Mapping the Railway formalism onto different domains

Zhong Xi Lu

Department of Mathematics
and Computer Science
University of Antwerp, Belgium
zhong-xi.lu@student.uantwerpen.be

July 2019

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 at 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], here we also allow the user to modify the model. In the appendix (section 9), a few examples are worked out with a few analyses for illustration.

In figure 1, a small overview is given with the different formalisms and mappings in between them.

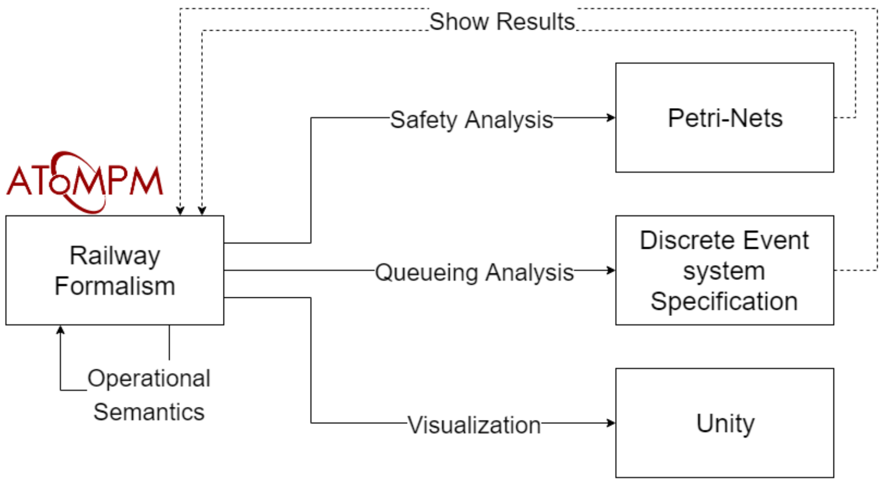


Figure 1: Overview of the different formalisms and mappings

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 project 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 network. 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.

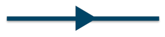
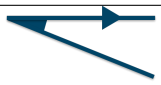
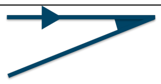
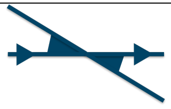
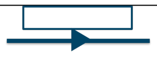
Name:	**Symbol:**	**Description:**
Straight		a basic segment that connects and is connected by one other segment
Turnout		a segment with internally a switch, which can be used to control its outgoing rail (either going straight or make a turn)
Junction		similar to a turnout, but instead of controlling 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

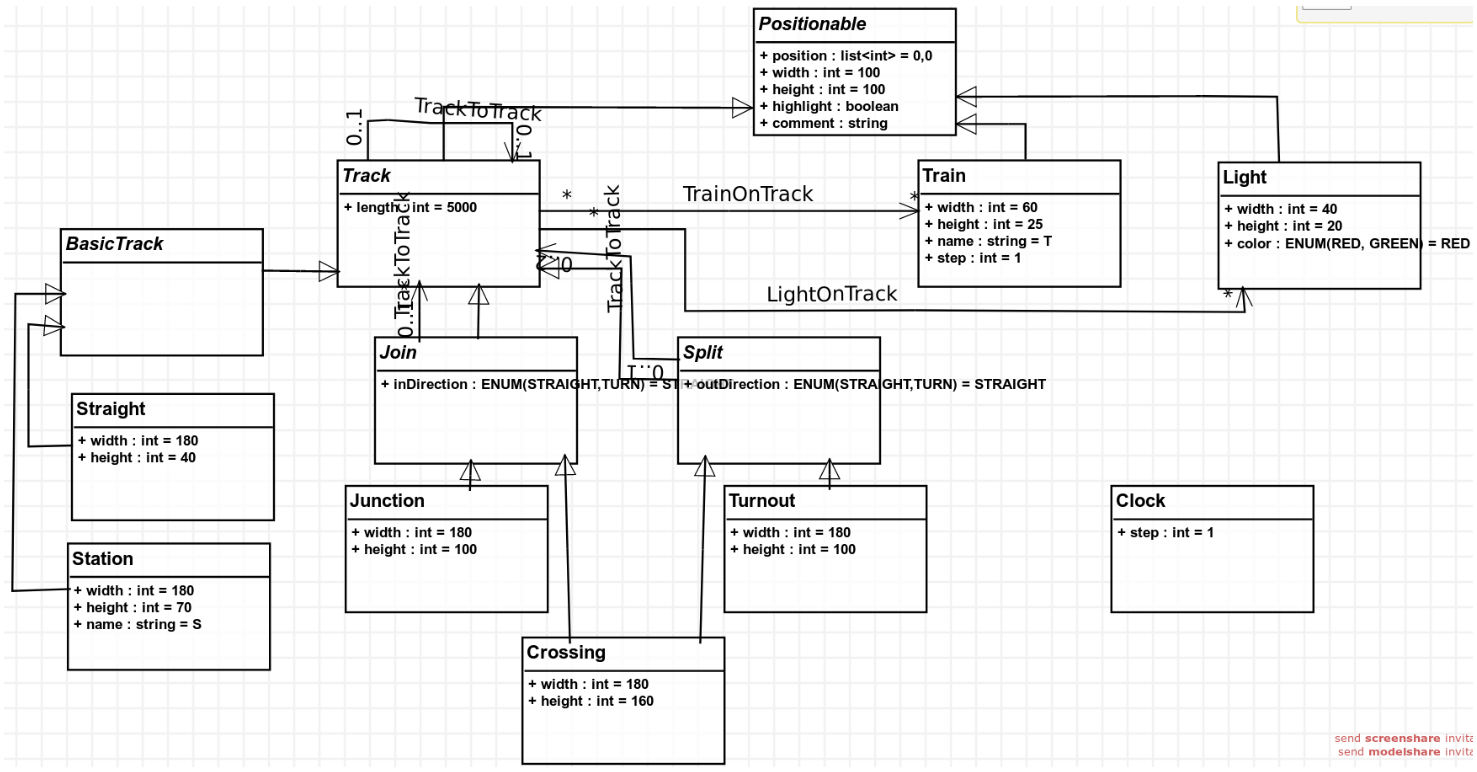


Figure 2: Abstract syntax of the Railway formalism

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, `Join`s can have two incoming one's and `Split`s 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 `Train`s, which is self-explanatory, and `Light`s 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

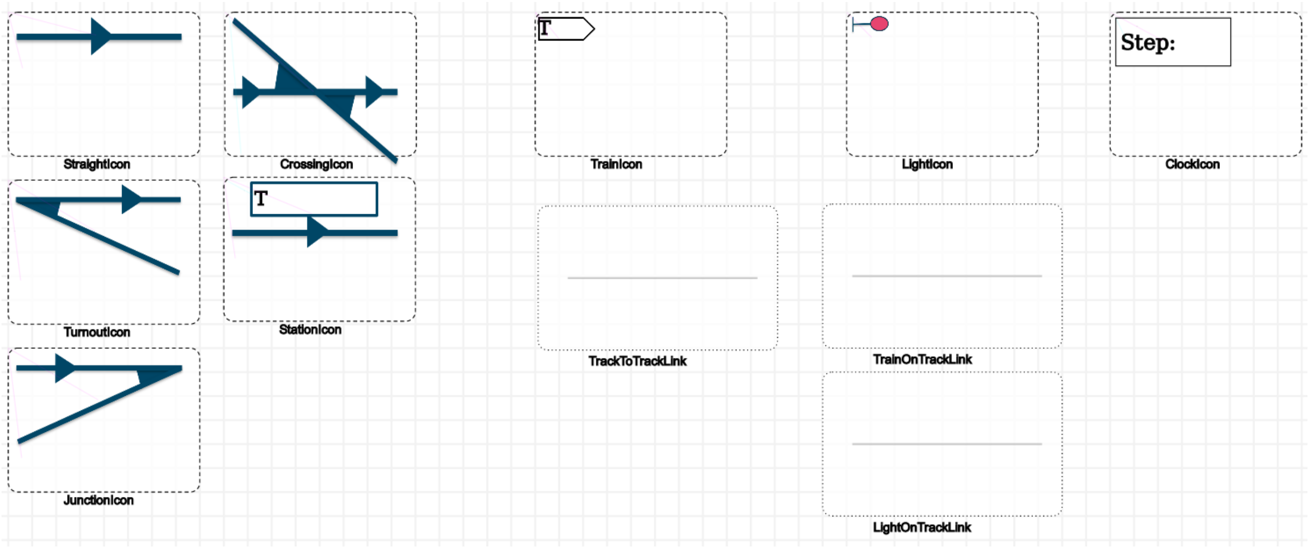


Figure 3: Concrete syntax of the Railway formalism

Having modelled the abstract syntax, we can now define the icons for our formalism (see figure 3). 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 station) 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 4 and the concrete in figure 5. 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.

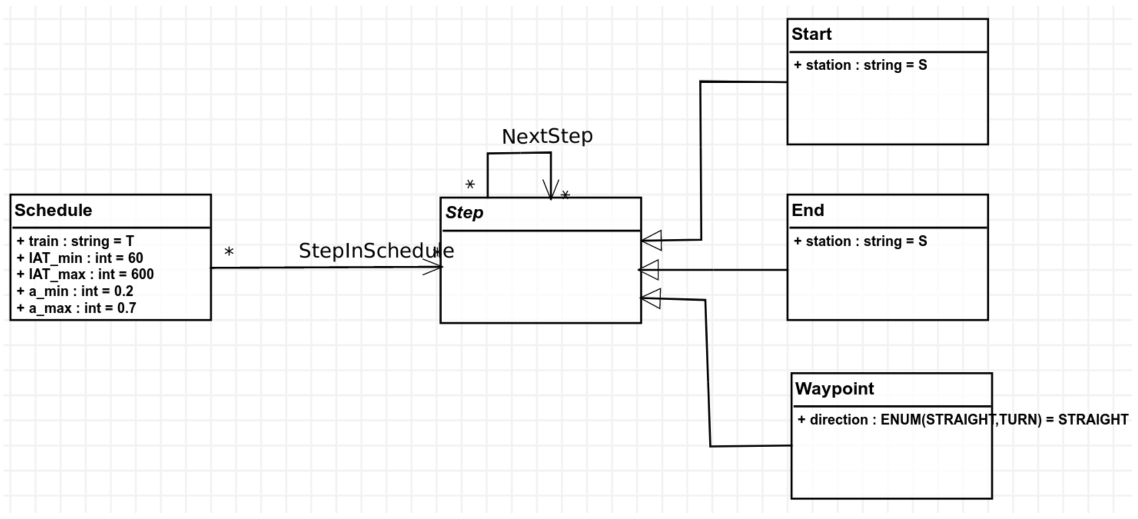


Figure 4: Abstract syntax of the Train Schedule formalism

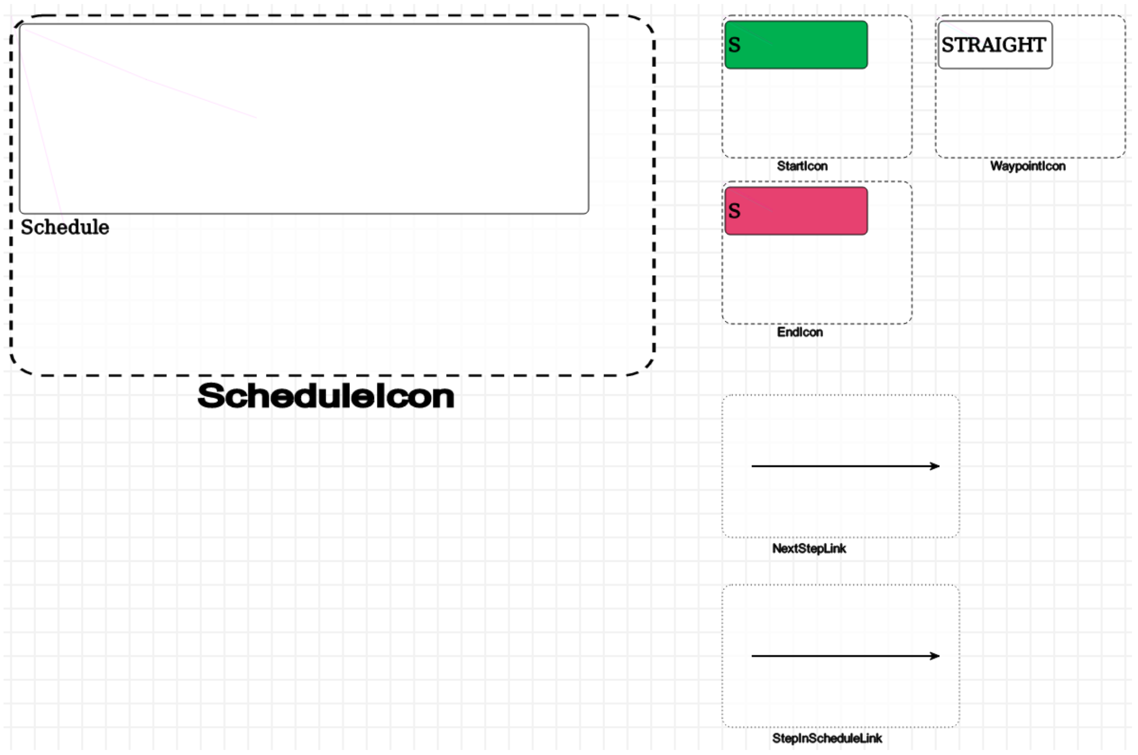


Figure 5: Concrete syntax of the Train Schedule formalism

4.2 Operational Semantics

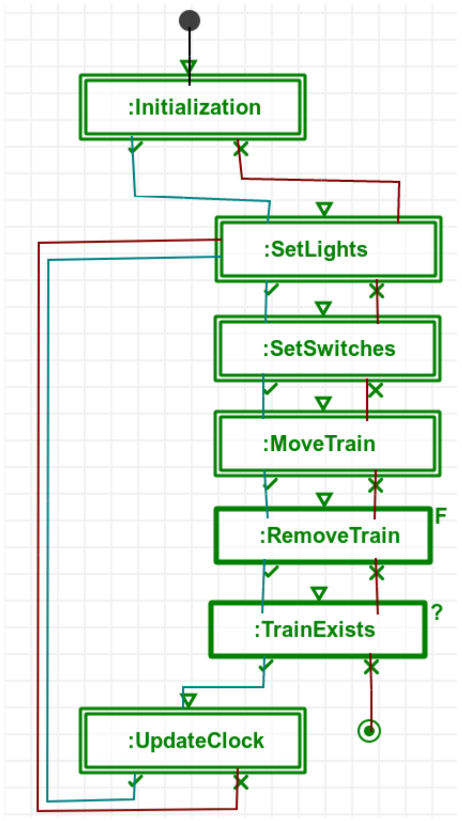


Figure 6: Schedule for the operational semantics

To explain the semantics, I will go over the MoTif schedule (figure 6):

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 on the track the light is on 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 7)

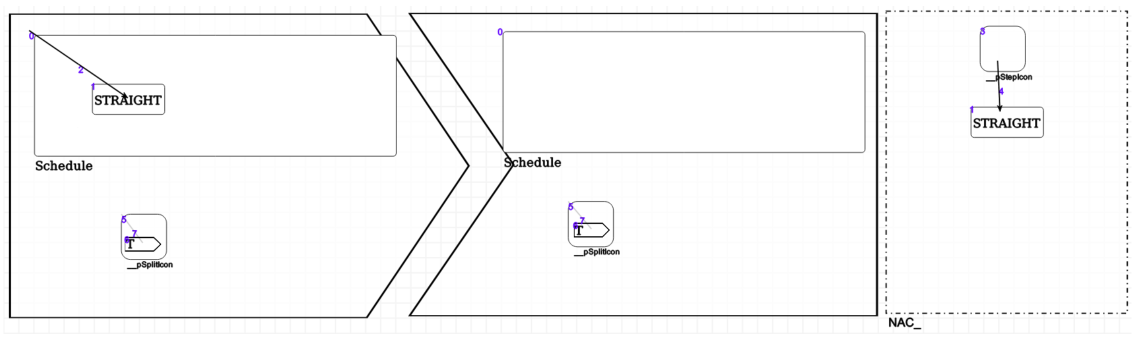


Figure 7: Transformation rule for setting switch on splits

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 also verify if the in/out-direction is set correctly, e.g. a train cannot move to a junction when the switch is not 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 serves as 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 kind 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. The reason *LoLA* is used, is because, as the name already suggests, it is a low level analyzer, that way, we can easily ”integrate” it in our transformations just by running a few commands. Another advantage is that it is possible to specify custom properties through *Computation Tree Logic* (CTL) formulas, this will come in handy when we define the safety properties for our railway network.

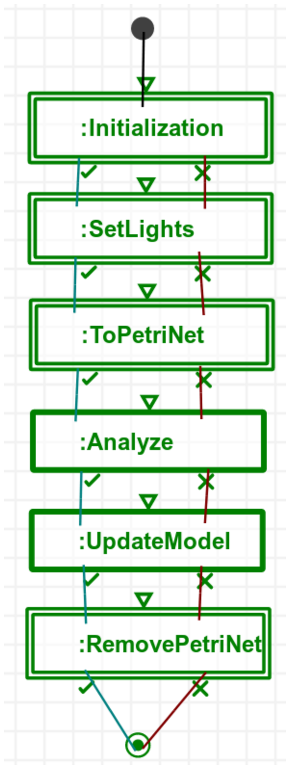


Figure 8: Schedule for the safety analysis

The full MoTif schedule for the safety analysis can be found in figure 8. First, we initialize the model so the trains are on the right tracks and the lights are all set correctly. Then we actually do the petri-net mapping and finally we can analyze the intermediate petri-net with a few LoLA calls. Note that during this step an intermediate file is created to represent the petri-net, this will then consequently be used by *LoLA* to perform the analyses. Lastly, we update the railway model according to the results.

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, section 5.3 will go over how we communicate from `AToMPM` to `LoLA` and vice versa, i.e. what their interfaces are.

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 the tokens of these places will be equal to 1.
- To make sure the petri-net can run 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 intermediate petri-net, we can run several analyzes using *LoLA* to verify if the model satisfies a particular property. As a small side note, the way `LoLA` generally does an analysis is by building a reachability graph for a given petri-net and then make a search depending on the 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.

This property is already predefined in *LoLA* ("`EF DEADLOCK`"), this simply means if there exists (`E`) a path, starting from the initial marking, where it eventually (`F`) reaches a deadlock state (no transition is enabled). This condition holds true if a possible deadlock state can be reached.

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 initial 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 from the initial marking.

The CTL formula to verify this is "`EF T > 0`" (with `T` the name of the track/place), this translates to whether it's possible to reach a state where the marking value of this place is strictly greater than 0.

5.2.3 Safenty

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.

As a CTL formula, it would be written as "`AG T` $\leq$ `1`" (with `T` the name of the track/place), here we check if on all paths (`A`) and globally (`G`), i.e. every step on this path, the marking of this place is never greater than 1. Again, we run this property check for every track in our railway network.

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 the tokens of these two places (of a light) is equal to 1 (invariant).

This invariant we can model as "`AG (G = 1 OR R = 1)`" (where `G` and `R` stand for the places that respectively correspond to the green and red light of a track). Similarly to the previous property, this will make sure that on all paths (`A`) and globally (`G`), either there is a token in the "green" place or one in the "red" place. Of course, here we do this same call for each track, since we want to verify this invariant for every light.

5.2.5 Custom Properties

It's all well and good having these safety properties, but it doesn't allow much for extensibility in the sense that we can define our own properties if that's needed. Hence, we can again define our own little language that has the ability to model these properties. However, this language is merged into the railway formalism, so we can reference particular tracks and individually check certain tracks.

This property formalism is heavily based on the CTL language that we pass onto LoLA, so in the end a user has to know a little bit about this to define a property.

5.3 Interfaces

As slightly mentioned, to be able to call `LoLA`, we first generate a petri-net file suitable for analysis. Then we can simply call `LoLA` via a command and make the necessary property checks. This is all done in one transformation rule (the `Analyze` step in figure 8).

When analyzing and generating the results, we want to be able to show them to the user somehow. The way this is achieved is also through a small model transformation that reads the (*LoLA* generated) 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.

Another way of showing these results, is to show a replay of a counterexample if one exists and is given by LoLA. This replay is of course done through almost the same transformation rules as the ones in the operational semantics. However, this time we don't take the train schedules into account, instead we base the semantics on the generated trace by LoLA. This trace tells us precisely which actions have been taken (which transitions have been fired), so all that we have to do, is translate and replay these actions on our railway model.

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.

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 Discrete Event System Specifications (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].

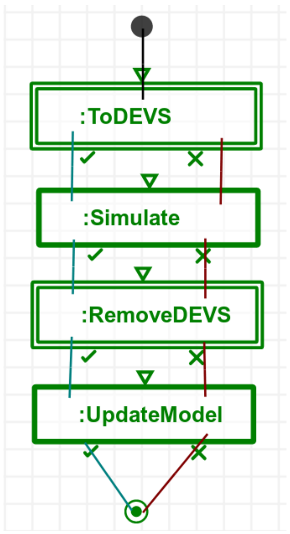


Figure 9: Schedule for the queueing analysis

6.1 DEVS model

Before mapping the Railway elements to DEVS elements, one must first specify the DEVS model. 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 into 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].

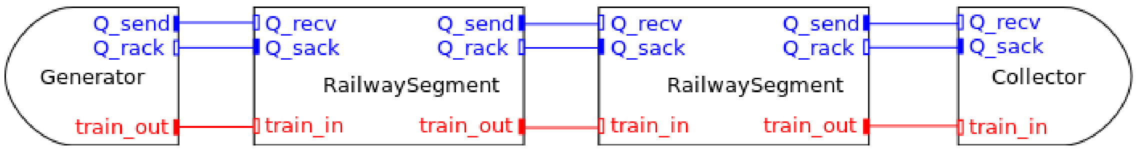


Figure 10: Example of coupled DEVS model for the Railway formalism [8]

Essentially there are a couple of atomic DEVS, which we can link together to form a coupled DEVS model. Most of these models actually somewhat correspond to the abstract syntax (figure 2); there are four main atomic 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.

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.

A small example of how the atomic models connect to each other can be found in figure 10. At the very first step of the simulation, i.e. at time 0, all the atomic models will enter their initial state depending on their initial condition. Together with that, it will schedule the next internal transition based on the time advance function which will tell how long it will take to reach the next transition depending on the current state. For example, a generator might have the first transition at time 10 where it will generate a new train. After all these transitions have been scheduled, the simulator will run until the next transition is enabled, then it will take this transition and execute the related action code (e.g. calculate a train's velocity), send some output if necessary and finally, it will schedule the next transition.

Next to internal transition, external transitions can also occur, this is determined by another model; for example, when a new train arrives at a segment, it will receive this and handle this event. During this, it will of course go to a certain state and schedule a new transition. This continues until the end condition is met (e.g. termination time).

6.2 Mapping

For the mapping, we need to create instances of these aforementioned DEVS models and encapsulate them in a coupled DEVS and of course, we use transformation rules yet again to transform our railway model. Since we based those atomic models on our abstract syntax, the transformation can easily be done by mapping tracks to their corresponding atomic models and the links between the tracks (`TrackToTrack`) become so-called channels which connect to in- and out-ports.

6.3 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 are 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.

6.3.1 Average Transit Time of a Schedule

In the railway formalism we allowed 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.

6.3.2 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 amount of trains that passed this section is also kept, so we can easily track the trains.

6.4 Interfaces

Like with the safety analysis, the `Analyze` step here in the MoTif schedule (figure 9) is responsible for creating the actual DEVS file (which contains the complete coupled DEVS model) after the mapping and for calling *PythonPDEVS* via a command, which will then simulate the DEVS model.

On the other hand, after the simulation has ended, the results are automatically written to a file, this can then be read by the `UpdateModel` transformation rule to display the useful information. Again, we make use of id's to have backward traceability, so we can precisely relate the results to a track.

7 Visualization

This whole time we stayed in *AToMPM*'s environment, both for creating the railway model and showing the results of the analyses. In the back-end, we did use some other tools though, such as *LoLA* and *PythonPDEVS*, but in the end we went back to the railway formalism. This section will go over how we can visualize a railway model in a more user-friendly way by creating a 3D world in *Unity*.

7.1 Model Generation

The first step is generating all the 3D objects, unfortunately we cannot directly use transformation rules to transform our model to one written for *Unity*. Therefor, an intermediate step is taken; I created a small (`xml`) file that contains the essential information of the railway network, such as how the railway is structured or what parameter values are associated with a track. This file can then be used to instantiate the objects in *Unity*.

7.2 Simulation

Not only do we want to visualize a model, we also want to see the interactions in the model. To do this, we simulate the model (in *AToMPM*) by mapping it to DEVS and the let the *PythonPDEVS* simulator do the work. During this simulation, it will create a trace file, telling precisely at each point in time the dynamics of the system. For this, however, a custom tracer was written to incorporate additional information about the models.

After the creation of a trace, we can start our "gameloop" in `Unity` (`Update()`); first, we generate the world by loading the `xml` file that represented the railway network. Then the `Update` function is called each frame (note that we can also use `FixedUpdate` to get a consistent update). During this frame, we check the trace if there are any events that occurred in the past, i.e. the timestamp of the event is less or equal than the time since the startup. If this is the case, we simulate all these past events, otherwise we just wait for the next frame and repeat the process. The psuedocode for this gameloop can be found in listing 1. In this manner, we respect the timings of the simulation. Note that since the whole simulation was already ran, we can easily allow the user to set a speed up factor for a faster simulation time.

```
Update():
    while next event exists:
        if timestamp of next event <= time since startup:
            simulate next event
        else:
            break
```

Listing 1: Pseudocode Gameloop

Finally, there is also support for live simulation, i.e. visualizing the railway system while it's running. This is done by simulating the DEVS model in real time and each time a transition is taken or an event is occurring, the simulator (or rather the tracer) will then pass a message to *Unity* through sockets, which will then parse it and display the results of it. It's basically the same as before, but instead of reading the event from a tracefile, we now get these messages live from the simulator. This also means that we don't really have a gameloop anymore or have to worry much about the timestamps since all that is dealt with in the original simulator in *PythonPDEVS*. Note that here, we also have to load the world prior to the actual simulation, this means that an `xml` must already exist before we can do the live simulation. A small overview of all this is given in figure 11.

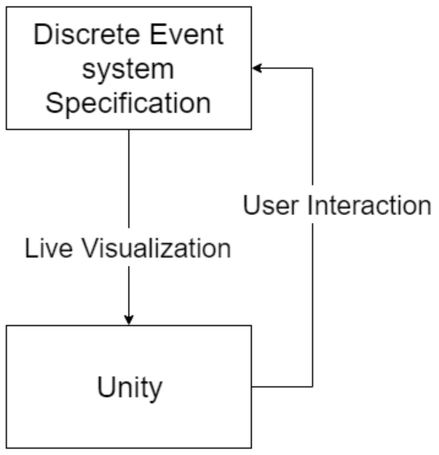


Figure 11: Overview of the live simulation and visualization

7.3 User Interaction

Since we support live simulation, it might also be very useful to allow user interaction. By doing so, a user can easily tweak a few parameters in *Unity* and immediately see the effect of it. The way this is achieved is by sending a message back to the DEVS simulator from *Unity* (again, through sockets). This message will contain information of which model to update (we can precisely specify the model with the id), based on this the simulator will relay this message to the appropriate model. This is then interpreted as an external (or user) event in the DEVS model, meaning that it will handle this event, go to a (new) state and schedule the next transition. The received message also stores the parameters with their (new) values, so the model can update its parameters while the simulation is still running.

8 Future Work

This research project serves as a proof of concept, we created a few mappings starting from a railway formalism. However during this work, I took many shortcuts that could be improved in the future; these are some some of things that may need some work:

- The base railway formalism can be extended; for example, to allow the railway formalism to construct tracks that allow trains to transit in two directions.
- Currently the DEVS model is implemented in *PythonPDEVS*, but this could of course also be modelled explicitly and replace the current one. On top of that, the current simulator is missing some features as well, such as explicit switches on junctions or allowing to bypass a station.
- It's possible to add additional properties for both the safety and queueing analysis.
- The visualization in *Unity* is at this point a little bit sloppy in the sense that the movement of trains is not really accurately being displayed (trains are not pixel-perfect on tracks). This could be improved as well as general user-friendliness.

Apart from improving existing translations or models, it is also possible to add more mappings. For example, one could maybe model the user interaction explicitly in statecharts.

9 Conclusion

The whole idea behind this project was to model as much as possible and use the many advantages modelling offers without coding too much. In this particular case, we used a railway system as our base formalism and tried to map it onto many different domains. We successfully defined several mappings with the goal that we can solve the ”problem” there, i.e. some sort of analysis and then link the results back. As earlier mentioned in the introduction, this way we tackled down the accidental complexity that programming brings with it and used modelling instead.

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.

Appendix: Examples

Operational Semantics

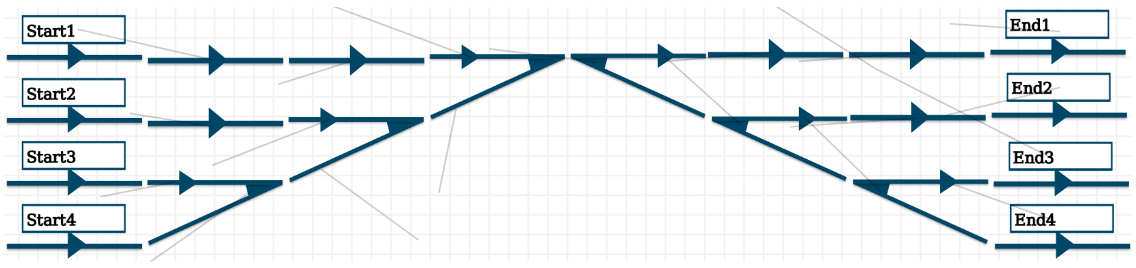


Figure 12: Railway model for the operational semantics

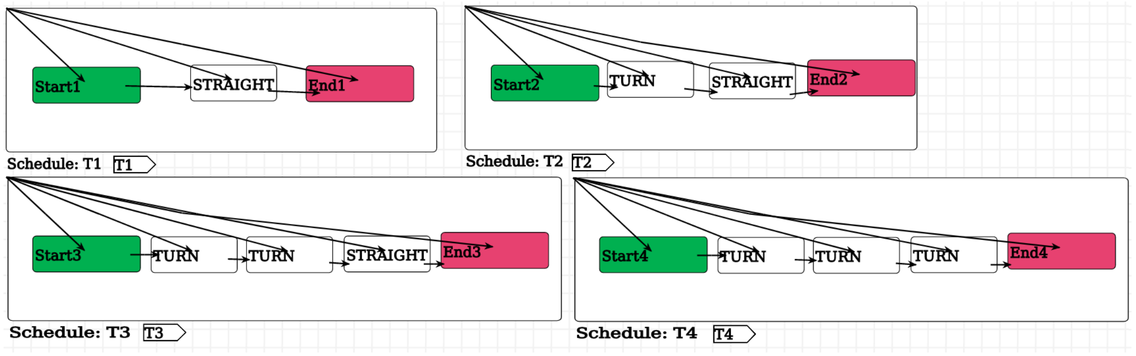


Figure 13: Train schedule models for the railway model in figure 12

The example model we will be working on for the operational semantics is shown in figure 12. The small network consists of four start and end stations that are connected to some sort of ladder. For each start station we also associate exact one train schedule, which can be found in figure 13. For example, train `T1` has to continuously go straight, while `T4` always has to take a turn on a turnout.

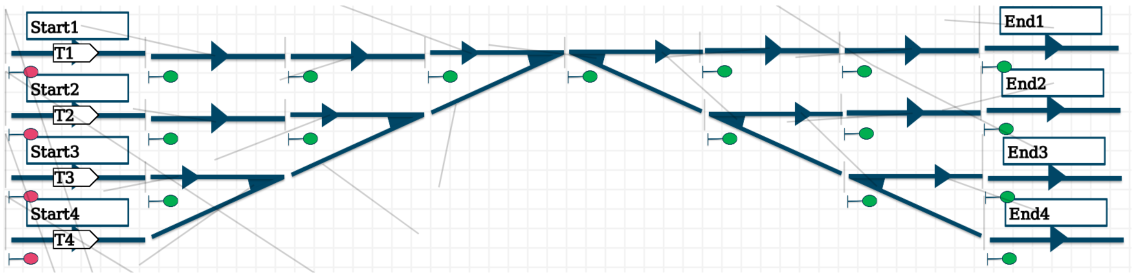


Figure 14: Model after initialization

After the initialization (figure 14), all the trains are placed on their start station and all the signalling lights are added and set correctly.

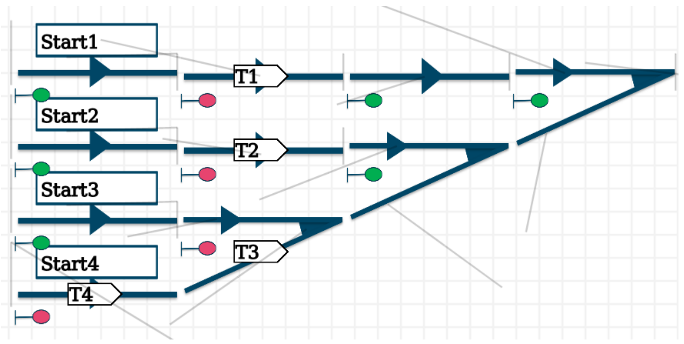


Figure 15: Model after first iteration

Now, the real simulation loop can start. The first iteration (figure 15) starts by setting the switches, in this case, only one switch (most bottom junction) will be set. There are however two trains (T3 and T4) who want to enter this junction, so it will randomly select one (T3 here). After this step, the trains can try to move to the next segment, all the trains can move, except T4 since the switch of the junction is not set correctly.

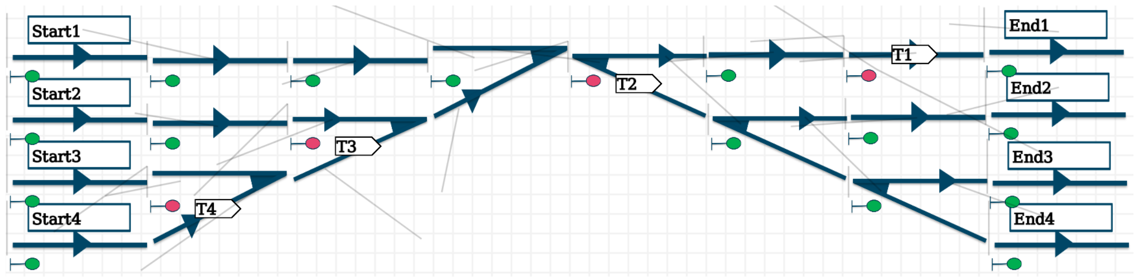


Figure 16: Model after several iterations

If we let the simulation run several iterations, we come to the following model in figure 16. Interestingly enough, T1 is almost already at its destination, while T4 just left its start station. This is due to the fact that T1 only has to enter one junction. Finally, if we let the simulation run until the end, all the trains will eventually reach their end location. However, I noticed that the order in which the trains arrive can differ in each simulation; for example T2 might arrive before T1, but more on this in the Queueing Analysis section.

Safety Analysis

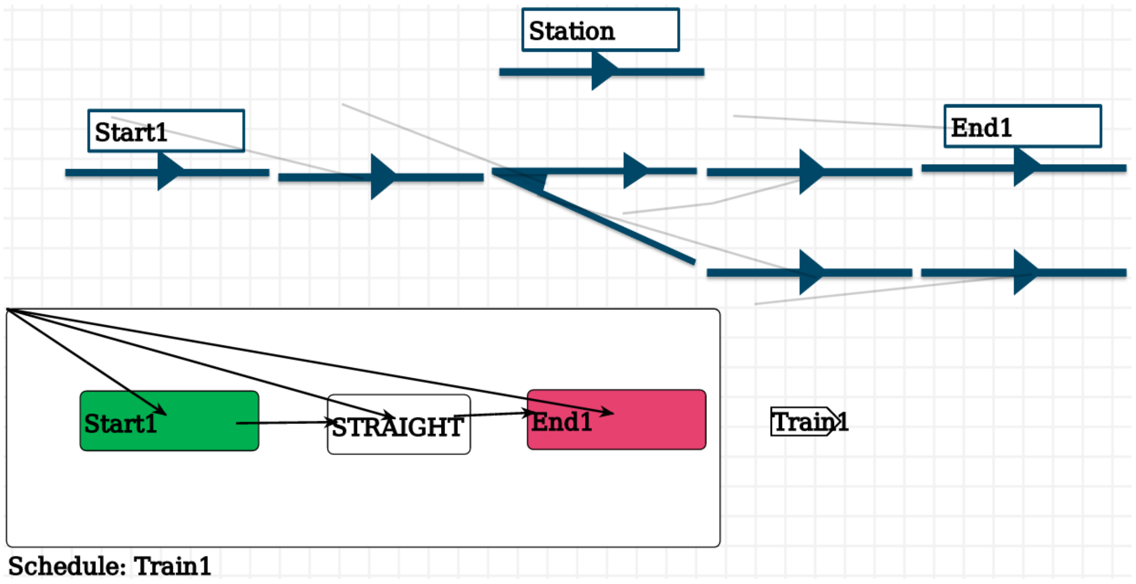


Figure 17: Railway and Train Schedule Model for the safety analysis

The model that we will be analyzing is pictured in figure 17. It contains only one train of which its schedule is simply going straight until it reaches its end station.

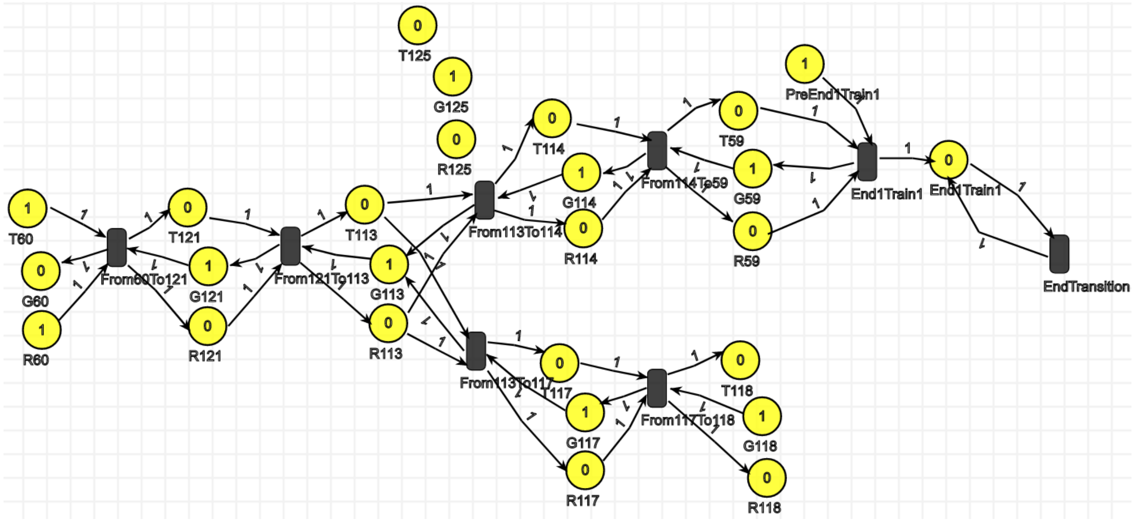


Figure 18: Intermediate petri-net model

When running the model transformations, an intermediate petri-net model is created, this can be viewed in figure 18. All of the tracks have three places, one for the actual track and two for the light. For each end station (of a schedule) we also associate one place (here it is `End1Train1`) where the train resides after it has reached its end station. Note that the `PreEnd1Train1` place holds one token, so that only one train finish in this station. Finally, if the end condition is met (all trains are in their end station), the `EndTransition` will then infinitely be enabled and fired.

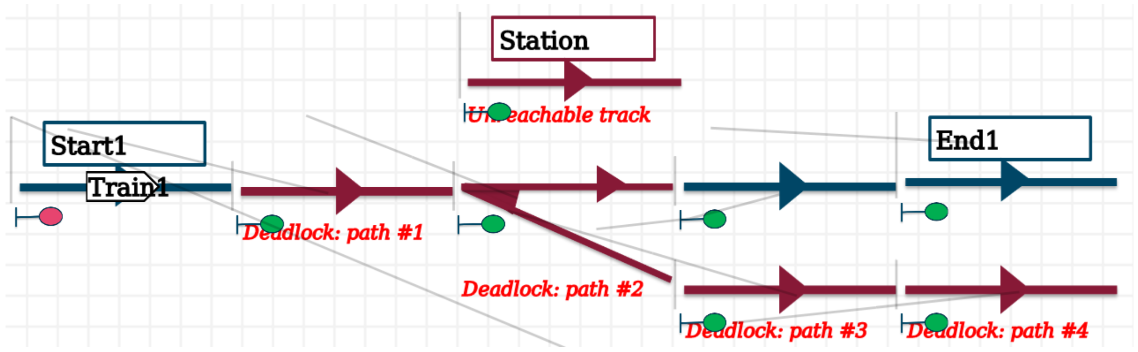


Figure 19: Safety analysis results

From this intermediate petri-net, we can run our analysis and import the results back onto our railway model. The effect of this can be seen in figure 19. First, there is an unreachable station `Station` and secondly, it might be possible to end up in a deadlock situation if we follow the indicated path on the model, i.e. instead of going straight on the turnout, the train takes a turn and goes into a dead path. Note that in practise this should not be possible if the control system is functioning correctly, i.e. setting the switch on the turnout according to a train's schedule.

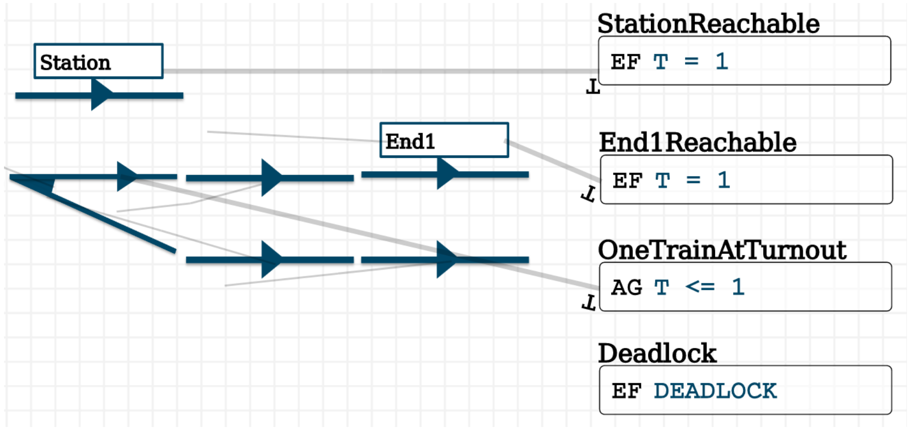


Figure 20: Some custom defined properties

In figure 20 a couple custom define properties are modelled. As earlier mentioned, this is in the form of CTL. Also note that in the formulas it is possible to reference a particular track, if we do this, we do have to create a link between the referenced name (`T` is used here) and the actual track in the railway model.

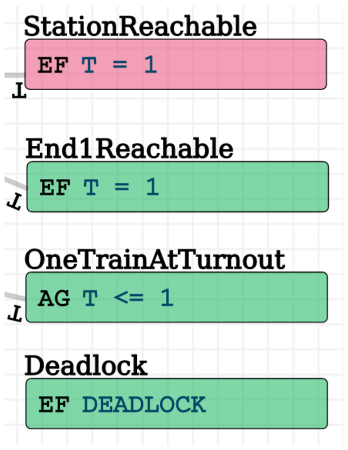

StationReachable
```
EF T = 1
```
End1Reachable
```
EF T = 1
```
OneTrainAtTurnout
```
AG T <= 1
```
Deadlock
```
EF DEADLOCK
```

Figure 21: Results for the custom defined properties

This time, we can call the custom analysis, which will take this CTL formulas and directly pass it to LoLA for analysis. After each call, we can easily give feedback on whether the property was satisfied or not (green is satisfied and red isn't). For example, the `EF DEADLOCK` is satisfied, meaning that it found a path to a deadlock state.

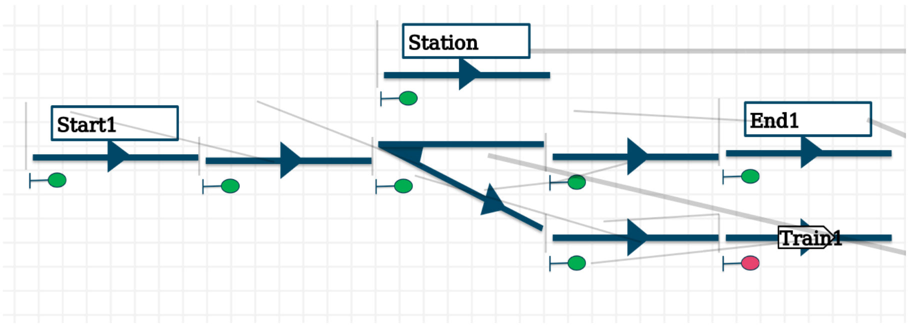


Figure 22: Results for the replay of the deadlock property

Lastly, we can ask for a replay to precisely give us the snapshots during the run. In figure 22, the last snapshot is given and as expected from the default analysis (figure 19), we concluded with the same path that led to a deadlock state.

Queueing Analysis

For the queueing analysis, we start off with the same model we used for the operational semantics (figure 12 and 13). Note that the railway segments are all five kilometers long, the inter arrival time (IAT) for all the generators are $[5, 10]$ seconds, the train's acceleration is $[0.2, 0.7]m/s^2$ and the simulation runtime is 100.000 seconds (or roughly 28 hours). We make the inter IAT short enough so that trains can get queued, so we can see how the train system will function under heavy load. As earlier mentioned, we could already tell that `T1` reached its end station way before the others, but to go deeper on this, we can make use of the queueing analysis.

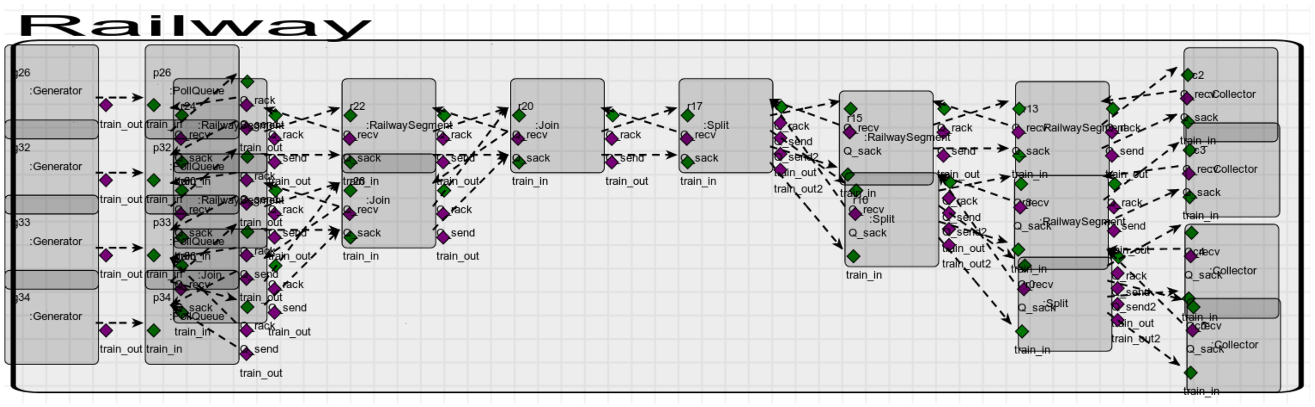


Figure 23: Intermediate coupled DEVS model

Similarly to the safety analysis, we create an intermediate coupled DEVS model here (figure 23). This we can then use to simulate the whole railway system and retrieve the necessary statistics.

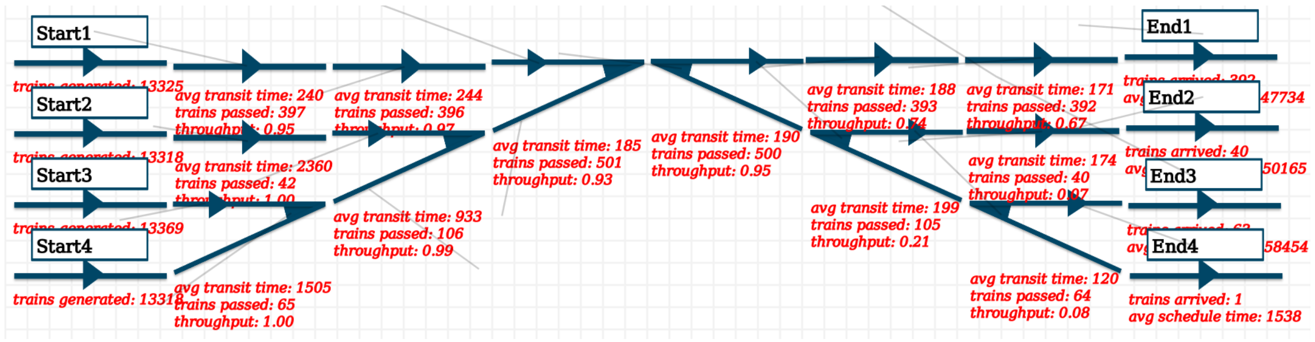


Figure 24: Queueing analysis results

Looking at the results, we immediately see that the middle junction and turnout have a high throughput, meaning a lot of traffic (in fact, all traffic) goes through this point. Another shocking fact is that only one train from schedule 4 (`Start4 - End4`) has arrived at their end station and that most of the trains that finished are from schedule 1. Furthermore, we can conclude that the tracks starting from `Start1` have a high throughput and a low transit time which is desirable. On the other hand, the tracks starting from `Start4` also have a high throughput (approx. 100% even), but they have an extremely long transit time since the trains probably have to wait for the other ones. Judging from this, there is some kind of indirect priority system (probably due to the pseudo-randomness), i.e. trains from schedule 1 have the most priority.