DEVS
Modelling and Simulation

Ken Bauwens 20143225
and Baturay Ofluoglu 20174797

December 20, 2017

1 Introduction

In this report we will briefly explain our models and our simulations as well as the results of those simulations.

1.1 Who did What

We worked on everything together

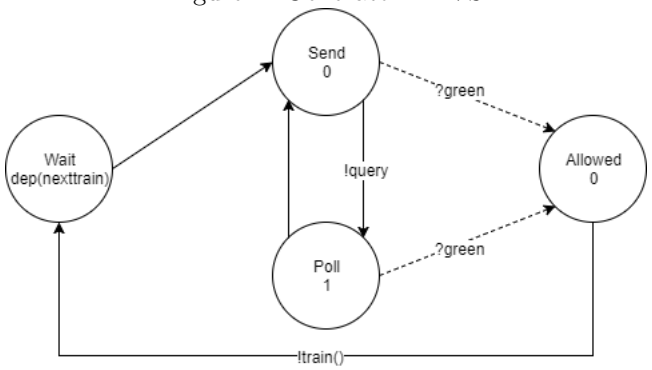
1.2 Time spent

We spent about 7hrs on the modeling, and about 3hrs on the simulation.
This does not include the time spent writing the report.

2 Modeling

2.1 Generator

Figure 1: Generator DEVS



Specification:

X	:	{ "green" }
Y	:	{ "query", train() }
S	:	{ Wait, Send, Poll, Allowed }
S_0	:	(Wait, 0)
Ta	:	{ Wait: max(0, deptime - elapsedtime) if train left else ∞, Send: 0, Poll: 1, Allowed: 0}
δ_{ext}	:	{ (Send,_, "green") $\rightarrow$ Allowed, (Poll,_, "green") $\rightarrow$ Allowed}
δ_{int}	:	{ Wait $\rightarrow$ Send, Send $\rightarrow$ Poll, Poll $\rightarrow$ Send, Allowed $\rightarrow$ Wait}
λ	:	{ Send $\rightarrow$ "query", Allowed $\rightarrow$ train()}

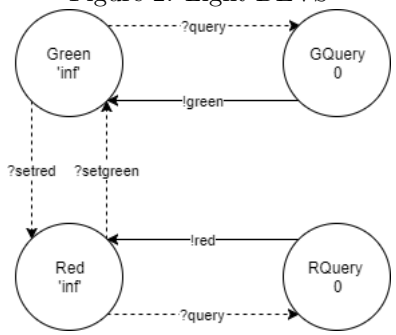
Explaination: When this model is initialized, it will instantly generate *numTrains* trains and add these to a list. Each train will get a value for *a_max* and *deptime*. This *deptime* is calculated as $deptime_{PrevTrain} + IAT$, with the first train having a deptime of IAT.
The generator will start in a state *Wait*. When $t_a(Wait)$ runs out it will transition to a state *Send*, where it will start polling every 1 second until it receives an external event *"green"*. It will then transition to an intermediate state *Allowed*, where the train will be sent out, before returning to the *Waitstate* where it will restart the loop.
Using this construction and the variable value for $t_a(Wait)$, the generator will always start polling when *self.elapsedtime* $\geq$ *nexttrain.deptime*, ensuring that a train does not leave too early but also leaves as soon as it is allowed.

2.2 Railroad

We decided to model the railroad as a CompositeDEVS. This CompositeDEVS will consist of a light-atomicDEVS and a railway-atomicDEVS. This is done to prevent having to add too many unnessecary transitions, reducing the complexity of railroads.

2.2.1 Light

Figure 2: Light DEVS



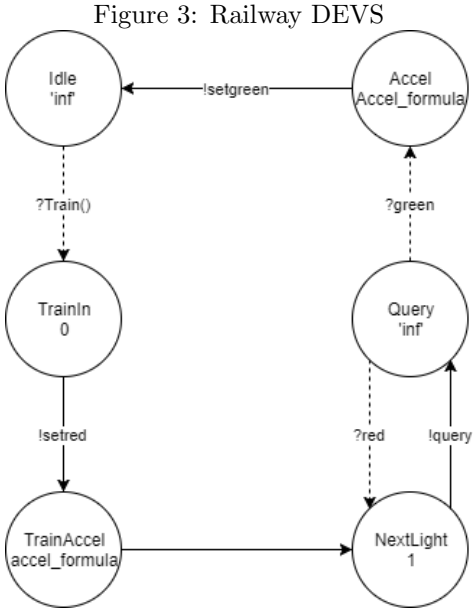
Specification:

X	:	{ "query", "setgreen", "setred" }
Y	:	{ "red", "green" }
S	:	{ Green, Red, GQuery, RQuery }
S_0	:	(Green, 0)
Ta	:	{ Green: ∞, Red: ∞, GQuery: 0, RQuery: 0 }
δ_{ext}	:	{ (Green,_, "query") $\rightarrow$ GQuery, (Red,_, "query") $\rightarrow$ RQuery}
δ_{int}	:	{ GQuery $\rightarrow$ Green, RQuery $\rightarrow$ Red}
λ	:	{ GQuery $\rightarrow$ "green", RQuery $\rightarrow$ "red"}

Explaination: The light will start in a state *Green*, where it will stay indefinitely. Only on receiving an external event "Setred" will it transition to state *red*. It can only transition out of this state when receiving an event "Setgreen".
Each of these states will also transition to another state *RQuery* or *GQuery* (depending on which state it is in), which indicates that a train in a previous segment wants to enter. It will then immediately return to the correct state, and it will generate an event "red" or "green".

2.2.2 Railway

Figure 3: Railway DEVS



Specification:

X	:	{ "green", train(), "red" }
Y	:	{ "setred", "query", "setgreen" }
S	:	{ Idle, TrainIn, TrainAccel, NextLight, Query, Accel }
S_0	:	(Idle, 0)
Ta	:	{ Idle:∞, Trainin: 0, TrainAccel: accel_formula(), NextLight: 1, Query: ∞, Accel: accel_formula() }
δ_{ext}	:	{ (Idle,_, train()) $\rightarrow$ TrainIn, (Query,_, "red") $\rightarrow$ NextLight, (Query, _, "green") $\rightarrow$ Accel }
δ_{int}	:	{ TrainIn $\rightarrow$ TrainAccel, TrainAccel $\rightarrow$ NextLight, NextLight $\rightarrow$ Query, Accel $\rightarrow$ Idle}
λ	:	{ TrainIn $\rightarrow$ "setred", Accel $\rightarrow$ "setgreen", Accel $\rightarrow$ train(), NextLight $\rightarrow$ "query"}

Explaination: The railway will start in a state *Idle*. It will stay there indefinitely. Only when it receives an external event "Train" will it transition to the *TrainIn* state. This state is a intermediate state which will immediately transition to state *TrainAccel*. On transitioning it will also raise an event "Setred", which sets it's corresponding light to red as explained in the previous section. The *TrainAccel* state represents the initial acceleration of the train when it enters the new track. It will therefor use the acceleration-formula to calculate it's $T_a()$. After this time, it will transition to state *NextLight*, where it will start polling the next track's light every second. If it receives an event "red" indicating that the next track is still occupied, it will recalculate the current position of the train, aswell as it's current velocity using the brake-formula. As soon as it receives an event "green", the system will transition to state *Accel*, which works the same way as the *TrainAccel* state except with the remaining distance on this track as parameter to the acceleration-formula.

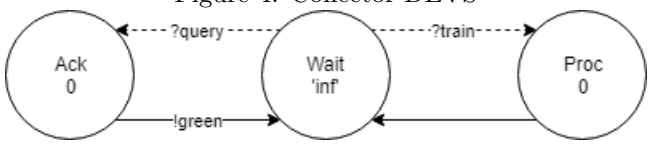
2.2.3 RailwaySegment

Explaination: An entire RailwaySegment will consist of a single railway and a single light. It will simply link the correct inputs to the correct outputs. The railway's *setred* and *setgreen* events will be linked to the light, to facilitate changing lights. The entire system receives/answers to, querries from the previous model and sends to/receives answers from, the next railway segment. These querries are linked to the light and the railway respectively. Lastly, the *train()* output from a railway will be an output of the composite DEVS, and will be linked to the *train()* input of the next track/Collector.
This model does not have a tie-breaker function, because it does not really mather which of the two subcomponents acts first.

2.3 Collector

The model for the collector can be seen in figure 4.

Figure 4: Collector DEVS



Specification:

X	:	{ "query", train() }
Y	:	{ "green" }
S	:	{ Wait, Ack, Proc}
S_0	:	(Wait, 0)
Ta	:	{ Wait: ∞, Send: 0, Poll: 0 }
δ_{ext}	:	{ (Wait,_, "query") $\rightarrow$ Ack, (Wait,_, train()) $\rightarrow$ Proc}
δ_{int}	:	{ Ack $\rightarrow$ Wait, Proc $\rightarrow$ Wait }
λ	:	{ Ack $\rightarrow$ "green"}

Explaination: The collector starts in a state *Wait*, where it will stay indefinitely. It will only transition on external events.
The first event that will cause a transition is a *Query*. This indicates that a train on the previous railway wants to enter. The state *Ack* will immediately return to *Wait*. Because the collector always allows trains to enter, it will generate an output *green* to indicate that the train can continue.
The second event indicates that a train has entered. This event is a train-instance. When receiving this event, the system will transition from *Wait* to *Proc*, where it will proces the train statistics before immediately returning to *Wait*.

2.3.1 Statistics

To find optimal parameters for the system, a user needs access to relevant statistics. In our case, this statistic is $(lights * 10) + TravelTime_{Average}$. To calculate this statistic, the collector will maintain the average traveltime. It will add the traveltime for each train and it will count how many trains it has processed yet. The average can then easily be calculated.

2.4 Entire system

Explaination: The entire system will consist of one generator, one collector and a specified amount of railwaysegments. These submodels are linked together according to the specification.
Each submodel will have it's *Q_send* linked to the next submodel's *Q_recv* (except of course for the collector), and each submodel will have it's *Q_sack* linked to the previous submodel's *Q_rack* (except of course for the generator). Each submodel will also be able to pass on a train with the correct connections as described in the specification.
The entire model needs a tiebreaker function to prevent a collision. Assume that two trains are waiting in the generatorqueue and that at elapsed time x, these two trains should have already left the generator. When the first train gets a green light, it will transition to the first traintrack. When this happens however, the next train will be scheduled to send after a time 0, so the generator will immediately query the first track. This track still has his light on green though, because the setred event has to still be generated when a train enters the track. This means that we need to make sure that the light is changed before the generator querries again.

3 Simulation

For the simulation results refer to simulation_< #ofTrains>.html files.

During simulation stage, we suppose that the total length of railway track is constant as 25,000 km and the maximum allowed speed is 150 km/h for all trains. The acceleration of trains are also uniformly distributed between 5 and 30. In this simulation, our purpose was to find the optimal or near optimal total cost by changing number of track segments, inter-arrival times of trains and total number of trains as given below. Also, to make a fair comparison we assign dynamic segment length to keep the same total railway length for all simulations. The length of segments are calculated as $segmentLength = totalLength/\#Segments$

- Number of Segment = {6,7,..,29}
- Inter-Arrival TimesIAT = {*uniform(50,100)*, *uniform(25,50)*, *uniform(12,25)*}
- Number of Trains = {100, 200, 300}

We make a simulation of each variable combinations that are indicated above. We simulate each model 10 times to obtain more accurate results. Note that if we simulate more than 10 times, the accuracy would increase but we did not prefer it due to time constraint. To observe the results of the simulation, we preferred to use box plot to see the cost difference of 10 models. If you hover the mouse over box plots then you can see median, quartiles and outliers of cost values for the simulations.

Based on simulation results, for all simulations 25 railway segments have the minimum cost with 275.0 score. If there is only 7 segments for example, then for *uniform(12,25)* IAT, the cost is much higher than other IATs because of queue is higher than the others. After 25 segments, each IAT have same costs since there is no queue for all different IAT's.

For number of trains variables, if we have more train in the system and we have less than 14 segments, then the total cost is higher. However, after 14 segments the cost does not change significantly if number of trains are increased from 100 to 300.