is visual, because a class can have a graphical appearance (Figure 4). In Scala we create instances of our classes and connect everything textually. Offcourse we could also write Scala code and we can inherit from Scala classes (create your own car, for instance LeftCar), in other words we have more freedom. Also the internal DSL can interop with other Scala or even Java code. In AToM3 the freedom is limited but we could load different formalisms and the graphical view of the different components is an advantage in AToM3.

4.3. Operational Semantics

The simulation in AToM3 could be achieved in two ways. The first option is to add a button that execute python code, ie. coded model transformation, the second option is to do it with rules, ie. rule-based model transformation. An advantage of AToM3 is that you can view the transformations (eg. cars moving from one road to another road). To simulate in Scala the only option is to execute the code and the simulation is outputted textually. But you could make a Scala or even a Java package that would visualize the traffic.

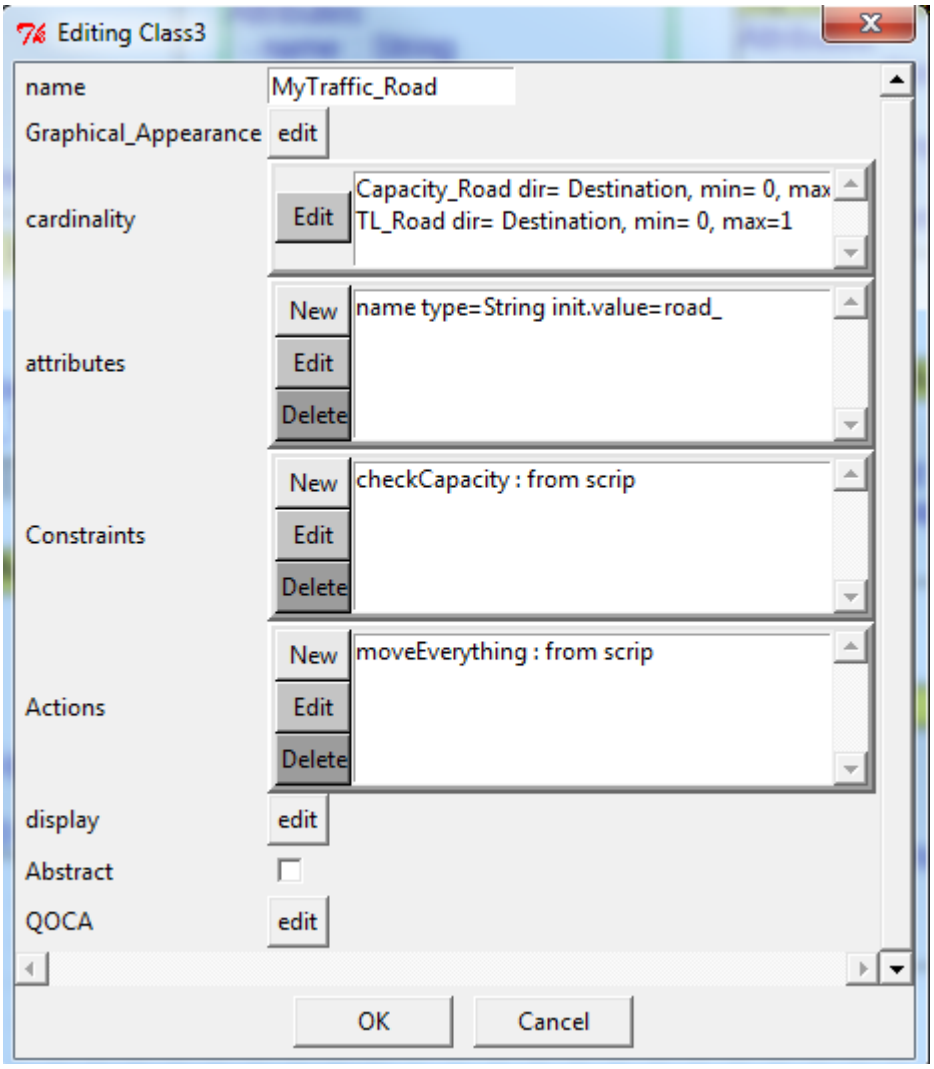

Figure 3: Screenshot of AToM3: Editing a class in the Class Diagram Formalism

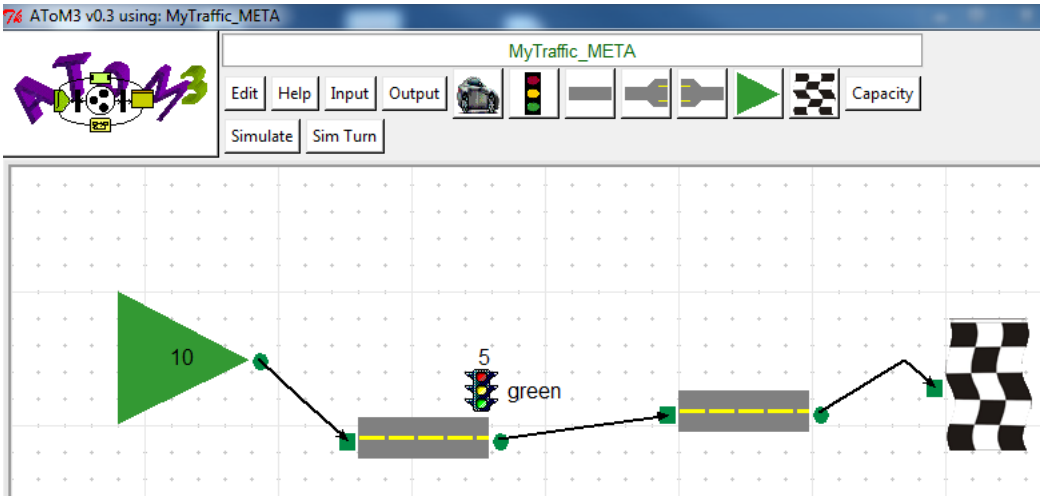

Figure 4: Screenshot of AToM3: Traffic formalism (concrete syntax)

5. Conclusion & Future Work

We can conclude that AToM3 is better to visualize the traffic and also the Class Diagram model gives you a good overview of all the classes and his corresponding actions, attributes, constraints and associations. In Scala you just have lines of code and you do not have a good overview. There is more freedom in Scala and Scala supports some nice things like for example implicit conversion, but it remains a programming language and not a modelling tool. If you want to have a language that interops with other Java or Scala code and that do something with the data, you could use this internal DSL. Otherwise use AToM3.
It is also possible to develop external DSLs in Scala[7] [8] [9] and we only focussed on internal DSLs in Scala. We could investigate this in the near future.

6. Bibliography

[1] Scala.
URL http://www.scala-lang.org/

[2] Domain-specific languages in scala.
URL http://programming-scala.labs.oreilly.com/ch11.html

[3] Designing internal dsls in scala.
URL http://debasishg.blogspot.com/2008/05/designing-internal-dsls-in-scala.htm

[4] A tour of scala.
URL http://www.scala-lang.org/node/104

[5] H. B. Song, Devs trace plotter, Tech. rep.

[6] Atom3.
URL http://atom3.cs.mcgill.ca/

[7] External dsls made easy with scala parser combinators.
URL http://debasishg.blogspot.com/2008/04/external-dsls-made-easy-with-scala.h

[8] Scala external dsl and combinator parsing experimentation.
URL http://www.mostlyblather.com/2010/03/scala-external-dsl-and-combinator.htm

[9] external dsls with scala.
URL http://log4p.com/2009/11/26/external-dsls-with-scala/