openTCS 2.5

User manual

openTCS 2.5: User manual

by Stefan Walter

Publication date December 2013

Table of Contents

1. Introduction 1
    1. Purpose of the software 1
    2. System requirements 1
    3. Further documentation 1
    4. Questions and problem reports 1
2. System overview 2
    1. System components and structure 2
    2. Structure of driving course models 3
3. Operating the system 4
    1. Starting the system 4
        1.1. Starting in modelling mode 4
        1.2. Starting in plant operation mode 4
    2. Creating transport orders using the plant overview client 5
    3. Withdrawing transport orders using the plant overview client 6
    4. Step by step: Constructing a new driving course 6
        4.1. Starting components for driving course modelling 6
        4.2. Adding elements to the driving course model 6
        4.3. Saving the driving course model 8
    5. Step by step: Operating the system 8
        5.1. Starting components for system operation 8
        5.2. Configuring vehicle drivers 8
        5.3. Creating a transport order 9
        5.4. Continuous creation of random orders 9
        5.5. Removing a vehicle from a running system 9
    6. Step by step: Manipulating the system configuration 10
        6.1. Selecting the cost function used for routing 10
        6.2. Configuring automatic parking 10
        6.3. Configuring order pool cleanup 11
4. Interfaces to other systems 12
    1. Creating orders via TCP/IP 12
        1.1. XML telegrams for creating orders 12
        1.2. XML telegrams referencing order batches 13
        1.3. Receipts for created orders 13
        1.4. Receipts for order batches 14
    2. Status messages via TCP/IP 14
    3. XML schemas for telegrams and scripts 15
5. Customizing and integrating openTCS 16
    1. Integrating custom vehicle drivers 16
        1.1. Important classes and interfaces 16
        1.2. Creating a new vehicle driver 17
        1.3. Requirements for using a vehicle driver 17
    2. Customizing the appearance of locations and vehicles 17
    3. Loading a model on kernel startup 17
    4. Running kernel and plant overview on separate systems 18

Chapter 1. Introduction

1. Purpose of the software

openTCS is a control system software for track-guided vehicles, with tracks possibly being virtual. It was primarily developed for the coordination of automated guided vehicles (AGV), but it is generally conceivable to use it with other automatic vehicles like mobile robots or quadrocopters, as openTCS controls the vehicles independent of their specific characteristics like track guidance system or load handling device.

2. System requirements

- Standard PC with at least 512 MB main memory (required processing power of the CPU and actual memory requirement depending on size and complexity of the system to be controlled)
- Java Runtime Environment (JRE), at least version 1.7 (The directory `bin` of the installed JRE, for example `C:/Program Files/Java/jre1.7.0/bin`, should be included in the enviroment variable `PATH` to be able to use the included start scripts.)

3. Further documentation

For information about the respective openTCS version you use - including a changelog for comparison with earlier versions -, please refer to the file `README.html` included in the openTCS distribution.

If you want to extend and customize openTCS, please also see the JavaDoc documentation that is part of the openTCS distribution. In addition to the API documentation, it contains multiple short tutorials aiming primarily at developers.

4. Questions and problem reports

If you have questions about this manual, the openTCS project or about using or extending openTCS, please contact the development team by using the discussion forums at `http://sourceforge.net/projects/opentcs/` or by sending an e-mail to `<info@opentcs.org>`.

If you encounter technical problems using openTCS, please remember to include the following data in your problem report:

- The applications' log files, contained in the subdirectory `log/` of both the kernel and the plant overview application
- The driving course model you are working with, contained in the subdirectory `data/` of the kernel application

Chapter 2. System overview

1. System components and structure

openTCS consists of the following components running as separate processes and working together in a client-server architecture:

- Kernel (server process), running vehicle-independent strategies and drivers for controlled vehicles
- Clients
  - Plant overview for modelling and visualizing the course layout
  - Arbitrary clients for comunicating with other systems, e.g. for process control or warehouse management

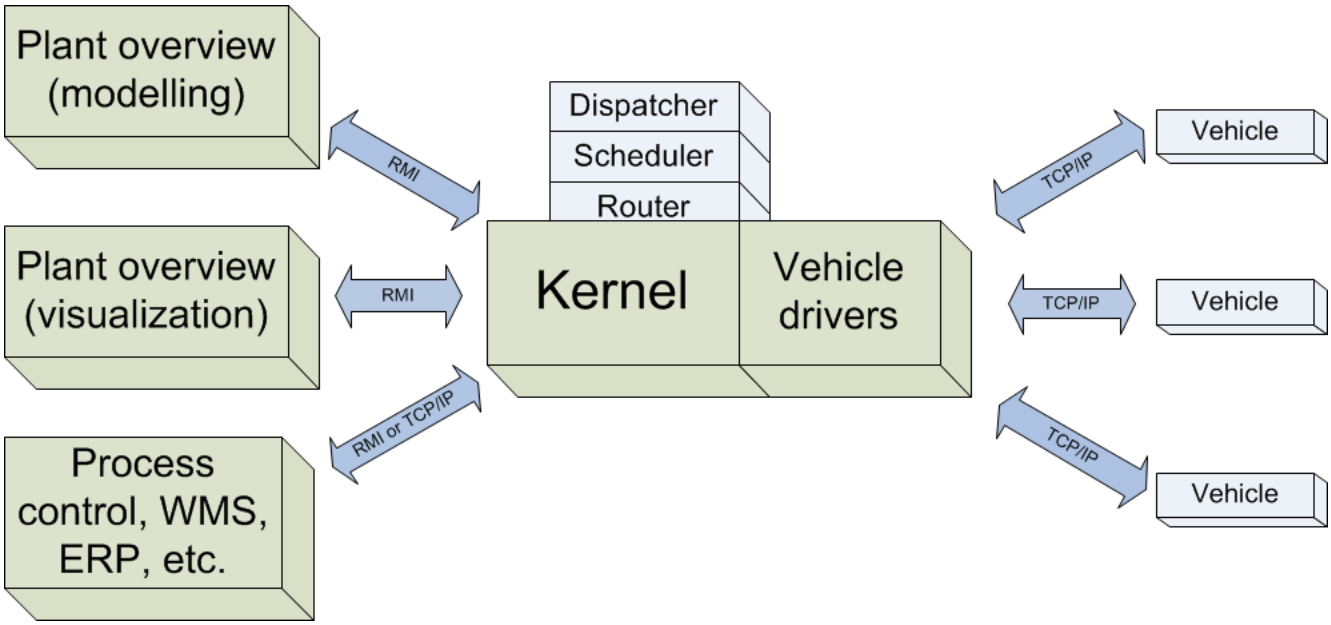

Figure 2.1. The structure of openTCS

The purpose of the openTCS kernel is to provide an abstract driving course model of a transportation system/ plant, to manage transport orders and to compute routes for the vehicles. Clients can communicate with this server process to, for instance, modify the driving course model, to visualize the driving course and the processing of transport orders and to create new transport orders. For user interaction, the kernel provides a graphical user interface titled *Kernel Control Center*.

The driver framework that is part of the openTCS kernel manages communication channels and associates vehicle drivers with vehicles. A vehicle driver is an adapter between kernel and vehicle and translates each vehicle-specific communication protocol to the kernel's internal communication schemes and vice versa. Furthermore, a driver may offer low-level functionality to the user via the kernel's graphical user interface, e.g. manually sending telegrams to the associated vehicle. By using suitable vehicle drivers, vehicles of different types can be managed simultaneously by a single openTCS instance.

The plant overview client that is part of the openTCS distribution allows editing of driving course models while the kernel is running in modelling mode. This includes, for instance, the definition of load-change stations, driving tracks and vehicles. In the kernel's plant operation mode, the plant overview client is used to display

the transportation system's general state and any active transport processes, and to create new transport orders interactively.

Other clients, e.g. to control higher-level plant processes, can be implemented and attached. For Java clients, the openTCS kernel provides an interface based on Java RMI (Remote Method Invocation). A host interface for creating transport orders using XML telegrams sent via TCP/IP connections is also available.

2. Structure of driving course models

A driving course model consists of the following elements:

- *Points* are logical mappings of discrete positions (reporting points) reported by a vehicle. In plant operation mode, vehicles move from reporting point to reporting point in the model.
- *Paths* are connections between reporting points that are navigable for vehicles.
- *Locations* are places at which vehicles may execute special operations (change their load, charge their battery etc.). To be reachable for any vehicle in the model, a location needs to be linked to at least one point.
- *Vehicles* map real vehicles for the purpose of visualizing their positions and other characteristics.

Furthermore, there is an abstract element that is only used indirectly:

- *Location types* group stations and define operations that can be executed by vehicles at these stations.

The attributes of these elements that are relevant for the driving course model, e.g. the coordinates of a reporting point or the length of a path, can be manipulated using the modelling client. Furthermore, it is possible to define arbitrary additional attributes as key-value pairs for all driving course elements, which for example can be read and evaluated by vehicle drivers or client software. Both the key and the value can be arbitrary character strings.

For example, a key-value pair `"IP address":"192.168.23.42"` could be defined for a vehicle in the model, stating which IP address is to be used to communicate with the vehicle; a vehicle driver could now check during runtime whether a value for the key `"IP address"` was defined, and if yes, use it to automatically configure the communication channel to the vehicle. Another use for these generic attributes can be vehicle-specific actions to be executed on certain paths in the model. If a vehicle should, for instance, issue an acoustic warning and/or turn on the right-hand direction indicator when currently on a certain path, attributes with the keys `"acoustic warning"` and/or `"right-hand direction indicator"` could be defined for this path and evaluated by the respective vehicle driver.

Chapter 3. Operating the system

1. Starting the system

To create or to edit the model of a transport system, openTCS has to be started in modelling mode. To use it as a transportation control system based on an existing model, it has to be started in plant operation mode. Starting a component is done by executing the respective shell script (Unix) or batch file (Windows).

1.1. Starting in modelling mode

1. Start kernel (`startKernel.bat`)

    a. Select existing model (that should be edited) from the list in the dialog window shown, select modelling mode and click *OK*, or click *Cancel* to work with a new, empty model

2. Start plant overview client (`startPlantOverview.bat`)

1.2. Starting in plant operation mode

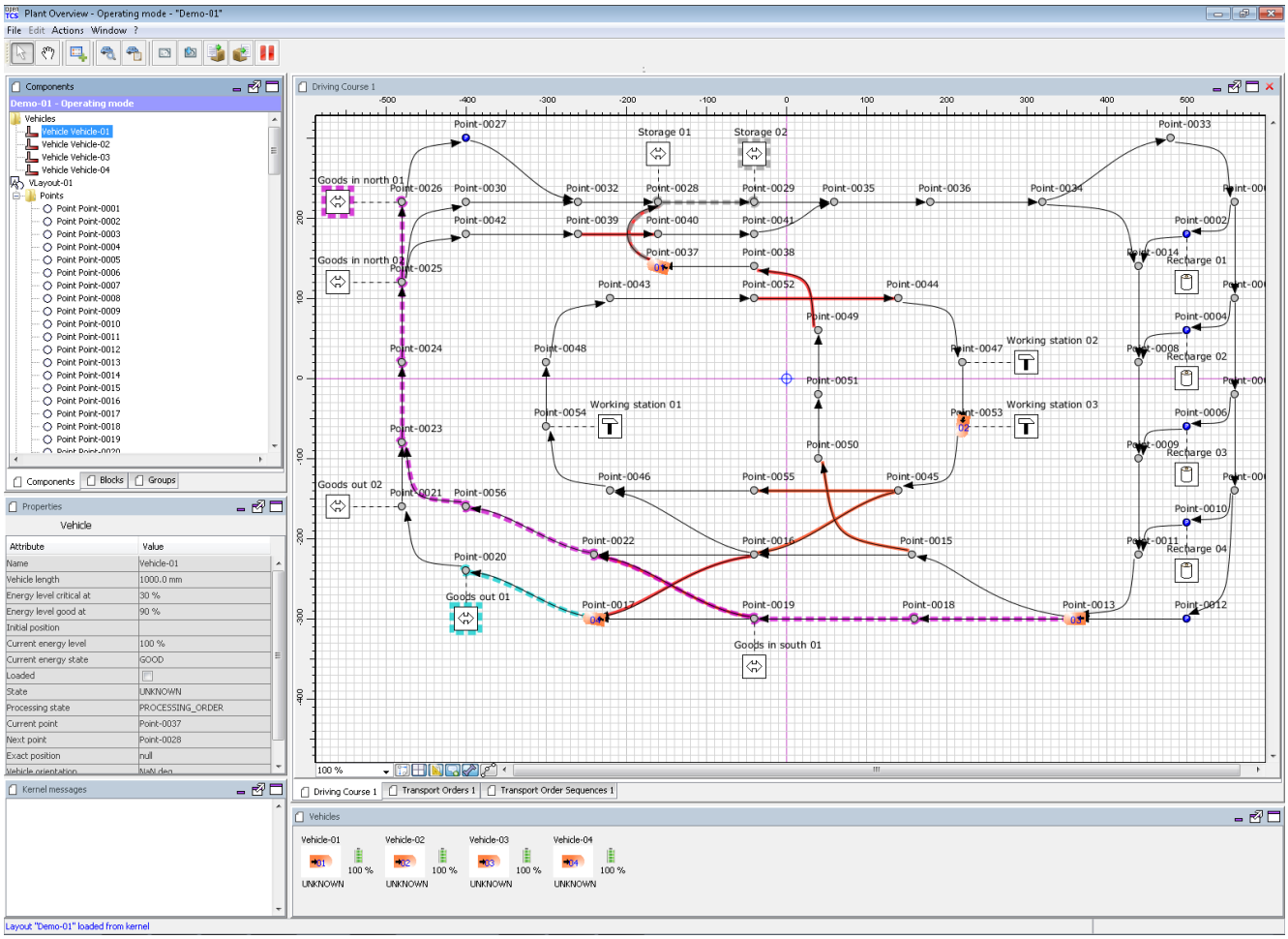

Figure 3.1. Plant overview client displaying driving course model

1. Start kernel (`startKernel.bat`)

a. Select existing model from the list in the dialog window shown, select plant operation mode and click *OK*.

2. Start plant overview client (`startPlantOverview.bat`)

3. Select tab *Vehicle drivers* in the kernel control center. Select, configure and start driver for each vehicle in the model

   a. The list on the left-hand side of the window shows all vehicles in the chosen model.

   b. A detailed view for a vehicle can be seen on the right-hand side of the driver panel after double-clicking on the vehicle in the list. The specific design of this detailed view depends on the driver associated with the vehicle. Usually, status information sent by the vehicle (e.g. current position and mode of operation) is displayed and low-level settings (e.g. for the vehicle's IP address) are provided here.

   c. Right-clicking on the list of vehicles shows a popup menu that allows to attach or detach drivers for selected vehicles.

   d. For a vehicle to be controlled by the system, a driver needs to be attached to the vehicle and enabled. (For testing purposes without real vehicles that could communicate with the system, the so-called loopback driver can be used, which provides a virtual vehicle or simulates a real one.)

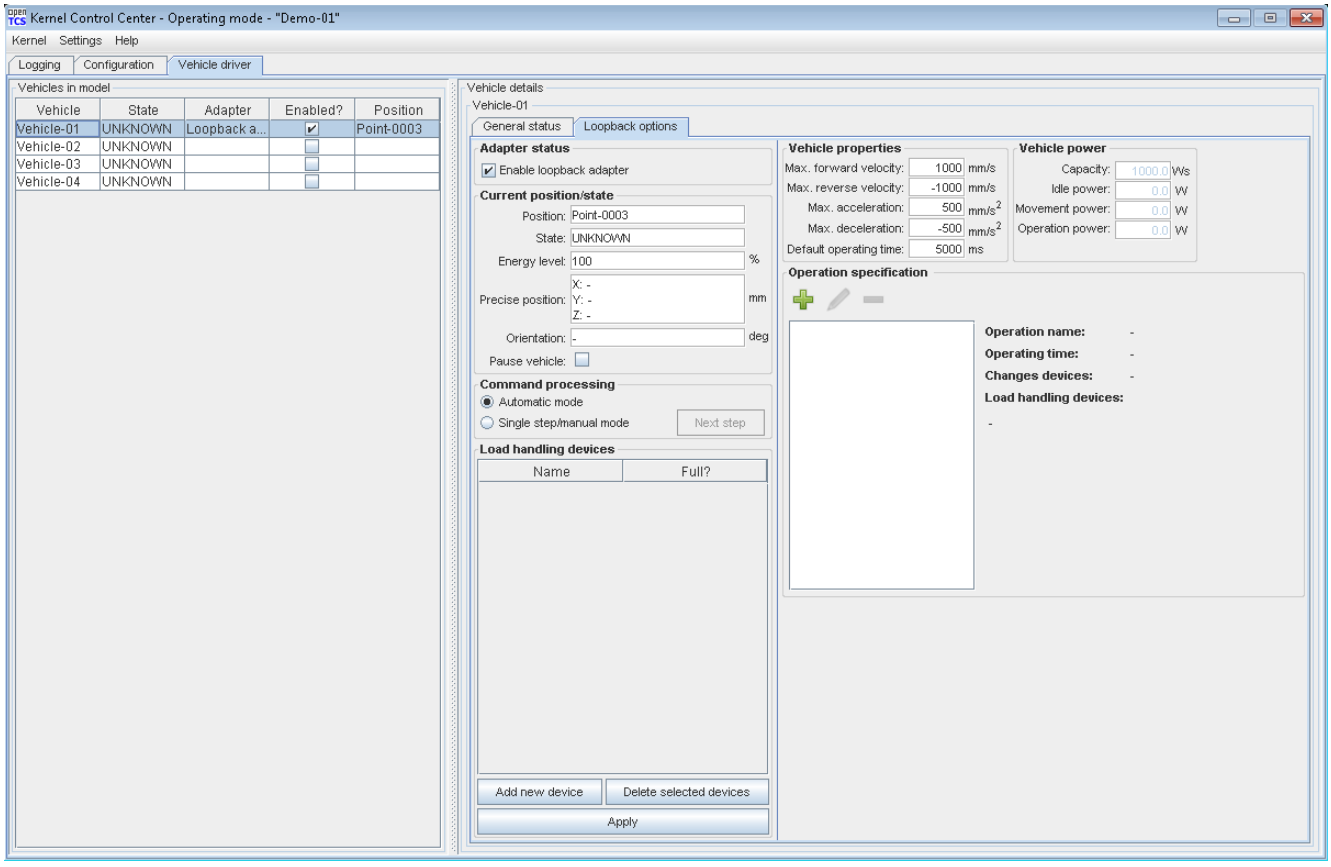

Figure 3.2. Driver panel with detailed view of a vehicle

2. Creating transport orders using the plant overview client

To create a transport order, the plant overview client provides a dialog window presented when selecting *Actions* → *Transport Order* in the menu. Transport orders are defined as a sequence of destination locations at which

actions are to be performed by the vehicle processing the order. The user can select the desired station and action from the dropdown menu which appears after pressing the *Add* button. The user may also optionally choose the vehicle for this order; alternatively, the guidance system automatically chooses the vehicle that will most likely finish the transport order the soonest. Furthermore, a transport order can be given a deadline specifying the point of time at which the order should be finished at the latest. This deadline will be considered when dispatching the transport orders in the pool.

3. Withdrawing transport orders using the plant overview client

A transport order can be withdrawn from a vehicle that is currently processing it. This can be done by right-clicking on the respective vehicle in the plant overview client and selecting *Withdraw Transport Order* in the context menu shown. The processing of the order will be cancelled and the vehicle (driver) will not receive any further drive orders. Processing of this transport order *cannot* be resumed later. Instead, a new transport order will have to be created.

4. Step by step: Constructing a new driving course

These step by step instructions roughly show how a new driving course model is created and filled with driving course elements so that it can eventually be used in plant operation mode.

4.1. Starting components for driving course modelling

1. Start kernel (`startKernel.bat`).

2. Wait until a dialog for selecting a driving course model is shown.

3. Click *Cancel* to open a new, empty model instead of loading an existing one, and to leave the kernel running in modelling mode.

4. Start the plant overview client (`startPlantOverview.bat`).

5. Wait until the graphical user interface of the plant overview client is shown.

4.2. Adding elements to the driving course model

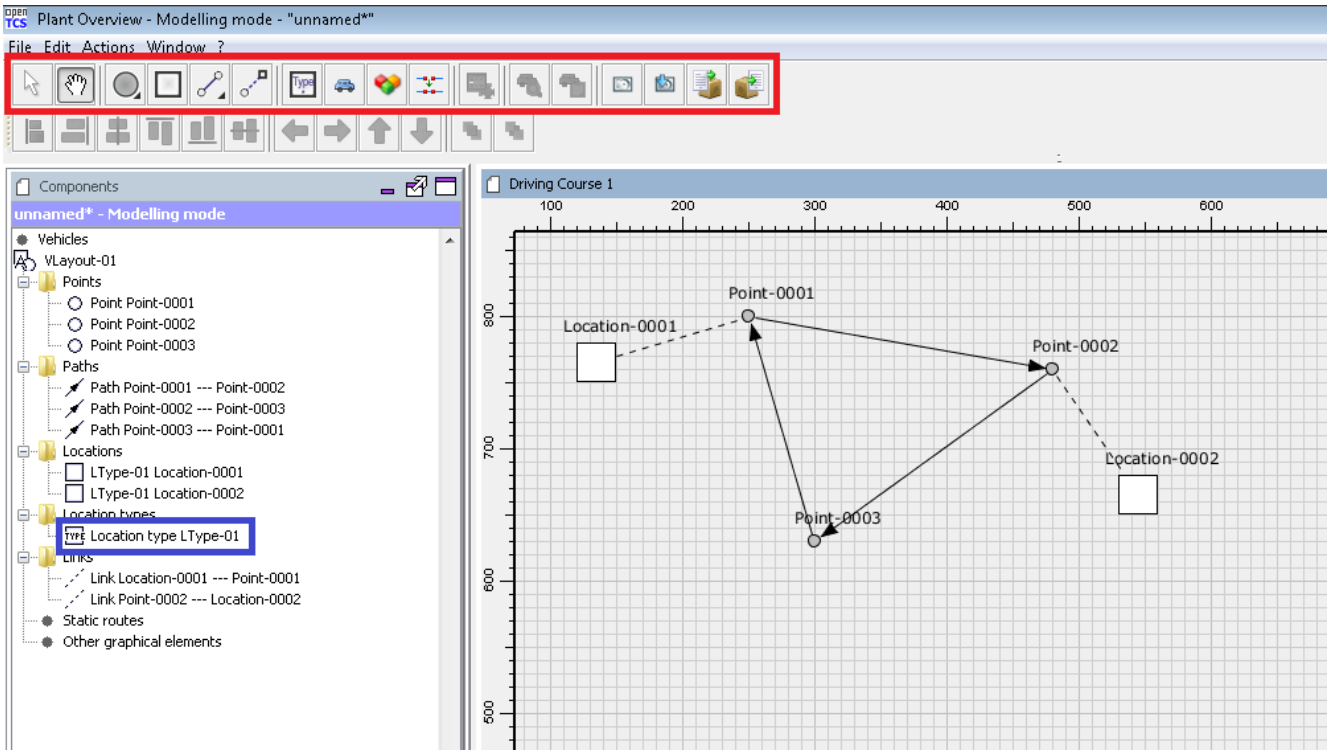

Figure 3.3. Control elements in the plant overview client (modelling mode)

1. Create three reporting points by selecting the point tool from the driving course elements toolbar (see red frame in Figure 3.3) and click on three positions on the drawing area.

2. Link the three points with paths to a closed loop by

   a. selecting the path tool by double-click.

   b. clicking on a point, dragging the path to the next point and releasing the mouse button there.

3. Create two stations by double-clicking the station tool and clicking on any two free positions on the drawing area. As a station type does not yet exist in the course model, a new one is created implicitly when creating the first station, which can be seen in the tree view to the left of the drawing area.

4. Link the two stations with (different) points by

   a. double-clicking on the link tool.

   b. clicking on a station, dragging the link to a point and releasing the mouse button.

5. Create a new vehicle by clicking on the vehicle button in the course elements toolbar.

6. Define the allowed operations for vehicles at the newly created stations by

   a. selecting the stations' type in the tree view to the left of the drawing area (see blue frame in Figure 3.3).

   b. clicking the value cell `"Actions"` in the property window below the tree view.

   c. entering the allowed actions as arbitrary text in the dialog shown, for instance `"Load cargo"` and `"Unload cargo"`.

d. Optionally, you can choose a symbol for stations of the selected type by editing the property `"Symbol"`.

4.3. Saving the driving course model

1. Select the menu entry *File* → *Save Model As...* and enter an arbitrary name for the model.
2. Close the plant overview client.
3. Shut down the kernel.

The newly created driving course model now contains a minimum of elements and can be used in operation mode.

5. Step by step: Operating the system

These step by step instructions show how the newly created model can be used in plant operation mode, how vehicle drivers are used and how transport orders can be created and processed by a vehicle.

5.1. Starting components for system operation

1. Start the kernel (`startKernel.bat`).
2. Wait until the dialog for selecting a driving course model is shown.
3. Select the model you created, select plant operation mode, and click *OK*.
4. Start the plant overview client (`startPlantOverview.bat`) and wait until its graphical user interface is shown.

5.2. Configuring vehicle drivers

1. Associate the vehicle with the loopback driver by right-clicking on the vehicle in the vehicle list of the driver panel and selecting the menu entry *Driver* → *Loopback adapter (virtual vehicle)*.
2. Open the detailed view of the vehicle by double-clicking on the vehicle's name in the list.
3. In the detailed view of the vehicle that is now shown to the right of the vehicle list, select the tab *Loopback options*.
4. Enable the driver by ticking the checkbox *Enable loopback adapter* in the *Loopback options* tab or the checkbox in the *Enabled?* column of the vehicle list.
5. In the *Loopback options* tab or in the vehicles list, select a point from the driving course model to have the loopback adapter report this point to the kernel as the (virtual) vehicle's current position. (In a real-world application, a vehicle driver communicating with a real vehicle would automatically report the vehicle's current position to the kernel as soon as it is known.)
6. In the vehicle driver's *Loopback options* tab, set the vehicle's state to `IDLE` to let the kernel know that the vehicle is now in a state that allows it to receive and process orders.
7. Switch to the plant overview client. An icon representing the vehicle should now be shown at the point on which you placed it using the loopback driver.
8. Right-click on the vehicle and select *Dispatch Vehicle* in the menu shown to allow the kernel to dispatch the vehicle. The vehicle is then available for processing orders, which is indicated by the *Processing state* `IDLE`

in the property panel at the bottom left of the plant overview client's window. (You can revert this by right-clicking on the vehicle and selecting *Withdraw TO and Disable Vehicle* in the context menu. The processing state shown is now `UNAVAILABLE` and the vehicle will not be dispatched for transport orders any more.)

5.3. Creating a transport order

1. Select the menu entry *Actions* → *Transport Order*.

2. In the dialog shown, click on the *Add* button and select a station as the destination and an operation which the vehicle should execute there. You can add an arbitrary number of destinations to the order this way. They will be processed in the given order.

3. After creating the transport order with the given destinations by clicking *OK*, the kernel will check for a vehicle that can process the order. If a vehicle is found, it is assigned the order immediately and the route computed for it will be highlighted in the plant overview client. The loopback driver simulates the vehicle's movement to the destinations and the execution of the operations.

5.4. Continuous creation of random orders

Important

The *generic client* used here is primarily intended to be used during software developement and for testing of individual functions of the system that are not accessible via the plant overview client, yet. The generic client is not designed to be very user-friendly or to prevent errors and can, if operated incorrectly, cause errors in the running openTCS instance. If necessary, such problems can be corrected by restarting openTCS.

1. Start the generic client (`startGenericClient.bat`).

2. Select the tab *Continuous load*.

3. Choose a trigger for creating new transport orders: New orders will either be created once only, or if the number of active orders in the system drops below a specified limit, or after a specified timeout has expired.

4. By using an *Order profile* you may choose if the transport orders' destinations should be selected randomly or if you want to select them yourself.

   Using *Create orders randomly*, you define the number of transport orders that are to be generated at a time, and the number of destinations a single transport order should contain. Since the destinations will be selected randomly, the orders created might not necessarily make sense for a real-world system.

   Using *Create orders according to definition*, you can define an arbitrary number of transport orders, each with an arbitrary number of destinations and properties, and save and load your list of transport orders.

5. Start the order generator by activating the corresponding checkbox at the bottom of the *Continuous load* tab. The load generator will then generate transport orders according to its configuration until the checkbox is deactivated.

5.5. Removing a vehicle from a running system

There may be situations in which you want to remove a single vehicle from a system, e.g. because the vehicle temporarily cannot be controlled by openTCS due to a hardware defect that has to be dealt with first. The following steps will ensure that no further transport orders are assigned to the vehicle and that the resources it might still be occupying are freed for use by other vehicles.

1. In the plant overview client, right-click on the vehicle and select *Withdraw TO and Disable Vehicle* to disable the vehicle for transport order processing.

2. In the kernel control center, disable the vehicle's driver by unticking the checkbox *Enable loopback adapter* in the *Loopback options tab* or the checkbox in the *Enabled?* column of the vehicle list.

3. In the kernel control center, right-click on the vehicle in the vehicle list and select *Reset vehicle position* from the context menu to free the point in the driving course that the vehicle is occupying.

6. Step by step: Manipulating the system configuration

These step by step instructions demonstrate how system parameters that influence e.g. the routing or parking strategies, can be manipulated.

6.1. Selecting the cost function used for routing

1. Switch to the kernel control center's *Configuration* tab.

2. Find and select the configuration entry `org.opentcs.kernel.module.routing.DijkstraRouter.costType` and click the *Configure* button.

3. In the dialog shown, set the configuration item's value to one of the following:

   - `LENGTH_BASED` (default): Routing costs are based on the lengths of paths travelled.

   - `TIME_BASED`: Routing costs are based on the time required for travelling. The time is computed using the length of a path and the maximum speed with which a vehicle may move on it.

   - `EXPLICIT`: Routing costs are explicitly specified by the modelling user. They can be specified for every single path in the model using the plant overview client. (Select a path and set its *Costs* property to an arbitrary integer value.)

4. Shut down and restart the kernel for the changes to take effect.

6.2. Configuring automatic parking

6.2.1. Activating/deactivating automatic parking of idle vehicles

1. Switch to the kernel control center's *Configuration* tab.

2. Find and select the configuration entry `org.opentcs.kernel.module.dispatching.OrderSequenceDispatcher.parkIdleVehicles` and click the *Configure* button.

3. Set the configuration item's value to `true` (activated) or `false` (deactivated).

6.2.2. Select a parking strategy

1. Switch to the kernel control center's *Configuration* tab.

2. Find and select the configuration entry `org.opentcs.kernel.module.dispatching.OrderSequenceDispatcher.parkingStrategy` and click the *Configure* button.

3. You may choose from the following parking strategies:

   - `OffRouteParkingStrategy` (default): With this parking strategy, idle vehicles will be sent to the parking position that is unoccupied and closest to their current position.

- `PriorityParkingStrategy`: With this parking strategy, idle vehicles will be sent to the parking position that is unoccupied and has the highest priority. Parking priorities can be manipulated with the plant overview client.

  To do this, select a point that is marked as parking position. In the property view, click on the *Miscellaneous* property's value cell, which will show a dialog for editing arbitrary properties as key-value pairs. To assign a priority to the parking position, you may set two properties with the following keys:

  - `tcs:parkPriorityGroup`: The group priority.
  - `tcs:parkPriority`: The point priority.

  The values should be integer values representing the respective priorities, where lower values represent higher priorities. Among all unoccupied parking positions, the parking strategy will choose the one with the highest priority in the group with the highest priority.

  If you do not define any priorities or if two unoccupied points have the same (highest) priority, the one closest to the vehicle's position is chosen.

6.3. Configuring order pool cleanup

By default, openTCS checks in intervals of ten minutes if the number of finished transport orders in the pool exceeds 200. If this is the case, the oldest of these orders are removed from the pool until only 200 are left. To customize this behaviour, the following steps are required:

- Switch to the kernel control center's *Configuration* tab.
  - Find and select the configuration entry `org.opentcs.kernel.OrderCleanerTask.orderSweepInterval`. The default value is 600000 (milliseconds, corresponding to an interval of 10 minutes). Set this value according to your needs.
  - Find and select the configuration entry `org.opentcs.kernel.OrderCleanerTask.orderSweepThreshold`. The default value is 200 (orders to be kept in the pool). Set this value according to your needs.
  - If you do not want to clean up old orders when their number passes a certain threshold, but want to remove them depending on their age, do the following:
    1. Find and select the configuration entry `org.opentcs.kernel.OrderCleanerTask.orderSweepType`. Its default value is `BY_AMOUNT`, which means that orders will be cleaned up when reaching a certain amount. Change this value to `BY_AGE`. This will remove finished transport orders once they have passed a certain age.
    2. Finally, find and select the configuration entry `org.opentcs.kernel.OrderCleanerTask.orderSweepAge` to change the maximum age of finished orders. The default value is 3600000 (milliseconds, corresponding to one hour that a finished order should be kept in the pool). Set this value according to your needs.

Chapter 4. Interfaces to other systems

openTCS offers the following interfaces for communication with other systems:

- An RMI interface providing access to all functions of the kernel
- A bidirectional interface via a TCP/IP connection for the creation of transport orders
- An unidirectional interface via a TCP/IP connection for receiving status messages, e.g. about transport orders being processed

The RMI interface is described in the API documentation of the openTCS distribution. The descriptions of the interface `org.opentcs.access.Kernel` and the class `org.opentcs.access.rmi.DynamicRemoteKernelProxy` as well as the package `org.opentcs.data.order` are good points to get started.

The TCP/IP interfaces are described in the following sections.

1. Creating orders via TCP/IP

For creating transport orders, the openTCS kernel accepts connections to a TCP port (default: port 55555). The communication between openTCS and the host works as follows:

1. The host establishes a new TCP/IP connection to openTCS.
2. The host sends a single XML telegram (described in detail in Section 1.1 and Section 1.2) which either describes the transport orders to be created or identifies batch files that are available with the kernel and that contain the transport order descriptions.
3. The host closes its output stream of the TCP/IP connection or sends two consecutive line breaks (i.e. "`\r\n \r\n`"), letting the kernel know that no further data will follow.
4. openTCS interprets the telegram sent by the host, creates the corresponding transport orders and activates them.
5. openTCS sends an XML telegram (described in detail in Section 1.3) to confirm processing of the telegram.
6. openTCS closes the TCP/IP connection.

The following points should be respected:

- Multiple sets of transport orders are not intended to be transferred via the same TCP connection. After processing a set and sending the response, openTCS closes the connection. To transfer further sets new TCP/IP connections need to be established by the peer system.
- openTCS only waits a limited amount of time (default: ten seconds) for incoming data. If there is no incoming data from the peer system over a longer period of time, the connection will be closed by openTCS without any transport orders being created.
- The maximum length of a single XML telegram is limited to 100 kilobytes by default. If more data is transferred, the connection will be closed without any transport orders being created.

1.1. XML telegrams for creating orders

Every XML telegram sent to openTCS via the interface described above can describe multiple transport orders to be created. Every order element must contain the following data:

- A string identifying the order element. This string is required for unambiguous matching of receipts (see Section 1.3) and orders.
- A sequence of destinations and destination operations defining the actual order.

Furthermore, each order element may contain the following data:

- A deadline/point of time at which the order should be finished.
- The name of a vehicle in the system that the order should be assigned to. If this information is missing, any vehicle in the system may process the order.
- A set of names of existing transport orders that have to be finished before the new order may be assigned to a vehicle.

Figure 4.1 shows how an XML telegram for the creation of two transport orders could look like.

```
<?xml version="1.0" encoding="UTF-8"?>
<tcsOrderSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <order deadline="2008-10-30T12:14:48.717+01:00" id="TransportOrder-01"
         intendedVehicle="Vehicle-01" xsi:type="transport">
    <destination locationName="Storage 01" operation="Load cargo"/>
    <destination locationName="Storage 02" operation="Unload cargo"/>
  </order>
  <order id="TransportOrder-02" xsi:type="transport">
    <destination locationName="Working station 01" operation="Drill"/>
    <destination locationName="Working station 02" operation="Drill"/>
    <destination locationName="Working station 03" operation="Cut"/>
  </order>
</tcsOrderSet>
```

Figure 4.1. XML telegram for the creation of two transport orders

1.2. XML telegrams referencing order batches

Alternatively, an XML telegram may also reference order batches which are kept in files on the openTCS system. The (parameters of the) transport orders to be created will then be read from the referenced batch files. A batch file may contain/create an arbitrary number of transport orders and needs to be placed in the kernel application's subdirectory `scripts`. In the openTCS distribution, this directory already contains a couple of templates for batch files (`template.tcs` and `test.tcs`).

Figure 4.2 shows an example of an XML telegram referencing a batch file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsOrderSet>
  <order xsi:type="transportScript" fileName="test.tcs" id="test.tcs"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</tcsOrderSet>
```

Figure 4.2. XML telegram referencing a batch file

1.3. Receipts for created orders

In response to an XML telegram for the creation of transport orders, an XML telegram will be sent back to the peer, reporting the operation's outcome. In the response telegram, every order element of the original telegram will be referenced by a response element with the same ID. Furthermore, every response element contains:

- A flag reflecting the success of creating the respective order
- The name that openTCS internally assigned to the created order. (This name is relevant for interpreting the messages on the status channel - see Section 2.)

Figure 4.3 shows how a response to the telegram in Figure 4.1 could look like.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsResponseSet>
    <response xsi:type="transportResponse" executionSuccessful="true"
              orderName="TOrder-0001" id="TransportOrder-01"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
    <response xsi:type="transportResponse" executionSuccessful="true"
              orderName="TOrder-0002" id="TransportOrder-02"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</tcsResponseSet>
```

Figure 4.3. XML telegram with receipts for created orders

1.4. Receipts for order batches

For referenced order batches, receipts will be sent back to the peer, too. The response contains an element for every batch file referenced by the peer. If the batch file was successfully read and processed, a response for every single order definition it contains will be included.

Figure 4.4 shows a possible response to the batch file reference in Figure 4.2. In this case, the batch file contains two transport order definitions which have been processed successfully.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsResponseSet>
    <response xsi:type="scriptResponse" parsingSuccessful="false"
              id="test.tcs"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <transport orderName="TOrder-0003" executionSuccessful="true"
                   id="test.tcs"/>
        <transport orderName="TOrder-0004" executionSuccessful="true"
                   id="test.tcs"/>
    </response>
</tcsResponseSet>
```

Figure 4.4. XML telegram with receipts for orders in batch file

2. Status messages via TCP/IP

To receive status messages for transport orders in the system, connections to another TCP port (default: port 44444) may be established. Whenever the state of a transport order changes, an XML telegram will be sent to each connnected client, describing the new state of the order. Each of these telegrams is followed by a string that does

not appear in the telegrams themselves (by default, a single pipe symbol: "|"), marking the end of the respective telegram. Status messages will be sent until the peer closes the TCP connection.

The following points should be respected:

- From the peer's point of view, connections to this status channel are purely passive, i.e. openTCS does not expect any messages from the peer and will not process any data received via this connection.
- A peer needs to filter the received telegrams for relevant data itself. The openTCS kernel does not provide any filtering of status messages for clients.
- Due to concurrent processes within openTCS, it is possible that the creation and activation of a transport order and its assignment to a vehicle is reported via the status channel before the peer that created the order receives the corresponding receipt.

Figure 4.5 shows a status message as it would be sent via the status channel after the first of the two transport orders defined in Figure 4.1 has been created and activated.

```
<?xml version="1.0" encoding="UTF-8"?>
<tcsStatusMessageSet timeStamp="2008-10-31T09:49:38.177+01:00"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <statusMessage orderName="TOrder-0001" orderState="ACTIVE"
                 xsi:type="orderStatusMessage">
    <destination locationName="Storage 01" operation="Load cargo"
                 state="PRISTINE"/>
    <destination locationName="Storage 02" operation="Unload cargo"
                 state="PRISTINE"/>
  </statusMessage>
</tcsStatusMessageSet>
```

Figure 4.5. Status message for the generated order

3. XML schemas for telegrams and scripts

XML schemas describing the expected structure of XML order telegrams and order batch files as well as the structure of receipt telegrams as sent by openTCS are part of the openTCS distribution and can be found in the directory containing the documentation.

Chapter 5. Customizing and integrating openTCS

1. Integrating custom vehicle drivers

openTCS supports dynamic integration of vehicle drivers that implement vehicle-specific communication protocols and thus mediate between the kernel and the vehicle. Due to its function, a vehicle driver is also called a communication adapter. The following sections describe which requirements must be met by a driver and which steps are necessary to create and use it. A basic prerequisite for the integration of drivers is their implementation in the Java programming language. Therefore, these instructions are directed primarily at developers who are familiar with programming in Java. They are also primarily a rough guide, while the implementation details can be found in the API documentation.

1.1. Important classes and interfaces

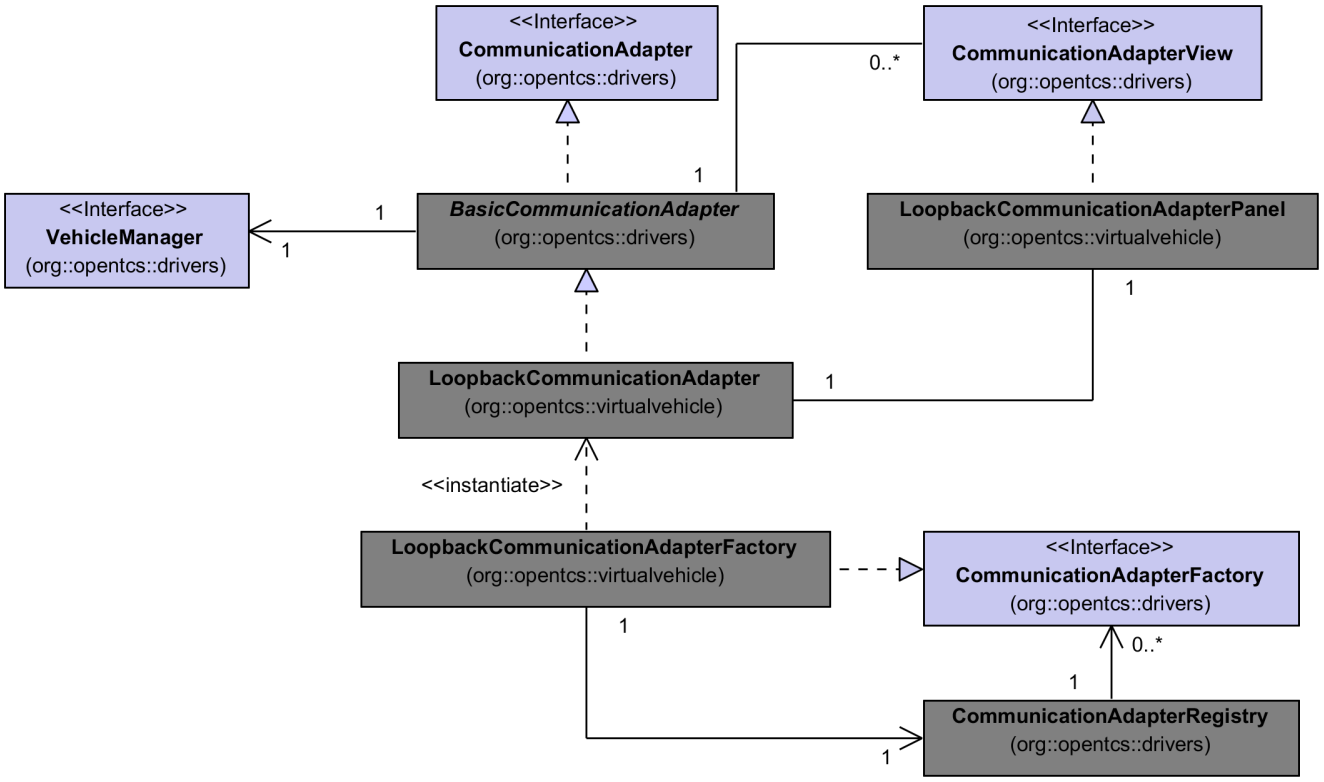

Figure 5.1. Structure of driver classes

openTCS defines some Java classes and interfaces that are relevant for the development of vehicle drivers. These classes and interfaces are part of every openTCS distribution and are (among others) included in the JAR file `openTCS-Base.jar`. The most important classes and interfaces are the following:

- `org.opentcs.drivers.CommunicationAdapter` declares methods that every driver must implement. These methods are called by the kernel, for instance when a vehicle is supposed to move to the next position in the driving course.
- `org.opentcs.drivers.VehicleManager` offers methods that the driver may call when certain events occur, e.g. to report a change of the vehicle's position.

- `org.opentcs.drivers.BasicCommunicationAdapter` is the base class for all drivers. Every driver implemented needs to be derived from this class. It allows the integration of the driver into the driver framework and the graphical user interface. Furthermore, it includes sensible default implementations for some of the methods declared by `CommunicationAdapter`. Only those methods concerning the vehicle-specific communication protocol are declared as abstract and thus must be implemented by subclasses.
- `org.opentcs.drivers.CommunicationAdapterView` needs to be implemented by all driver-specific panels that are to be shown in the driver application and whose contents depend on the state of the respective `CommunicationAdapter`. Calls to the method `update()` inform the panel that the state of the driver or of the vehicle has changed and the graphical user interface may need to be updated.
- `org.opentcs.drivers.CommunicationAdapterFactory` declares methods that have to be implemented by the factory class of the driver. The factory class creates and configures instances of the actual `CommunicationAdapter` implementation before they are made available to the driver application.
- `org.opentcs.drivers.CommunicationAdapterRegistry` is the central registry for all factory classes. A factory class is found automatically by the registry if it is in the Java class path and has been declared as an implementation of the service `org.opentcs.drivers.CommunicationAdapterFactory`. Only factory classes that are declared as service implementations and found in the class path will be offered for association with a vehicle by the driver framework. (See also the documentation for the class `java.util.ServiceLoader` in the Java standard class library.)

  Figure 5.1 shows the classes' relations in a concrete driver implementation, the loopback driver.

1.2. Creating a new vehicle driver

See the API documentation for package `org.opentcs.drivers`. It contains some details about the implementation steps to create a new vehicle driver.

1.3. Requirements for using a vehicle driver

See the API documentation for package `org.opentcs.drivers`. It contains some details about what to do for a custom vehicle driver to be recognized and integrated at runtime.

2. Customizing the appearance of locations and vehicles

Locations and vehicles are visualized in the plant overview client using pluggable themes. To customize the appearance of locations and vehicles, new theme implementations can be created and integrated into the plant overview client. Instructions for creating such a location or vehicle theme can be found in the API documentation for package `org.opentcs.util.gui`.

3. Loading a model on kernel startup

When running the kernel using the startup script that is part of the openTCS distribution (`startKernel.bat` or `startKernel.sh`, depending on the operating system), a dialog is shown that allows you to select the driving course model to be loaded and the desired kernel mode (modelling or operating the system). Alternatively, you can let the kernel load a specific model on startup without any user interaction by doing the following:

1. Open the kernel's startup script in a text editor.
2. Find the line containing the kernel startup parameter `-choosemodel`.
3. Replace `-choosemodel` with `-loadmodel MODELNAME`, where `MODELNAME` is the name of the model you want the kernel to load on startup.

4. Running kernel and plant overview on separate systems

The kernel and the plant overview client communicate via Java's Remote Method Invocation (RMI) mechanism. This makes it possible to run the kernel and the plant overview client on separate systems, as long as a network connection between these systems exists and is usable.

To connect a plant overview client to a kernel running on a remote system, do the following:

1. Open the plant overview client's startup script that comes with the openTCS distribution (`startPlantOverview.bat` or `startPlantOverview.sh`, depending on the operating system) in a text editor.

2. Find the line containing the Java VM startup parameter `-Dopentcs.kernel.host=localhost`.

3. With this parameter, change `localhost` to one of the following:

   - If you want the plant overview client to always connect to a kernel on a specific host, replace `localhost` with the host name or IP address of this host.

   - If you do not know the host name or IP address of the system the kernel will be running on in advance, simply remove `localhost` from the parameter, leaving only `-Dopentcs.kernel.host=`. This will result in a dialog being shown every time the plant overview client is started, allowing you to enter the host name or IP address to be used for that session.