General information

- The due date is December 20th, before 23:55.
- Submissions must be done via BlackBoard. Beware that BlackBoard's clock may differ slightly from yours. *All* results must be uploaded to BlackBoard and accessible from links in the main (index.html) file.
- The assignment must be made in groups of maximum 2 people. It is understood that all partners will understand the complete assignment (and will be able to answer questions about it). Clearly identify who did what.
- Grading will be done based on correctness and completeness of the solution. Do not forget to document your requirements, assumptions, design, implementation and modelling and simulation results in detail!
- To get feedback about the assignment workload, provide the number of hours you spent on this assignment.
- Contact: Simon Van Mierlo.

Goals

This assignment will make you familiar with modelling, simulation, and performance analysis in DEVS.

The problem

The purpose of this assignment is to model the behaviour of train traffic on a straight stretch of train tracks. The stretch is made of a sequence of smaller railway segments. For safety reasons, a single railway segment can only hold a single train, despite being much longer. Each segment is guarded by a traffic light in the beginning, which will be put to red if there is a train on that segment, or green otherwise.

You will use your created model to determine the ideal length of a railway segment. This is a trade-off situation: smaller segments will decrease the space between trains (increasing performance), but costs more money due to more traffic lights (decreasing cost efficiency). This can be seen in Figure 1. You may assume a simple cost function: *10 * number of lights + time in seconds to traverse total distance*. You will have to determine, for a few different settings, what is the ideal length of a single railway track for this cost function.

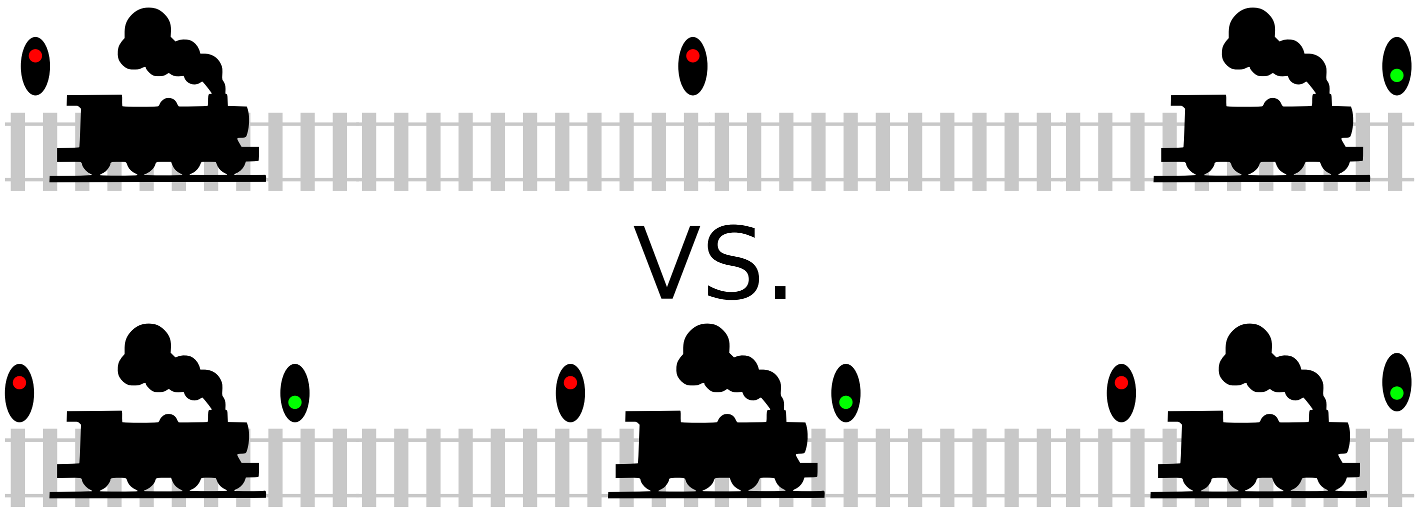

Figure 1: different configurations lead to different performance.

The model will consist of a Coupled DEVS model named `TrainNetwork`. This model is made up of a concatenation of one `Generator` Atomic DEVS model, followed by a series of `RailwaySegment` models (either Coupled or Atomic DEVS models), and terminated by a `Collector` Atomic DEVS model, as depicted in Figure 2.

Note how the modelling view we will take is called active resource (as opposed to the active entity or agent-based view). In the active resource view, it is the resources for which there is competition (and as a result, queueing), the railway segments, which are modelled as dynamic entities. In contrast, the trains moving around will be modelled as passive entities, passed around as (structured/parametrized) events between active resources. This is an antropopomorphic view, as railway segments are obviously not active, and do not make any decisions. It is however a convenient abstraction/view which is commonly used when modelling queueing systems.

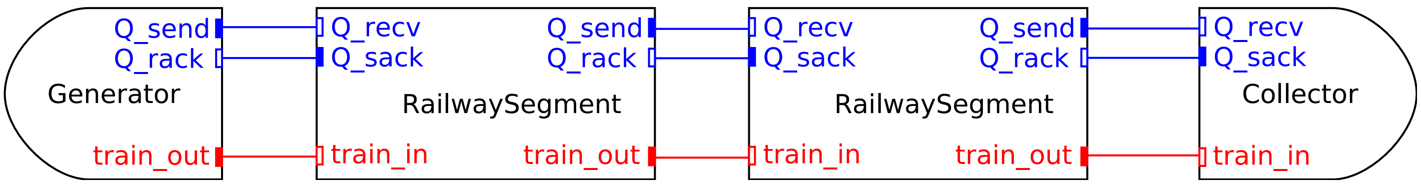

Figure 2: overview of the `TrainNetwork` structure.

The following is a detailed description of the three different DEVS models, as well as of the `Train`, `Query` and `QueryAck` entities sent (as events) between the DEVS building blocks.

Train

`Train`s are instances of the `Train` class, and are generated by the `Generator` model. They pass through a sequence of `RailwaySegment` models, and finally end up at the `Collector` model.

The `Train` class is a container for all relevant information pertaining to a train. It has the following attributes set at instantiation time:

- *ID*: a unique ID for the train.
- *a_max*: the maximal acceleration of the train in its current configuration (a positive number). It is different for each train, and is therefore sampled from a random distribution by the `Generator`.
- *departure_time*: the time at which the train leaves the `Generator`. It is the `Generator`'s responsibility to assign this value.

We give the `Train` class the following additional attributes, which are modified during simulation:

- *v*: the current speed at which the train is moving.
- *remaining_x*: the remaining distance on this railway segment.

Query

The moment a `Train` gets close to the next segment, a `Query` message will be sent to the railway segment that is being approached. This `Query` is used to model, at a discrete event level of abstraction, the train driver's observation of the next railway segment for the current state of the traffic light. The railway segment will reply to the `Query` with a `QueryAck`, described next.

QueryAck

The `QueryAck` is sent as the reply to the `Query` message. It contains information on the state of the traffic light, so either if it is green or red.

Generator

A `Generator` generates `Train` instances on its output port. Each time, the Inter Arrival Time (IAT) of trains is sampled from a uniform distribution over the interval *[IAT_min, IAT_max[*. Every time a `Train` instance is generated, it is passed a value for *a_max* by the `Generator`. This value is sampled from a uniform distribution over the interval *[a_min, a_max[*. *a_max* is a strictly positive number. For sampling from a uniform distribution, you should use Python's `random` module which implements pseudo-random number generators for various distributions. The initial velocity is always 0.

The basic behaviour of a `Generator` is: keep output-ing trains, with IAT and acceleration sampled from a distribution. The problem with this is that it is likely that at the moment of departure, there is still a train in the first railway segment.

So, the `Generator` should more accurately reflect what happens in the real world. You might have "scheduled" to leave at a certain time, but the moment you want to leave, you first look ahead. If there is nothing in your way, you will leave, otherwise you will wait until you can leave. This is what we need to model.

Even if the previous train cannot leave yet, the next train might already be scheduled to leave. The `Generator` will therefore need to have a queue of trains that are ready to leave (in *First-In First-Out order*). It will poll every 1 second to check if the traffic light of the segment became green. If so, it will output the first train of the queue to the first railway segment.

Collector

The `Collector` will receive `Train`s and will act as if it is a railway segment with a traffic light that is always green. It will therefore always respond with a `QueryAck` which allows the train to enter.

When the `Collector` receives a train, it will calculate performance metrics. In our case, it will compute the transit time of the train: how long did it take between the generation of the train, and arriving in the `Collector`. This transit time includes the time spent in the `Generator`'s queue.

Ideally, we would like as much insight as possible in the distribution of the above performance metric. For simplicity, you should however not collect the distribution in a table, but only the average.

Railway Segment

A `RailwaySegment` has the following parameters:

- *L*: the length of the railway segment

The `RailwaySegment` also keeps track of the train that is currently present on that segment.

When a Query arrives in the railway segment, it will reply immediately (*i.e.*, with no delay) with a `QueryAck`. This will grant permission (green light) if there is no train present in the `RailwaySegment` at the moment, or will refuse entry (red light) if there is a train on the `RailwaySegment`. Upon receiving a `QueryAck`, the resulting action depends on whether the `RailwaySegment` is clear or not.

If the `RailwaySegment` is clear (green light), the train will accelerate as fast as possible. It will keep going full throttle for the remainder of the railway segment, of course capped by its maximum speed. Some math is required to determine how long it takes before the end of the railway segment is reached, and at what velocity the train will leave the railway segment. Don't worry: the file formulas.py includes 2 functions which define both the acceleration and braking behaviour. The first function (`acceleration_formula`), computes the acceleration behaviour, and takes the parameters:

- *v_0*: the current velocity, expressed in meters per second.
- *v_max*: the maximum allowed speed, expressed in meters per second.
- *x_remaining*: the length of the railway segment that needs to be traversed, expressed in meters.
- *a*: the maximum acceleration of the train, expressed in meters per second squared.

It will return a tuple containing the new *velocity* (in meter per second) at the end of the provided distance, and the *time* it will take (in seconds). The second function (`brake_formula`), computes the braking behaviour, and takes the parameters:

- *v_0*: the current velocity, expressed in meters per second.
- *t_poll*: the time between two polls.
- *x_remaining*: the total distance that remains to be travelled before the red light.

It will return a tuple containing the new *velocity* (in meter per second) after *t_poll* seconds have passed, and the *distance* travelled in that period (in meters).

After the time, returned by the `acceleration_formula` function, has passed, the train needs to leave the current railway segment, and enters the next railway segment. The train will keep accelerating here too, but only until the traffic light of the next railway segment is within sight. We assume that a traffic light can be seen from a distance of 1000 meters. This means that your first call to the `acceleration_function` has to pass the distance of the first part of the segment (the total distance minus 1000). Therefore, the train will keep accelerating until it is 1000 meters before the traffic light, and will then look at the traffic light. During this observation, the train will just maintain its previous acceleration. From there on, it continues as specified above. Note that, if the length of the railway track is smaller than 1000, the driver will sometimes see multiple traffic lights. The driver will then only be interested in the traffic light of the next track, and not look at the further ones (yet).

If the railway segment is occupied (red light), the train will brake gradually. While it is possible to keep accelerating, and finally brake as hard as possible at the last second, this is not very comfortable for the passengers, nor is it particularly safe. Therefore, the train will decrease its velocity linearly, by using a negative (but constant) acceleration. This is automatically computed by the provided formulas, and will make it such that the train would stop right in front of the traffic light. We assume that the tracks are sufficiently long for a train driving the maximum speed, to come to a halt before reaching the traffic light. As such, there is no need to model cases where the train can not stop in time.

However, the train driver will poll the traffic light every second, by sending a new `Query`. If entry is still prohibited (red light), this behaviour will be repeated. If entry is allowed (green light), the train will again accelerate for the remaining distance of the track. Note that we assume that it is impossible for a train driver to see a green light, which later on turns red.

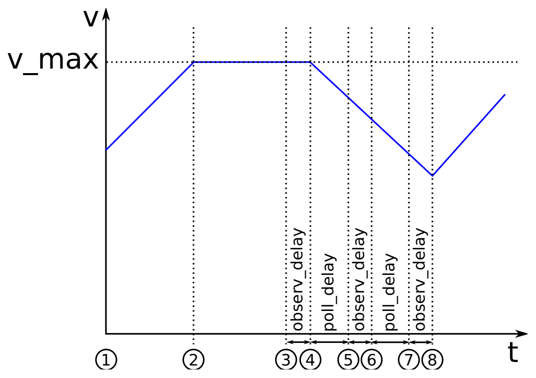

Figure 3: The evolution of the speed in different phases.

An example is shown in Figure 3. At point 1, the segment is entered, and the train accelerates until it reaches *v_max*. At point 2, *v_max* is reached, and the speed is capped. At point 3, the first `Query` message is sent to the next railway segment. During this phase, the train does not change its velocity yet. At point 4, a `QueryAck` is received from the next railway segment, which says that it is red. Therefore, the train starts to brake. At point 5, the second `Query` is sent. The train keeps braking until a reply is received. At point 6, the `QueryAck` is received, which still results in a red light, thus further braking the train. At point 7, the same happens as on point 5. At point 8, finally, a permissive `QueryAck` (green light) is received. The train starts to accelerate again, until it leaves the railway segment.

With the provided formulas, you would need to call the `acceleration_formula` at point 1, which will return you the time of point 3. Point 2 is completely taken care of by the functions. At point 3, a `Query` is sent, but the train keeps accelerating, so the `acceleration_formula` is again needed. At point 4, the train knows that it has to brake, and it will call the `brake_formula`, and will poll again after the polling time. At point 5, a new `Query` is sent, but because we keep braking, the `brake_formula` is called, thus computing the distance travelled during the wait for poll. At point 6, exactly the same happens as in point 4. Point 7 is again identical to point 5. Then, at point 8, the train gets a green light, and therefore starts accelerating again for the remaining distance. Here again, the `acceleration_formula` is called, which will return the time at which the railway segment has to be exitted.

Note that in our abstraction, a driver that saw a green light will never respond to a red light anymore. This is because it is a straight stretch, and the light would have no reason to switch to red again after it was green.

Simulation

Make a plot showing both the performance and the cost of the railway network for a varying length of segments. Do this for a fixed set of parameters. Plot the value of the cost function for a fixed set of parameters, thus showing which length is ideal for these parameters.

Finally, run these simulations for a varying IAT, plotting the ideal railway segment size for each possible configuration of the IAT. Plot your results. You can do the same while varying other parameters.

Perform a reasonable number of simulation experiments and be sure to sample with enough trains. Make sure that the total length of the railway track is equal: 10 segments of 1 kilometer should not be compared to 10 segments of 2 kilometers! Always choose the simulation duration sufficiently long enough to get statistically relevant measurements. Discuss your results!

Practical issues

You will use the PythonPDEVS simulator. You are strongly advised to first study the Classic DEVS TrafficLight example in the examples directory before starting on this assignment. Installing PythonPDEVS can be done by executing `python setup.py install --user`. Note that you need Python 2.7 to be able to run PythonPDEVS. Finally, you can run the simulation by executing `python experiment.py`.

Note that in `PythonPDEVS`, when an external transition is triggered, this means that some external input has arrived on *one or more of the ports*. The inputs will be passed to the method in the form of a dictionary, which is the only argument of the `extTransition` method. The key values of this dictionary are the ports. If a port is not present in the dictionary, there was no input on that port. The elapsed time can be accessed using the `elapsed` attribute, but note that this is only correct in the `extTransition` method. It is therefore *not allowed* to access the `elapsed` attribute in an `intTransition` (and its value will be undefined).

Apart from the model, you will also need to create an experiment file. In that file, you instantiate the model and the simulator, and run the simulation with a specific configuration. Make sure that you run the simulation as Classic DEVS, with the configuration option `setClassicDEVS(True)`. After the simulation, it is possible to access the model and its updated state.

As you will be comparing different approaches, you would normally have to create multiple simulators, one for each model being compared. The statistics can be printed after the `simulate` call and printed to the console, to be plotted at the end (*e.g.*, using gnuplot).

In Python, you can use the "infinity" value using the `float("inf")` construct.