Development Of OpenTCS-ROS2 Driver

Table of Contents

Table of Figures

1 Research Existing Work

While an open-source OpenTCS-ROS2 driver is a one of a kind at the time of writing, this does not mean that no other types of OpenTCS vehicle drivers exist. Two vehicle drivers are available that have been used as an example to develop the ROS2 vehicle driver.

1.1 Built-in Loopback Driver

The OpenTCS software comes standard with a vehicle driver: the Loop-back driver. This driver is simple in terms of functionality. All instructions are returned with a success response, and instructions for moving the vehicle are simulated. The movement simulations allow the loopback vehicle to effectively move over the plant overview if it has received the order to do so.

Since this vehicle driver is embedded in the software, the source code can be downloaded from the OpenTCS page [20]. The loopback driver is in a standalone project within the OpenTCS source code. The image below illustrates how the major classes of the vehicle driver relate to each other.

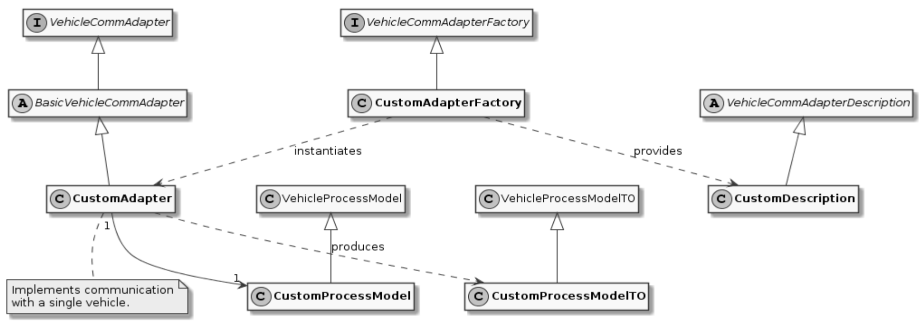


Fig. 1: Conceptual information model of OpenTCS embedded loopback driver [41].

Vehicle Comm Adapter (interface)
An interface that every vehicle adapter must implement. Methods in this interface are called by kernel components, for example, to tell a vehicle to move to the next position. Classes implementing this interface are expected to perform actual communication with a vehicle, e.g. via TCP, UDP or a Fieldbus.

Basic Vehicle Comm Adapter
The '*BasicVehicleCommAdapter'* is the recommended base class for implementing a VehicleCommAdapter. It mainly provides functionality for basic command queuing.

Vehicle Comm Adapter Factory
This class describes a factory for '*VehicleCommAdapter*' instances. The kernel starts and uses such factory per vehicle driver to create instances of the respective '*VehicleCommAdapter*' implementation upon request.

Vehicle Process Model
Each '*VehicleCommAdapter*' instance, in which state of both the vehicle and the comm adapter must be maintained, should hold a single VehicleProcessModel instance. This model instance should be updated to notify the kernel of any relevant changes about the vehicle. The communication adapter implementation must, e.g. update the current position of the vehicle in this model when it receives information so that the kernel and the kernel control center can use it.

Likewise, other components can set values that affect the behaviour of the communication adapter in the model, e.g. a time interval for periodic messages that the comm adapter sends to the vehicle. The VehicleProcessModel can be used because it contains members for all the information the OpenTCS kernel itself needs. However, developers can use driver-specific subclasses of VehicleProcessModel to exchange the comm adapter and other components more than the standard set of attributes.

1.2 Integration Project Example

There usually are two ways to add functionality to OpenTCS. The first way is to download the OpenTCS source code and make adjustments to it. The code is completely open-source and can be changed relatively easily. A disadvantage of this approach is that the "added" code is closely intertwined with the OpenTCS source code. Suppose there is a partner with many different types of AGVs, then all additions or changes to the functionality of all AGVs must be present in one code source base. An additional drawback is that this approach does not go well when a new version of OpenTCS is launched. If the partner wishes to upgrade, all source code relevant to the operation of the AGVs must be transferred to the latest version of OpenTCS.

For this reason, applying an OpenTCS Integration Project is a better approach. With an Integration Project, an application is written that works as an extension on top of OpenTCS. It is not even necessary to download the source code of OpenTCS. In the case of multiple types of AGVs, and thus multiple vehicle drivers, ideally each driver will be a separate Integration Project. Because an Integration Project is a standalone project, updating the OpenTCS software is a lot easier.

An Integration Project can be built with Gradle, which creates JAR files. These JAR files must be placed in an extension folder of the appropriate OpenTCS application, such as the Kernel, Kernel Control Center or the Plant Overview.

A sample integration project can be downloaded from the download page [41] of the OpenTCS website. The Development Guide offers instructions on how to set up an integration project.

1.3 Integration

When developing the OpenTCS-ROS2 driver, it is tried as much as possible to keep the OpenTCS design conventions. For example, the exact class structure and exact class naming from the built-in Loopback driver will be applied. Furthermore, the way of implementing the Control Panel GUI for the Kernel Control Center will be implemented in a similar way.

The driver is also developed through an Integration Project, so that the OpenTCS software retains its ease of updating. However, there is a number of deliberate deviations from the OpenTCS standard, which is described in “5 Design Deviations”.

2 Software Requirements

Several partners are involved in project NeNa. The partners use different types of AGVs, of which all AGVs operate in a unique company setting. Also, there is a difference in the policy to be applied for sending orders to the AGVs.

For example, with a preventive policy, the AGV will stop driving if it loses its connection with the fleet manager. The AGV will, therefore, not receive new instructions and will stop working. With this policy, the AGV will always receive instructions for short distances.

With an assertive policy, the fleet manager will send as many instructions in advance as possible. This approach allows the AGV to continue operating for much longer if it loses its connection to the fleet manager.

The case described above typifies one of the many ways to manage an AGV. For this reason, the ROS2 driver will have to be modular. Modular software will make it possible to change specific parts of the driver without having substantive knowledge of the entire software package. An additional advantage of a modular approach is that new functionality can easily be added, such as controlling a turn signal.

Of course, for the development of the base driver, it is necessary to predetermine certain choices. Agreements must also be made about which functionality the driver must have (at least). In consultation with the research group, requirements have been drawn up for this using the MoSCoW model:

- **M**: Must have this requirement to meet the business needs
- **S**: Should have this requirement if possible, but project success does not rely on it
- **C**: Could have this requirement if it does not affect anything else on the project
- **W**: Would like to have this requirement later, but delivery won't be this time

Must	
#	***Requirement***
1	Dispatch the ROS2 AGV to a user-defined coordinate in the control center pane.
2	Dispatch the ROS2 AGV to an OpenTCS Plant Model point.
3	Support for executing OpenTCS Transport Orders.
4	Properly handle ROS2 AGV navigation failures, such an unreachable destination.
5	Support for executing OpenTCS Operations.
6	Support for ROS2 namespaces, which allows usage for multiple ROS2 AGVs simultaneously.
7	Maintain a modular software layout that allows easy integration of additional operations, tasks and other functionality.

Should	
#	***Requirement***
1	Show a continuously updated ROS2 navigation status in the control center panel.
2	Show the connection status in the control center panel.
3	Show the live position of a ROS2 AGV in the OpenTCS Plant Overview.
4	Visualize the orientation of the ROS2 AGV in the OpenTCS Plant Overview.
5	Show a battery percentage of the vehicle in the OpenTCS Plant Overview.

Could	
#	***Requirement***
1	Implement smooth navigation: drive to a destination without stopping at passing points along the route.
2	Support for Plant Model scaling, allowing better representation of tiny/huge plant maps.
3	Get a percentage of how much a current navigation goal has been completed.
4	Support for dynamic ROS2 domain IDs.
5	Set the initial position of a ROS2 AGV.

Would	
#	***Requirement***
1	Support for other languages in the menus (e.g. Dutch).
2	Show a map in the control center panel that is similar to the RViz application.

2.1 Software Domains

The ROS2 vehicle driver is a modular software package. This approach means that a great deal of attention has been paid during its development to split the functionality into independent pieces of software as much as possible. Some advantages of the modularity are scalability, maintainability and reduced complexity. An additional benefit is that the separate components are easier to test.

The figure below shows a schematic overview of all packages in a Package Diagram. This shows which package belongs to which package (fixed line), and which package depends on which package (dotted line).

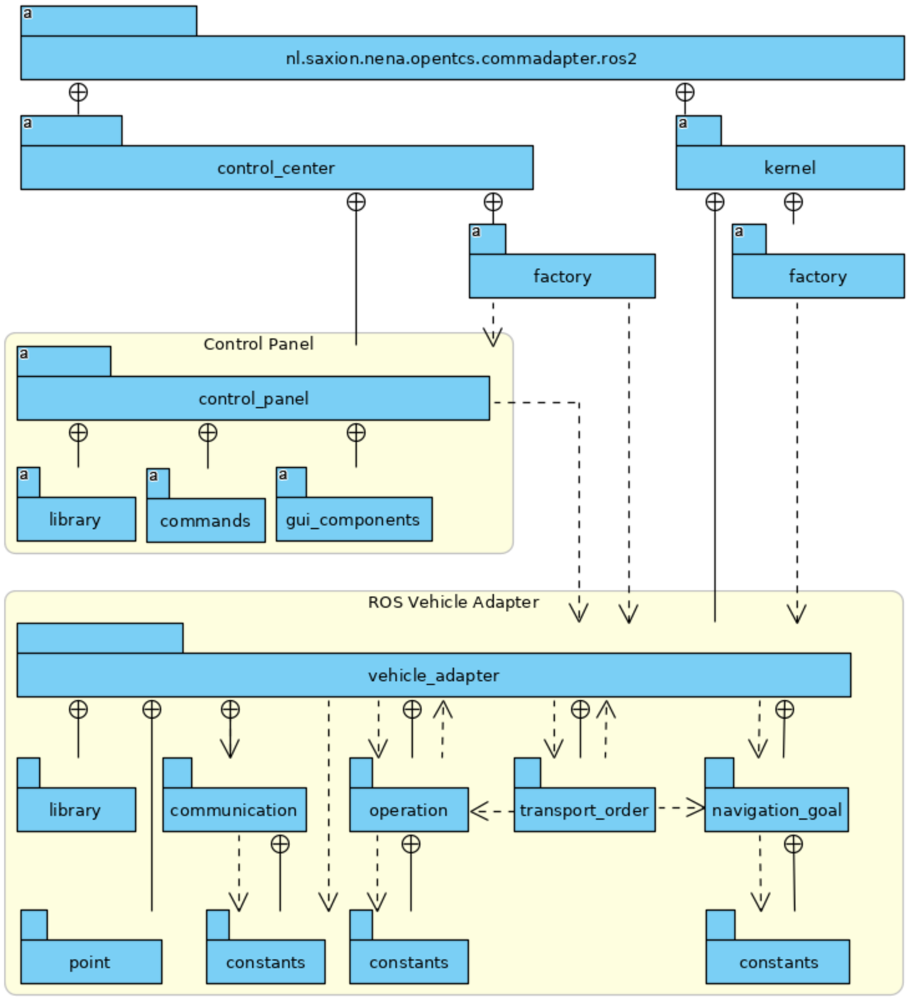


Fig. 2: Domain Diagram of all ROS2-driver components.

The diagram above illustrates which domains depend on other domains. For example, the Control Panel cannot work without the *Vehicle Adapter*, but it is possible the other way around. It is also clearly visible that transport orders depend on operations and navigation goals. Finally, it can be deducted that communication can function independently.

3 Control Center Panel (GUI)

The Kernel Control Center is a GUI application from OpenTCS in which drivers and vehicles can be configured. The ROS2 driver is also set from this platform, which allows OpenTCS to communicate with the ROS2 AGV.

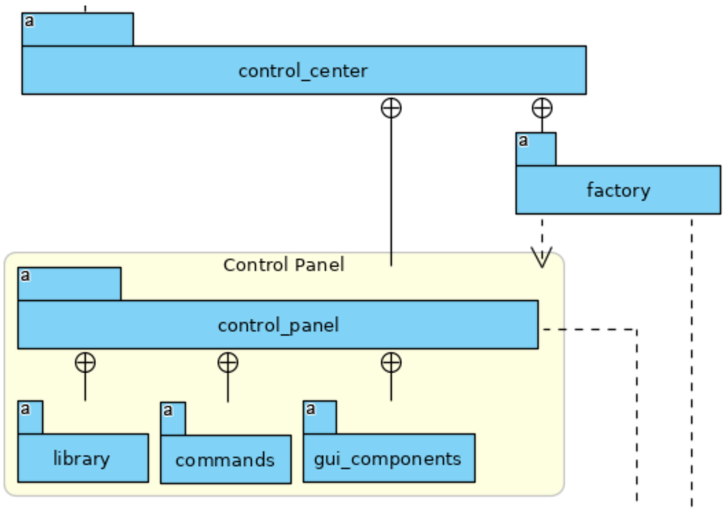


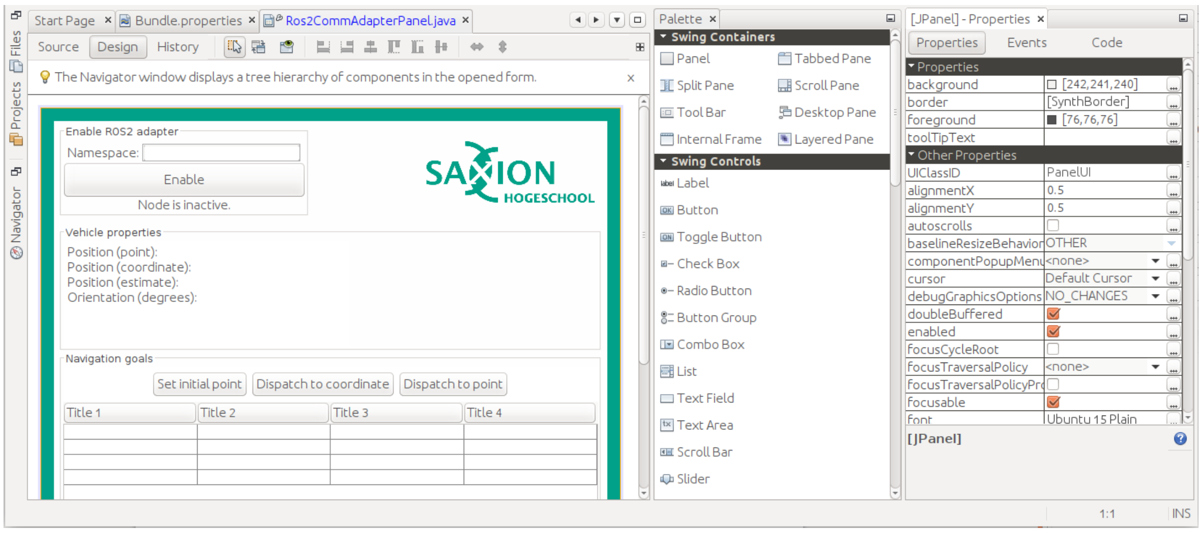
Fig. 3: Control Center Panel domains.

3.1 Factory

The main reason for choosing a Fleet Manager is to manage several AGVs at the same time. This means that per active AGV, an instance of the Control Center Panel will run. The factory ensures that an instance of the Control Panel is available for each active vehicle.

3.2 Java Swing

The OpenTCS developers have used the Apache Netbeans GUI Builder to develop the Kernel Control Center. The Control Panel for the ROS2 driver is developed in the same way. It is possible to build a GUI entirely from a Java file, but for a more complex layout, it is easier to use a builder, such as the Apache Netbeans GUI Builder. The image below illustrates the development of the driver panel in Netbeans.



The Navigator window displays a tree hierarchy of components in the opened form.

Enable ROS2 adapter
Namespace:
Enable
Node is inactive.

Vehicle properties
Position (point):
Position (coordinate):
Position (estimate):
Orientation (degrees):

Navigation goals
Set initial point | Dispatch to coordinate | Dispatch to point

Title 1	Title 2	Title 3	Title 4

Properties	
background	☐ [242,241,240]
border	[SynthBorder]
foreground	■ [76,76,76]
toolTipText	
Other Properties	
UIClassID	PanelUI
alignmentX	0.5
alignmentY	0.5
autoscrolls	☐
baselineResizeBehavior	OTHER
componentPopupMenu	<none>
cursor	Default Cursor
debugGraphicsOptions	NO_CHANGES
doubleBuffered	✓
enabled	✓
focusCycleRoot	☐
focusTraversalPolicy	<none>
focusTraversalPolicyPr	☐
focusable	✓
font	Ubuntu 15 Plain

[JPanel]

Fig. 4: Designing the control panel in Java Swing using Apache Netbeans GUI Builder.

In addition to the main panel visible above, additional panels have been developed for entering and verifying coordinates, for example.

3.3 COMMANDS

The Kernel Control Center is a standalone application but relies on an underlying Kernel. After all, without Kernel, there is little to configure. The link between both applications is done through commands. A command is located in the Kernel Control Center, describing what needs to be done at the Kernel. An example of this is the "enable" button. When pressing this button, a command is sent to the Kernel to invoke a particular function there. A command consists of a single simple class, as visible in the example below:

```java
/**
 * Instruct the kernel to dispatch the vehicle to a {@link Triple} coordinate.
 *
 * @author Niels Tiben
 */
@AllArgsConstructor
public class DispatchToCoordinateCommand implements AdapterCommand {
    private final Triple destinationCoordinate;

    @Override
    public void execute(@Nonnull VehicleCommAdapter adapter) {
        Ros2ProcessModel ros2ProcessModel = (Ros2ProcessModel)
adapter.getProcessModel();
        ros2ProcessModel.dispatchToCoordinate(this.destinationCoordinate);
    }
}
```

4 Kernel ROS2 Driver

The ROS2 driver is intended for the Kernel, and thus responsible for handling communications, operations, transport orders and navigation targets of the ROS2 AGV. As with the Control Center Panel, a factory is used to ensure that each AGV receives an instance of the ROS2 driver. The code is functionally divided into one of the appropriate packages:

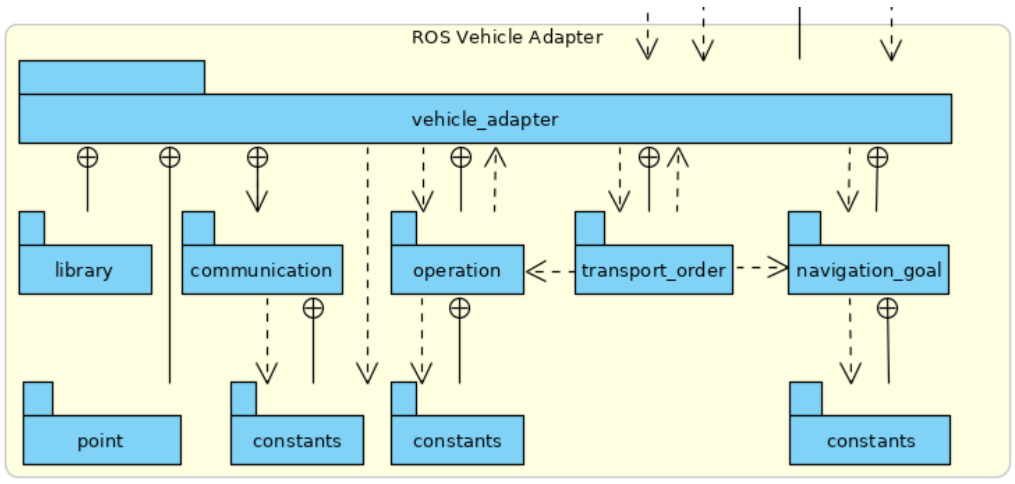


Fig. 5: Kernel ROS2 Driver domains.

4.1 Process Model

The Process Model class receives instructions from the parent adapter, Control Center panel and various callback functions from, for example, the Node Manager and the Navigation Goal Tracker. All these dependencies make the Process Model a central point in the application. A conceptual information model has been drawn up to visualize the position within its dependencies:

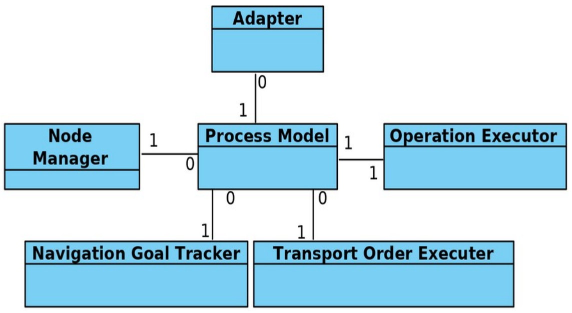


Fig. 6: Conceptual information model of the Process Model.

4.2 Communication

A crucial part of the driver is the communication part. In this chapter, further explanation is given about the development and implementation choices of the communication part.

RCLJava Integration

ROS2 for Java (rcljava) is used for communication. Ideally, Java developers download dependencies through an online repository. However, this package relies heavily on the Linux system files and is therefore not portable, so ROS2 for Java *must* be built en sourced locally [11]. This is because the rcljava environment must be sourced before an application can use the ROS2-Java functionality. This is a design decision which is made by ROS2 [15].

A Gradle script has been written to simplify importing the correct rcljava package. It is only necessary to enter the ROS2-Java installation path, after which the script handles the rest. The script validates whether the path is correct. Then it automatically extracts the correct rcljava dependencies from the ROS2-Java build folder. If the ROS2-Java build folder could not be found, a plain error is shown to the developer:

```
Build file '/home/nielstiben/Documents/opentcs/integration/openTCS-NeNa/openTCS-NeNa-CommAdapter-ROS2/build.gradle'

A problem occurred evaluating project ':openTCS-NeNa-CommAdapter-ROS2'.
> Ros2-Java not found OR not build. Specify the path in build.gradle.
```

If this message is not shown above, it means that all dependencies have been successfully imported.

Non-blocking Nodes

Setting up and running a Node is a blocking task. In order not to hinder other processes, a solution has been developed, in which the Node runs in a separate thread. There is a separate instance that starts and stops the node, which is called the Node Manager. The Node manager uses callbacks to notify other class instances of the running status. The sequence diagram below shows how to set up a new driver instance, and thus a new Node.

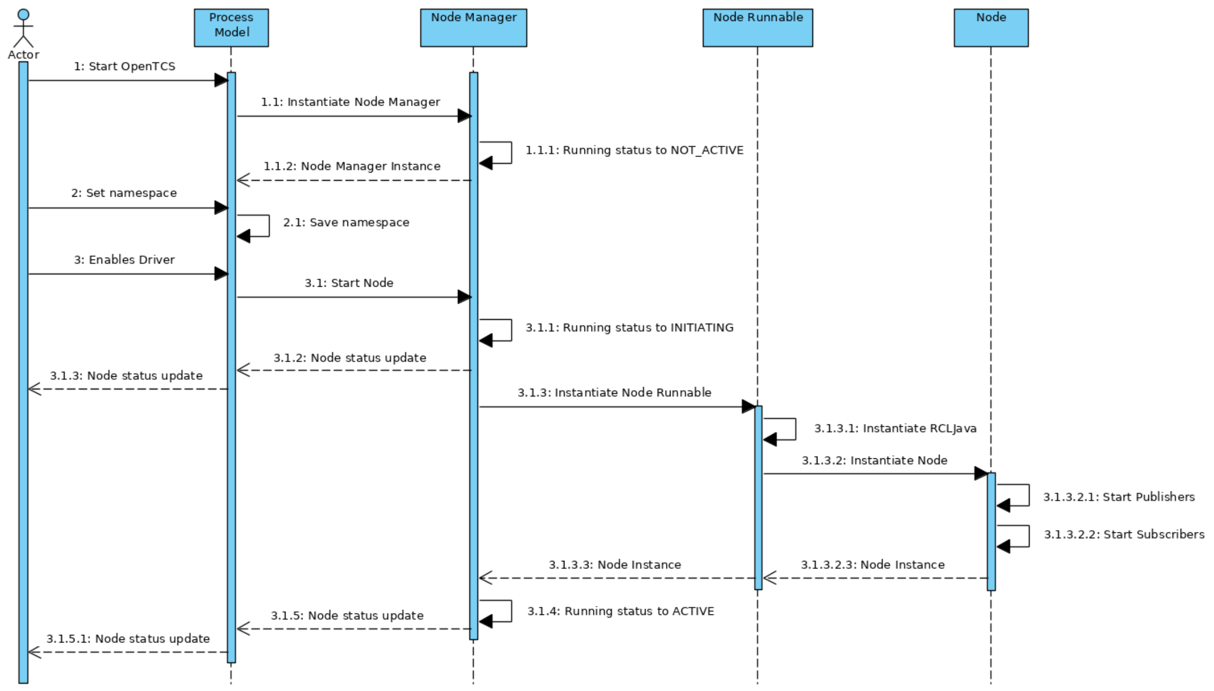

Actor
Process Model
Node Manager
Node Runnable
Node
1: Start OpenTCS
1.1: Instantiate Node Manager
1.1.1: Running status to NOT_ACTIVE
1.1.2: Node Manager Instance
2: Set namespace
2.1: Save namespace
3: Enables Driver
3.1: Start Node
3.1.1: Running status to INITIATING
3.1.2: Node status update
3.1.3: Node status update
3.1.3: Instantiate Node Runnable
3.1.3.1: Instantiate RCLJava
3.1.3.2: Instantiate Node
3.1.3.2.1: Start Publishers
3.1.3.2.2: Start Subscribers
3.1.3.3: Node Instance
3.1.3.2.3: Node Instance
3.1.4: Running status to ACTIVE
3.1.5: Node status update
3.1.5.1: Node status update

Fig. 7: Sequence diagram of how to establish a node instance.

Node State Transitions

It takes time to initiate a node in a separate thread. For this reason, the introduction of *node Running States* was chosen. This allows the application to know the live state of the and prevent a node from receiving instructions even though it is not yet ready. It also prevents a node in the initiating phase from receiving a start command because of an inpatient end-user repeatedly presses the "Enable driver" button.

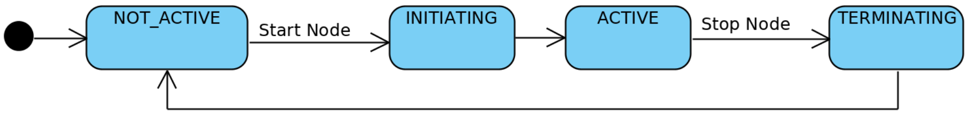


Fig. 8: Machine state diagram visualising the allowed node running state transitions.

Publishing And Subscribing

If the node is in the active phase, it is ready to use the publishers and subscribers. Publishers are intended to send data to the ROS2 AGV. Currently, publishers are in use on the following topics:

- "/initialpose": To set the initial position of an AGV.
- "/move_base_simple/goal": For sending navigation commands.

Publishing messages on the above topics can be done from the Process Model (see “4.1 Process Model”). In addition to publishers, subscribers can receive data from the ROS2 AGV. The following subscribers on the following topics are in use:

- "/NavigateToPose/_action/status": To receive the current ROS2 navigation statuses.
- "/amcl_pose": To receive the AGV’s current (exact) position and orientation.

The subscribers give a callback to the Process Model for each incoming message, which further processes the message.

It is possible to implement more subscribers or publishers for more functionality. Think of publishers and subscribers for handling specific tasks such as loading and unloading goods.

4.3 NAVIGATION

ROS2 uses Navigation 2 to drive an AGV from point a to point b safely [42]. To move an AGV, one must send a navigation goal to Navigation 2. Sending a navigation goal is a typical example of a ROS2 action. An action is a communication type that is designed for longer-term tasks and consists of a goal, result and feedback [43]. A goal is a message that starts the action. This message is sent by the application that wants to move the AGV. The AGV sends the result and feedback during or after the task is performed.

Unfortunately, actions are not yet supported in ROS2 Java. However, this does not mean that sending navigation targets is impossible. Since actions are built on top of topics and services, it is possible to send or receive a goal, result and feedback separately. The exact approach of independently handling the channels is what is implemented at for the ROS2 driver at the moment.

Sending Goals

The fleet manager has a publisher active, which sends messages on "/move_base_simple/goal". This message contains a coordinate and orientation of the position where the AGV should drive. This message is picked up by the navigation node, which then starts a navigation action.

Receiving Goal Execution Feedback

While the AGV is performing the navigation action, the fleet manager needs to know what the current navigation status is. If the AGV has already reached its destination, new instructions may need to be sent. Even if the AGV has not been able to reach its destination, for example, due to an obstacle on the course, it must be known to the fleet manager.

Finding out the navigation status of was challenging. On the internet, you can find a lot of information about the predecessor of ROS1: the Navigation Stack. However, information about the navigation progress for ROS1 is not fully applicable to Navigation 2. General information can be found about Navigation 2 [42], but no substantive information about the topics used is offered. Research has been done in two ways to find a way to receive feedback on the Navigation 2 node.

First, the decomposition methodology is applied to the source code of Navigation 2. This node is developed in C ++ (*rclcpp*). Console messages are sent from the node, such as “Navigation Succeeded”. However, these console messages are only visible in the terminal where the relevant node is running, so that other nodes such as the OpenTCS node cannot see these messages. Besides, these logging messages are scant and not intended for this type of purpose.

For example, a console message does not include a unique identifier, so it is never sure that the console messages belong to the navigation goal started by OpenTCS.

It was further noted that the Navigation 2 feedback channel had not been implemented. The feedback channel is typically used to send interim updates about the action that is currently being executed. The placeholders are present in the source code, but no practical functionality has been implemented for handling feedback.

Although the Navigation 2 action’s feedback topic cannot (yet) be used, this can be used with the action’s result topic. Each time a navigation action is started, messages are repeatedly published on the topic `/NavigateToPose/_action/status`. A message looks like this:

```
/niels@laptop:~$ ros2 topic echo /NavigateToPose/_action/status
status_list:
- goal_info:
    goal_id:
      uuid: [252, 126, 236, 51, 95, 235, 87, 156, 252, 98, 102, 225, 45, 118, 160, 13]
    stamp:
      sec: 1586252361
      nanosec: 703727920
  status: 6
- goal_info:
    goal_id:
      uuid: [41, 55, 100, 99, 191, 18, 167, 153, 57, 243, 1, 90, 68, 37, 116, 3]
    stamp:
      sec: 1586252365
      nanosec: 466342743
  status: 4
- goal_info:
    goal_id:
      uuid: [236, 4, 47, 230, 142, 173, 162, 72, 214, 90, 203, 227, 250, 192, 60, 180]
    stamp:
      sec: 1586253246
      nanosec: 14198261
  status: 4
- goal_info:
    goal_id:
      uuid: [15, 142, 123, 28, 148, 183, 150, 200, 227, 38, 255, 190, 197, 21, 148, 231]
    stamp:
      sec: 1586253278
      nanosec: 662794261
  status: 4
---
```

This message shown above was sent when an action was started and stopped, and sometimes in the meantime. The first step was to understand this message.

First, it can be seen that the message consists of a list of multiple "goal_info" objects. Initially, this was confusing, because sometimes a list was sent with multiple objects, while only one navigation goal had started. However, the list also includes older, already completed, navigation goals and is sorted from newest to oldest. It can, therefore, be assumed that the top object is the most recent goal info.

Each goal info object has a unique value (UUID), a timestamp and a status. The source code Navigation 2 source code reveals a list of status codes, along with a description of each status. Because an updated list is published after every navigation event, such as starting or ending a goal, it is possible to track a navigation goal. Therefore, a goal tracker has been designed.

Navigation Goal Tracker Development

An essential task of the fleet manager is to send navigation goals. It is at least as necessary to keep track of whether a sent-out navigation goal has been successfully completed. The Navigation Goal Tracker has been designed for this purpose. An instance of the Navigation Goal Tracker processes incoming navigation goal lists in the following way:

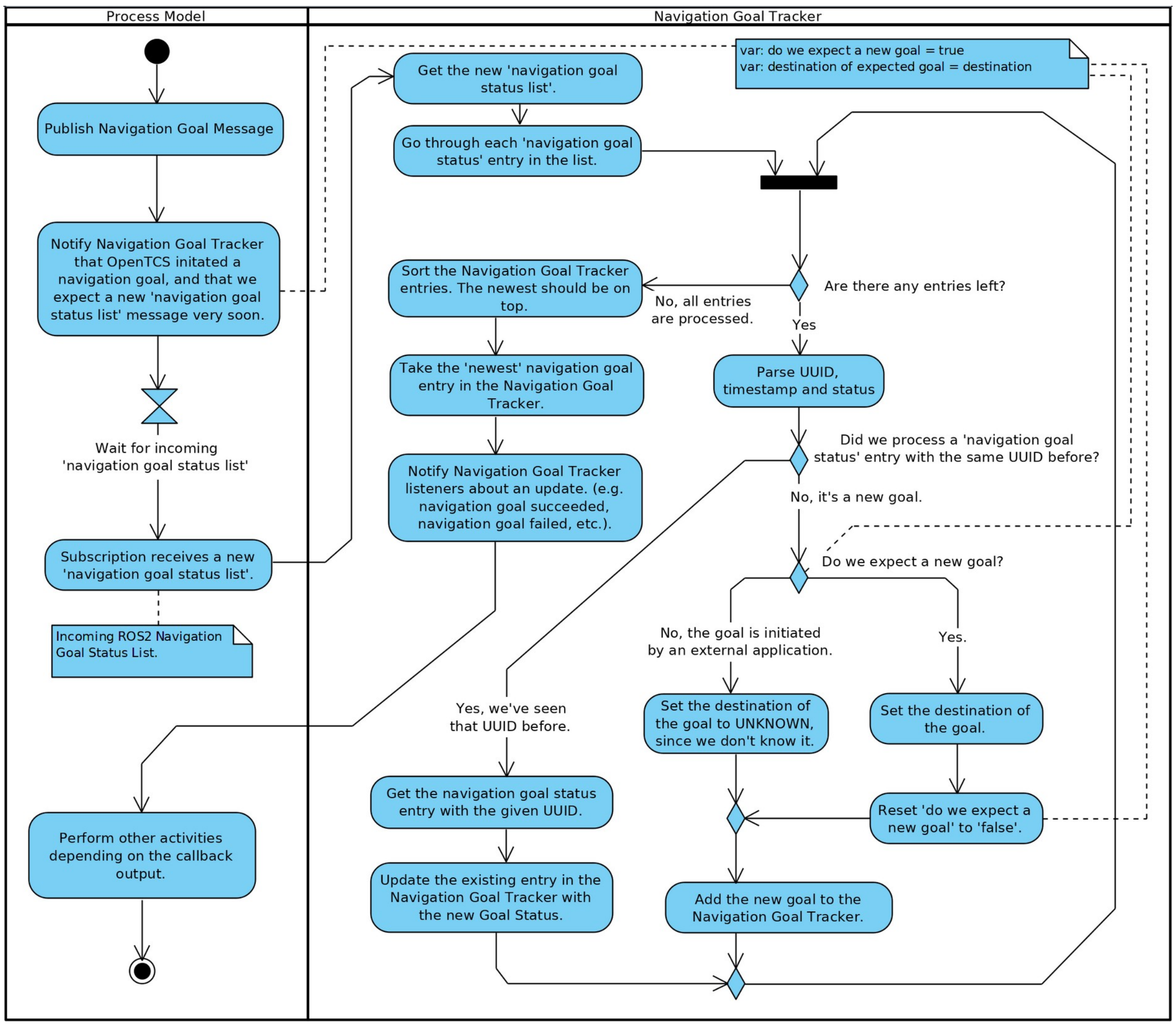

Process Model
Navigation Goal Tracker
Publish Navigation Goal Message
Notify Navigation Goal Tracker that OpenTCS initated a navigation goal, and that we expect a new 'navigation goal status list' message very soon.
Wait for incoming 'navigation goal status list'
Subscription receives a new 'navigation goal status list'.
Incoming ROS2 Navigation Goal Status List.
Perform other activities depending on the callback output.
var: do we expect a new goal = true
var: destination of expected goal = destination
Get the new 'navigation goal status list'.
Go through each 'navigation goal status' entry in the list.
Are there any entries left?
No, all entries are processed.
Yes
Sort the Navigation Goal Tracker entries. The newest should be on top.
Take the 'newest' navigation goal entry in the Navigation Goal Tracker.
Notify Navigation Goal Tracker listeners about an update. (e.g. navigation goal succeeded, navigation goal failed, etc.).
Parse UUID, timestamp and status
Did we process a 'navigation goal status' entry with the same UUID before?
No, it's a new goal.
Yes, we've seen that UUID before.
Do we expect a new goal?
No, the goal is initiated by an external application.
Yes.
Set the destination of the goal to UNKNOWN, since we don't know it.
Set the destination of the goal.
Reset 'do we expect a new goal' to 'false'.
Get the navigation goal status entry with the given UUID.
Update the existing entry in the Navigation Goal Tracker with the new Goal Status.
Add the new goal to the Navigation Goal Tracker.

Fig. 9: Activity Diagram visualising the procedure of processing navigation goals.

4.4 TRANSPORT ORDER

A transport order is a sequence of movements and operations performed by an AGV [22]. A transport order can be created and started in the OpenTCS Plant Overview. The ROS2 vehicle driver processes the transport order and provides feedback to the Plant Overview. The ROS2 vehicle driver has implemented a workflow class that handles a transport order in the following (non-blocking) way:

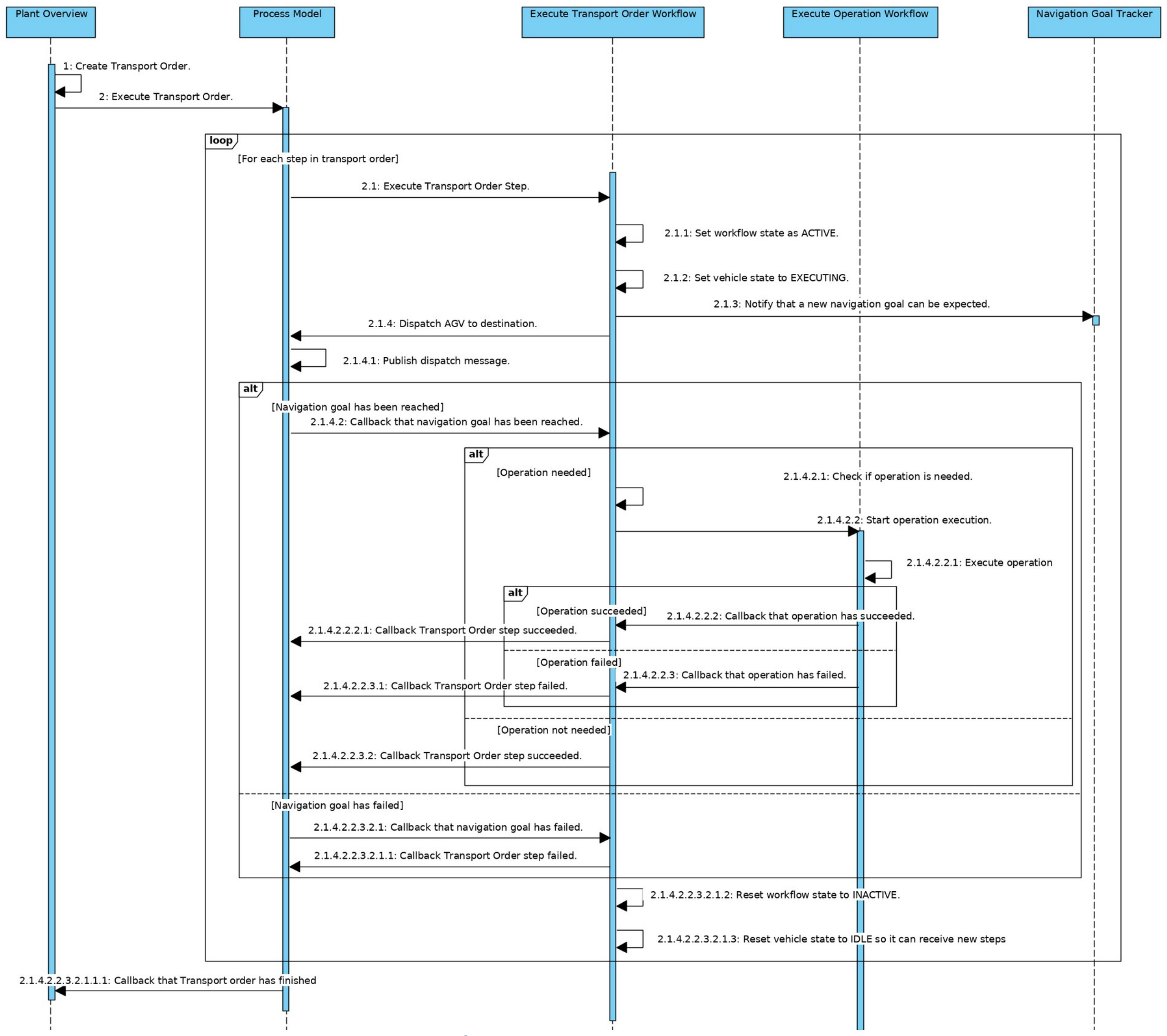

Plant Overview
Process Model
Execute Transport Order Workflow
Execute Operation Workflow
Navigation Goal Tracker
1: Create Transport Order.
2: Execute Transport Order.
loop
[For each step in transport order]
2.1: Execute Transport Order Step.
2.1.1: Set workflow state as ACTIVE.
2.1.2: Set vehicle state to EXECUTING.
2.1.3: Notify that a new navigation goal can be expected.
2.1.4: Dispatch AGV to destination.
2.1.4.1: Publish dispatch message.
alt
[Navigation goal has been reached]
2.1.4.2: Callback that navigation goal has been reached.
alt
[Operation needed]
2.1.4.2.1: Check if operation is needed.
2.1.4.2.2: Start operation execution.
2.1.4.2.2.1: Execute operation
alt
[Operation succeeded]
2.1.4.2.2.2: Callback that operation has succeeded.
2.1.4.2.2.2.1: Callback Transport Order step succeeded.
[Operation failed]
2.1.4.2.2.3: Callback that operation has failed.
2.1.4.2.2.3.1: Callback Transport Order step failed.
[Operation not needed]
2.1.4.2.2.3.2: Callback Transport Order step succeeded.
[Navigation goal has failed]
2.1.4.2.2.3.2.1: Callback that navigation goal has failed.
2.1.4.2.2.3.2.1.1: Callback Transport Order step failed.
2.1.4.2.2.3.2.1.2: Reset workflow state to INACTIVE.
2.1.4.2.2.3.2.1.3: Reset vehicle state to IDLE so it can receive new steps
2.1.4.2.2.3.2.1.1.1: Callback that Transport order has finished

Fig. 10: Sequence diagram showing the process of handling OpenTCS Transport Orders.

4.5 OPERATIONS

Every point in the plant model has a specified type. Characteristic of point types is that specific tasks can only be performed on certain point types. These tasks are called operations. Typical operations for an AGV are, for example, loading and unloading of cargo.

These operations usually are part of a transport order. The AGV is sent from point to point, where it can perform operations on the points.

While an AGV takes time to execute an operation, operations are fully non-blocking integrated into the ROS2 vehicle driver since callbacks are applied. A schematic sequence diagram describes how an operation is performed:

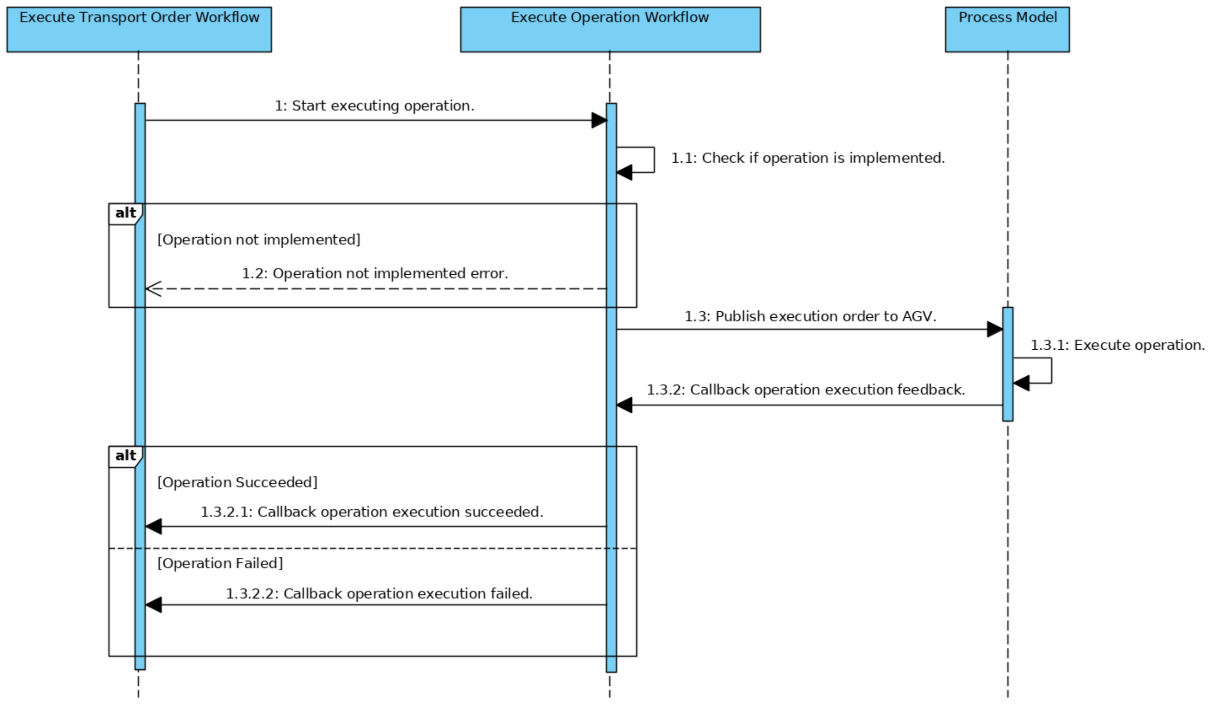

Execute Transport Order Workflow
Execute Operation Workflow
Process Model
1: Start executing operation.
1.1: Check if operation is implemented.
alt
[Operation not implemented]
1.2: Operation not implemented error.
1.3: Publish execution order to AGV.
1.3.1: Execute operation.
1.3.2: Callback operation execution feedback.
alt
[Operation Succeeded]
1.3.2.1: Callback operation execution succeeded.
[Operation Failed]
1.3.2.2: Callback operation execution failed.

Fig. 11: Sequence diagram showing the process of operation executions.

5 Design Deviations

The development of the ROS2-OpenTCS vehicle driver has deviated from the standard code design of OpenTCS in several areas. These deviations are covered in this chapter.

5.1 Library Classes

The same functionality is used in different Java classes. Examples of functions that would be widely reused include converting units or parsing outgoing ROS2 messages. These types of functions are therefore located in Library Classes in which these types of functions are freely accessible to any class that benefits from this shared functionality.

Typical of Library classes is that they are not holding data; they only convert data. Because all functions are static, it is not even possible to hold an instance of a Library Class. Using an instance would not be a logical choice because the functions themselves would not change the (underlying) Library Class object [44].

An example from the Unit Converter Library, where a coordinate in millimetres is converted to a coordinate in meters:

```
public static Triple convertCoordinatesInMeterToTriple(double x, double y, double z) {
    long xInMillimeter = convertMetersToMillimeters(x);
    long yInMillimeter = convertMetersToMillimeters(y);
    long zInMillimeter = convertMetersToMillimeters(z);

    return new Triple(xInMillimeter, yInMillimeter, zInMillimeter);
}
```

5.2 Workflow Classes

Tasks such as executing Transport Orders or executing Operations consist of many steps that must be executed sequentially. Workflow classes are classes with functions that are always intended to be executed sequentially (from top to bottom). Each workflow step is explicitly named in the workflow class. It is also stated how the next step is activated, which can be done, for example, by a callback.

Below is an example of a single step in the Transport Order workflow. The example shows a function where an operation is started if it is needed:

```
//==================================================================================
// 4: Execute Operation (if needed).
//==================================================================================

@SneakyThrows
private void executeOperationIfNeeded() {
    if (this.currentCommand.isWithoutOperation()) {
        // No operation in this command => skip this step.
        setCommandWorkflowSucceeded();
    } else {
        this.executeOperationWorkflow.executeOperationByName(this.currentCommand.getOperation());
    }
}
// Next step (5) is activated by callback. Step 5 is skipped when there are no operations.
```

5.3 Automatic Code Injection

Lombok was added to this project to keep the code cleaner and more readable. Lombok is a Java library and plugin that automatically injects code based on annotations. It is also used in (other) large Java projects such as the Spring Framework.

Usually, most Java classes have getters, setters and one or more constructors. With Lombok, writing this type of code is unnecessary. Only annotations have to be placed for this functionality. The code below describes a class with three parameters. Due to the annotations, it is no longer necessary to write a constructor, getters or setters.

```java
/**
 * A serializable representation of a {@link Ros2ProcessModel}.
 *
 * @author Niels Tiben
 */
@Getter
@Setter
@NoArgsConstructor
public class Ros2ProcessModelTO extends VehicleProcessModelTO {
    private String nodeStatus;
    private Triple estimatePosition;
    private String[][] navigationGoalTable;
}
```

6 Documentation

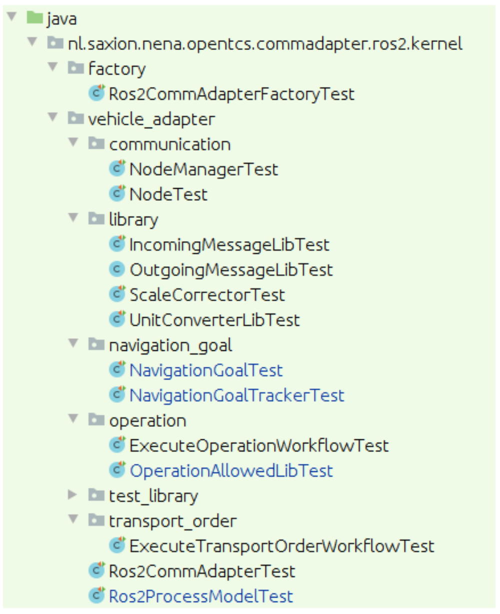
All packages and classes are described by Javadoc. Javadoc is an industry-standard and Java documentation generator that creates HTML pages to describe a program.

The code has been developed in accordance with the "self-explanatory code" method. All functions and variables are written in such a way that comments are no longer needed. With this method, function names and variable names are written out in full, and each step in the code is written on a separate line or in a separate function. This makes the code longer but easier to understand.

7 Code Quality Measurements

Various tooling has been used to ensure code quality. The tooling is described in this chapter.

7.1 Unit Tests



```
▼ java
  ▼ nl.saxion.nena.opentcs.commadapter.ros2.kernel
    ▼ factory
        Ros2CommAdapterFactoryTest
    ▼ vehicle_adapter
      ▼ communication
          NodeManagerTest
          NodeTest
      ▼ library
          IncomingMessageLibTest
          OutgoingMessageLibTest
          ScaleCorrectorTest
          UnitConverterLibTest
      ▼ navigation_goal
          NavigationGoalTest
          NavigationGoalTrackerTest
      ▼ operation
          ExecuteOperationWorkflowTest
          OperationAllowedLibTest
      ▶ test_library
      ▼ transport_order
          ExecuteTransportOrderWorkflowTest
        Ros2CommAdapterTest
        Ros2ProcessModelTest
```

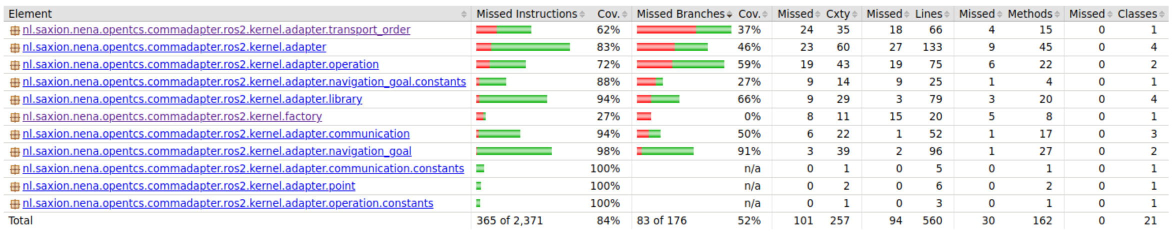
Unit testing is a software test method in which individual pieces of code are tested separately from each other. The purpose of the tests is to validate whether each software component functions as designed.

Following the same structure as the source code, unit tests have been developed for various parts of the ROS2 Driver. This also includes crucial elements such as the communication and Transport Order workflow. In addition to good-weather tests, a variety of bad-weather tests have been developed to ensure that code provides expected error messages when it should.

Fig. 12: An overview of all Unit test files.

7.2 Test Coverage Tooling

The aim is to have at least 80 per cent of the driver code covered by unit tests. In software engineering, 80 per cent is a common rule of thumb when it comes to testing coverage. Code coverage tooling was used to gain insight into the extent to which software components are covered with Unit tests. The JaCoCO tool was used for this.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.transport_order		62%		37%	24	35	18	66	4	15	0	1
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter		83%		46%	23	60	27	133	9	45	0	4
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.operation		72%		59%	19	43	19	75	6	22	0	2
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.navigation_goal.constants		88%		27%	9	14	9	25	1	4	0	1
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.library		94%		66%	9	29	3	79	3	20	0	4
nl.saxion.nena.opentcs.commadapter.ros2.kernel.factory		27%		0%	8	11	15	20	5	8	0	1
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.communication		94%		50%	6	22	1	52	1	17	0	3
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.navigation_goal		98%		91%	3	39	2	96	1	27	0	2
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.communication.constants		100%		n/a	0	1	0	5	0	1	0	1
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.point		100%		n/a	0	2	0	6	0	2	0	1
nl.saxion.nena.opentcs.commadapter.ros2.kernel.adapter.operation.constants		100%		n/a	0	1	0	3	0	1	0	1
Total	365 of 2,371	84%	83 of 176	52%	101	257	94	560	30	162	0	21

Fig. 13: Code test coverage per software package in the ROS2 driver.

7.3 Linter

A linter is static code analysis tooling that recognizes errors, bugs, duplicate code, style-related errors and strange programming structures. The IDE IntelliJ IDEA has Linter tooling, which was applied when developing the ROS2 driver. The standard rules for Java have been used for this.