Appendix

A. Railway system DTM

For the correct operation of railway system, it is necessary to track the location, condition and other characteristics of all vehicles (various passenger trains, freight trains, technical trains and other equipment that moves on rails) plying the railway tracks. Having the models of all vehicles and keeping them up-to-date makes it possible to increase the efficiency of logistics of transportation and movement of vehicles. With the help of sensors giving coordinates in space, speed of movement, heading angle, as well as actuators that perform various commands to change the parameters of vehicle movement, it is possible to receive up-to-date information in real time, process it and take specific actions.

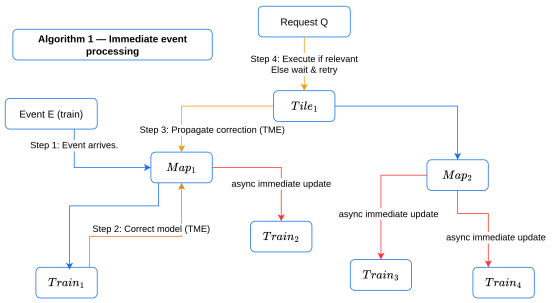

Fig. 1. Algorithm 1. Immediate event processing.

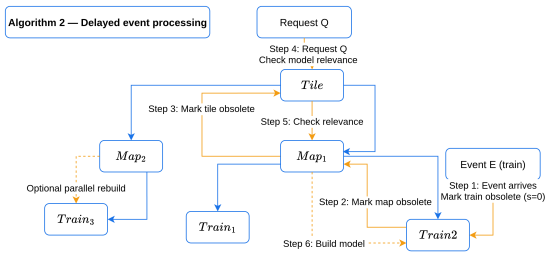

Fig. 2. Algorithm 2. Delayed event processing.

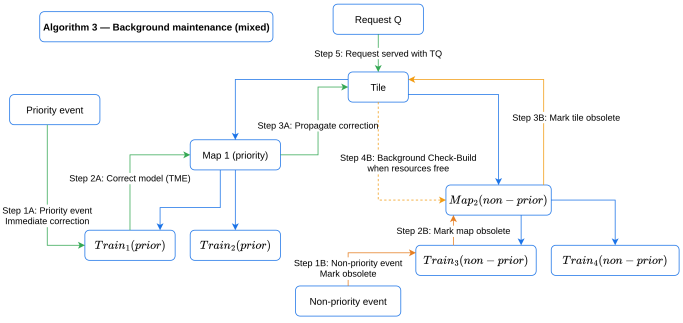

Fig. 3. Algorithm 3. Maintaining specific models in the background.

The diagram of models relevance update is shown on Fig. 4.

For the railway field the model has the following structure: at the lower level there are sensors that gather and send data (coordinates, speed, etc.) received from vehicles. At the next level there are vehicles which model may be in relevant or irrelevant state. State of the vehicle model is considered relevant if there are no changes in the data received from the sensors. If the sensor monitors the change, then an event is sent, which must be processed. If at least one sensor has sent an event, then the vehicle model becomes irrelevant. The model of the upper level combines several vehicles and is called a tile (map fragment). The tile state becomes irrelevant if the state at least of one of the vehicles is irrelevant. The next level, is the root one, it combines several tiles (map fragments) into a single map. The state of the map becomes irrelevant if the state at least of one of the tile is irrelevant. The functioning of the proposed algorithms to keep the model up-to-date is illustrated in Fig. 1, 2 and 3.

B. Implementation of algorithms

To implement the proposed algorithms, the following packages were used:

- Python3 was used to implement model classes, as well as to process data from sensors and actuators;
- ROS2 was used to simulate data received from sensors and actuators;
- Docker was used for virtualization of sensors and actuators simulation.

To solve the problem of system modeling, the following classes were developed:

- Template basic model class with a list of child models, based on which the model state (relevant / irrelevant) is updated;
- Vehicle that receives data from sensors and actuators and updates its relevance based on them;
- Tile (map fragment) which stores a list of vehicles;
- Map which stores a list of tiles.

A limited queue has been created to update vehicle models. The relevance of the state of the vehicle model is estimated based on the relevance of all messages taken from a fixed by length queue. The queue length is set by the parameter N (in the test cases below, the queue length is set to five messages). The message is a built-in structure from the ROS2 library, which is called PointStamped. To implement the experiment the class was written for publishing ROS2 messages. The console output of the published ROS2 message is shown in Fig. 5. In the figure the timestamp when the message was generated, the name of the class that publishes the messages, as well as the coordinates on the circle trajectory along which the vehicle is moving, can be seen.

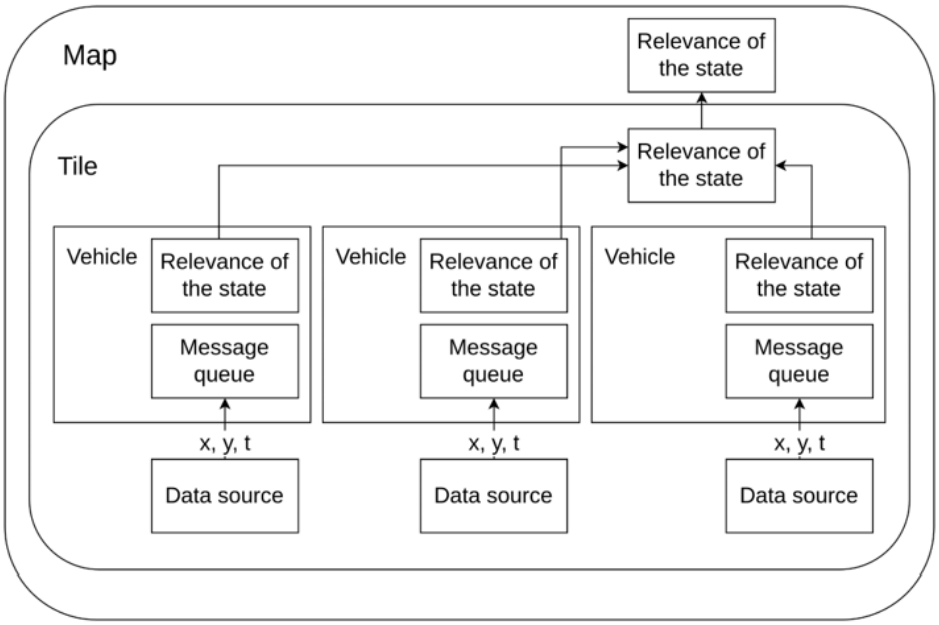

Fig. 4. The diagram of models relevance update

```
[INFO] [1721389639.666157778] [data_publisher_test_0]: Publisher - data_publisher_test_0
x - 9.950041652780259, y - 0.9983341664682815
[INFO] [1721389640.151581106] [data_publisher_test_0]: Publisher - data_publisher_test_0
x - 9.800665778412416, y - 1.9866933079506122
[INFO] [1721389640.651346171] [data_publisher_test_0]: Publisher - data_publisher_test_0
x - 9.55336489125606, y - 2.955202066613396
[INFO] [1721389641.151219118] [data_publisher_test_0]: Publisher - data_publisher_test_0
x - 9.210609940028851, y - 3.8941834230865053
[INFO] [1721389641.651043136] [data_publisher_test_0]: Publisher - data_publisher_test_0
x - 8.775825618903728, y - 4.79425538604203
[INFO] [1721389642.151127496] [data_publisher_test_0]: Publisher - data_publisher_test_0
x - 8.253356149096783, y - 5.6464247339503535
```

Fig. 5. The console output of the published ROS2 message

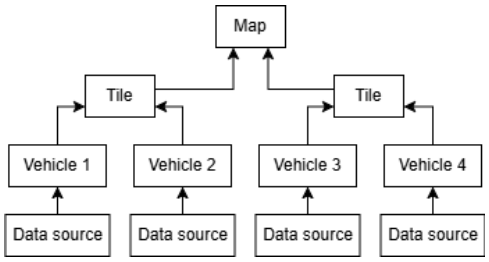

Fig. 6. Diagram of models hierarchy in the system.

The hierarchical structure of the constructed model is presented in Fig. 6. Update requests go from the highest level (Map) to the lowest (Vehicle). Further, after getting a request to update the relevance of the model state, the requests are processed in accordance with the algorithms: immediate event processing, delayed event processing, keeping individual models up-to-date in the background. Requests to models can only originate from the model at the top level, i.e. "Vehicle" model cannot execute an update request for "Tile" model, but "Tile" can update the relevance of "Vehicle".

After each of the "child" models is updated to relevant/irrelevant state, the “Tile" model becomes relevant/irrelevant. An example of how the status update works is shown in Fig. 7, where you can see the console output of the system which model consists of tile, vehicles and data sources.

The system consistently receives data from "Data source" models and updates states of models that are part of the system. Changes of states in each model of the system can be seen in Fig. 7. Each model is initialized in UNDEF (undefined) state and after five messages from data sources are received, states of all models change to VALID.

To execute the queries, algorithms were written for updating the model state. During the immediate event processing, the method for updating the state of the specified model is called. For example, test_1 and test_2 models were selected. The processing of requests for different cases of operation of data sources is shown in Fig. 7 - 9. In Fig. 8 you can see that only a single data source is active, so the only model named “test_2”

```
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is UNDEF
 Tile - Tile_test is UNDEF
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is UNDEF
 Tile - Tile_test is INVALID
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is UNDEF
 Tile - Tile_test is INVALID
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is UNDEF
 Tile - Tile_test is INVALID
Vehicle - test_0 is VALID Vehicle - test_1 is VALID Vehicle - test_2 is VALID
 Tile - Tile_test is INVALID
```

Fig. 7. Example of updating the “Tile” state.

```
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF
Vehicle - test_1 is in idle mode and updates frequantly, current state - UNDEF
Vehicle - test_2 is VALID
Tile - Tile_test is INVALID


Made instant update for test_2, state - VALID
Made instant update for test_1, state - UNDEF
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF
Vehicle - test_1 is in idle mode and updates frequantly, current state - UNDEF
Vehicle - test_2 is VALID
Tile - Tile_test is INVALID
```

Fig. 8. Example of immediate update of the “Vehicle” models (test_1 and test_2), with a single data source for the test_2 model.

```
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is VALID
Tile - Tile_test is INVALID


Made instant update for test_2, state - VALID
Made instant update for test_1, state - UNDEF
```

Fig. 9. Example of immediate update of the “Vehicle” models (test_1 and test_2), with two data sources for the test_2 and test_0 models.

will be VALID, all other vehicles will not be initialized, because there are no other active data sources.

Fig. 9 shows an option when there are two data sources for the test_1 and test_0 models. In this case, the event is processed instantly for the test_1 and test_2 models.

Fig. 10 shows an option when there are all three data sources for the test_1, test_0 and test_2 models. In this case, the event is also processed instantly for the test_1 and test_2 models.

For delayed event processing, implemented in the second algorithm (delayed event processing), the synthesis of “child” models takes place. The event is processed after receiving a request for delayed event processing. Model, named “test_0”, was chosen as an example. The processing of requests in different cases of data sources operation is shown in Fig. 11 - 14. In Fig. 11 it can be seen that with only a single data source for the test_2 model, all vehicles will not be initialized except for the test_2 vehicle.

To maintain models up-to-date in the background, a list of model names that will be updated on an ongoing basis need to be specified in the configuration. To do this, during the synthesis of child models, it is checked whether it is necessary to raise the update flag on a permanent basis for the created model, i.e. idle variable, which can take only two values - True or False. An example of maintaining the test_0 model in the background for a different number of data sources is shown in Fig. 15 - 17.

Time of processing the event is important, because trains have a long braking distance, so every millisecond might be significant in different situations. To compare the algorithms with each other in terms of the average time between receiving information about an event and processing table I was composed. The table represents the time difference between receiving the event and processing it. The experiment consists of twenty consecutive event reception and processing, then the

```
Vehicle - test_0 is UNDEF Vehicle - test_1 is VALID Vehicle - test_2 is VALID
Tile - Tile_test is VALID


Made instant update for test_2, state - VALID
Made instant update for test_1, state - VALID
```

Fig. 10. Example of immediate update of the “Vehicle” models (test_1 and test_2), with three data sources for all “child” models of the Tile_test model.

```
Vehicle - test_0 is VALID Vehicle - test_1 is UNDEF Vehicle - test_2 is UNDEF
Tile - Tile_test is INVALID


Made delayed update for test_0 state - UNDEF
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is UNDEF
Tile - Tile_test is INVALID
```

Fig. 11. Example of delayed updating of “Vehicle” models (test_0), with three data sources for all “child” models of the Tile_test model.

```
Vehicle - test_0 is VALID Vehicle - test_1 is VALID Vehicle - test_2 is VALID
Tile - Tile_test is VALID


Made delayed update for test_0 state - UNDEF
Vehicle - test_0 is UNDEF Vehicle - test_1 is VALID Vehicle - test_2 is VALID
Tile - Tile_test is VALID
```

Fig. 12. Example of delayed updating of “Vehicle” models (test_0), with three data sources for all “child” models of the Tile_test0 model.

```
Vehicle - test_0 is VALID Vehicle - test_1 is UNDEF Vehicle - test_2 is VALID
Tile - Tile_test is INVALID


Made delayed update for test_0 state - UNDEF
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is VALID
Tile - Tile_test is INVALID
```

Fig. 13. Example of delayed updating of “Vehicle” models (test_0), with two data sources for the test_0 and test_2 models.

TABLE I
AVERAGE TIME BETWEEN RECEIVING INFORMATION ABOUT AN EVENT AND PROCESSING IT, OBTAINED AS A RESULT OF THE PROPOSED ALGORITHMS

Case of keeping DT model up-to-date	**Average operating time between receiving information about twenty events and processing them, second**
Delayed event processing	0.008
Instant event processing	0.000114
Maintaining models in the background	0.000114

time for the twenty events is averaged.

As a result of modeling, the most effective algorithm for updating a given model turned out to be the algorithm that assumes maintaining specific models up-to-date. This is due to the fact that this method does not require traversing the entire model system to find each model that is required to get information about the state. The second fastest method is instant event processing, which demonstrates slightly slower execution due to the need to search for the appropriate model when processing a request. The delayed event processing method is on the third place, it requires repeated synthesis of selected “child” models, which in the experiments clears the message queue, which requires processing at least five new messages to determine whether the model is relevant or not.

```
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is VALID
Tile - Tile_test is INVALID


Made delayed update for test_0 state - UNDEF
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF Vehicle - test_2 is VALID
Tile - Tile_test is INVALID
```

Fig. 14. Example of delayed updating of “Vehicle” models (test_0), with a single data source for the test_2 model.

```
Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF
Vehicle - test_1 is in idle mode and updates frequantly, current state - UNDEF
Vehicle - test_2 is UNDEF
Tile - Tile_test is INVALID


Vehicle - test_0 is UNDEF Vehicle - test_1 is UNDEF
Vehicle - test_1 is in idle mode and updates frequantly, current state - UNDEF
Vehicle - test_2 is VALID
Tile - Tile_test is INVALID
```

Fig. 15. Example of maintaining the test_1 “Vehicle” model in the background, with a single data source for the test_2 model.

```
Vehicle - test_0 is VALID Vehicle - test_1 is VALID Vehicle - test_2 is VALID
 Tile - Tile_test is VALID
Made instant update for test_1, state - VALID
Made instant update for test_0, state - VALID
Vehicle - test_0 is VALID Vehicle - test_1 is VALID Vehicle - test_2 is VALID
 Tile - Tile_test is VALID
Made instant update for test_1, state - VALID
Made instant update for test_0, state - VALID
```

Fig. 16. Example of maintaining the test_1 “Vehicle” model in the background, with two data sources.

```
Vehicle - test_0 is UNDEF Vehicle - test_1 is VALID Vehicle - test_2 is UNDEF
 Tile - Tile_test is INVALID
Made instant update for test_1, state - VALID
Made instant update for test_0, state - UNDEF
Vehicle - test_0 is UNDEF Vehicle - test_1 is VALID Vehicle - test_2 is UNDEF
 Tile - Tile_test is INVALID
Made instant update for test_1, state - VALID
Made instant update for test_0, state - UNDEF
```

Fig. 17. Example of maintaining the test_1 “Vehicle” model in the background, with three data sources.