Parallel DEVS Modelling of Traffic in AToM^3

Ximeng Sun
School of Computer Science, McGill University
xsun16@cs.mcgill.ca

Abstract

***Traffic**, a timed visual formalism for vehicle traffic networks, is introduced. The syntax of **Traffic** models is meta-modelled [2] in the **Entity-Relationship Diagrams** formalism. The semantics of the **Traffic** formalism is modelled by mapping **Traffic** models onto **Parallel DEVS** [1] models. From this, codes which are suitable for simulation by the **PythonDEVS** [4] simulator (an implementation of the standard **Classic DEVS** simulation algorithm) can be generated. Based on the simulation, analyses (i.e., performance analysis) of a user-defined traffic network can be performed. Graph rewriting is used to transform models. All of these are implemented in **AToM^3**, A Tool for Multi-formalism and Meta-Modelling [5].*

1. Introduction

DEVS formalism [1] is a well known for modelling and simulation discrete-event systems. Some of the advantages of the DEVS formalism are that it allows the hierarchical description of systems, that it provides natural ways for modular design and implementation of systems, and that there are efficient algorithms for their simulation. The basic DEVS formalism is also called Classic DEVS [1] which has some limitations for parallel implementation. For example, the *select* function used in Classic DEVS coupled model for collision tie-breaking, is less controllable as the tie-breaking decision can only be made in the global level. Parallel DEVS [1], as an extension to Classic DEVS, which eliminates the *select* function in coupled model and introduces the *confluent* function in atomic model, gives the modeller complete control over the collision behavior. Parallel DEVS also uses bags as the message structures. This allows that inputs of a component arrive in any order and that more than one input with the same identity may arrive from one or more sources.

In this project, the DEVS formalism we meta-modelled is Parallel DEVS; and so are the automatically generated models from mapping Traffic models to DEVS models by using graph transformation. Due to the time limitation of this project, code generation is only capable of generating codes suitable for simulation by PythonDEVS so far. However, based on the transformed DEVS models, the implementation of code generation for other DEVS simulation frameworks (i.e., DEVSJAVA [7]) is only a practical issue.

Traffic and DEVS meta-modelling, model transformation and simulation code generation are implemented in AToM^3 V0.3 [5].

The rest of the report is organized as follows. Section 2 presents the Traffic formalism for modelling vehicle traffic networks and Traffic meta-modelling in AToM^3. Section 3 presents the Parallel DEVS formalism and meta-modelling in AToM^3. Section 4 presents model transformation which maps Traffic models to DEVS models in AToM^3. Finally, section 5 presents the code generation from DEVS models to PythonDEVS.

2. Traffic formalism and meta-modelling

The Traffic formalism discussed here is an extension of the one described in [2]. This extension is also called the Timed Traffic formalism because we add timing elements to the original Traffic formalism. Based on our extension, the simulation of traffic system is more reasonable and more realistic.

Figure 1 shows a traffic system in which vehicles arrive into the system via a source *Start1* or *Start2*; go along road sections *Lorne and Milton*, or go along *Pine*; then go across an intersection to *Parc* which has entries from *Milton* and *Pine* (each of both controlled by a traffic light and synchronized with each other); finally leave via an exit *End*.

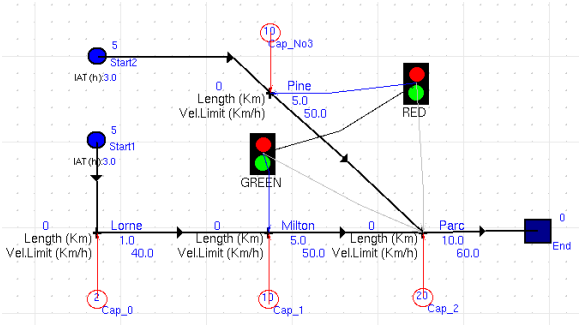


Figure 1: A **Traffic** model

Vehicle arrival is denoted by a filled circle which has three other properties besides its name: *inter_arrival_time (IAT)*, *number_vehicles* and *infinite_supply* (an invisible boolean property). Vehicle departure is denoted by a filled rectangle which has two properties: *name* and *number_vehicles*. A cross denotes a road section which has four other properties besides its name: *length*, *velocity_limit*, *state* (normal or jammed), and *number_vehicles* (a time-varying number of vehicles in it). Road sections are connected by arrows. Multiple arrows departing from a single road section indicates a divergence; multiple arrows arriving to a single road section indicates a convergence which should be coordinated by several synchronized traffic lights. A traffic light is denoted by a black rectangle in which there are a red circle and green circle. The traffic light has no name but three properties: *state* (green or red), *green_time*, *red_time*. A capacity constrain circle may be connected to a number of road sections. The total number of vehicles in all those sections may not exceed the capacity.

2.1. Traffic Meta-Model

To build a meta-model for the Traffic formalism with AToM^3, we use the default meta-formalism **Entity Relationship Diagrams**. The Traffic meta-model shown in Figure 2 describes which entities are allowed in the formalism with their attributes, how they may be connected, and what cardinalities between them are. For example, a source can only connect into one road section and a road section can only have single source connected into; the cardinality between road section and sink is the same as the previous. Not shown is the definition of the graphical appearance (seen in Figure 1) of these entities, global attributes (such as the model name and author), actions, nor are constraints.

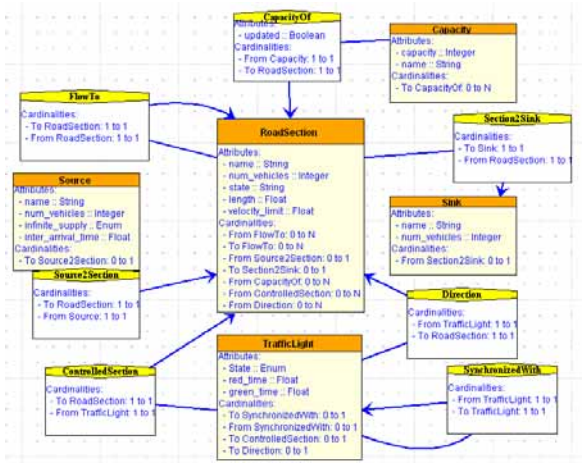


Figure 2: **Traffic** meta-model

3. Parallel DEVS formalism and meta-modelling

The **Parallel DEVS** formalism discussed here is an extension of the one described in [6]. Figure 3 shows a DEVS model which is transformed from the traffic system described in Figure 1. The top-level DEVS model *Traffic* is a coupled model which is a composition of several sub-models (atomic or coupled). For our traffic system example, all sub-models are atomic DEVS models. Each entity in Traffic formalism, such as source, sink, or road section is transformed into a corresponding atomic DEVS model. A group of synchronized traffic lights are transformed into a single traffic light atomic DEVS model; each of unsynchronized traffic lights is transformed into a traffic light atomic DEVS model. A capacity entity is eliminated after transformation.

Sub-models have ports which are connected by channels. There are two types of ports: input and output. A channel must go from an output port of some model to an input pot of a different model, from an input port in a coupled model to an input port of one of its sub-models, or from an output port of a sub-model to an input port of its parent model. For our traffic system example, we have only the first situation, a channel connects the two atomic DEVS model. There are two channels between a source model and a road section model: the one from source to road section for sending messages of cars and the one from road section to source for sending messages of road section state; every two consecutive road section models have a couple of similar channels. There is only one channel between a road section model and a sink model which goes from the former to the later. There is one channel

starts from a traffic light model to each of controlled road section models.

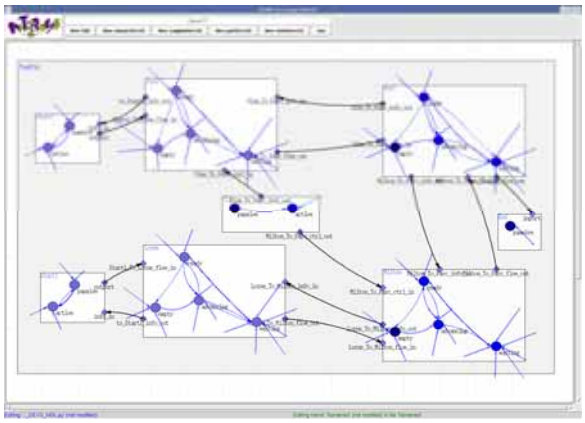


Figure 3: A **DEVS** model transformed from figure 1

An atomic model has, in addition to ports, a set of states, one of which is the initial state, and three types of transitions between states: internal, external and confluent. Associated with each state are a time-advance and an output function. A source atomic DEVS model has two states: *passive* and *active*, four internal transitions, and three external transitions; the *active* is the initial state. A sink atomic DEVS model has one state: passive, and one external transition. A road section atomic DEVS model has four states: *empty*, *advancing*, *waiting* and *ready*, ten internal transitions, and fourteen external transitions; the initial state is *empty*. A traffic light atomic DEVS model has two states: *passive* and *active*, and two internal transitions.

3.1. Parallel DEVS Meta-Model

To build a meta-model for the **Parallel DEVS** formalism with AToM^3, we use the default meta-formalism **Entity Relationship Diagrams**. The **Parallel DEVS** meta-model shown in Figure 4 describes which entities are allowed in the formalism with their attributes, how they may be connected, and what cardinalities between them are. For example, a coupled model can contain multiple atomic models or multiple coupled models. Not shown is the definition of the graphical appearance (seen in Figure 1) of these entities, global attributes (such as the model name and author), actions, nor are constraints. The **Parallel DEVS** meta-model we build here is base on the project work done by Denis Dube [6].

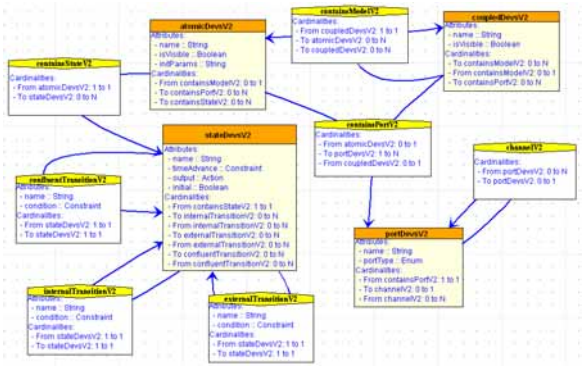


Figure 4: **Parallel DEVS** meta-model

4. Model transformation

As models and meta-models are all in essence attributed and typed graphs, we can transform them by means of graph rewriting. The rewriting is specified in the form of **Graph Grammar** models. A graph grammar is composed of rules. Each rule consists of Left Hand Side (**LHS**) and Right Hand Side (**RHS**) graphs [2].

4.1. Traffic Semantics

To model the semantics of Traffic formalism we build a **Graph Grammar** model of its dynamics. In this project, we map Traffic models onto **Parallel DEVS** models.

Figure 5, 6, 7, 8, 9 and 10 depict our **Graph Grammar** model of the mapping. The model starts with an initial action followed by nine rules. Each rule has a LHS and a RHS as well as an optional pre-condition and post-action. Nodes and connections in LHSs and RHSs are identified by means of labels (numbers). See [2] for the details of how these work in AToM^3.

In the initial action of our model, to avoid infinite application of several following rules, we set a global flag variable *rootTrafficGenerated* as *false* which means the top-level coupled model hasn't generated; and this trick is also used for all nodes, such as *Source*, *Sink*, *RoadSection*, *TrafficLight* and *Capacity*. Rule 1 creates a top-level coupled model for the whole traffic system. Rule 2 transforms **Traffic** *Source* nodes into **DEVS** *Generator* atomic models, with a link to the original *Source* node. Rule 3 transforms **Traffic** *Sink* nodes into **DEVS** *Collector* atomic models, with a link to the original *Sink* node. Rule 4 transforms **Traffic** *RoadSection* nodes into **DEVS** *Road* atomic models, with a link to the original *RoadSection* node. Rule 5 transforms **Traffic** *Source2Section* connections

between *Source* nodes and *RoadSection* nodes into **DEVS** channels with appropriate **DEVS** ports. Rule 6 transforms **Traffic** *Section2Sink* connections between *RoadSection* nodes and *Sink* nodes into **DEVS** channels with appropriate **DEVS** ports. Rule 7 transforms **Traffic** *FlowTo* connections between *RoadSection* nodes into **DEVS** channels with appropriate **DEVS** ports. Rule 8 copies the capacity attribute of **Traffic** *Capacity* node into the corresponding **DEVS** *Road* atomic model. Rule 9 removes the no longer needed **Traffic** *Capacity* nodes. Rule 10 transforms **Traffic** synchronized *TrafficLight* nodes into **DEVS** *TrafficLight* atomic model, with links to the original *TrafficLight* nodes. Rule 11 is similar to Rule 10 except that the **DEVS** *TrafficLight* atomic model already exists. Rule 12 transforms **Traffic** unsynchronized *TrafficLight* nodes into **DEVS** *TrafficLight* atomic models, with a link to the original *TrafficLight* node. Rule 13 transforms **Traffic** *ControlledSection* connections between *TrafficLight* nodes and *RoadSection* nodes into **DEVS** channels with appropriate **DEVS** ports. Rule 14 removes the special link between **DEVS** *TrafficLight* model and **Traffic** *TrafficLight* node. Rule 15 removes the no longer needed **Traffic** *TrafficLight* nodes. Rule 16 removes the special link between **DEVS** *Road* model and **Traffic** *RoadSection* node. Rule 17 removes the no longer needed **Traffic** *RoadSection* nodes. Rule 18 connects **DEVS** *Road* atomic models with **DEVS** *Traffic* coupled models. Rule 19 connects **DEVS** *TrafficLight* atomic models with **DEVS** *Traffic* coupled models. Finally, rule 20 connects **DEVS** *Generator* atomic models and *Collector* atomic models with **DEVS** *Traffic* coupled models.

Appendix A illustrates the application of the rules. It starts from a very simple Traffic model with a source, two connected road segments, a sink, and a traffic light. The transformation ends with a **DEVS** representing the behavior of the **Traffic** model.

```
INITIAL ACTION:
self.rootTrafficGenerated = False
for node in graph.listNodes["Source"]:
  node.sourceGeneratorGenerated = False
for node in graph.listNodes["Sink"]:
  node.sinkCollectorGenerated = False
for node in graph.listNodes["RoadSection"]:
  node.roadAtomicGenerated = False
for node in graph.listNodes["Capacity"]:
  node.capacityInfoGenerated = False
for node in graph.listNodes["TrafficLight"]:
  node.trafficLightWithRoadGenerated = False
  node.trafficLightSyncGenerated = False
self.TL_count = 0
```

Figure 5: Model transformation rules (part 1)

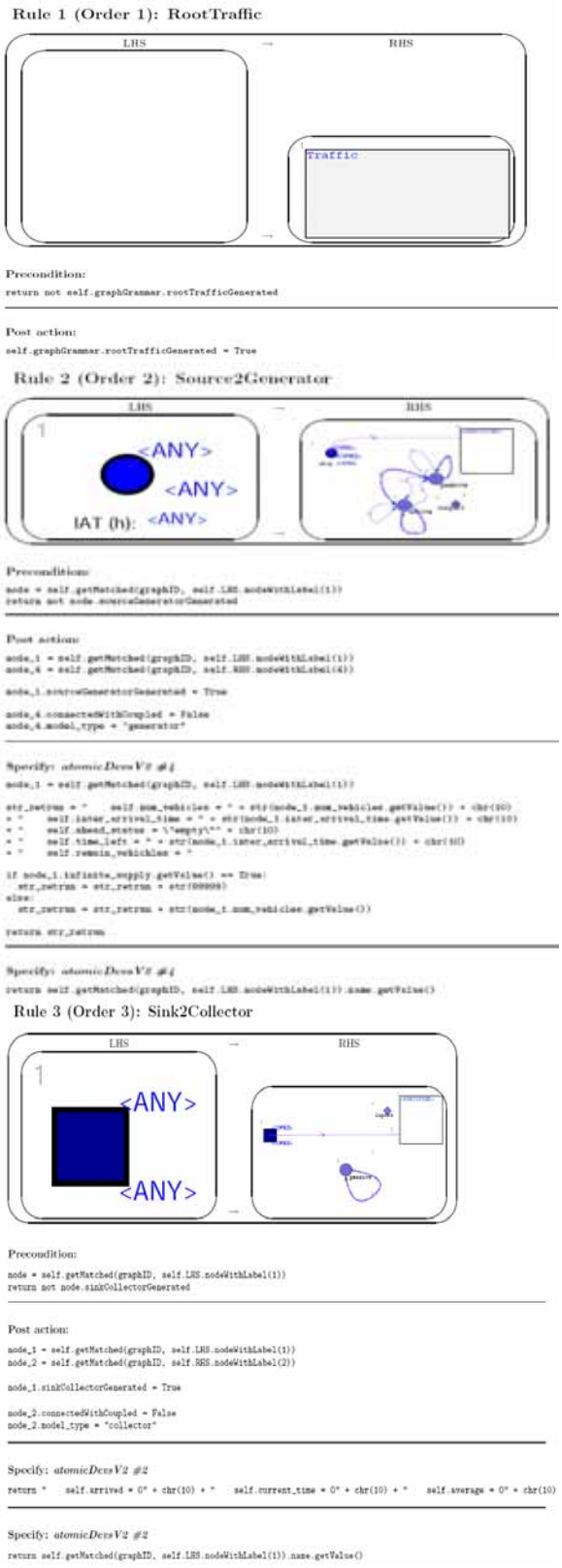


Figure 6: Model transformation rules (part 2)

Rule 4 (Order 4): Road2Road

Precondition:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not node.roadAtomicGenerated
```

Post action:

```
node_1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node_2 = self.getMatched(graphID, self.RHS.nodeWithLabel(2))

node_1.roadAtomicGenerated = True

node_2.connectedWithCoupled = False
node_2.model_type = "road"
```

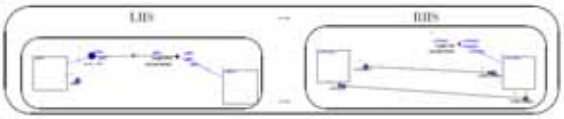
Specify: *atomicDevsV2 #2*

```
node_1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node_2 = self.getMatched(graphID, self.RHS.nodeWithLabel(2))

str_return = "    self.length = " + str(node_1.length.getValue()) + chr(10)
+ "    self.velocity_limit = " + str(node_1.velocity_limit.getValue()) + chr(10)
+ "    self.queue = []" + chr(10) + "    self.traffic_light = \"green\"" + chr(10)
+ "    self.ahead_status = \"empty\"" + chr(10) + "    self.changed = \"false\"" + chr(10)
+ "    self.current_time = 0" + chr(10) + "    self.time_left = 99999" + chr(10)
+ "    self.capacity = 99999" + chr(10)

return str_return
```

Specify: *atomicDevsV2 #2*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue()
```

Rule 5 (Order 5): GeneratorWithRoad



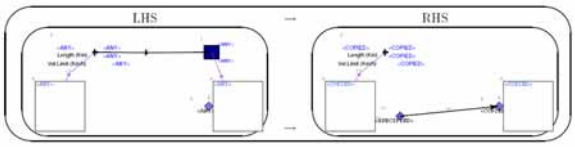
Specify: *portDevsV2 #10*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(5)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_flow_in'
```

Specify: *portDevsV2 #18*

```
return 'to_' + self.getMatched(graphID, self.LHS.nodeWithLabel(5)).name.getValue() + '_info_out'
```

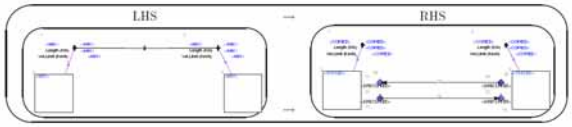
Rule 6 (Order 6): RoadWithSink



Specify: *portDevsV2 #11*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue() + '_flow_out'
```

Rule 7 (Order 7): Flow2Channel



Specify: *portDevsV2 #10*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_flow_out'
```

Specify: *portDevsV2 #12*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_flow_in'
```

Specify: *portDevsV2 #17*

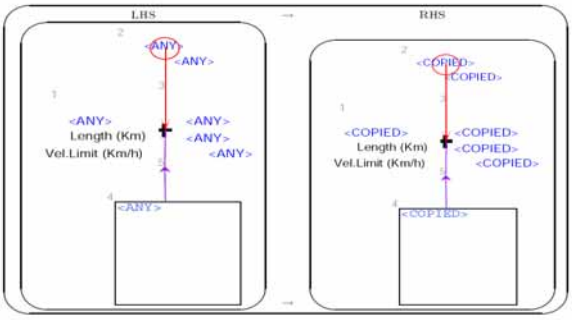
```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_info_in'
```

Specify: *portDevsV2 #19*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_info_out'
```

Figure 7: Model transformation rules (part 3)

Rule 8 (Order 8): CapacityWithRoad



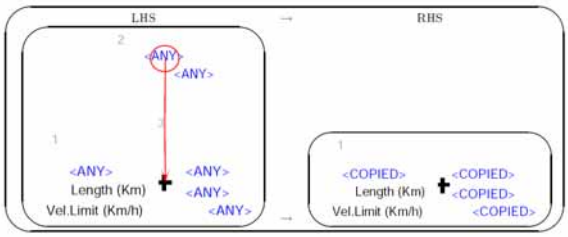
Precondition:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
return not node.capacityInfoGenerated
```

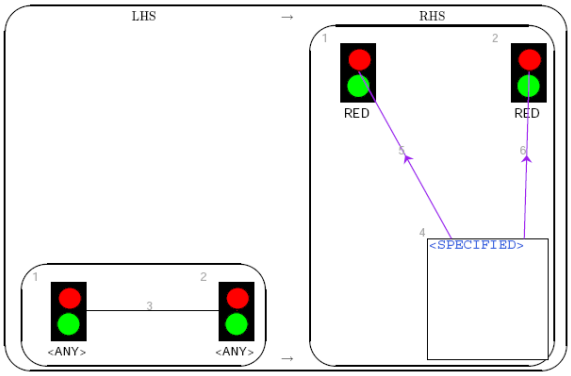
Post action:

```
self.getMatched(graphID, self.LHS.nodeWithLabel(2)).capacityInfoGenerated = True
```

Rule 9 (Order 9): CapacityCleanup



Rule 10 (Order 10): TrafficLightSync1



Precondition:

```
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node2 = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
return (not node1.trafficLightSyncGenerated) and (not node2.trafficLightSyncGenerated)
```

Post action:

```
self.getMatched(graphID, self.LHS.nodeWithLabel(1)).trafficLightSyncGenerated = True
self.getMatched(graphID, self.LHS.nodeWithLabel(2)).trafficLightSyncGenerated = True

node_4 = self.getMatched(graphID, self.RHS.nodeWithLabel(4))
node_4.connectedWithCoupled = False
node_4.model_type = "trafficLight"

self.graphGrammar.TL_count = self.graphGrammar.TL_count + 1
```

Specify: *atomicDevsV2 #4*

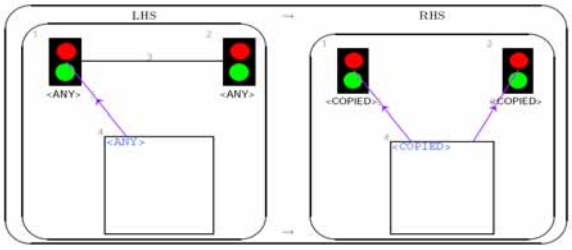
```
return ""
```

Specify: *atomicDevsV2 #4*

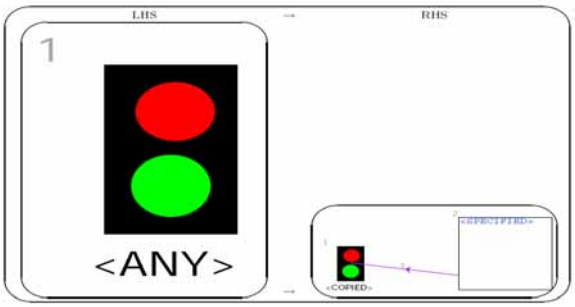
```
return "TL" + str(self.graphGrammar.TL_count)
```

Figure 8: Model transformation rules (part 4)

Rule 11 (Order 11): TrafficLightSync2



Rule 12 (Order 12): TrafficLightSync3



Precondition:

```
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return not node1.trafficLightSyncGenerated
```

Post action:

```
self.getMatched(graphID, self.LHS.nodeWithLabel(1)).trafficLightSyncGenerated =True

node_2 = self.getMatched(graphID, self.RHS.nodeWithLabel(2))
node_2.connectedWithCoupled = False
node_2.model_type = "trafficLight"

self.graphGrammar.TL_count = self.graphGrammar.TL_count + 1
```

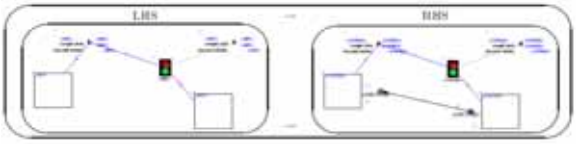
Specify: *atomicDevsV2 #2*

```
return ""
```

Specify: *atomicDevsV2 #2*

```
return "TL" + str(self.graphGrammar.TL_count)
```

Rule 13 (Order 13): TrafficLightWithRoad



Precondition:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(3))
return not node.trafficLightWithRoadGenerated
```

Post action:

```
self.getMatched(graphID, self.LHS.nodeWithLabel(3)).trafficLightWithRoadGenerated = True
```

Specify: *portDevsV2 #11*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_ctrl_in'
```

Specify: *portDevsV2 #13*

```
return self.getMatched(graphID, self.LHS.nodeWithLabel(1)).name.getValue() + '_To_'
+ self.getMatched(graphID, self.LHS.nodeWithLabel(2)).name.getValue() + '_ctrl_out'
```

Rule 14 (Order 14): TrafficLightCleanup1

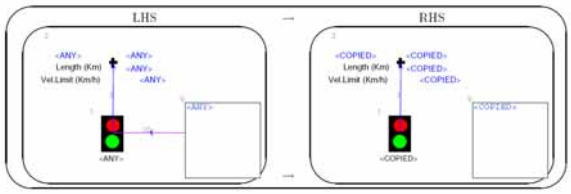
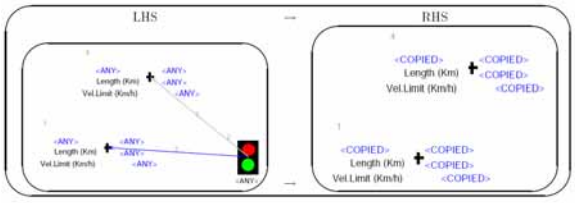
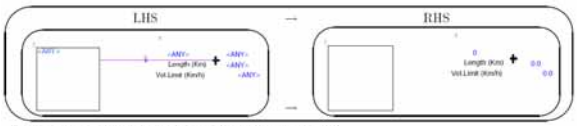


Figure 9: Model transformation rules (part 5)

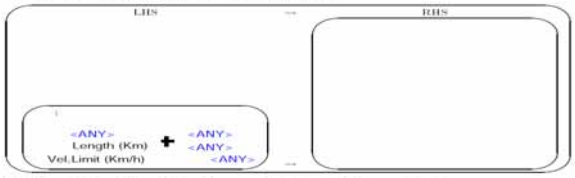
Rule 15 (Order 15): TrafficLightCleanup2



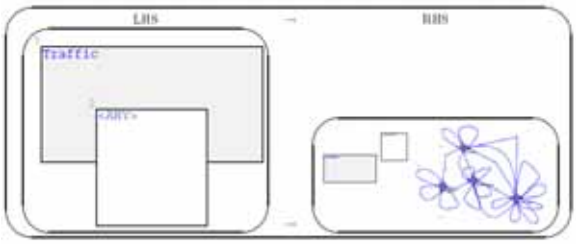
Rule 16 (Order 16): RoadCleanup1



Rule 17 (Order 17): RoadCleanup2



Rule 18 (Order 18): ConnectAtomic2Coupled1



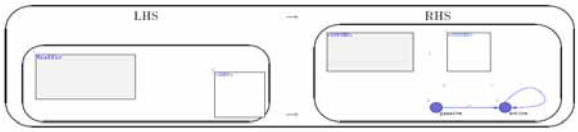
Precondition:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
return (not node.connectedWithCoupled) and (node.model_type == "road")
```

Post action:

```
self.getMatched(graphID, self.LHS.nodeWithLabel(2)).connectedWithCoupled = True
```

Rule 19 (Order 19): ConnectAtomic2Coupled2



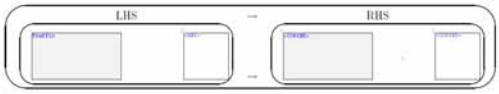
Precondition:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
return (not node.connectedWithCoupled) and (node.model_type == "trafficLight")
```

Post action:

```
self.getMatched(graphID, self.LHS.nodeWithLabel(2)).connectedWithCoupled = True
```

Rule 20 (Order 20): ConnectAtomic2Coupled3



Precondition:

```
node = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
return (not node.connectedWithCoupled) and (node.model_type == "generator" or node.model_type == "collector")
```

Post action:

```
self.getMatched(graphID, self.LHS.nodeWithLabel(2)).connectedWithCoupled = True
```

Figure 10: Model transformation rules (part 6)

5. Code generation

PythonDEVS [4] is a modelling and simulation package which provides an implementation of the standard classic **DEVS** simulation algorithm as introduced in [3]. The package consists of two files, the first of which (DEVS.py) provides a class architecture that allows hierarchical classic DEVS models to be easily defined by sub-classing the **AtomicDEVS** and **CoupledDEVS** classes. The codes generated from transformed **DEVS** models are suitable for simulation by **PythonDEVS**. In this project, the implementation of code generation is an extension of the work described in [3]. Each model (atomic and coupled) is compiled into a class, which is a subclass of AtomicDEVS or CoupledDEVS.

Figure 12 shows, as an example, when generating codes for a *RoadSection* atomic model, all parts related to the code generation process in AToM^3. Each model (atomic and coupled) is compiled into a class, which is a subclass of **AtomicDEVS** or **CoupledDEVS**. To avoid infinite states in an atomic model, such as *RoadSection* atomic model, condition scripts are added into transitions. For example, when a message of Car enters into a *RoadSection*, to determine which state should be the next one, the external transition has to check the current length of the queue in which cars are stored. This condition script is as follows:

```
def getDirection():
  for outport in outports_list:
    outport_name = outport[0]
    if (outport_name[-9:] == "_flow_out"):
      return outport_name
  return None

for inport in inports_list:
  inport_name = inport[0]
  inport_value = inport[1]
  if inport_value != None:
    if (inport_name[-8:] == "_flow_in"):
      if (len(self.queue) > 0) and (self.changed == "false"):
        self.current_time = self.current_time + e
        direction = getDirection()
        self.queue = [(inport_value, direction, self.current_time)] +
self.queue
        self.changed = "true"
        self.time_left = self.time_left - e
        return True
return False
```

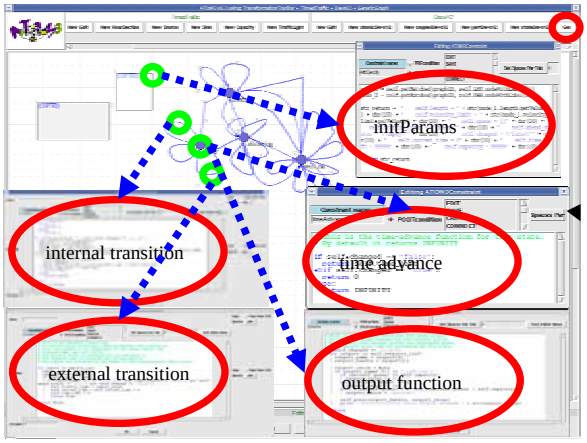


Figure 12:

The above code script also shows that to determine from which kind of ports the incoming messages are, we have some naming conventions for every generated ports:

For *Source* atomic model: the name of output port is *outport* and the name of input port is *info_in*;

For *Sink* atomic model: the name of input port is *inport*;

For *Road* atomic model: the name of output port to which messages of *Cars* go out is *(self model name)_(next model name)_flow_out*, the name of input port from which messages of *Cars* come in is *(previous model name)_(self model name)_ flow_in*, the name of output port to which messages of *State* go out is *(self model name)_(previous model name)_info_out*, the name of input port from which messages of *Cars* come in is *(next model name)_(self model name)_ info_in* and the name of input port from which messages of *TrafficLightStatus* come in is *(model name controlled by traffic light)_(model name directed by traffic light)_ ctrl_in*;

For *TrafficLight* atomic model: the name of output port to which messages of *TrafficLightStatus* go out is *(model name controlled by traffic light)_(model name directed by traffic light)_ ctrl_in*.

6. Conclusions

Domain-specific modelling, such as **Traffic** modelling we showed, can maximally constrains users, allowing them, by construction, to only build syntactically and semantically correct models.

The **Parallel DEVS** formalism as an extension of **Classic DEVS** from which all its advantages that allows the rigorous description of complex dynamic systems, the definition of component-based models and the efficient simulation algorithms for these

models are inherited, also overcomes some limitations of **Classic DEVS** for parallel simulation.

Meta-modelling which alleviates the problem of building and interconnecting a plethora of different tools when modelling complex systems, combined with model transformation which is based on Graph Grammar, smoothly bridges between domain-specific modelling (i.e., **Traffic**) and general modelling and simulation (i.e., **Parallel DEVS**).

This project shows how the whole process from **Traffic** modelling, to **Traffic** models to **DEVS** models mapping, and to **PythonDEVS** simulation code generation can be done in **AToM^3**.

Future work will address code generation for other **DEVS** simulation framework (i.e., **DEVSJAVA**) and in other languages (i.e., **Java**). Another line of work that needs attention is the use of a neutral language for specifying transition conditions, time-advancing and output functions of atomic **DEVS** models in **AToM^3**.

Acknowledgments

We wish to thank Denis Dube and Sagar Sen for their suggestions and helps in this project.

References

[1] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim, *Theory of Modeling and Simulation*, Academic Press, 2000.

[2] Hans Vangheluwe, Juan de Lara, *Computer Automated Multi-Paradigm Modelling for Analysis and Design of Traffic Networks*, Proceedings of the 2004 Winter Simulation Conference.

[3] Ernesto Posse and Jean-Sebastien Bolduc, *Generation of DEVS Modeling and Simulation Environments*, Proceedings of the 2003 Summer Simulation MultiConference, 2003.

[4] Jean-Sebastien Bolduc and Hans Vangheluwe, *A modelling and simulation package for classic hierarchical DEVS*, Technical report, McGill University, School of Computer Science, 2002.

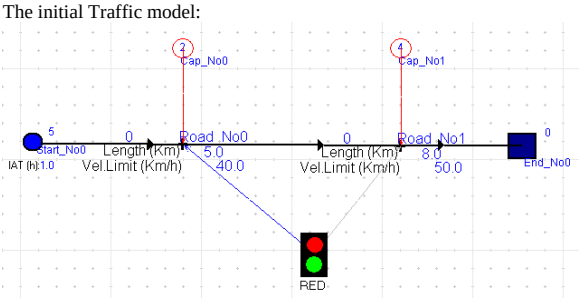
[5] Modelling, Simulation and Design Lab, *AToM3 V0.3: A Tool for Multi-formalism and Meta-Modelling*, http://msdl.cs.mcgill.ca/MSDL/research/

[6] Denis Dube, *GenGed vs. AToM3: Creating a visual DEVS modelling environment*, http://moncs.cs.mcgill.ca/people/hv/teaching/MSBDesign/presentations/050324.DenisDube.pdf

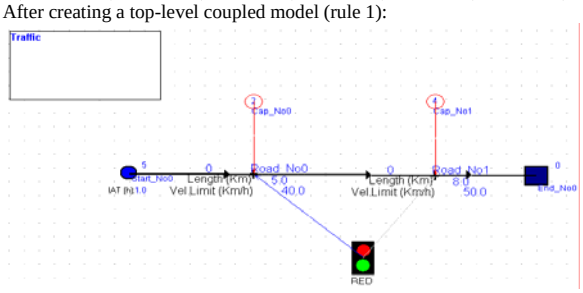
[7] Bernard P. Zeigler, Hessam S. Sarjoughian, *Introduction to DEVS Modeling and Simulation with JAVA*, http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA

Appendix A – An example of the application of the transformation rules described in Section 4.1

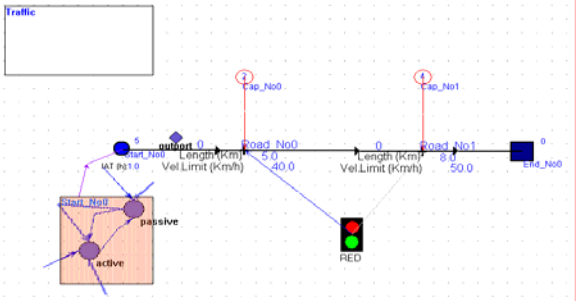
The initial Traffic model:



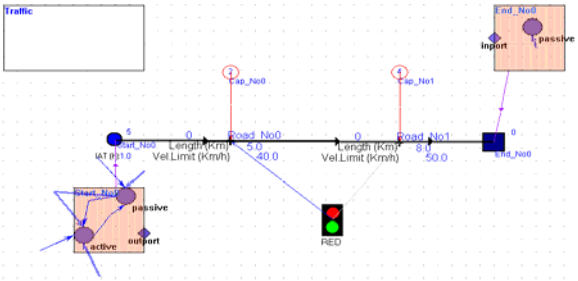
After creating a top-level coupled model (rule 1):



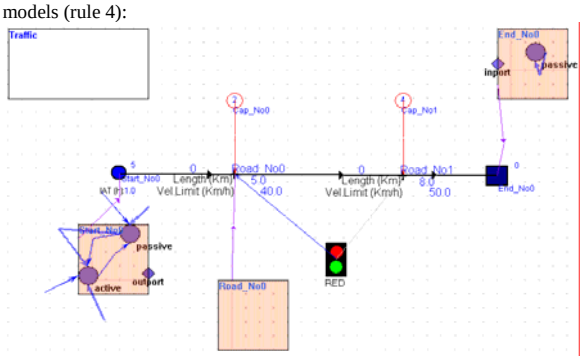
After transforming **Traffic** *Source* nodes into **DEVS** *Generator* atomic models (rule 2):



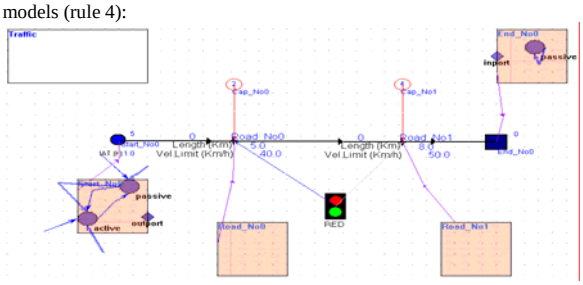
After transforming **Traffic** *Sink* nodes into **DEVS** *Collector* atomic models (rule 3):



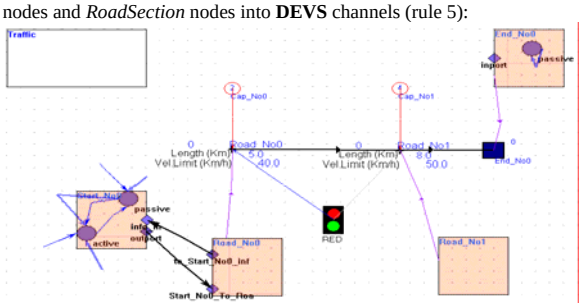
After transforming **Traffic** *RoadSection* nodes into **DEVS** *Road* atomic models (rule 4):



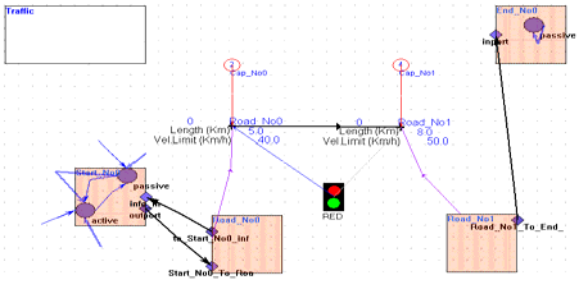
After transforming **Traffic** *RoadSection* nodes into **DEVS** *Road* atomic models (rule 4):



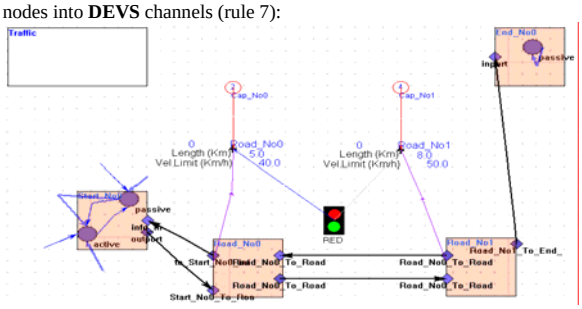
After transforming **Traffic** *Source2Section* connections between *Source* nodes and *RoadSection* nodes into **DEVS** channels (rule 5):



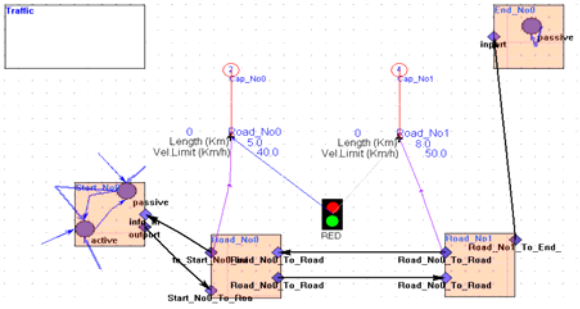
After transforming transforms **Traffic** *Section2Sink* connections between *RoadSection* nodes and *Sink* nodes into **DEVS** channels (rule 6):



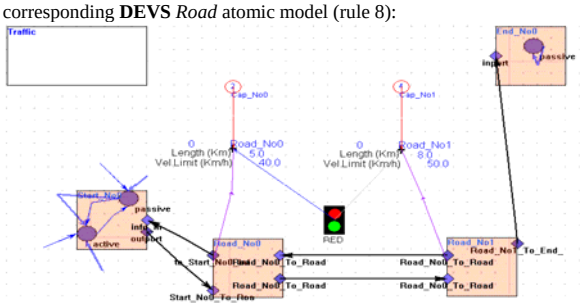
After transforming **Traffic** *FlowTo* connections between *RoadSection* nodes into **DEVS** channels (rule 7):



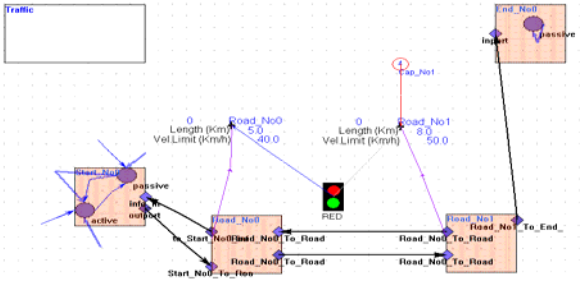
After copying the capacity attribute of **Traffic** *Capacity* node into the corresponding **DEVS** *Road* atomic model (rule 8):



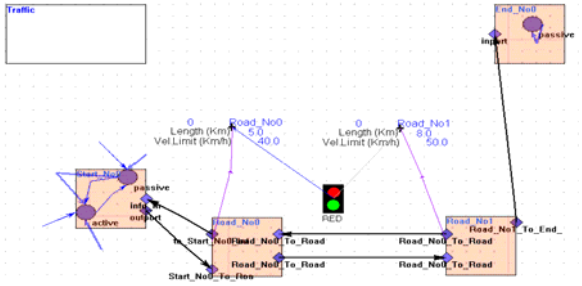
After copying the capacity attribute of **Traffic** *Capacity* node into the corresponding **DEVS** *Road* atomic model (rule 8):



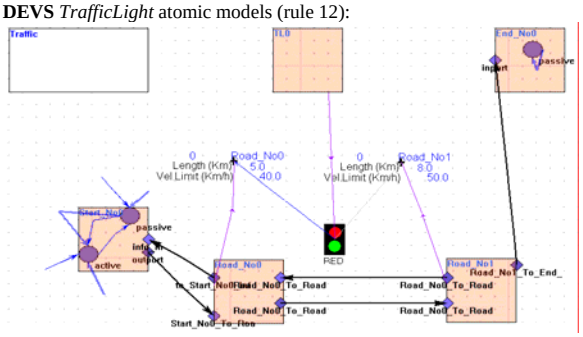
After removing the no longer needed **Traffic** *Capacity* nodes (rule 9):



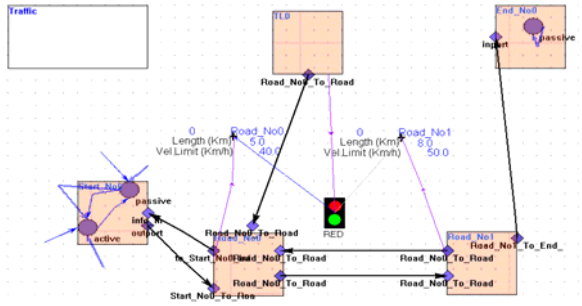
After removing the no longer needed **Traffic** *Capacity* nodes (rule 9):



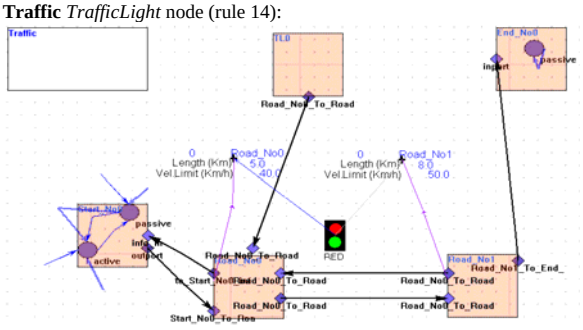
After transforming **Traffic** unsynchronized *TrafficLight* nodes into **DEVS** *TrafficLight* atomic models (rule 12):



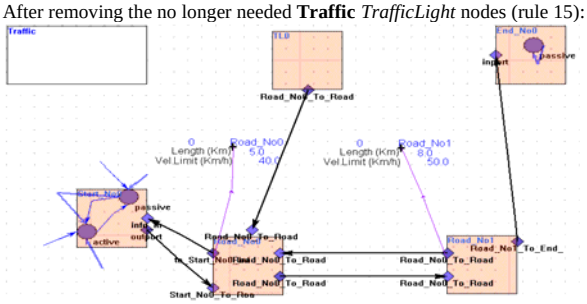
After transforming **Traffic** *ControlledSection* connections between *TrafficLight* nodes and *RoadSection* nodes into **DEVS** channels (rule 13):



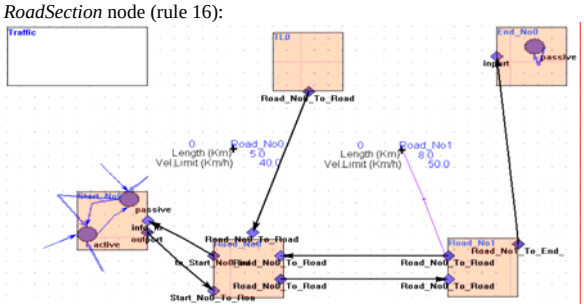
After removing the special link between **DEVS** *TrafficLight* model and **Traffic** *TrafficLight* node (rule 14):



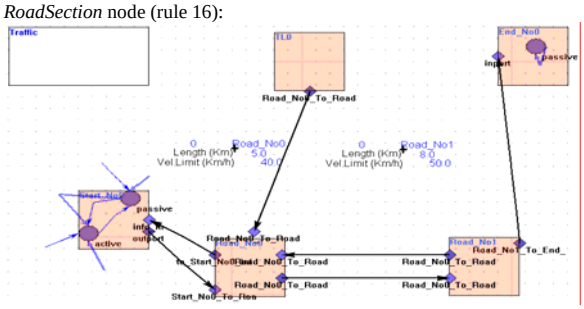
After removing the no longer needed **Traffic** *TrafficLight* nodes (rule 15):



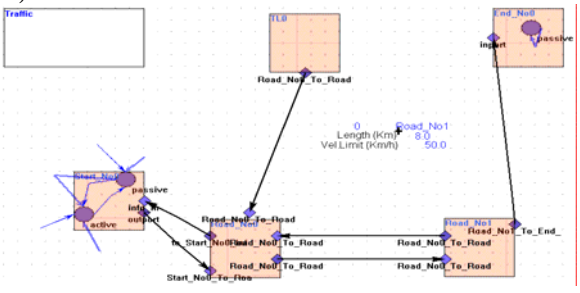
After removing the special link between **DEVS** *Road* model and **Traffic** *RoadSection* node (rule 16):



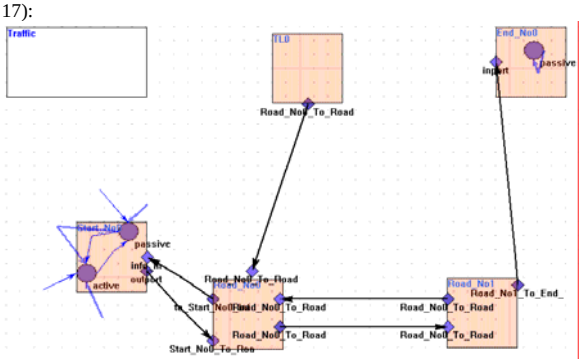
After removing the special link between **DEVS** *Road* model and **Traffic** *RoadSection* node (rule 16):



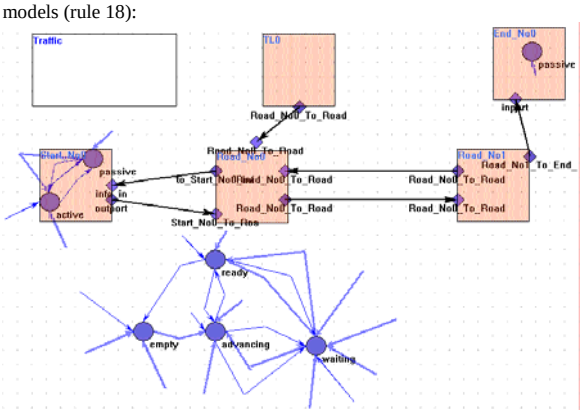
After removing the no longer needed **Traffic** *RoadSection* nodes (rule 17):



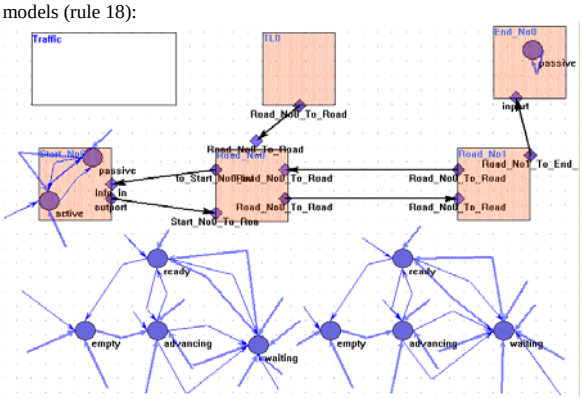
After removing the no longer needed **Traffic** *RoadSection* nodes (rule 17):



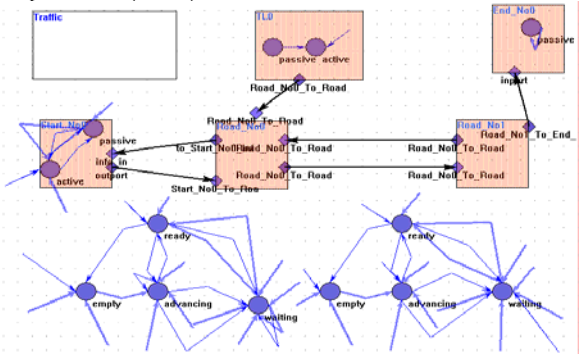
After connecting **DEVS** *Road* atomic models with **DEVS** *Traffic* coupled models (rule 18):



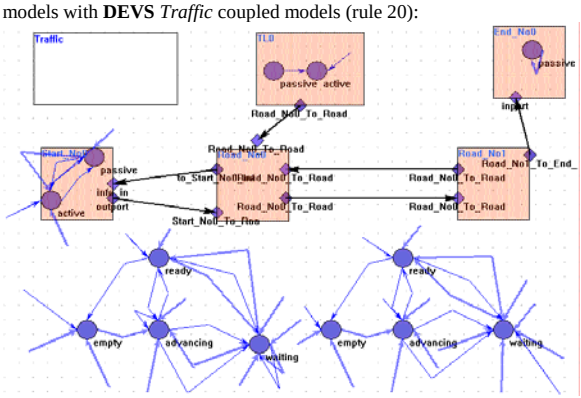
After connecting **DEVS** *Road* atomic models with **DEVS** *Traffic* coupled models (rule 18):



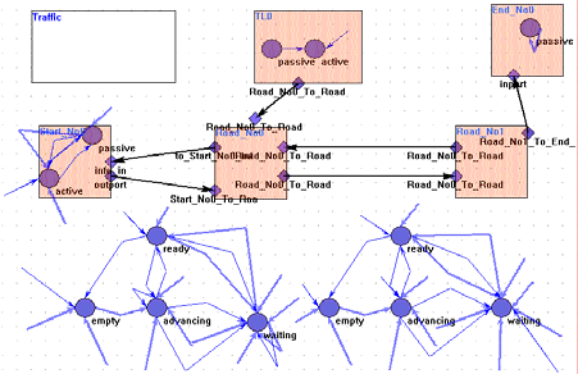
After connecting **DEVS** *TrafficLight* atomic models with **DEVS** *Traffic* coupled models (rule 19):



After connecting **DEVS** *Generator* atomic models and *Collector* atomic models with **DEVS** *Traffic* coupled models (rule 20):



After connecting **DEVS** *Generator* atomic models and *Collector* atomic models with **DEVS** *Traffic* coupled models (rule 20):



The final model after some adjustments: