Mapping the Railway formalism onto different domains

Zhong Xi Lu

March 2019

remove railway_wd/railway.py

add source to formulas.py

1 Introduction

Modelling is a powerful technique which allows us to work on the right abstraction level and avoid accidental complexity. Nowadays, we have a lot of different formalisms at our disposal, so it's necessary to choose the most appropriate language when building a model. Aside from that, it's also possible to define a mapping from one formalism to another, so that we can expose the functionality of one another.

This paper will revolve around the *Railway* formalism, which is mostly based on *Railway Operation and Control* [1] by *Joern Pachl* and some of the assignments [2] given in the *Model Driven Engineering* course in the *University of Antwerp*. This formalism is first modelled in the tool *AToMPM* [3] to create a basic visual modelling environment, here we can also simulate a model by defining its operational semantics. To analyze if a model satisfies certain properties, we will map it to petri-nets, where we can do a reachability, coverability, deadlock, ... analysis. Next to that, we can also map it to *Discrete Event System Specifications* (DEVS), which is more appropriate when it comes to queueing, throughput, ... analysis. Finally, to visualize and animate the model, we make use of *Unity* [4].

finish introduction

add diagram with the different formalisms

2 Railway Formalism

As earlier mentioned, the book *Railway Operation and Control* by *Joern Pachl* [1] was a starting point for this Railway formalism. However, the language used in that book is heavily simplified to make the steps throughout this paper much easier. On top of that, the focus mainly lies on the railway (that consists of different segments) and not much on scheduling, signaling, ... This section will give a brief introduction on this simplified Railway formalism.

At its core, a model consists of multiple segments which can be connected to each other to form a railway. These segments also have a signalling light equipped which will inform an approaching train about its current state: green light means that there's no train represent on the segment and red light means there is. The different types of segments supported by this formalism can be found in table 1.

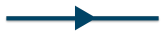
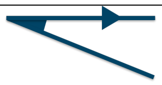
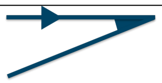
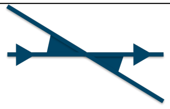
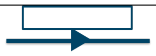
Name:	**Symbol:**	**Description:**
Straight		a basic segment that connects and is connected by one other segment
Turnout		a segment with internally a switch, which can be used to control its outgoing rail (either going straight or make a turn)
Junction		similar to a turnout, but instead of controlling its outgoing rail, it will control the ingoing rail (trains can arrive straight or in a turn)
Crossing		a combination of a turnout and a junction, has two switches available, allowing to control both the incoming and outgoing rails
Station		a segments with a train station next to it

Table 1: All the different types of segments supported by the Railway formalism

3 Abstract and Concrete Syntax

Now that we have defined the initial concepts of our formalism, we can start by building the syntax. This is split in two parts, namely the abstract and concrete syntax. For more information on this topic, I refer to *AToMPM*'s documentation [5].

3.1 Abstract Syntax

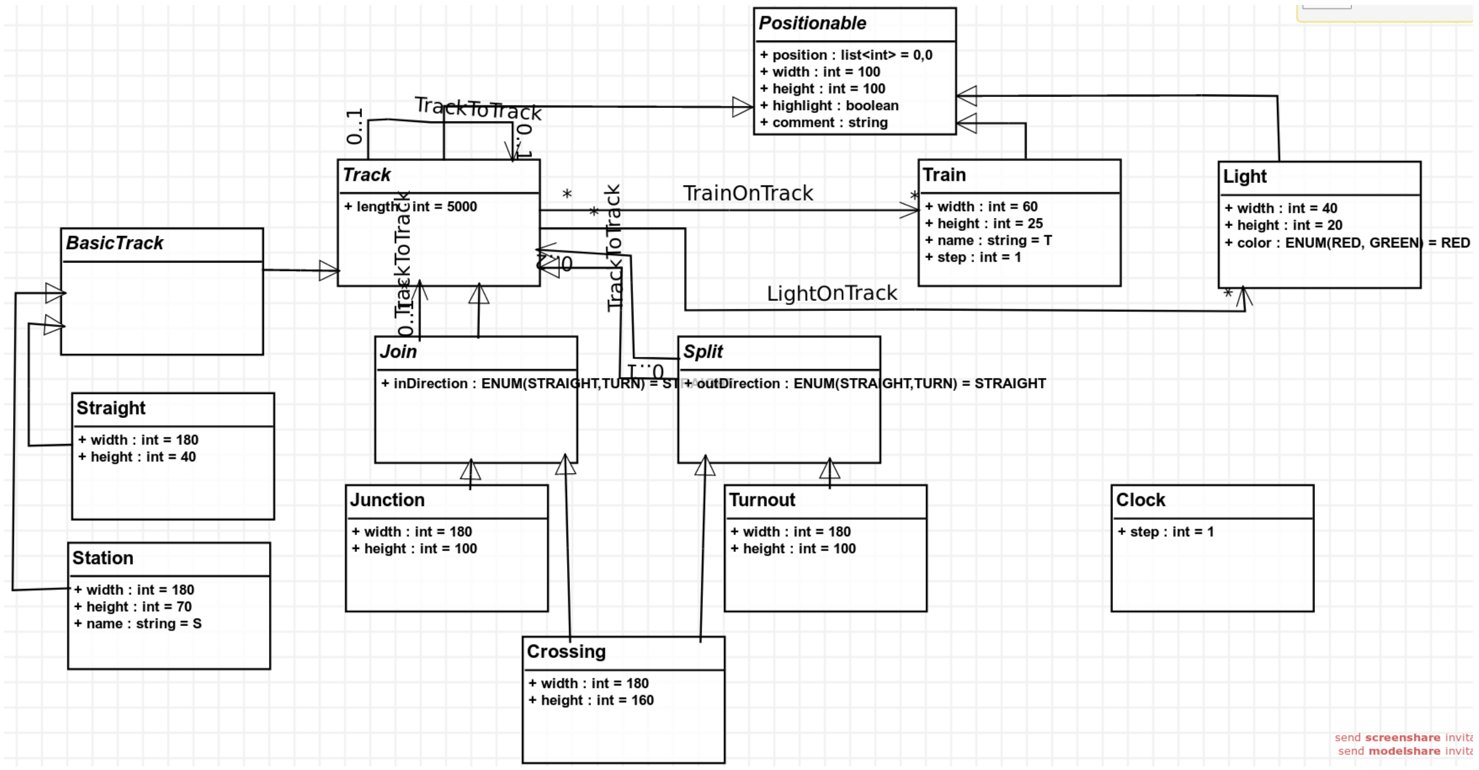


Figure 1: Abstract syntax of the Railway formalism

update figure of abstract syntax

Since we deal with multiple types of segments, an inheritance tree would be suitable for this problem. At the root, we have an abstract class `Track`, from here on, we have three (abstract) subclasses:

- `BasicTrack`: All the most basic tracks that have at most one incoming and outgoing track.
- `Join`: A track where two incoming tracks converge, in other words, a junction. It also has an attribute (`inDirection`) which tells in which the switch is set.
- `Split`: A track that has two outgoing tracks. Similar to `Join`, this also has an attribute (`outDirection`) to indicate the current direction of the outgoing track.

To actually connect the tracks to one another, the TrackToTrack link is used, by default a Track can only connect and be connected by one other track. However, there are of course segments where this is not the case and where we have to override the existing cardinality constraint constraint; for example, Joins can have two incoming one's and Splits two outgoing one's. This link also has an extra attribute direction to store to which "port" it has been connected in case it's connected to a junction; for example STRAIGHT means that it is connected to the STRAIGHT rail of the junction.

Aside from track, we can also create Trains, which is self-explanatory, and Lights that are used for signalling purposes. These objects can of course be linked and placed on tracks, this is managed by the TrainOnTrack and LightOnTrack links. Finally, a Clock is explicitly modelled here as well, this is to keep track of the simulation steps and make the simulation process easier later on.

Note that there are some "visual" attributes present in the model: position, width and height. These attributes are mainly used to automatically connect segments to each other when they are linked. In true nature, this is not the ideal method to store this in the abstract syntax, but this is just a slight work around to store some visual information. As such, there exists an abstract base class Positionable to deal with this.

3.2 Concrete Syntax

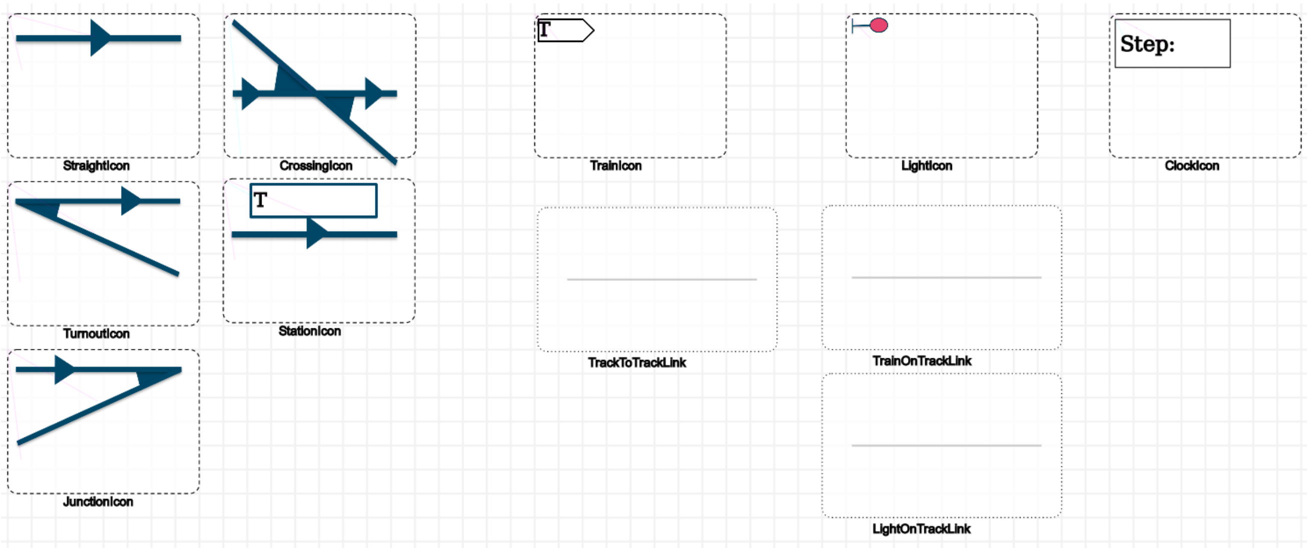


Figure 2: Concrete syntax of the Railway formalism

Having modelled the abstract syntax, we can now define the icons for our formalism (see figure 2). Most of these notation are based on the one used in the book *Railway Operation and Control* by *Joern Pachl* [1]. Besides that, these icons also change depending on the state; for example, a junction will show the current direction of the switch (indicated by the arrow). Either way, most these symbols are pretty straightforward.

4 Operational Semantics

This section will go over the operation semantics of the Railway formalism, i.e. how the whole system behaves and operates. To model this, we make use of transformation rules, again I refer to *AToMPM*'s documentation [5] for more detail.

4.1 Train Schedule Formalism

Before we implement the rules, a second domain specific language is modelled to define train schedules (the path it takes from start to end) as was suggested in the [2] given in the *Model Driven Engineering* course [2]. This way, we can easily operate the switches by looking at the train schedules.

The abstract syntax can be found in figure 3 and the concrete in figure 4. Basically, a schedule consists of one start station and one ending station. In between are zero or more steps that tell in which direction the train should move when it encounters a waypoint (turnout or crossing). One schedule is associated with exactly one train and vice versa.

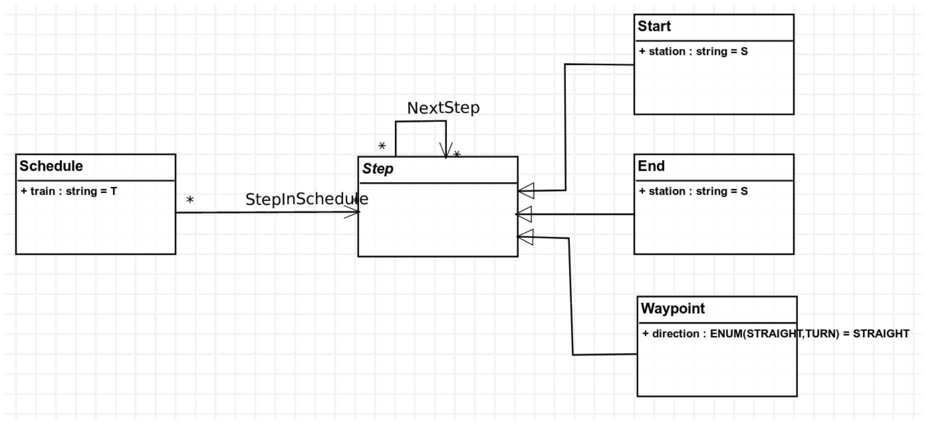


Figure 3: Abstract syntax of the Train Schedule formalism

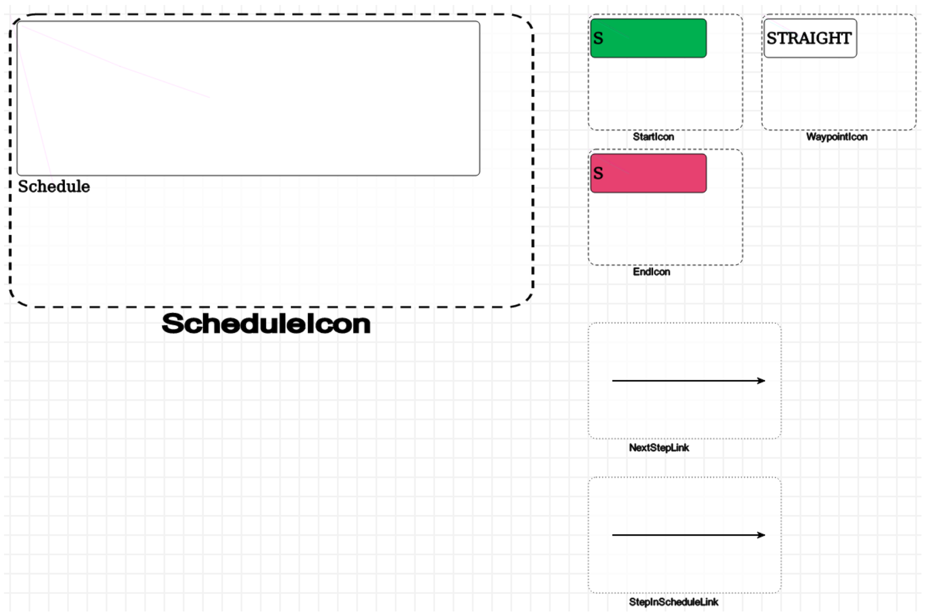


Figure 4: Concrete syntax of the Train Schedule formalism

4.2 Operational Semantics

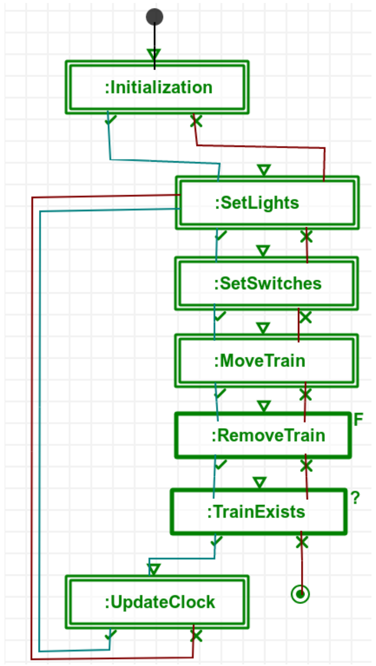


Figure 5: Schedule for the operational semantics

To explain the semantics, I will go over the MoTif schedule (figure 5):

1. Initialization: Adds signalling lights to all tracks if that wasn't the case already and it will place all the trains on their starting station according to their unique train schedule.

2. Set Lights: Set the lights correctly depending on their state; set the light green if there's no train present and red otherwise.

3. Set switches: Set the switches on splits and joins:

    - Joins: if a train wants to enter a junction, the control system will set the direction of the incoming track correctly so that the train can enter. If two trains want to enter, it will "randomly" choose one.
    - Split: to set the switch for splits, we look at the train schedule of the train that is currently on this split. This schedule will tell us in which direction we should move. We then remove that step, indicating it was taken. (see figure 6)

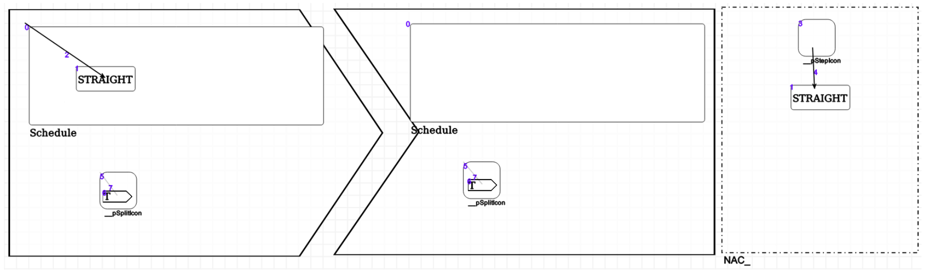


Figure 6: Transformation rule for setting switch on splits

4. Move Train: This step will try to move a train to the next segment. There are however several cases that we have to keep in mind; for example, we can only move if the light on the next section is green. Most of these cases verify if the in/out-direction is set correctly.

5. Remove Train: Whenever a train reaches its end (station), we will remove it from the model, so that potential future train can enter this station as well.

6. Train Exists: A simple query rule to check if there are still trains left on any track. This is the end condition and the transformation will halt if there cannot be a train found.

7. Update Clock: Finally, if the `TrainExists` was successful, we can move to the next simulation cycle: this step will update the clock as well as the step internally of all the trains so that they are synchronized with the clock.

5 Safety Analysis

Given a model written in our railway formalism, one would also want to do some analysis regarding its safety, for example if there is a reachable deadlock state. To do all this, we can define a mapping (using transformation rules) to petri-nets. These nets are highly suitable to these kinds of analyses. In this case specifically, the tool *LoLA* (*a Low Level Petri net Analyzer*) [6] will be used to analyze a given petri-net by building the reachability graph etc.

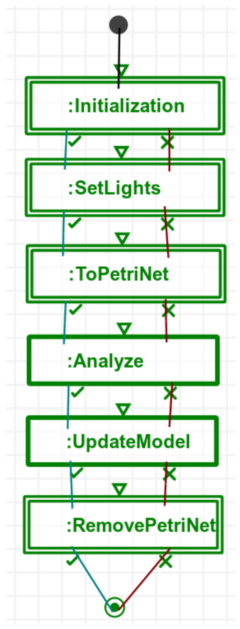


Figure 7: Schedule for the safety analysis

The full MoTif schedule for the safety analysis can be found in figure 5.2. First, we initialize the model so the trains are on the right tracks and the lights are all correctly set. Then we actually do the petri-net mapping and finally we can analyze the intermediate petri-net with a few LoLA calls and update the railway model if that's necessary.

This section is divided in three parts; section 5.1 will define the mapping itself, section 5.2 will discuss some ”safety” properties and how we can define those in *LoLA* and finally, in section 5.3 will go over how we can use the results generated by *LoLA* to update our initial model to show it to the user.

5.1 Mapping

The complete mapping is fairly trivial and can be split in several steps/rules:

- Tracks become places. If one contains a train, the place has a marking value of 1.
- Links between tracks (`TrackToTrack`) become transitions. Note that when a train goes from one track to another, it will also look at the places that correspond to the lights (see next point).
- Lights can be split up in two places: one for the green color and one for the red color. This will mean that at all time the sum of these places will always be equal to 1.

- To make sure the petri-net can go on forever, we add one additional place for each end station in a train schedule and one transition that will take all the tokens in those places and put them back. This means that if all the trains have reach their end station, this transition can fire an infinite amount of times.

Combining all these rules, we now have an analyzable petri-net that represents the Railway formalism.

5.2 Analysis

On this petri-net, we can run several analyzes using *LoLa* to verify if the model satisfies a particular property.

5.2.1 Deadlock

A simple property is deadlock; we want to be able to verify if there is a reachable deadlock state from the start state. If there is any, the user may want to change the initial model to prevent this from happening. In this context, a deadlock means that it is possible for a train to get stuck somewhere in the railway network and never get moving again, which is of course not what we want. This property will also implicitly tell us whether the trains can all reach their end station.

5.2.2 Reachability

Petri-nets are highly suitable for reachability analysis, more specifically, with this we can verify if a specific place is reachable from the the initials state. In other words, tracks that cannot be reached by a train (in the initial setup). For this, we make n LoLA calls ($n =$ number of tracks) where we check each time individually if that track can be reached.

5.2.3 Safeness

Since we're dealing with trains on rails, it's of course desirable to have that trains cannot crash into each other, i.e. two trains on one single track. To verify this, we can check if our whole petri-net is 1-bounded or safe, meaning in every place there cannot be more than one token or on every track there can be at most one train present.

5.2.4 Lights Invariant

In our petri-net mapping, we introduced two places for each signalling light. Implicitly, this means that only one token can be in those two places combined or else the lights are both green and red (or neither of them are on). This property will then check if at all time the sum of two places (of a light) is equal to 1 (invariant).

5.3 Updating model

When analyzing and finally generating the results, we want to be able to show them to the user somehow. The way we do this is also through a small model transformation that reads those results in and updates the railway model accordingly. For example, highlight (and add a few comments about) the path that led to a deadlock state.

One important thing to note is when mapping the results (of the analysis) back to our original model, we need some form of traceability. We can create traces by making use of unique identifiers for the objects in the railway model and then use the same ones in the petri-net model. This way, we can correspond a petri-net element to its original element.

add reference to appendix?

6 Queueing Analysis

Another analysis we can perform is queueing analysis, for example, how long it averagely takes for a train to reach its end station or how long a train has to wait to enter its next segment. To do this, we create another mapping, this time we map the Railway formalism to DEVS. Here we can simulate a given DEVS model and retrieve the necessary information from it. The DEVS library used for this is *PythonPDEVS* [7].

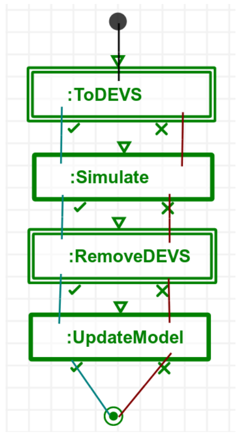


Figure 8: Schedule for the queueing analysis

6.1 Mapping

When mapping the Railway elements to DEVS elements, one must specify the DEVS model as well. For instance, we need a DEVS model for a track that describes its behaviour. One such model has different components, such an internal transition function, states, output function, etc. However, I did not choose to take this path, instead I already had these models implemented with the *PythonPDEVS* library, so I saved myself some time, but ideally one should also model this (in *AToMPM*). I will not go in further details on the semantics of the DEVS models, but more on how I implemented these can be found in the DEVS assignment, part of the *Modelling of Software-Intensive Systems* course [8].

Either way, we still need to create instances of these DEVS models and encapsulate them in a coupled DEVS and of course, we use transformation rules yet again to transform our railway model. Similarly to the petri-net mapping, we can define some basic rules that will take care of this mapping:

- Tracks become DEVS instances. There are four main DEVS models, namely `RailwaySegment`, `Join`, `Split` and `Crossing`. Most of these instances also link to in- and out-ports. The in-ports are `train_in`, `Q_recv` and `Q_rack` and the out-ports `train_out`, `Q_send` and `Q_rack`. What these ports specifically mean, I refer to the DEVS assignemnt [8]. Note that some of these models can have more ports, depending on their type; for example, a split will have a `train_out` and `train_out2` port.
- Links between tracks (`TrackToTrack`) become so-called channels which connect to in- and out-ports.
- Since we are performing a queueing analysis, we most likely want more than a few trains in our network. Hence, there is also a `Generator` DEVS model; this will generate trains following an inter arrival time distribution. This way, a more in depth analysis can be performed and trains actually get queued. The railway element that gets mapped to this generator is the `Station` that is also the start (station) of a train schedule.
- On the other hand, a `Collector` is also represent. This will simply remove a train object and store some statistics. Here, the `Station` that corresponds to an end station of a schedule will get translated to this collector element.

6.2 Analysis

Simulating a complete coupled DEVS model with *PythonPDEVS* will create a detailed trace that specifically tells us at each point in time what has happened, but to retrieve more information, we need to add more logic to the models, but this is quite easily done. On top of that, there some extra attributes in the abstract model (e.g. segment length, train acceleration, ...), so that the user can manually (or by script) tweak the settings and try to optimize a given railway model. In the following small subsections, the few statistics integrated are discussed.

6.2.1 Average Transit Time of a Schedule

In the railway formalism we allowed to the user to define the schedule for a particular train, but it might also be interesting to see if the schedule is actually effective, i.e. getting fast from the start station to the end station and having to wait minimally. Maybe some schedules are interfering with each other, i.e. a train has always to wait for another train (some sort of priority), so knowing the actual transit time can be quite useful. To retrieve this information, we add an extra attribute to a train, namely the train's departure time and when a train arrives at its end station (i.e. a collector), we can subtract the current time with that departure time. At the end of the simulation, we can take the average of all the transit times to get the final result.

Next to this, we can also show the number of trains that have been generated (at their start station) and that have arrived to their destination, this can then also tell us whether all the trains seemingly can drive to their end location.

6.2.2 Throughput and Average Transit Time of a Railway Segment

It might also be interesting to look at an individual segment instead of the complete schedule. In such manner, we can isolate single tracks and look at their own performance. First, we can record the time that a train needs to pass through this segment. This implicitly also says whether a train has to wait a lot while being on this segment. Knowing this, we can try to avoid this by changing the initial schedule and for instance, take another route.

Secondly, we can also store a track's throughput, i.e. the percentage that a train is present on the track. Generally, having a higher throughput is better, but together with the average time to pass this segment, one might expose some conjunction points in the railway network. For example, a certain junction has a high throughput which might mean that trains get clustered before that junction. Additionally, the mount of trains that passed this section is also kept, so we can easily track the trains.

6.3 Updating Model

Like with safety analysis, we update our initial railway model with the gathered results from the DEVS simulation. Again, we make use of id's to have backward traceability, so we can precisely relate the results to a track.

7 Conclusion

References

[1] J. Pachl, *Railway Operation and Control.* VTD Rail Publishing, 1 ed., 2002.

[2] S. Van Mierlo and H. Vangheluwe, "Assignments domain-specific modelling." `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/201516/assignments/`, 2015.

[3] "AToMPM homepage." `https://atompm.github.io/`.

[4] "Unity homepage." `https://unity3d.com/`.

[5] "AToMPM documentation." `https://msdl.uantwerpen.be/documentation/AToMPM//`.

[6] K. Schmidt, *LoLA: a Low Level Petri net Analyzer*, September 2000.

[7] "PythonPDEVS homepage." `http://msdl.cs.mcgill.ca/projects/DEVS/PythonPDEVS`.

[8] S. Van Mierlo and H. Vangheluwe, "Devs modelling and simulation." `http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/assignments/DEVS`, 2018.