optimising railways

SMA and partners Ltd

Gubelstrasse 28
8050 Zurich
Switzerland
Phone +41 44 317 50 60
info@sma-partner.com
www.sma-partner.com

An Architectural Overview of the Robustness Analysis: Train Simulation via Algorithm Platform

Abstract

In this post we want to outline the high-level architecture of Viriato's macroscopic train simulator, which will be used for Viriato's new robustness analysis module currently being implemented by our development team. Roughly speaking robustness can be seen as the ability of a timetable to recover from delays caused by unforeseen events, including the potential rescheduling efforts. For a more precise definition we refer the interested reader to our annual report 2021 (pp. 18-20). In a Viriato robustness analysis we investigate the effects on the planned timetable, and how long it takes to return to normal train operation, after a disturbance has occurred on the network. The core of our robustness tool is a macroscopic train simulator. For a background about the relationship between train simulation and robustness we recommend our previous posts 04.11.2020 Robustness vs. Train Simulation and 11.06.2021 - Robustness Analysis through the Algorithm Platform.

Problem Context

For each case where there are deviations from the planned timetable, the infrastructure manager has to implement a dispatching strategy to solve the conflicts which arise. This strategy differs between different organisations because it is possible to prioritise according to a wide range of properties, such as the train type (e.g. long distance vs. short distance) or the train's route and the available infrastructure capacity. We divide the activities that are necessary for a robustness analysis between two main components: the Simulator for the general procedure and the Dispatcher for the implementation-specific strategy. Furthermore, we use an event-based representation of the timetable which enables the simulator to propose an arrival or departure event which the dispatcher can accept or refuse. Both the simulator and the dispatcher are controlled and coordinated by Viriato's Algorithm Platform (see Figure 1 for an overview).

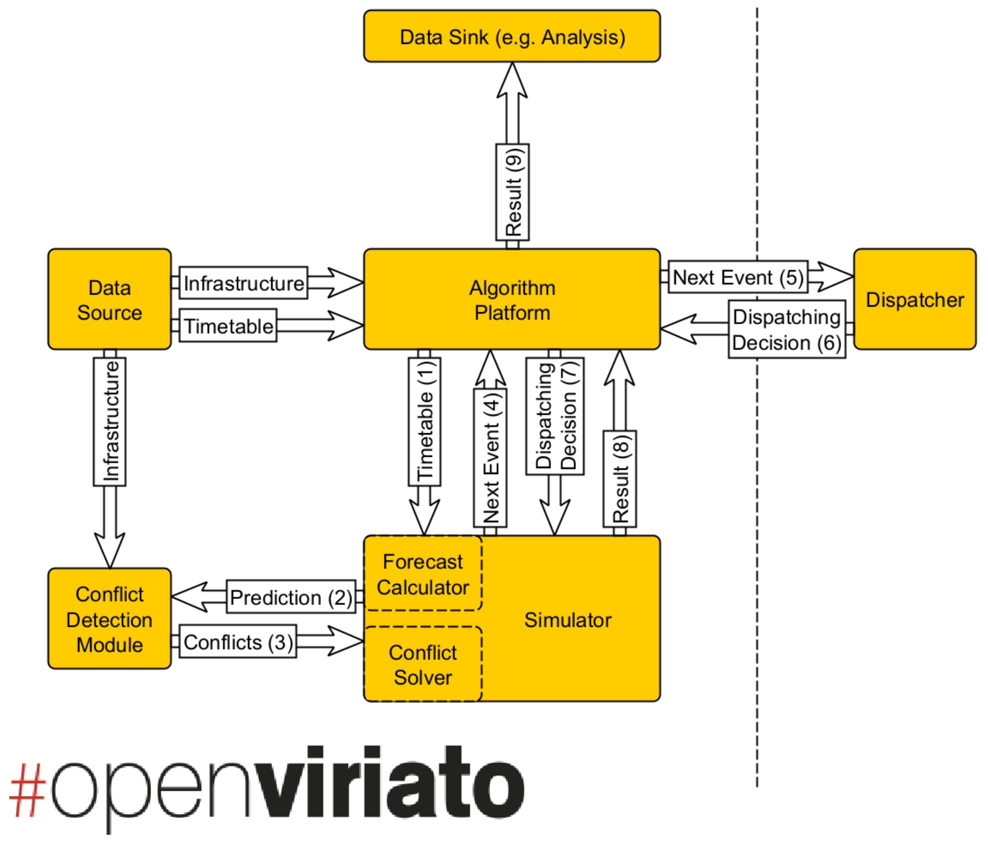


Figure 1 Overview of the relevant components and the data flow (in the direction of the arrows) between them. Note that the dashed line separates the two actors in the architecture.

Conflict Resolution during the Simulation

When a robustness analysis is started the timetable is converted into the event-based model and sent to the simulator (Step 1 in Figure 1). Typically, this is the timetable as planned in Viriato but it is also possible that another algorithm provides a timetable to the Algorithm Platform (or modifies the Viriato one) before it is passed to the simulator.

The main task for the simulator is to find the next realisable event (i.e. the next event that can be performed). For this, the future traffic situation needs to be predicted by taking into account the current position of trains on the railway network, their current delays and the possible reduction of these delays through the consumption of their planned time reserves. The next event in chronological order is chosen and checked for conflicts that would hinder it from being actually realised (Steps 2 and 3). If there is a conflict then the event is postponed as long as necessary to solve the conflict and the search starts over again. This means that the simulator manages compliance with the abstracted interlocking rules. For example, this models the real-world situation where a train has to wait to enter a section because it is already blocked by another train.

To accomplish this task, the simulator uses Viriato's existing conflict detection model. Only conflicts between trains are regarded by the simulator, and conflicts between a train and the infrastructure (e.g. driving in the wrong direction) are not taken into account. We assume that there are no conflicts of this type in the original timetable as it is impossible to cause or resolve them simply by postponing a train. For conflicts between two trains we distinguish between the initiating and the affected train, where the initiating train does something that hinders the affected train. E.g. consider a single-track section where two trains are to travel in opposing directions. The initiating train will be able to depart, blocking the section track and preventing the affected train from entering the section. That means that the conflict is not relevant for the initiating train but will have an impact on the affected train until the line section is clear. The affected train will be delayed by the simulator to provide a solution to this conflict, leading to an updated forecast time for its next event. This procedure is continued iteratively until a realisable event is found, which is then sent to the dispatcher for processing via the Algorithm Platform (Steps 4 & 5).

Customisable dispatcher

The dispatcher is a self-contained algorithm running outside of Viriato. Therefore, it can be replaced or modified for the differing needs of railway organisations. When the dispatcher is given an event, it has two main options (Step 6) - to realise or to postpone the event. This is sufficiently flexible to implement more complex operations such as changing the order through overtaking at a station or to decide where trains will cross.

A very basic dispatcher would simply realise all events, thereby letting the simulator solve all conflicts through delaying the affected train. However, a more useful dispatcher would also have the option to postpone the initiating train itself if the affected train has a higher priority. Another reason for a dispatcher to introduce an additional delay could be to wait for the arrival of another train to satisfy connections or other operational constraints. To enable a dispatcher to make a good decision requires a prediction showing the probable consequences of each option. As it has access to the Algorithm Interface, the dispatcher can also make use of the rich features of the Algorithm Platform such as infrastructure exploration or running time calculations.

Each dispatching decision is returned to the simulator (Step 7) where it is processed and included in the forecast calculation to determine the next realisable event.

Analysing the result

After a successful run the result can be analysed in various ways. The most important KPIs (e.g. average delays, total delay and time to recover) are calculated automatically by Viriato and presented to the user. But there is also the option to export raw data (like the delay log) which can be used to calculate further KPIs. Another feature is to write a simulated timetable back to Viriato where it can be compared with the planned one, for example by using Viriato's graphic timetable.

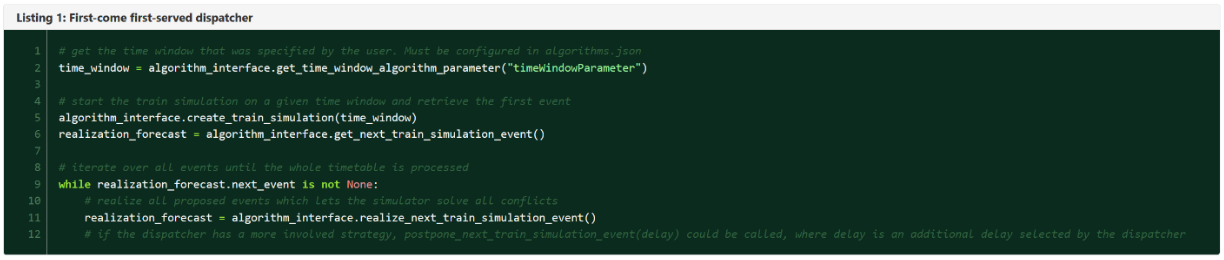
Layered Domain Model

We have seen above that the simulation covers various aspects specific to the railway domain: Planned arrival and departure times, the route of a train on the infrastructure and the minimum running and stopping times all stem from the timetable. The conflict detection and resolution in the simulator and the train dispatching by an actor are other operations. Each of these aspects forms a domain layer and are reflected by the architectural components depicted in Figure 1.

Implementing a Dispatching Strategy

Implementing a dispatcher is done in the same way as any other algorithm that communicates with Viriato via the Algorithm Interface. As the interface itself is a REST interface it supports X-language development. However, to increase development productivity we offer the py_client and the CSharpClient as native client implementations to support type hints and explicit typing respectively. Both provide a native interface and abstract the communication layer in order to simplify the interaction with the REST backend. The py_client is opensource and is available at GitHub.

A simple dispatcher in Python is straightforward. The following code snippet starts the simulation and provides a dispatcher working in a first-come first-served manner which realises all events.



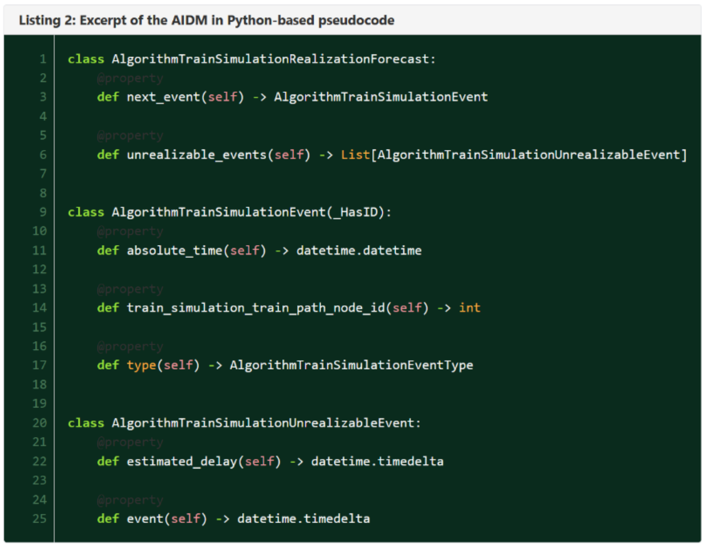
Listing 1: First-come first-served dispatcher

```python
# get the time window that was specified by the user. Must be configured in algorithms.json
time_window = algorithm_interface.get_time_window_algorithm_parameter("timeWindowParameter")

# start the train simulation on a given time window and retrieve the first event
algorithm_interface.create_train_simulation(time_window)
realization_forecast = algorithm_interface.get_next_train_simulation_event()

# iterate over all events until the whole timetable is processed
while realization_forecast.next_event is not None:
    # realize all proposed events which lets the simulator solve all conflicts
    realization_forecast = algorithm_interface.realize_next_train_simulation_event()
    # if the dispatcher has a more involved strategy, postpone_next_train_simulation_event(delay) could be called, where delay is an additional delay selected by the dispatcher
```

When the next event is requested, the dispatcher receives an instance of the type AlgorithmTrainSimulationRealizationForecast. This object includes the next event and also all events that had to be postponed by the simulator. Listing 2 and Figure 2 show the current state of the corresponding AIDM in a Python-based pseudocode and in UML notation.



Listing 2: Excerpt of the AIDM in Python-based pseudocode

```python
class AlgorithmTrainSimulationRealizationForecast:
    @property
    def next_event(self) -> AlgorithmTrainSimulationEvent

    @property
    def unrealizable_events(self) -> List[AlgorithmTrainSimulationUnrealizableEvent]


class AlgorithmTrainSimulationEvent(_HasID):
    @property
    def absolute_time(self) -> datetime.datetime

    @property
    def train_simulation_train_path_node_id(self) -> int

    @property
    def type(self) -> AlgorithmTrainSimulationEventType


class AlgorithmTrainSimulationUnrealizableEvent:
    @property
    def estimated_delay(self) -> datetime.timedelta

    @property
    def event(self) -> datetime.timedelta
```

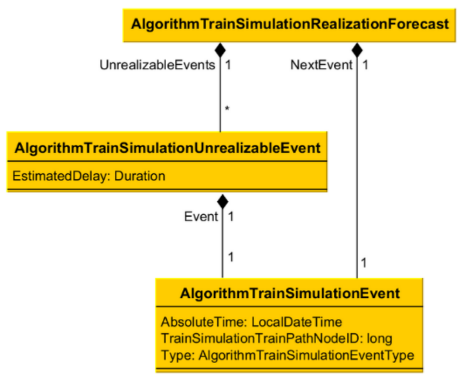

AlgorithmTrainSimulationRealizationForecast
UnrealizableEvents
1
NextEvent
1
*
AlgorithmTrainSimulationUnrealizableEvent
EstimatedDelay: Duration
Event
1
1
1
AlgorithmTrainSimulationEvent
AbsoluteTime: LocalDateTime
TrainSimulationTrainPathNodeID: long
Type: AlgorithmTrainSimulationEventType

Figure 2: Excerpt of the AIDM in UML

Outlook

In the future it will be possible to define a probability distribution which will be used to generate delays that are introduced into the simulation. Multiple iterations can then be run with the same distribution to carry out a Monte Carlo analysis. By the nature of the problem, it is possible to parallelise these iterations arbitrarily so that an analysis on multiple iterations takes almost the same time as one iteration if