Chapter 9. Rails and trains

The AnyLogic *Rail Library* allows you to efficiently model, simulate and visualize any kind of rail transportation of any complexity and scale. Classification yards, rail yards of large plants, railway stations, rail car repair yards, subways, airport shuttle trains, rail in container terminals and factories, trams, or even rail transportation in a coal mine can be easily yet accurately modeled with this library.

The Rail Library integrates well with other AnyLogic libraries. This means you can readily combine rail models with the types of models supported by these libraries, namely: auto transport, cranes, ships, passenger flows, warehouses, oil refineries, manufacturing or business processes, and so on.

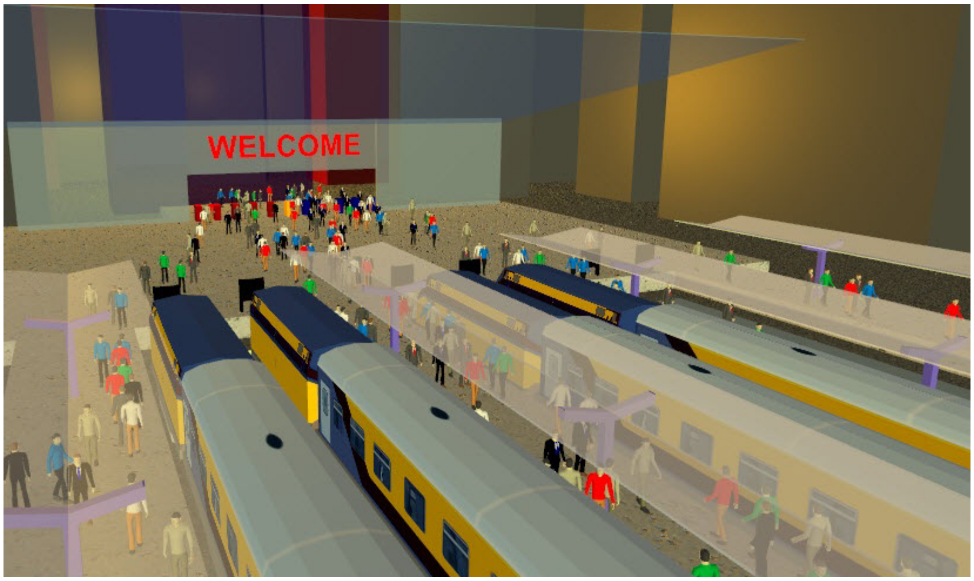


Figure 9.1 A Railway station model – Rail Library works together with Pedestrian Library

While the library supports detailed and accurate modeling (dimensions of individual cars, exact topology of tracks and switches, accelerations and decelerations of trains), the simulations it produces are of very high performance. This is particularly important when you use the optimizer to identify the best management policies or most efficient operations.

The Rail Library supports 2D and 3D animation of railway tracks, switches, and rail cars. The **3D Objects** palette contains ready-to-use 3D objects for locomotives, several types of freight cars and passenger cars of most common dimensions.

Since all AnyLogic libraries support 3D animation, you can easily create dynamic 3D models of systems where rail transportation is combined with processes simulated with other libraries, see Figure 9.1.

The two main components of a rail model are the rail topology and the operation logic. We will now discuss these in turn.

9.1. Defining the rail topology

The rail topology is defined by specific space markup shapes – *railway tracks* and *switches*. Railway networks can either be drawn manually using the AnyLogic graphical editor or created programmatically, see Example 9.5: "Creating a rail yard by code").

Switches are created automatically when you connect railway tracks following these guidelines:

1. Each switch must contain exactly three railway track ends (see Figure 9.2)
2. At each switch there must be at least two obtuse angles out of three between the track ends. The switch determines the routes based on those angles.

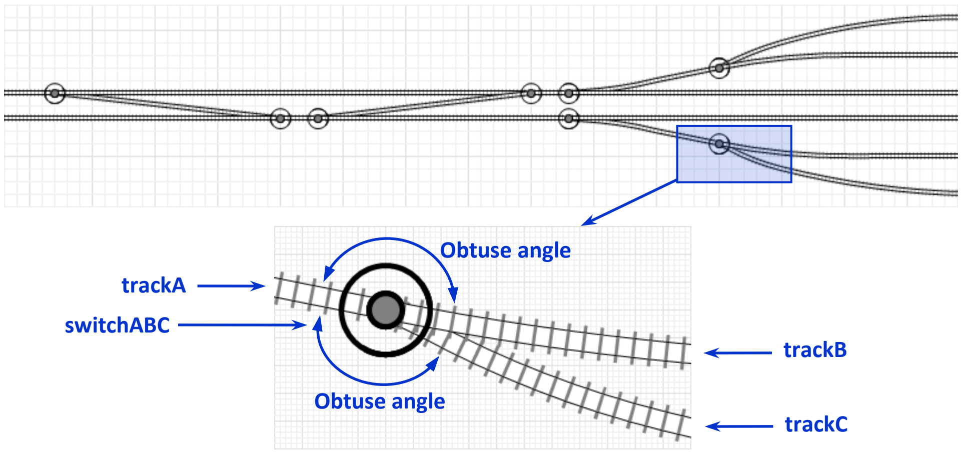


Figure 9.2 A fragment of rail topology near a railway station. Defining a switch

Train collisions at switches are detected automatically and errors are signaled. But note that the Rail Library will not automatically detect track crossings (i.e. places where two tracks cross each other without a switch), and it is the user's responsibility to make sure there are no train collisions in such places.

If you can find the GIS shapefile with the data on the existing railways, you can avoid drawing tracks manually and simply convert the shapefile into the AnyLogic railway tracks and switches see Section “Converting GIS shapefiles to rail space markup shapes”.

Alternatively, if a rail yard is drawn manually it may be convenient to have a CAD drawing or an image of the yard as a background, lock it, and draw the railway tracks above the image.

Railway track

The track has a start point and an end point, and therefore has an orientation. Railway tracks can have multiple segments, both linear and curved.

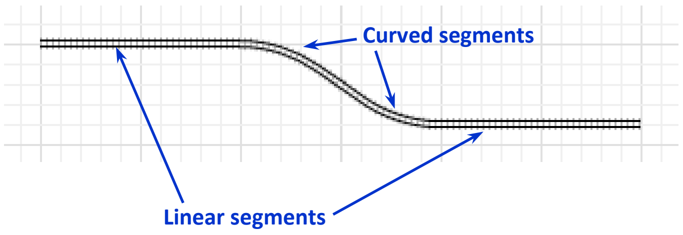


Figure 9.3 Railway track

▶ To draw a railway track

1. Double-click the **Railway Track** element in the **Rail Library** palette.
2. Click in the graphical editor where the start point of the track should be placed.
3. To add linear segments, continue clicking to place the end points.
4. To draw a curved track segment, press the mouse button in the graphical editor where the end point of the segment should be placed. Hold the mouse button pressed and move the mouse. Having achieved the intended curve, release the mouse button.
5. To finish drawing the railway track, double-click the mouse button.

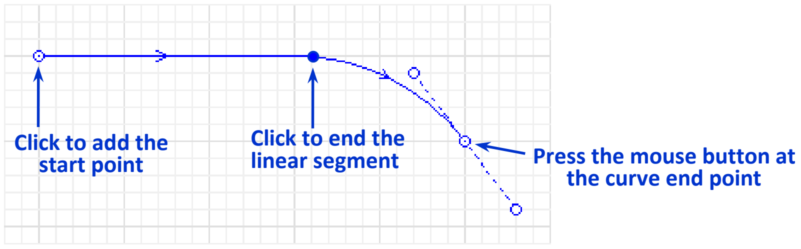

Click to add the start point
Click to end the linear segment
Press the mouse button at the curve end point

Figure 9.4 Drawing a curved segment

▶ To straighten a curved railway track segment

1. Right-click the railway track and select **Make Segment Linear/Arc-Based** from the context menu.
2. Click the segment whose shape you want to change.
3. Right-click the track and deselect the **Make Segment Linear/Arc-Based** option to switch off this editing mode.

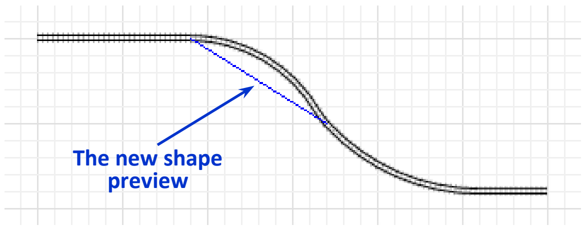


Figure 9.5 Straightening the curved segment

▶ To curve a linear railway track segment

1. Right-click the railway track and select **Edit Using Guiding Lines** from the context menu. The guiding lines will appear for editing points.
2. Drag the guiding line's handle to change the curve of the adjacent segments. To make the curve smooth, extend the guiding line.
3. Right-click the track and deselect the **Edit Using Guiding Lines** to switch off this editing mode.

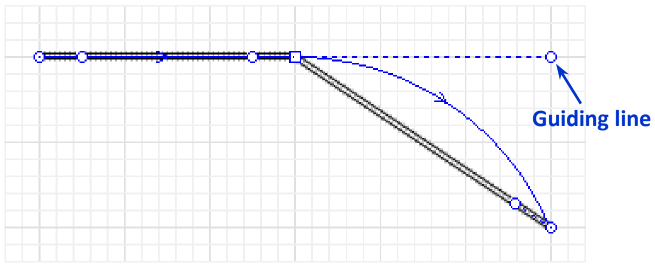


Figure 9.6 Curving a linear segment

Using the guiding lines, you can adjust the shape of the curved track segments.

▶ To add a segment to the railway track

1. Right-click the railway track and select **Append Line** from the context menu.
2. Click the track's end point where you want to append a line.
3. You are now in the drawing mode. You can add as many new segments as you need, both linear and curved.
4. Put the final point of the railway track with the double-click.

The exact position on the track is defined by the distance from the start point of the track, often referred as *offset*. To graphically define some exact point on the track, use the **Position on track** element.

The track knows about the presence of switches at the either end, if there are any. If there is no switch at one of the sides (an *open-ended track*), and a rail car exits the track at that side, the car leaves the rail yard model.

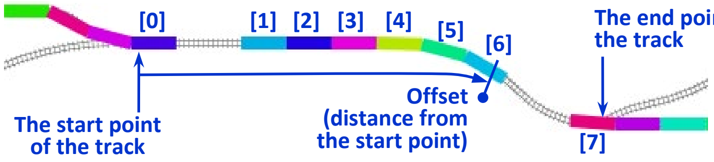

[0]
[1]
[2]
[3]
[4]
[5]
[6]
The end point of the track
Offset (distance from the start point)
The start point of the track
[7]

Figure 9.7 Trains on a railway track at runtime

The track knows all cars that are (fully or partially) located on it, and you can obtain those cars using the **RailSettings** block (if you create trains of a custom type, see Section "Custom train types") or **RailwayTrack** object API. The track in Figure 9.7 will say that it has 8 cars on it and will return them by index as shown.

For each continuous track of the yard with no switches in the middle, AnyLogic creates a Java object of class **RailwayTrack**. In some cases, the **RailwayTrack** API might be useful.

Here are some functions of class **RailwayTrack**, see *Library Reference Guides: Rail Library* (The AnyLogic Company, 2020) for the full list:

- **boolean isEmpty()** – tests if the track is empty, i.e. there are no cars that are (even partially) on the track. Returns **true** if the track is empty, **false** otherwise.
- **int nCars()** – returns the number of cars on the track (including those that are only partially on this track).
- **Agent getCar(int index)** – returns a car on the track at a given position (**index**) counted from the beginning of the track, or null if there is no such car. All cars count: moving, standing, coupled, and cars that are only partially on this track.
- **double getFreeSpace(boolean fromstart)** – tests the availability of space on the track. If there are no cars on the track, returns **infinity**. If there are cars, returns the distance from the track start or end point (depending on the parameter **fromstart**) to the nearest car. If there is a car partially entered or exited the track at a given side, returns a negative value.

- **RailwaySwitch getSwitch(boolean atend)** – returns the switch at the beginning or at the end of the track, depending on **atend** parameter.
- **double length()** – returns the length of the track in meters.

Railway switch

Railway switch models a two-way railroad switch that connects three tracks: 0, 1, and 2. Depending on its state, the switch will direct the trains coming from track 0 (*face point movement*) to either track 1 or track 2. The trains coming from track 1 or 2 (*trailing point movement*) will always proceed to track 0 regardless of the state of the switch and will always force the switch to the corresponding state.

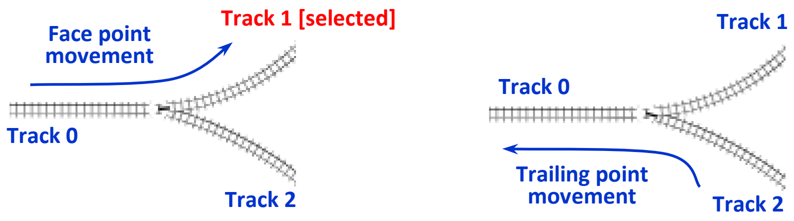


Figure 9.8 Railway switch

You can toggle the switch state directly by clicking it at model runtime in 2D animation or via API.

When you are drawing the railyard, AnyLogic automatically creates railway switches where required, see Section 9.1. This makes the **Railway Switch** element presence in the palette redundant.

▶ To connect two railway tracks with a switch

1. Double-click the **Railway Track** element in the **Rail Library** palette.
2. Click on the railway track in the graphical editor where you want to connect another railway track.
3. Draw the track and place the end point with a double-click of the mouse button. The railway switch will appear at the connection point.

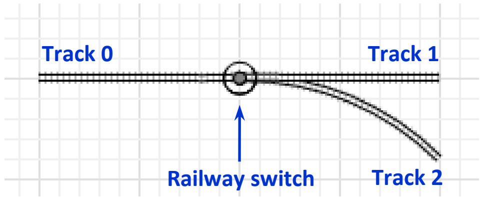


Figure 9.9 Railway switch

Note, that the creation of the switch divides the initial railway track into two separate tracks.

For each railway switch, AnyLogic creates a Java object of class **RailwaySwitch**. You won't need to use the objects of class **RailwaySwitch** and their functions too often; the Rail Library block **TrainMoveTo** can set up the switch states automatically when the train moves along a defined route. However, in some cases the **RailwaySwitch** API might be useful. Here are some functions of class **RailwaySwitch**:

- **RailwayTrack getSelectedTrack()** – returns the currently selected track (track 1 or 2).
- **setSelectedTrack(RailwayTrack track)** – selects a given **track** (which should be either 1 or 2). If a car is over the switch, signals error.
- **toggle()** – toggles the selected tracks.
- **getToggleCount()** – returns the number of switch toggles up to this moment: both automatic and initiated by the user during the model run.
- **RailwayTrack nextTrack(RailwayTrack from)** – based on the state of the switch, returns the next track, given the switch is approached from a given track **from**.
- **boolean isTrailingPoint(RailwayTrack from)** – tests if movement from a given track **from** through the switch is a trailing point movement or face point. Returns **true** if trailing point, **false** if face point.
- **RailwayTrack getTrack(int index)** – returns the track connected to the switch with a given **index** (0, 1 or 2).

Position on track

Position on track defines the exact point on the track where a specific action will happen, such as the train's entry point or the destination of the train's movement.

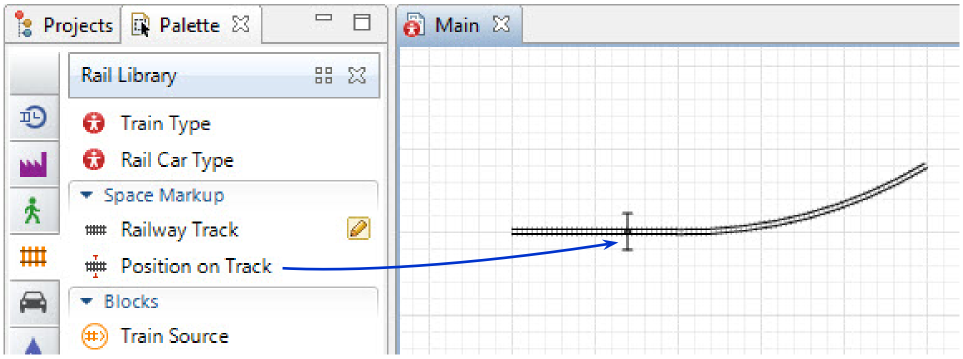


Figure 9.10 Position on track

▶ To define the position on the railway track

1. Drag the **Position on Track** element from the **Rail Library** palette to the railway track where you want to mark the position.

The location of this element on track is calculated as an offset from the starting point of the track. When you resize the track, the position's offset remains the same. You can change this offset by dragging the **Position on track** element along the track manually. Note, that once placed, the **Position on track** exists only on this same track and cannot be dragged to a different one.

Position on track can be used to perform some actions with the train or some other part of the model when the train passes through the element.

The **PositionOnTrack** API is not extensive, but the following functions may prove useful as well:

- **getTrack()** – returns the railway track where this particular position on track is located.
- **getOffset(LengthUnits units)** – returns the distance from the starting point of the track to the position on track in the specified length units. The **units** argument takes one of AnyLogic's length unit constants: **METER**, **FOOT**, etc.

Converting GIS shapefiles to rail space markup shapes

In this example we will convert the GIS shapefile data into AnyLogic railway tracks and switches. It may help you avoid drawing numerous tracks manually.

▶ **Follow these steps:**

1. Obtain the GIS shapefile with the data on the required railways. Note that the shapefile needs to be in Mercator projection.
2. Drag the **GIS Map** element from the **GIS** section of the **Space Markup** palette to the graphical editor.

Note that you can add GIS map only on the diagram that does not contain any rail space markup shapes yet.

3. Expand the **Shapefiles** section of the map properties and add a shapefile by clicking the **Plus** button.
4. The shape files package may contain different files dedicated to the specific GIS markup. Choose the one containing the railways shapes. Select the file and click **Open**.
5. The GIS map will show the file data, and the shapefiles will be copied to the model folder.
6. Double-click the GIS map to edit it and scroll to zoom in to the required rail yard until you see all the railway tracks you need to consider. We do this to avoid converting numerous tracks that we do not need really in our model.
7. To zoom in smaller steps, press the Ctrl key while scrolling.
8. Click outside of the map to finish editing.

9. Right-click the GIS map and choose **Convert Shapefile to Space Markup** from the context menu.
10. In the dialog box, select **Railway** in the **Convert to** list. Make sure that **Visible only** is set for **Conversion boundaries**, which will only convert the tracks within the region that we previously selected.
11. Click **OK.**
12. Confirm the action in the dialog box.

The GIS map will be substituted with the image. On top of this image you will see the rail network composed of AnyLogic's **Railway Track** and **Railway Switch** space markup elements.

9.2. Defining the operation logic of the rail model

The operational logic definition in the Rail Library is based on an easy-to-use process flowchart methodology. The agent moving in a rail yard flowchart is a *train* – any sequence of coupled rail cars. The following flowchart blocks of the Rail Library describe all the actions specific to trains:

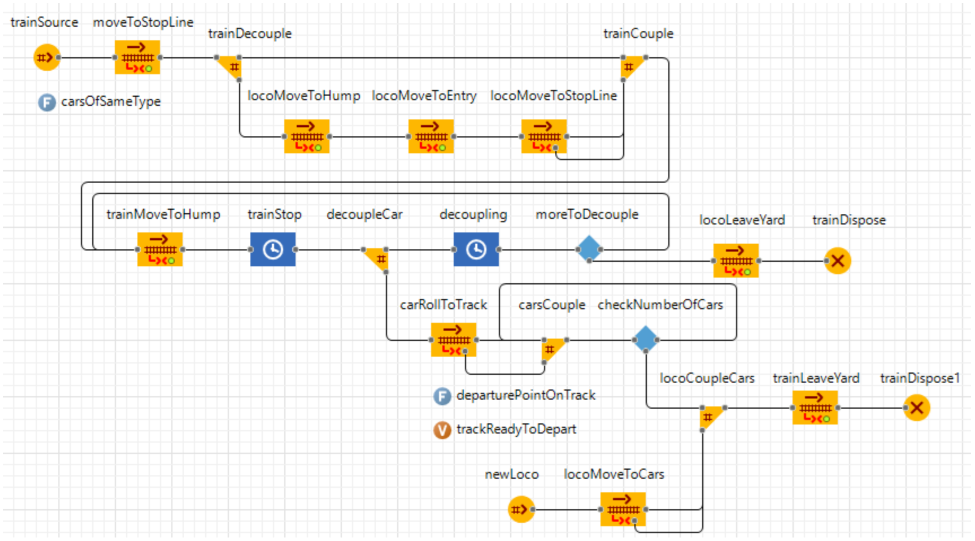
Block	Description
TrainSource	Creates trains, performs initial setup and puts the trains in the rail yard. Starts any rail yard process flowchart. Supports several types of arrivals scheduling.
TrainDispose	Removes trains from the model; either those that have exited the rail yard via an open-ended track, or by "deleting" a train that is still on a track.
TrainMoveTo	Controls movement of trains. Can calculate routes and set switch states as the train goes along the route. Supports acceleration and deceleration.
TrainCouple	Couples two trains that "touch" each other into one train. Has two queues and supports coupling of trains on several tracks simultaneously and independently.
TrainDecouple	Decouples cars from the incoming train and creates a new train from those cars. Supports extreme cases when 0 or all cars are decoupled.

TrainEnter	Takes the train agent and places it on the railway track. Together with **TrainExit**, is used to model part of the train movement on a higher abstraction level.
TrainExit	Removes the train from the railway network and passes the train agent on to the regular process flowchart where it can go through delays, queues, decisions, etc.
RailSettings	Offers the lower-level interface for the rail yard management based on Java functions and callback technique. You can use the callbacks, for example, to collect statistics on the rail car movement throughout the whole rail yard.

These blocks can be mixed in a flowchart with blocks from the Process Modeling Library, such as **Delay**, **SelectOutput**, **Hold**, **Seize**, **Release**, **Queue**, etc. The latter are used to define time delays, make decisions, and manage sharing of the rail yard's resources – tracks and switches.

For example, if a part of the yard should be locked to allow a train to pass through, you may associate a resource with it. Then a train that enters that part would need to seize the resource, and the other trains would wait in the queue of the **Seize** block. For the same purpose you can use the **Hold** block and the pair **RestrictedAreaStart/End.**

The **SelectOutput** block can be used in the rail yard process flowcharts to choose between the different process branches, and **Delay** can naturally model the stops durations or duration of operations such as coupling/decoupling or loading/unloading.



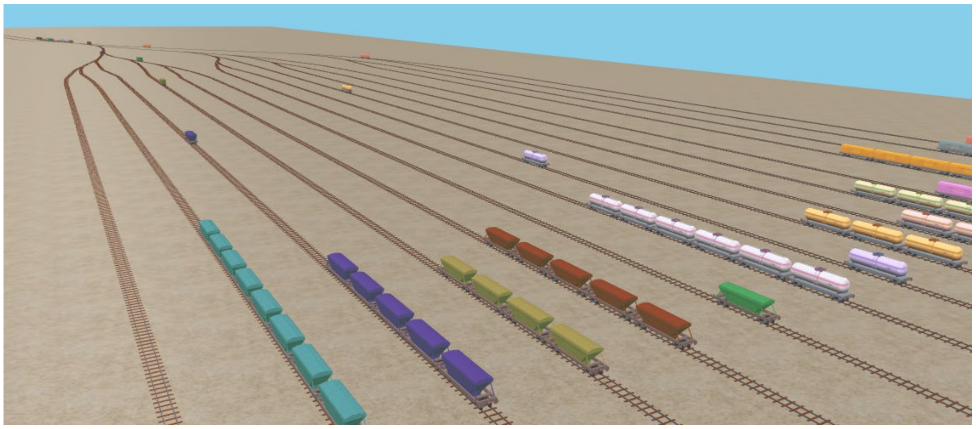


Figure 9.11 A simple hump yard and the flowchart describing its operation

The rail yard operation flowcharts created with AnyLogic Rail Library tend to be very compact. For example, the full logic of a hump yard where arriving trains with different types of cars get disassembled and trains containing cars of the same type are assembled can be defined by a flowchart with about 20 blocks as shown in Figure 9.11.

Example 9.1: Train stop

We will now create a very simple model with one straight railway track and no switches. Passenger trains will stop in the middle of the track for one minute and then will continue moving in the same direction.

▶ **Create a train and let it move along the track:**

1. Create a new model. In the **New Model** wizard, set **minutes** as the **Model time units.**
2. Open the **Rail Library** palette and draw a straight railway track with one segment. Double-click the **Railway Track** in the palette, then click at the starting point (0,100) and then double-click at the end point (1200, 100). In the message window that appears when you finish drawing the track, confirm changing the animation scale to 2 pixels per meter. This scale is effective for drawing a rail yard in the boundaries of the default model window frame of 1000x600 pixels.
3. Leave the default name of the track: **railwayTrack**.

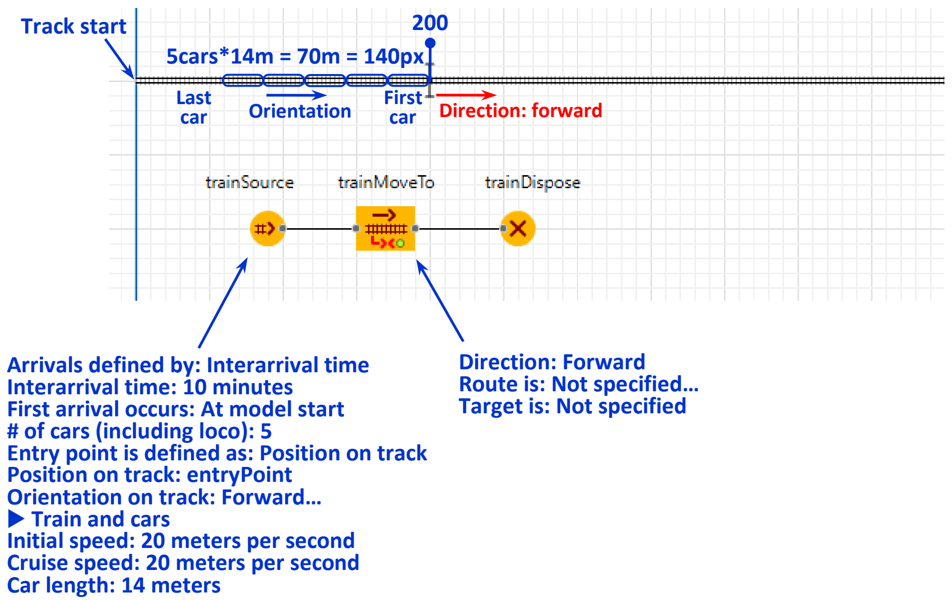
While drawing the track, you can pan the graphical diagram by right-dragging.

4. Drag the **Position on Track** element from the palette onto the railway track. Place it at X=200. Name it **entryPoint**.
5. From the same palette drag three blocks: **TrainSource**, **TrainMoveTo**, and **TrainDispose.** Place them in a sequence and connect as shown in Figure 9.12 (place the blocks close to each other to let them connect automatically).
6. Select **trainSource**. Set the following parameters of that block:
   First arrival occurs: At model start
   # of cars (including loco): 5
   Entry point defined as: Position on track
   Position on track: entryPoint
7. Run the model. At the model start, and then at 10, 20, 30, etc. minutes a train appears at the beginning of the track, moves to the right and exits the track.

Look at the parameters of **trainSource**.

We have set up the **trainSource** to create a train in our rail yard and place it on the **railwayTrack** with the front side of the first car located at the **entryPoint** position on track. This point is located 200 pixels from the beginning of the track. The animation scale is 2 pixels per meter, so 200 pixels correspond to 100 meters. The train will have forward orientation relative to the track (parameter **Orientation on track**), so the rest of train will be between the beginning of the track and the **entryPoint**. At the time of creation, the train must be fully on the track, so you need to make sure there is enough space. Our train has 5 cars, and the default length of a rail car is 14 meters. As 5*14 = 70 <100 meters, there is enough space.

If you get the “The car being created must fully be on one track” runtime error, it means that you have placed the **entryPoint** too close to the railway track start (or have not changed the number of cars in a train from 11 to 5).



Arrivals defined by: Interarrival time
Interarrival time: 10 minutes
First arrival occurs: At model start
# of cars (including loco): 5
Entry point is defined as: Position on track
Position on track: entryPoint
Orientation on track: Forward...
▶ Train and cars
Initial speed: 20 meters per second
Cruise speed: 20 meters per second
Car length: 14 meters

Direction: Forward
Route is: Not specified...
Target is: Not specified

Figure 9.12 The simplest train process flowchart: train appears and moves along the track

The default arrival schedule of **TrainSource** is one train every 10 minutes, so at times 0, 10, 20, 30 minutes and so on. Our **trainSource** will create a new train at the same place, and we need to make sure the previous train has moved and gotten out of the way.

The *cruise speed* that was specified in **trainSource** (20 meters per second, which equals 72 km/h) will be the default speed for train movement.

At the time of creation by **TrainSource** we set the *initial speed* of a train to its cruise speed as if the train has just been moving with the cruise speed. This is done to enable immediate continuation of movement at the same speed.

Now look at the parameters of **trainMoveTo**. You will see that the **Start options** is set to **Continue at current speed**, and the **Direction** is **Forward**, which means the train will appear in the model moving to the right at 72 km/h. No target and no routing options are specified in **trainMoveTo**, so the train will naturally exit the **railwayTrack** at its right end and the train agent will exit the **trainMoveTo** block and will be consumed by **TrainDispose**.

Now we will add a short train stop.

▶ Add the train stop

8. Drag the **Position on Track** element from the **Rail Library** palette onto the railway track. Place it at X=800. Name it **stopPoint**.
9. Select the **trainMoveTo** block and set the following parameters:

   Route is: **Calculated automatically from current to target track**

   Target is: **A given position on track**

   Position on track: **stopPoint**
10. Run the model. You will see that now trains disappear once they reach the **stopPoint** position.
11. Modify the flowchart. Move the **trainDispose** block to the right to give space for two new blocks.
12. Rename the **trainMoveTo** block to **toStop**.
13. Open the **Process Modeling Library** palette and drag the **Delay** block in the flowchart after **toStop**. Call the block **trainStop**. Set its parameter **Delay time** to **1** minute.
14. Open the **Rail Library** palette again and drag another **TrainMoveTo** block between **trainStop** and **trainDispose**. Name it **toExit**. Make sure the ports of all blocks are connected.
15. Run the model again. Now the train stops at the specified point, waits there for one minute, and then continues to the right end of the track.

In the first **TrainMoveTo** block, which is now called **toStop**, we specified the target position of the movement, which is graphically defined by the **stopPoint** Position on track element. Having reached this point, the train agent exits the **toStop** block and enters the **trainStop** block of type **Delay.** While the train agent waits in the **trainStop** block, the train does not move. Then, after a one-minute delay, the train agent exits **trainStop** and enters the second **TrainMoveTo** block (**toExit**) which moves it further.

In this flowchart we mixed the blocks from the Rail Library with the blocks from the Process Modeling Library. This is possible because trains are also agents and can be handled by process flowchart blocks from the Process Modeling Library.

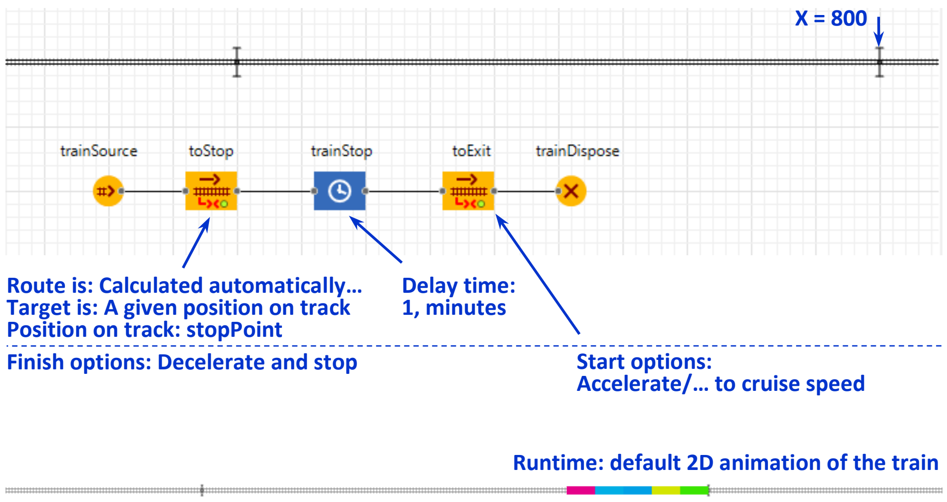

X = 800
trainSource
toStop
trainStop
toExit
trainDispose
Route is: Calculated automatically...
Target is: A given position on track
Position on track: stopPoint
Finish options: Decelerate and stop
Delay time:
1, minutes
Start options:
Accelerate/... to cruise speed
Runtime: default 2D animation of the train

Figure 9.13 The complete process flowchart for the Train stop model. Default 2D animation.

If, however, you investigate the parameters of **toStop**, you will see that **Finish options** is set to **Finish at current speed.** We did not ask the train to decelerate and stop during the first movement. Therefore, the train original speed (20 meters/sec) is "remained set" during the stop, although the train is not physically moving. This is obviously an unrealistic behavior: the train cannot stop immediately and cannot immediately gain speed. This is not an accurate representation of the physics of trains, though it may be useful for certain models with a higher level of abstraction. Now, however, let's add deceleration before the stop and acceleration afterwards.

▶ Add acceleration / deceleration

16. Select **toStop** and set **Finish options** to **Decelerate and stop.**
17. Select **toExit** and make sure that **Start options** is set to **Accelerate/decelerate to cruise speed.**
18. Run the model. Play with different cruise speed, acceleration and deceleration values (they are set up in **TrainSource**). Try to slow down the simulation.

As a final part of this example, we will add the 3D animation.

▶ Add 3D animation

19. Open the **Presentation** palette and drag the **3D Window** to the canvas below the flowchart. Change its size to, say, 900x300 pixels.
20. Run the model and view the 3D picture of the track. Move the camera to view the track and the trains better.

The default 3D animation of trains is very schematic. However, you can always create custom rail car types with specific 3D models as animations. You can also find ready-to-use 3D models of rail cars in AnyLogic **3D Objects** palette, the **Rail Transport** section.

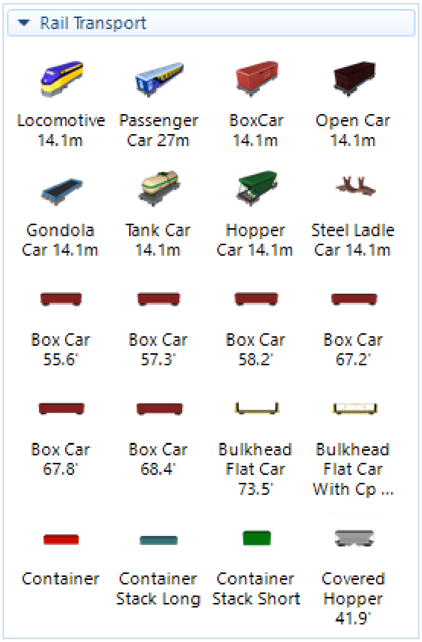


Figure 9.14 3D models of rail cars

▶ Create custom rail car types

21. Drag the **Rail Car Type** element from the **Rail Library** palette to the **Main** diagram.
22. In the wizard enter the name of the new rail car type: **Locomotive**. Click **Next.**
23. AnyLogic provides ready-to-use 3D animations for different types of rail cars of most common dimensions. Select the 3D model from the list: **Locomotive 14.1m.** Click **Finish.**
24. On the last page of the wizard you may optionally define parameters for the rail cars of this type. Click **Finish.**
25. When done, the graphical diagram of the rail car type will be opened. Here you can add any AnyLogic elements to define custom logic.
26. The graphical editor opens for the new rail car type. We do not need to modify the type, so you can close the editor.
27. Create one more rail car type: **PassengerCar**. In the wizard, select the **Passenger Car 27m** 3D model for this rail car type.
28. Return to the editor of **Main** where you have the rails.

29. Select **trainSource**.
    Switch the **New rail car** property to the **Dynamic value** mode and type:
    carindex == 0 ? new Locomotive() : new PassengerCar()

This Java expression is checked for every rail car on the train's creation. We use the conditional operator to check the index of the rail car and set its type. The first rail car is a locomotive, so if the **carindex** equals zero, we use the **Locomotive** type constructor. The rest of the rail cars are passenger cars.

30. Run the model. View the 3D animation of the train. Notice that the cars heavily intersect.

It happens because the 3D objects for cars have different lengths (as real cars do), but **TrainSource** uses just one default length of the rail car, which is 14 meters. We need to specify the custom lengths of our cars. This can be done in the parameter **Car length** of **TrainSource.**

31. Select **trainSource**. Set the parameter **Car length** to **carindex == 0 ? 14 : 27 meters.**

Our 3D model of the locomotive has a length of 14 meters, and the passenger car is 27 meters long.

32. As our train is now longer, we need to adjust its initial location so all cars fit on the track. Drag the **entryPoint** element to the right to (100,250). This should be enough in our case (we need 14 + 27*4 = 122 meters = 244 pixels).
33. Run the model.

▶ Draw the platform at the train stop

34. Open the **Presentation** palette and drag **Rectangle** to the canvas.
35. The rectangle should appear just below the track to the left of the **stopPoint** position.
36. Set the **Line color** of the rectangle to **No color** and **Fill color** to **concrete** texture.
37. In the **Position and size** section of the rectangle edit the rectangle size and position this way:
    X: **550**
    Y: **103**
    Z: **2**
    Width: **250**
    Height: **10**
    Z-Height: **1**.
38. Run the model again.

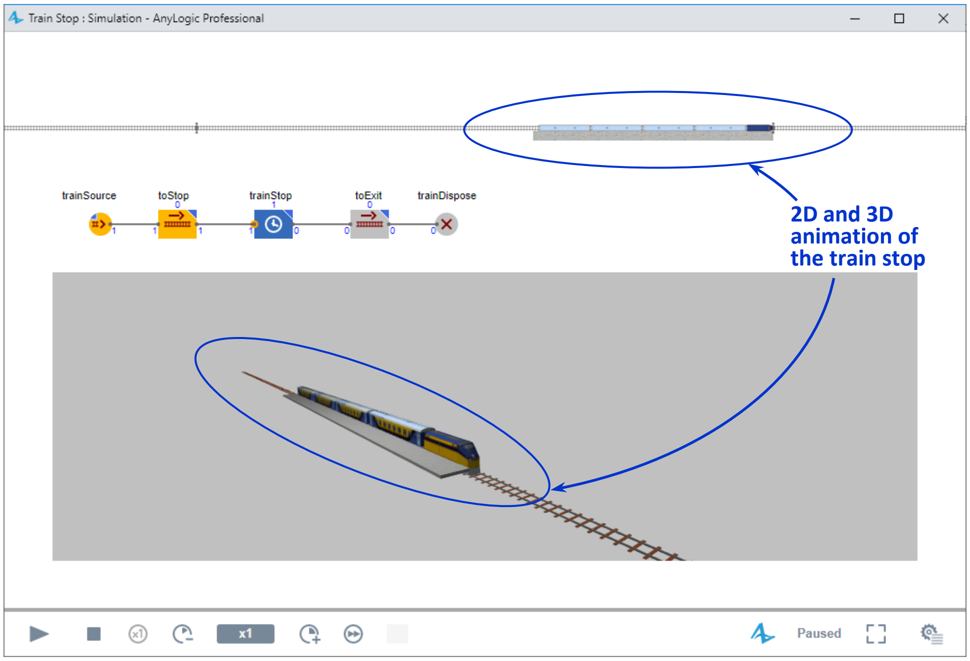

Train Stop : Simulation - AnyLogic Professional
trainSource
toStop
trainStop
toExit
trainDispose
2D and 3D animation of the train stop
x1
Paused

Figure 9.15 A screenshot of the Train Stop model with an adjusted camera position

Example 9.2: Ensuring safe movement of trains

The AnyLogic Rail Library enables you to define rail topology and control the movement of trains but ensuring the safe train movement is the task of the modeler. The library will only detect train collisions at switches (a train arrives at a switch while another train is over the switch) or when a train collides with another train moving or standing on the same track.

In this example, we will show how you can use AnyLogic Process Modeling Library resources to model a simple safety control system. The implementation of the safety control in a real rail yard will contain various communication protocols between the train and the dispatchers, traffic lights, and other elements that are outside the Rail Library's scope. Here we are only suggesting one of the ways to map train management policy to AnyLogic modeling language.

We will create a very simple rail yard consisting of a main track and a branch line that may be used to wait while another train passes by. Assume trains are arriving from the left every 5 minutes. Every third train goes to the branch track, stays there from 3 to 10 minutes and then continues in the same direction. All other trains just move along the main track from left to right. As the trains merge at the **switchCBD**, collisions are possible, and we need to manage the traffic to avoid the collisions.

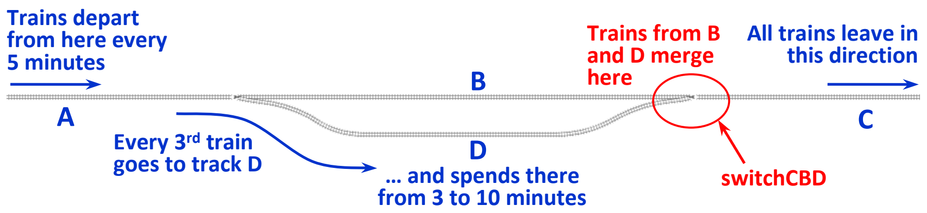

Trains depart from here every 5 minutes
A
B
Trains from B and D merge here
All trains leave in this direction
C
Every 3rd train goes to track D
D
... and spends there from 3 to 10 minutes
switchCBD

Figure 9.16 The problem definition for the example Ensuring safe movement of trains

Draw a rail yard:

1. Create a new model. In the **New Model** wizard, set **minutes** as the **Model time units.**
2. Open the **Rail Library** palette. Draw a horizontal **Railway Track** with one segment of length 1000. Confirm changing the animation scale to 2 pixels per meter.
3. Draw the branch railway track as shown in Figure 9.17. It consists of five segments: two curved segments followed by a linear one and two more curved segments. Double-click the **Railway Track** item in the **Rail Library** palette. Click the existing railway track at X=250 to start drawing a new track.
4. Start with drawing a curved track segment. While you draw a linear track segment with a simple click, to draw a curved segment press and hold the left mouse button where the segment ends and move the mouse until you get the segment of the required shape.
5. Draw the other track segments as shown in Figure 9.17. Finally double-click the main track at X=750 to finish drawing. The connected point should be highlighted in turquoise and the switch circle should appear, otherwise you will have to drag the end point on the main track to have a proper connection.
6. The original straight track is now divided into three separate railway tracks. Name them **trackA**, **trackB** and **trackC** as shown in the Figure 9.17. Name the branch track **trackD**.
7. Add two **Position on Track** elements: **pointMain** on **trackB** at X=700, and **pointBranch** on **trackD** at X=600 (see Figure 9.18).
8. Select the switch connecting the tracks **trackB**, **trackC** and **trackD** and name it **switchCBD**.

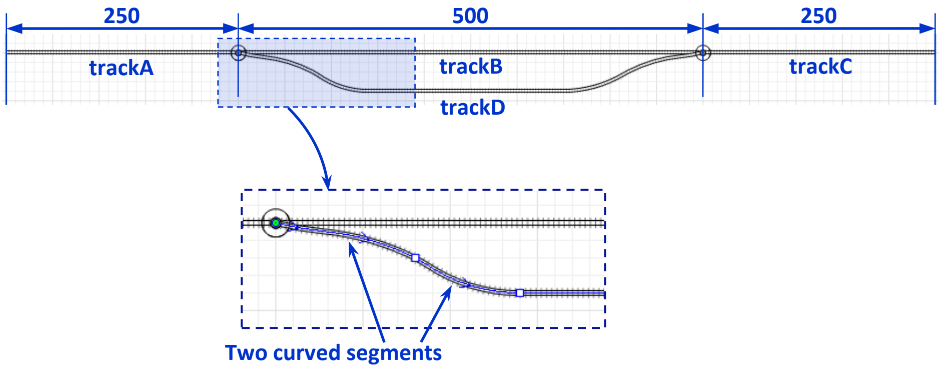


Figure 9.17 Drawing a very simple rail yard

Now the rail yard is fully defined and ready to be used. We will let the trains into the yard.

First, we will define the traffic without the safety control.

▶ **Set up the traffic without safety control:**

9. Open the **Rail Library** palette and drag the **TrainSource** block to the graphical editor. Set the following parameters of the **trainSource**:

   Interarrival time: **5**, **minutes**

   First arrival occurs: At model start

   # of cars (including loco): **8**

   Entry point defined as: **Offset on the track**

   Railway track: **trackA**

   Offset from: **Beginning of the track**

   Offset of 1st car: **120 meters**

   Cruise speed: **10 meters per second**

10. Open the **Process Modeling Library** palette and drag the **SelectOutput** block to the right of **trainSource**. Call it **notEachThird** and change its parameters:

    Select True output: **If condition is true**

    Condition: **self.in.count() % 3 != 0**

We need to treat each 3rd incoming train differently, and this **SelectOutput** block is here to distinguish between each 3rd train and all other trains. The easiest way to do it is to find out how many trains have entered the **SelectOutput**, divide it by three, and look at the remainder. For every 3rd train the remainder will be 0.

In the dynamic parameters of flowchart blocks you can always access the variable **self** – this is the flowchart block itself. So, the expression **self.in.count()** written in a dynamic parameter of **notEachThird** returns the number of agents that have entered **notEachThird** via its **in** port.

11. Continue the flowchart by adding two **TrainMoveTo** blocks after the two outputs of **notEachThird**.
12. Rename the **TrainMoveTo** at the T (true) output port of **notEachThird** to **moveToMain** and set the following parameters:

    Route is: **Calculated automatically**…

    Target is: **A given position on track**

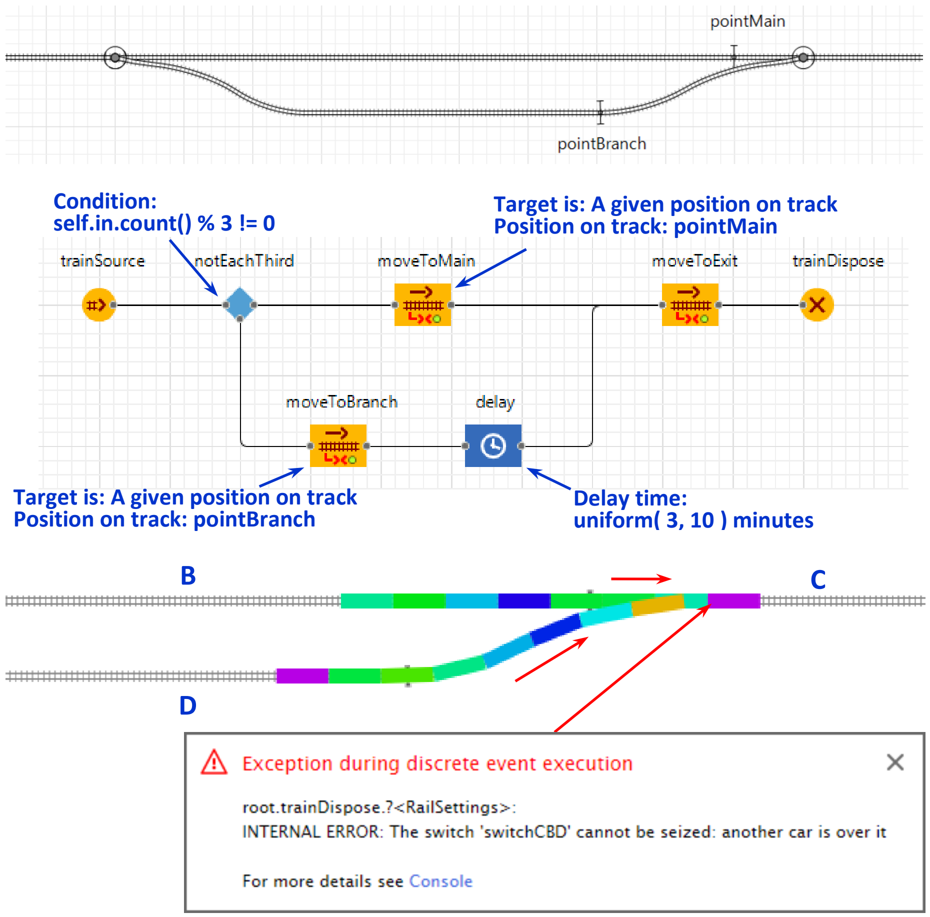
    Position on track: **pointMain**
13. Similarly, rename the **TrainMoveTo** at the F (false) port to **moveToBranch** and set the following parameters:

    Route is: **Calculated automatically**…

    Target is: **A given position on track**

    Position on track: **pointBranch**
14. Add a **Delay** block from the Process Modeling Library after **moveToBranch** and set its **Delay time** to **uniform(3, 10) minutes.**
15. Add the third **TrainMoveTo** block and connect the outputs of **moveToMain** and **delay** to its input. Name it **moveToExit**.
16. Finish the flowchart with **TrainDispose.**
17. Run the model. Watch the trains going through the yard. The first several trains may do fine, no collisions occur.
18. Run the model in virtual time mode (as fast as possible). The model will stop almost immediately with an error "The switch 'switchCBD' cannot be seized: another car is over it".

As long as the time the trains spend at **trackD** is nondeterministic and can be greater than the time interval between trains (5 minutes), collisions at **switchCBD** will inevitably occur. The error message you are getting (see Figure 9.18) is typical for that type of collision: while the train exiting **trackD** is still moving over the **switchCBD**, another train from **trackB** approaches the switch and tries to change its state (in this case as a result of trailing point movement).



Exception during discrete event execution

```
root.trainDispose.?<RailSettings>:
INTERNAL ERROR: The switch 'switchCBD' cannot be seized: another car is over it

For more details see Console
```

Figure 9.18 The unsafe train movement flowchart and the collision detected by the library

We will now add a safety management mechanism to our model. The idea is to associate a Process Modeling Library resource with the part of the rail yard shared by the conflicting train flows and have trains seize the resource before they enter that part and release it when they exit.

▶ Add safety control

19. Modify the flowchart by embracing the **moveToExit** block (the one where conflict actually occur) with the pair **Seize** / **Release** and a **ResourcePool** as shown in Figure 9.19. Those three blocks can be found in the **Process Modeling Library** palette.
20. In the properties of the **Seize** block, set the **Seize** option to **units of the same pool**, and select **resourcePool** in the **Resource pool** control. Leave all other parameters with the default values.

21. Run the model again. Try virtual time mode. No collisions occur.

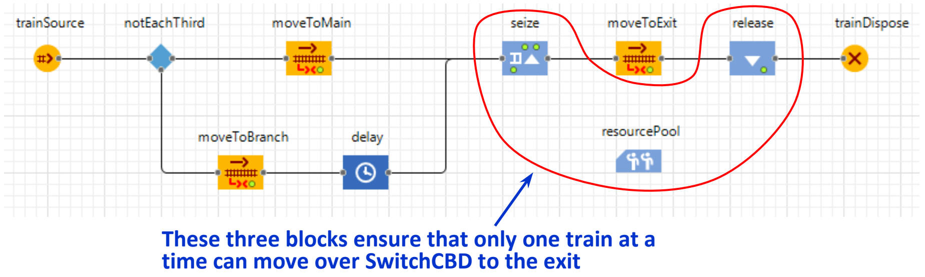


Figure 9.19 The flowchart modified to prevent train collisions

Associating **ResourcePool** with a shared section of a rail yard is not the only way of avoiding conflicts. Alternatively, you could use the block **Hold** or a pair **RestrictedAreaStart** / **RestrictedAreaEnd.**

The model of the safety control system is now in place. You may have noticed, however, that the movement of trains is not very realistic: they start and stop without acceleration and deceleration, the speed changes from 0 to 10 m/s (cruise speed) immediately. You already know that you can request **TrainMoveTo** to apply acceleration and deceleration. However, in this particular example, trains that move along the main track (A-B-C) should only decelerate before **pointMain** if the exit is seized by another train; otherwise they should just continue going at cruise speed. How can we implement such optional deceleration?

One approach is to determine whether the exit is busy (seized by another train) at some point before **pointMain.** If it is busy, the train will decelerate and stop at **pointMain**, where it will wait for the resource to become available. If the exit is free, the train will immediately seize it and proceed to the exit without deceleration.

▶ **Add acceleration and deceleration of trains:**

22. Add one more **Position on Track** element on **trackB** at X=600 (see Figure 9.20). Name it **pointCheck.**
23. Insert two blocks between the **true** port of **notEachThird** and the input port of **moveToMain**: **TrainMoveTo** and **SelectOutput** as shown in Figure 9.20. Call them **moveToCheck** and **exitIsBusy.**
24. Set the following parameters of **moveToCheck**:

    Route is: Calculated automatically…

    Target is: A given position on track

    Position on track: pointCheck

25. Change the parameters of **exitIsBusy**:
    Select True output: **If condition is true**
    Condition: **resourcePool.idle() == 0**
26. Change the start and finish options of the two **TrainMoveTo** blocks created earlier:
    moveToMain: **Finish options: Decelerate and stop**
    moveToBranch: **Finish options**: **Decelerate and stop**
27. Run the model. You may need to slow down the simulation to see how the trains decelerate.

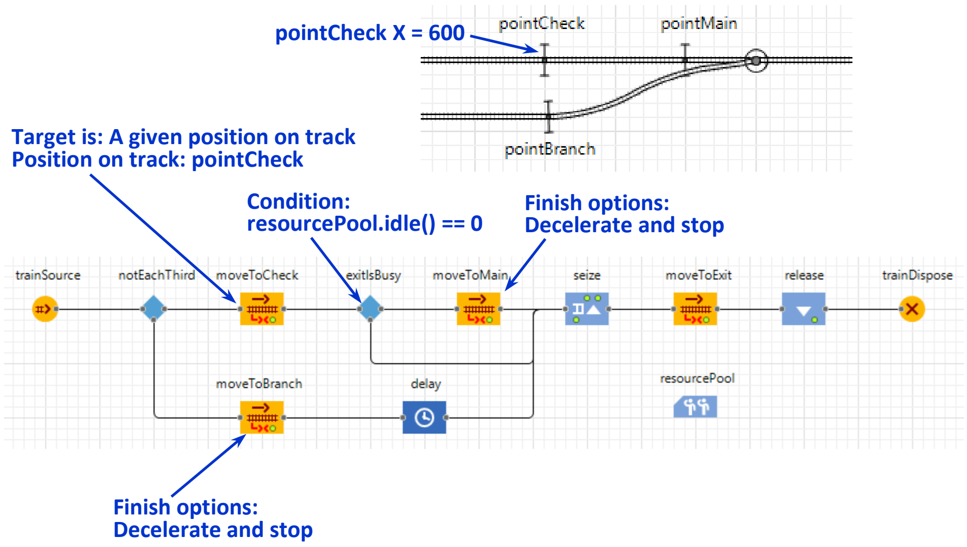


Figure 9.20 The flowchart modified to model deceleration

To check the state of the exit area of the rail yard in advance, we added an intermediate point on **trackB** marked with **pointCheck,** and then we divided the movement into two stages: move to **pointCheck** (without deceleration), and if the exit is idle, seize it immediately and proceed to the exit without deceleration. Otherwise (if the exit is busy), decelerate, stop at **pointMain** and wait there until the exit gets free. Note that:

- Continuous movement can be modeled by multiple **TrainMoveTo** blocks put in a sequence (like the movement along the main track A-B-C is modeled by **moveToCheck** and **moveToExit**).
- **TrainMoveTo** with the start option **Accelerate/decelerate to cruise speed** will either take a train already going at its cruise speed (in which case acceleration will not be done), or will accelerate a standing train – see **moveToExit**.

- The same **TrainMoveTo** block can take trains at different locations and bring them to the same or different destinations. For example, a train entering **moveToExit** can be either at **pointMain**, **pointBranch**, or **pointCheck**, but will be routed to the exit.

Example 9.3: Simple classification yard

In this example, we will create a very simple classification yard. Arriving trains will contain cars of two types: tanks and box cars. The tanks will be moved to the departure track and left there, and the box cars will continue to the exit. Once the departure track accumulates more than six tanks, they will be driven away by another locomotive. We will use the rail layout created in Example 9.2: “Ensuring safe movement of trains".

We will create custom rail car types with built-in statistics, and we will show how to access the properties of individual rail cars. We will also show how to *couple* and *decouple* rail cars.

▶ Create custom rail car types:

1. Repeat the steps 1-6 of Example 9.2: “Ensuring safe movement of trains".
2. Add the **Position on Track** element on **trackA** at X=200. Name it **pointA.**
3. Drag the **Rail Car Type** element from the **Rail Library** palette to the **Main** diagram.
4. In the wizard enter the name of the new rail car type: **Locomotive.** Click **Next.**
5. Select the 3D model from the list: **Locomotive 14.1m.** Click **Finish.**
6. The graphical editor opens for the new rail car type. We do not need to modify it, so you can close the editor.
7. Create two more rail car types: **BoxCar** and **TankCar**. Select **Box Car 14.1m** and **Tank Car 14.1m** 3D models respectively.

▶ Create a simple train flowchart and set up train creation:

8. On the **Main** diagram, create a flowchart of three Rail Library blocks: **TrainSource**, **TrainMoveTo**, and **TrainDispose.**
9. Set the following parameters of **trainSource**:
   First arrival occurs: At model start

   # of cars (including loco): 6

   Position on track: pointA
10. In the **Train and cars** properties section, set:
    Cruise speed: 5 meters per second
    New rail car:
    carindex == 0 ? new Locomotive() : (carindex < 3 ? new TankCar() : new BoxCar());
11. Run the model. At time 0 (and then at 10, 20, and so on) a new train appears on the main track and proceeds to the right.

In the field **New rail car** of **TrainSource**, we set up the rail cars of our train. The rail cars in our train are of custom types: **Locomotive**, **BoxCar** and **TankCar**. The type depends on the position of the car in the train, which is accessed as **carindex**. The first car (position 0) is the locomotive, so we call the default constructor of **Locomotive**. The locomotive is followed by two tank cars and three box cars.

▶ Model decoupling of tank cars and moving them to the departure track

12. Add three **Position on Track** elements: **pointB** on **trackB** at X=700, **pointC** on **trackC** at X=950, and **pointD** on **trackD** at X=400 (see Figure 9.21).
13. Modify the flowchart as shown in Figure 9.21.
14. Run the model.

The block after **TrainSource** is **toMain** (of type **TrainMoveTo**); it brings the incoming train to **pointMain**. The next block, **decoupleTanks** (type **TrainDecouple**), decouples 3 cars: the loco and two tanks. Next, **tanksToC** takes the decoupled part of the train to the **trackC**. Finally, the last block **tanksToD** pushes the loco and the tanks to **trackD** where they stop at **pointBranch**.

The block **TrainMoveTo** can only calculate the straight routes, i.e. the routes not requiring reverse movement of the train. Therefore, to bring the tanks from B to D we need to use two **TrainMoveTo** blocks: one that moves the train forward to C, and another that moves the train backwards to track D.

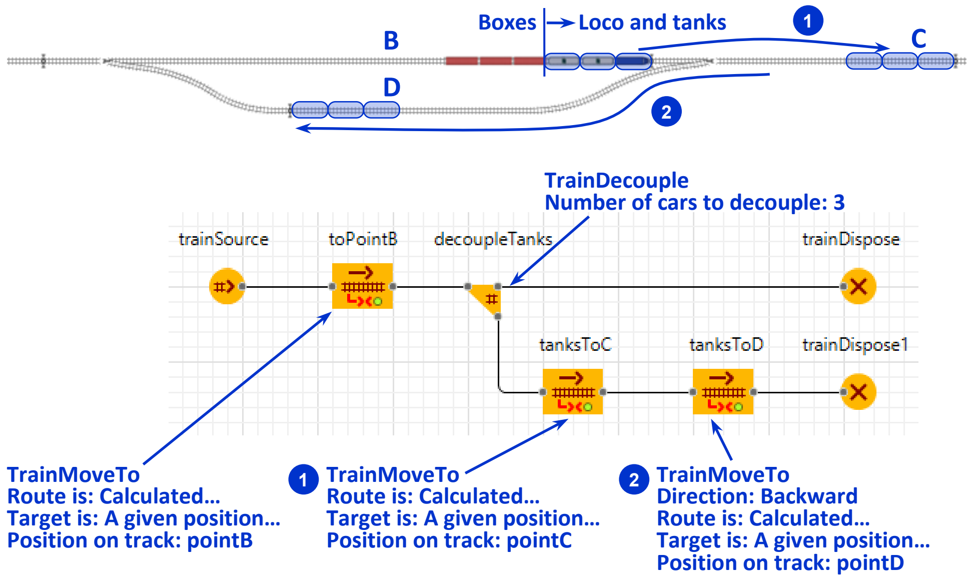


Figure 9.21 The first step of the classification process: tank cars are decoupled and moved to D

In the block **decoupleTanks**, we assumed that any train has only two tank cars after the locomotive. We will now generalize that block so that is will decouple any number of tank cars. To accomplish this, we need to access the contents of the incoming train, which requires creating a train type.

▶ **Generalize TrainDecouple to handle any number of tank cars**

15. Drag the **Train Type** element from the **Rail Library** palette to the **Main** diagram. In the wizard enter the name of the new type: **Train**. Click **Finish.**
16. Switch to the **Main** diagram. In the **Train and cars** section of the **trainSource** properties, set the parameter **New train** to **Train**.
17. Create a new function with the name **nCarsToDecouple** (**Function** element is in the **Agent** palette). In the function properties, select the **Returns value** option and select **int** in the **Type** drop-down list below.
18. In the **Arguments** section of the properties, add one parameter **train** of type **Train** (remember that AnyLogic is case sensitive).
19. Write the following code in the **Function body** section:

```
int n = 1; //the first car is locomotive
while( train.getCar(n) instanceof TankCar )
    n++;
return n;
```

20. Set the parameter **Number of cars to decouple** of **decoupleTanks** to: **nCarsToDecouple(train)**
21. Run the model. The behavior is the same. Try to modify the parameters of **trainSource** so that the incoming trains have different or variable number of tank cars following the locomotive.

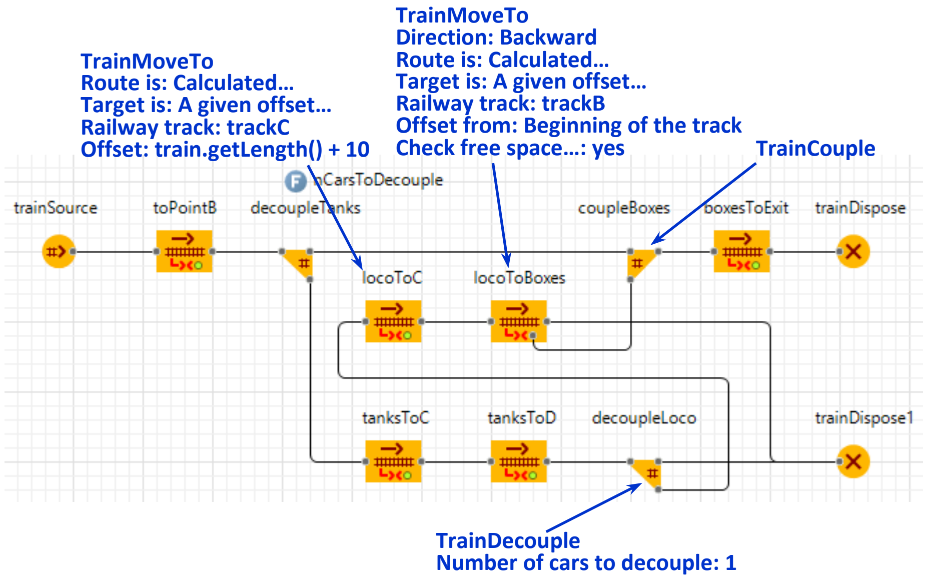
In the code of **nCarsToDecouple**, we access the agent of type **Train.** The function **getCar(n)** returns the rail car with a given index. However, the class of the returned rail car is a generic class **Agent.** Using the operator **instanceof** we check the actual type of the rail car. In the field **Number of cars to decouple** we call the function **nCarsToDecouple,** giving it the current train (**train**) as an argument.

We will now continue the classification process. Having moved the tanks to track D, the locomotive will decouple from them, return to track B, couple with the box cars waiting there, and leave the rail yard.

▶ **Model return of locomotive to box cars and their departure from the yard:**

22. Add more blocks to the flowchart and set their parameters as shown in Figure 9.22. Leave the parameters of the blocks **coupleBoxes** and **boxesToExit** at their default values. Note that the output port of **locoToBoxes** is different: it is **outHit** port.

23. Run the model. The locomotive now returns to box cars and drives them away. Again, the flowchart is still incomplete and the decoupled tank cars still disappear at **pointD**.



TrainMoveTo
Route is: Calculated...
Target is: A given offset...
Railway track: trackC
Offset: train.getLength() + 10

TrainMoveTo
Direction: Backward
Route is: Calculated...
Target is: A given offset...
Railway track: trackB
Offset from: Beginning of the track
Check free space...: yes

TrainCouple

TrainDecouple
Number of cars to decouple: 1

Figure 9.22 The 2nd step of classification: loco returns to box cars, picks them up, and drives away

Decoupling the locomotive is quite straightforward and can be accomplished in the following steps. First, decouple the first car of the train. Then, return to the box cars on track B, utilizing two moves: **locoToC** and **locoToBoxes**. The parameters of **locoToBoxes** are described as follows. The beginning of the track C is set as the target point. We know that there are box cars on track C, so the loco will inevitably hit them on its way to the target point. To let the locomotive move as close as possible to the nearest box car and then stop, we selected the parameter **Check free space on target track** of **TrainMoveTo.**

The option **Check free space on target track** allows you to finish movement at the first rail car met on the target track. The train would "touch" the car and stop. This option is mostly used when you are going to couple with that car. Please keep in mind that the free space on the target track is checked *at the time the train starts movement*, so if the situation on the track changes while the train is moving, it may stop at an incorrect position. If the train stops having touched another train, the train agent exits via **outHit** port of **TrainMoveTo**, otherwise the train proceeds to the target position, and exits via **out** port.

Next, look at the block **coupleBoxes** of type **TrainCouple.** It has two input ports for the two parts of the train that will be coupled into one train. When a train arrives to either of the two inputs, **TrainCouple** checks to see whether it "touches" any train waiting at another input. If yes, the trains are coupled into one (more precisely, the train at **in2** is added to the train at **in1**), and the train agent exits **TrainCouple.** Otherwise, the train waits in the queue associated with the input port.

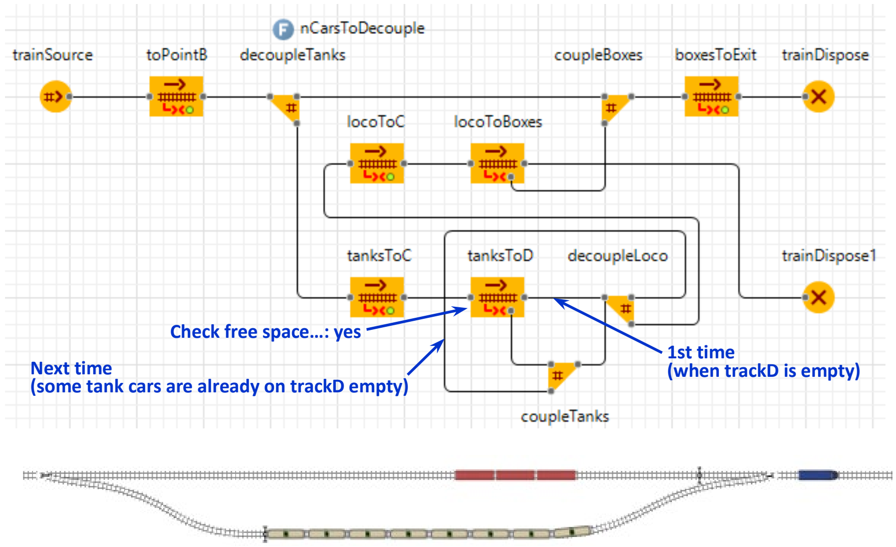


Figure 9.23 The tank cars are coupled on the departure track

Now we will show how to extend the classification process to couple the tank cars that are moved to **trackD** with other tank cars that are already there.

Model coupling of tank cars with each other:

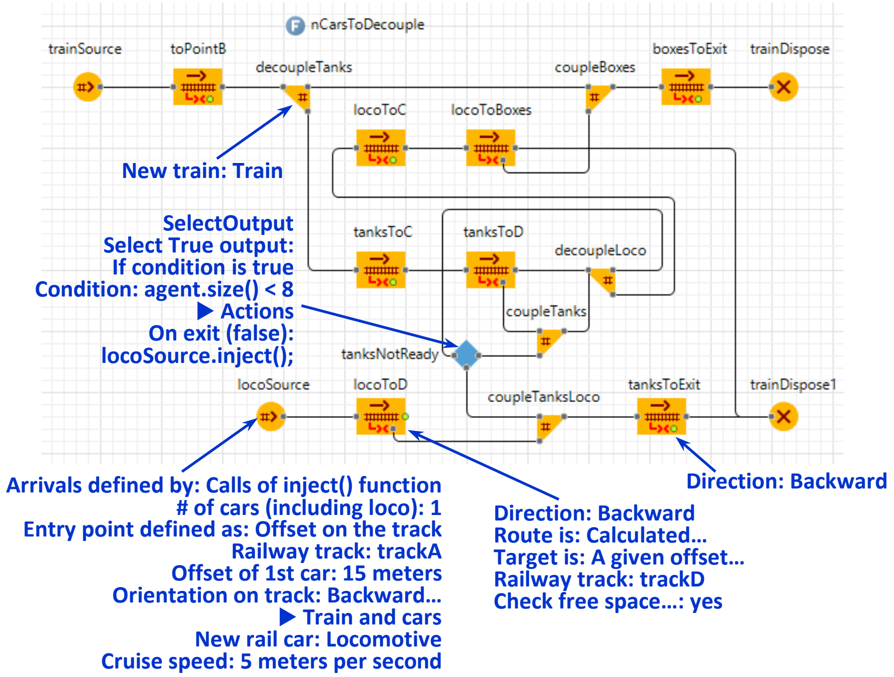
24. Add one more block to the flowchart: **coupleTanks** of type **TrainCouple** and connect it as shown in Figure 9.23.
25. Set the parameter **Check free space on target track** of **tanksToD** to **yes.**
26. Run the model. The tank cars now accumulate on the departure track and are coupled with each other.

The departure track has limited capacity and the cars that have accumulated on it need to be moved out of the way periodically. Another locomotive will be brought in as soon as eight tank cars have accumulated on the departure track. The locomotive will come from the left-hand side (track A), couple with the tanks, and drive them away back to the left.

Model departure of the tank cars

27. Modify the flowchart as shown in Figure 9.24.

28. Run the model. The tanks are now periodically driven away by a new locomotive and the model can run for an infinite amount of time.



New train: Train

SelectOutput
Select True output:
If condition is true
Condition: agent.size() < 8
▶ Actions
On exit (false):
locoSource.inject();

Arrivals defined by: Calls of inject() function
# of cars (including loco): 1
Entry point defined as: Offset on the track
Railway track: trackA
Offset of 1st car: 15 meters
Orientation on track: Backward...
▶ Train and cars
New rail car: Locomotive
Cruise speed: 5 meters per second

Direction: Backward
Route is: Calculated...
Target is: A given offset...
Railway track: trackD
Check free space...: yes

Direction: Backward

Figure 9.24 The final version of the classification flowchart

Note that in the flowchart block **tanksNotReady** we need to determine the size of the train. The **size()** function is specific to train agents only, so we need to tell this block that agents going through it are of our type **Train**. This is done by specifying **Train** in the **New train** field of the **decoupleTanks** block, which creates the trains that get into the **tanksNotReady** block.

The arrival of locomotives that drive away the tanks is triggered from the **SelectOutput** block **tanksNotReady**, as seen in the action code of its **On exit (false)** field. Correspondingly, the arrival type of **TrainSource** is set to **Calls of inject() function.**

There are a couple of additional things to notice in this flowchart. First, although the tank cars are all physically accumulated on track D, in the flowchart they are waiting either in the queue of **coupleTanks** block, or (if there are eight of them) in the queue of **coupleTanksLoco** block. Second, notice the orientation of the locomotive and the departure train. The initial orientation of the locomotive is backward (parameter **Orientation on track** of **locoSource**), such that it couples with the tanks at its rear side. After coupling, however, the loco is added to the tanks and not vice versa because it enters the **TrainCouple** block from its lower port. Therefore, the orientation of the departure train is same as orientation of the tank cars, and departure to the left is departure in backward direction – see the parameter **Direction** of the **tanksToExit**.

Now that the classification process is complete, we can collect some statistics. Let us obtain the length of stay of the tank cars in the yard.

▶ Record arrival time of each tank car

29. Open the **TankCar** diagram and create a new parameter with the name **timeEntered** (**Parameter** element is in the **Agent** palette).
30. Expand the **Train and cars** section of **trainSource** properties and type the following code in the **Car setup** field:

```
if (car instanceof TankCar)
  ((TankCar)car).timeEntered = time();
```

Here we set the **timeEntered** parameter to the current model time obtained by the function **time()**.

Now we will calculate the length of stay at the time the cars are exiting the yard.

▶ Add collection of length of stay statistics

31. Open the **Analysis** palette and drag the **Histogram Data** element to the **Main** diagram. Call it **lengthOfStay** and set the **Number of intervals** to 20.
32. Drag the **RailSettings** element from the **Rail Library** palette to graphical editor and set the following parameter for this block:

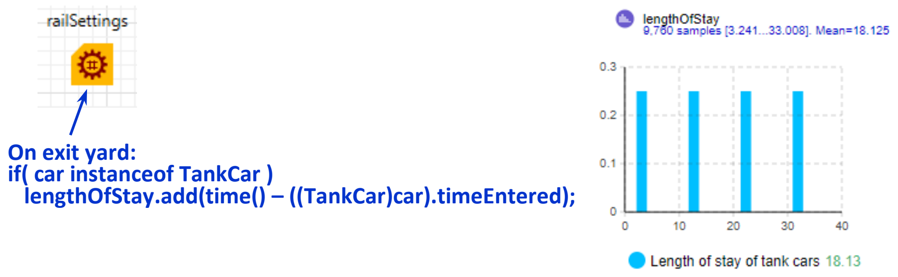
On exit yard:

```
if( car instanceof TankCar )
            lengthOfStay.add( time() – ((TankCar)car).timeEntered );
```

33. Drag the **Histogram** chart from the **Analysis** palette to the graphical editor.
34. Expand the **Data** section of the histogram properties and type **lengthOfStay** in the **Histogram** field. Change the **Title** to **Length of stay of tank cars**.
35. Run the model. Switch to virtual time model so that statistics is collected faster.

The length of stay appears to be deterministic and takes one of the four values, as shown in Figure 9.25. This is expected: the model itself is a deterministic model, and the variation of length of stay is due to the fact that tank cars are moved to the departure track in four portions of two cars each, so the first two rail cars will wait for a longer time than the last. You may add stochastic elements to the model, such as:

- Variation of arrival times of the original trains or the locomotive
- Stochastic time delays associated with coupling and decoupling
- Cruise speed variations


railSettings
lengthOfStay
9,760 samples [3.241...33.008]. Mean=18.125
0.3
0.2
0.1
0
0
10
20
30
40
Length of stay of tank cars 18.13

```
On exit yard:
if( car instanceof TankCar )
  lengthOfStay.add(time() – ((TankCar)car).timeEntered);
```

Figure 9.25 Collecting length of stay statistics for the tank cars

▶ Add deceleration and 3D animation of the model

36. In all **TrainMoveTo** blocks except for **boxesToExit** and **tanksToExit** set **Finish options** to **Decelerate and stop.**
37. Open the **Presentation** palette and drag the **3D Window** to the graphical editor below the flowchart. Change its size to, say, 900x300 pixels.
38. Run the model. View the 3D animation and move the camera to find a better viewpoint.

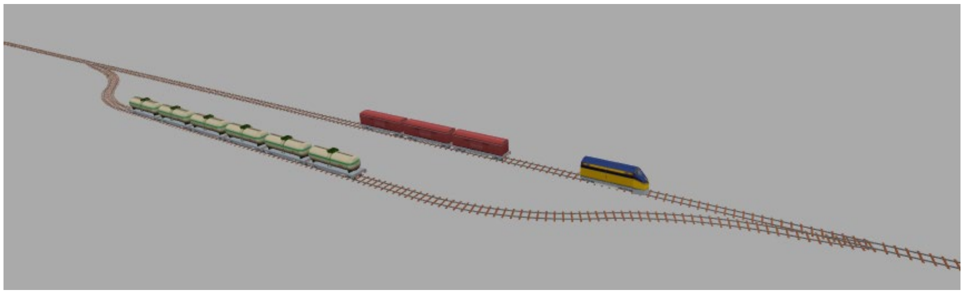


Figure 9.26 3D animation of the classification yard

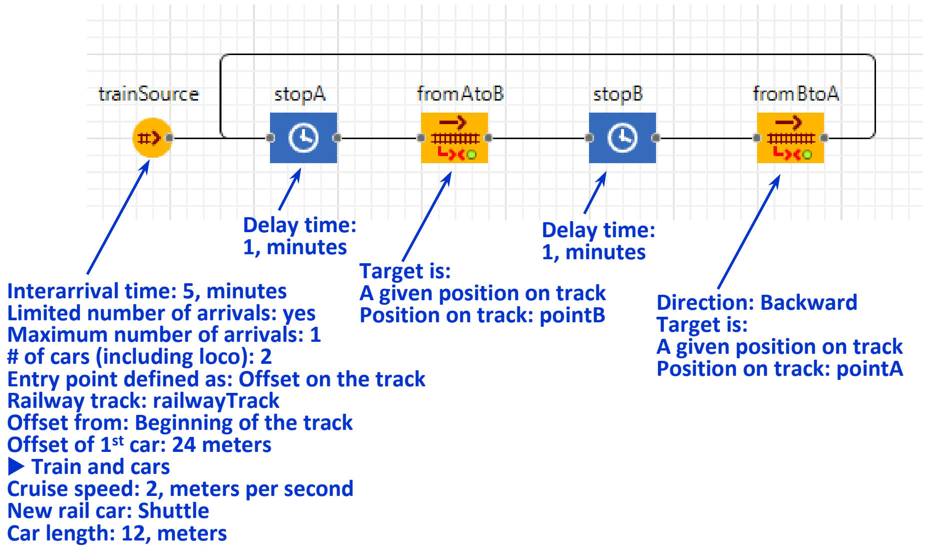
Example 9.4: Airport shuttle train (featuring AnyLogic Pedestrian Library)

We will now make the Rail Library and the Pedestrian Library work together by modeling a shuttle train that transfers passengers between two airport terminals. The passengers will be modeled using the AnyLogic Pedestrian Library when they are on the platform, and when they are on the train, they will be contained in a train agent. The rail system will be very simple: just one shuttle moving back and forth along one straight track.

▶ Model the rail part of the system:

1. Create a new model. In the **New Model** wizard, set **Model time units** to **minutes.**
2. Draw a straight railway track from (50,50) to (950,50). Confirm changing the animation scale to 2 pixels per meter.
3. Add **Position on Track** at the beginning of the railway track at (50,50). Name it **pointA**.

4. Add another position on track at the very end of the track. Name it **pointB**.
5. Drag the **Rail Car Type** element from the **Rail Library** palette to the graphical editor. Name the new rail car type **Shuttle**. Click **Next**.
6. On the next page of the wizard, set the **3D** animation shape for the rail car. In the list, scroll down and select the **Shuttle** item. Click **Finish**.
7. On the **Shuttle** diagram, you will see that the rail car animation is 120 pixels long, which corresponds to 12 meters.
8. In the **Main** agent, create the train process flowchart as shown in Figure 9.27.
9. Run the model.



Delay time: 1, minutes

Delay time: 1, minutes

Target is: A given position on track
Position on track: pointB

Interarrival time: 5, minutes
Limited number of arrivals: yes
Maximum number of arrivals: 1
# of cars (including loco): 2
Entry point defined as: Offset on the track
Railway track: railwayTrack
Offset from: Beginning of the track
Offset of 1st car: 24 meters
▶ Train and cars
Cruise speed: 2, meters per second
New rail car: Shuttle
Car length: 12, meters

Direction: Backward
Target is:
A given position on track
Position on track: pointA

Figure 9.27 The rail part of the Airport shuttle train flowchart

Note that the **TrainSource** block in this model is used to create only one train that remains in the system. We have prevented any generation of other trains by setting the **Maximum number of arrivals** to 1.

▶ Mark the pedestrian ground:

10. Open the **Pedestrian Library** palette and draw the space markup as shown in Figure 9.28. Use the **Target Line** element to draw four target lines denoting entry places and destinations for pedestrians.
11. Add a **Rectangular Area** below the start of the track. Name it **waitingArea** and make it invisible.

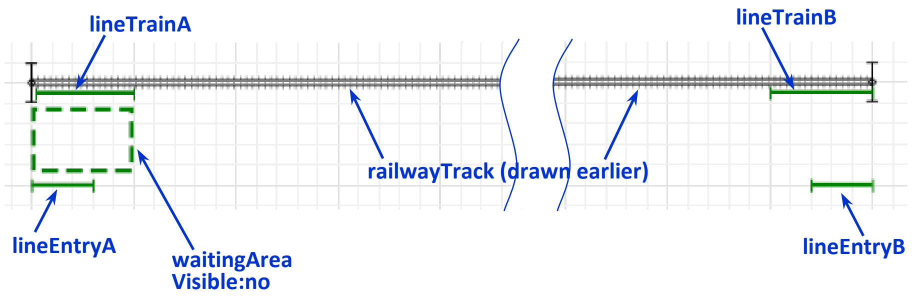


Figure 9.28 Markup of the pedestrian ground

Add the pedestrian part of the flowchart:

12. Open the **Process Modeling Library** palette and insert **Pickup** and **Dropoff** blocks into the rail flowchart as shown in Figure 9.29. Set their **Pickup/Dropoff** parameters to **All available agents.**
13. Open the **Pedestrian Library** palette. Add the pedestrian flowchart and set the parameters of the blocks as shown in Figure 9.29.
14. Add the entry and exit actions of the **stopA** block as shown.
15. Run the model.

Once the train arrives at the stop at terminal A, it lets the waiting passengers in by calling the function **freeAll()** of the **waitA** block (of type **PedWait**). Upon closing the doors, the passengers who could not board the train are returned to the waiting area by calling the function **cancelAll()** of the block **boardA**. Then they exit the block via the bottom port **ccl** and get back into **waitA**.

The block **exitPedModelA** of type **PedExit** temporarily removes the passengers who boarded the train from under control of the Pedestrian Library. The passengers are then picked up by the train agent and travel with the train to the terminal B, where they unboard (block **dropoffB**), are returned to the control of the Pedestrian Library, and appear again as pedestrians at **lineTrainB**.

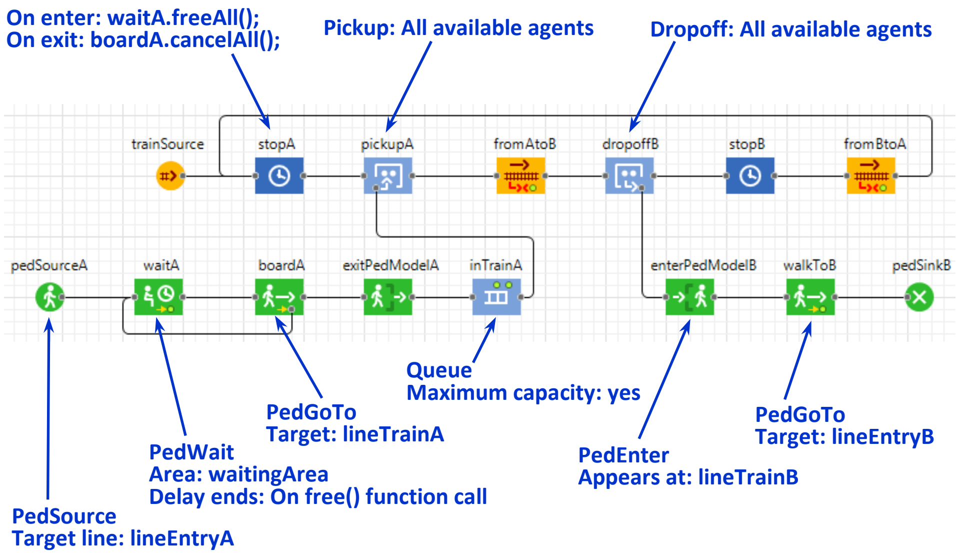


Figure 9.29 The complete flowchart including the pedestrian part

▶ **Add 3D animation:**

16. Drag the **Pedestrian Type** element from the **Pedestrian Library** palette to the graphical editor. Name the new pedestrian type **Passenger.** Click **Next.**
17. On the next page of the wizard, set the **3D** animation shape for the pedestrian. Leave the default **Person** 3D model. Click **Finish.**
18. Switch back to the **Main** diagram. Select the **pedSourceA** block, expand its **Pedestrian** properties section and choose **Passenger** from the **New pedestrian** drop-down list.
19. Open the **Presentation** palette and drag **3D Window** to the **Main** diagram, e.g. below the flowchart. Change its size to, say, 900 x 200.
20. Run the model and view the 3D animation.
21. Select the railway track and set its **Z** coordinate to -2.
22. Draw the platforms. Drag the **Rectangle** element from the **Presentation** palette, place it at the train stop at terminal A and resize to cover the area where passengers walk to the train. Set its properties:

    Line color: **No line**

    Fill color: **concrete** texture

    Z: **-2**

    Z-Height: **2**
23. Clrl+drag the rectangle to create its copy at the stop at terminal B.
24. Run the model again.

By setting the **Z** of the railway track to -2 we put the rails below the level 0, where the pedestrians move. Correspondingly, we set up the platforms so that their upper surface is at level 0.

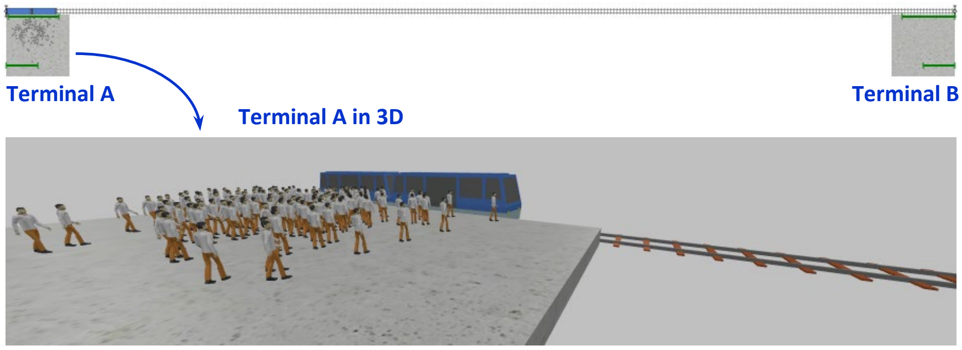


Figure 9.30 2D and 3D animation of the Airport shuttle train

Creating rail yards programmatically

Example 9.5: Creating a rail yard by code

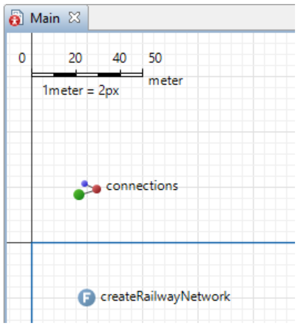
We will assume you have read the yard configuration from an external source and know the coordinates of switches and end points of track segments (curved tracks should be approximated by multiple straight segments). In this example we will create the railway network consisting of tracks, a switch and a position on track using AnyLogic API.

Follow these steps:

1. Create a new model. Right-drag the **Main** diagram down to see the **Scale** element above the axis. Select the scale item and change its properties. Set the **Scale is** option to **Specified explicitly** and set the **Scale** to **2** pixels per meter.
2. Open the **Agent** palette and drag the **Function** element to the graphical editor. Name the function **createRailwayNetwork**.
3. Enter the following code in the **Function body** section of the function properties:

```
// Create segments
MarkupSegmentLine ms1 = new MarkupSegmentLine( 50, 100, 0, 350, 100, 0);
MarkupSegmentLine ms2 = new MarkupSegmentLine( 350, 100, 0, 950, 100, 0);
MarkupSegmentLine ms3 = new MarkupSegmentLine( 350, 100, 0, 550, 200, 0);
MarkupSegmentLine ms4 = new MarkupSegmentLine( 550, 200, 0, 950, 200, 0);

// Create tracks
RailwayTrack rt1 = new RailwayTrack(this, SHAPE_DRAW_2D3D, true,
                        PATH_RAILROAD, black, 1.5, ms1);
RailwayTrack rt2 = new RailwayTrack(this, SHAPE_DRAW_2D3D, true,
                    PATH_RAILROAD, black, 1.5, ms2);
RailwayTrack rt3 = new RailwayTrack(this, SHAPE_DRAW_2D3D, true,
                    PATH_RAILROAD, black, 1.5, ms3, ms4);

// Create switch
RailwaySwitch rs = new RailwaySwitch( this, SHAPE_DRAW_2D3D, true,
               1.5, black, null, rt1, rt2, rt3 );

// Create position on track
PositionOnTrack pt = new PositionOnTrack( this, SHAPE_DRAW_2D3D, true,
                    black, rt2, 500);

// Create railway network
RailwayNetwork rn = new RailwayNetwork( this, "myRailwayNetwork",
                    SHAPE_DRAW_2D3D, 0 );

// Add tracks/switch/position on track to railway network
rn.addAll( rt1, rt2, rt3, rs, pt );

// Create and initialize level
Level l = new Level( this, "level", SHAPE_DRAW_2D3D, 0 );
l.add( rn );
l.initialize();

//Add level on presentation
presentation.add(l);
```

4. Click anywhere in the graphical editor to display the agent properties. In the **Agent actions** section, type the following in the **On startup** field:
   `createRailwayNetwork();`
5. Run the model. The three tracks, a switch and a position on track appear.



Runtime:

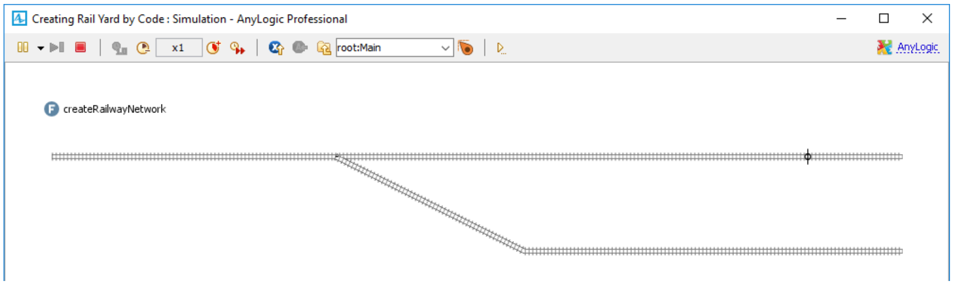

Creating Rail Yard by Code : Simulation - AnyLogic Professional
x1
root:Main
AnyLogic
createRailwayNetwork

Figure 9.31 A rail yard created by code

In this model, on startup, the function **createRailwayNetwork** executes. It creates all rail space markup shapes.

Train

A train is a sequence of one or several rail cars coupled with each other that can move in the rail yard and is controlled by the Rail Library and Process Modeling Library flowchart blocks.

Trains are created by **TrainSource** blocks and must be disposed by **TrainDispose** blocks. A train can be split into two trains by **TrainDecouple** and two trains can be combined into one by **TrainCouple.** Trains move in the rail yard under control of **TrainMoveTo** blocks. Trains are agents (objects of class **Agent**, or its subclasses if you are using custom train types) so trains can go through any Process Modeling Library blocks like **Delay**, **Seize**, **Release**, etc.

The cars in the train are ordered – there will always be the first car and the last car (which are the same in case the train contains only one car), as shown in Figure 9.32. The orientation of the cars at the time of the train creation is the same as that of the train, but later it can change as a result of coupling/decoupling. The length of the train (the length of the track portion occupied by the train) is equal to the sum of all the car lengths.

The Rail Yard Library does not have a concept of a locomotive, or of any other car type: all rail cars are considered to be equal. Any train can move at any speed or be coupled/decoupled at any side.

The train has *cruise speed*, *initial speed*, *acceleration* and *deceleration* properties. They are initially set up by **TrainSource** but can later be changed via the Train API. While the train is moving you can change its speed instantly or by applying acceleration and deceleration.

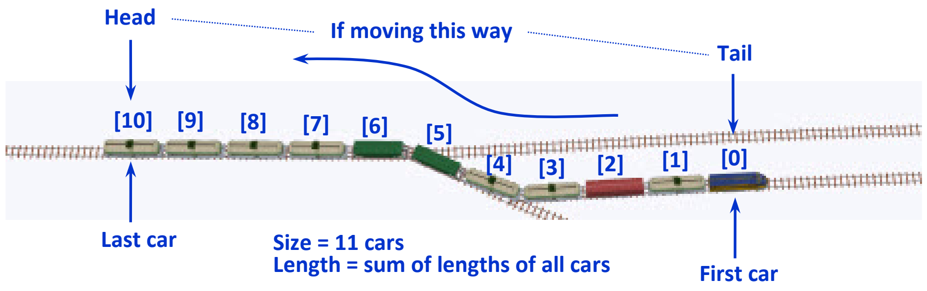


Figure 9.32 A train

Animation of a train is actually the animation of the rail cars of the train, – there are no specific graphics related to the train.

Custom train types

In case you need to access individual rail cars of a train or obtain some train properties dynamically, you should create a *custom train type*.

To create trains of a custom type:

1. Drag the **Train Type** element from the **Rail Library** palette to the graphical editor and follow the wizard instructions. When done, the graphical diagram of the train type will be opened. Here you can add parameters, variables, functions and any other AnyLogic elements to define custom logic.
2. In the **TrainSource** block, select the created custom train type in the **New train** property of the **Train and cars** section.

Here are some functions of interface **ITrain** that is implemented by custom train types, see *AnyLogic Help. Library Reference Guides: Rail Library* (The AnyLogic Company, 2020) for the full list:

Train contents and dimensions

- **boolean isEmpty()** – returns **true**, if the train contains no cars, **false** if it contains cars.
- **int size()** – returns the number of cars in the train.

- **double getLength(LengthUnits units)** – returns the length of the train (the sum of lengths of all cars) in specified length units. The **units** argument takes one of AnyLogic constants defining the length units. Some of the available constants are: **METER**, **FOOT**.*
- **Agent getFirst()** – returns the first car in the train.**
- **Agent getLast()** – returns the last car in the train.**
- **Agent getCar(int i)** – returns a car with a given index **i**, the first car index is 0, the last car – **size()-1**.**

Speed and acceleration

- **double getCruiseSpeed(SpeedUnits units)** – returns the cruise speed of the train in specified speed units. The **units** argument takes one of AnyLogic constants defining the speed units. Some of the available constants are: **MPS** (meters per second), **KPH** (kilometers per hour), **MPH** (miles per hour).*
- **setCruiseSpeed(double speed, SpeedUnits units)** – sets the new cruise speed of the train in specified speed units. The new speed must exceed 0.*
- **double getAcceleration(AccelerationUnits units)** – returns the acceleration of the train in specified acceleration units. The **units** argument takes one of acceleration units constants: **MPS_SQ** (meters per second^2) or **FPS_SQ** (feet per second^2).*
- **setAcceleration(double acceleration, AccelerationUnits units)** – sets the new acceleration of the train in specified acceleration units.*
- **double getDeceleration(AccelerationUnits units)** – returns the deceleration of the train in specified acceleration units.*
- **setDeceleration(double deceleration, AccelerationUnits units)** – sets the new deceleration of the train in specified acceleration units.*

Movement control

- **boolean isMoving()** – returns **true** if the train is moving, **false** if not.
- **boolean getDirection()** – returns **true** if the first car moves at the head of the train, **false** if not.
- **boolean getOrientation(boolean front)** – returns **true** if the orientation of the train is the same as the track where the given side (if **front** is **true**, this is front side, otherwise rear side) is located, **false** if not.
- **boolean isForwardOnTrack(boolean front)** – returns **true** if the train is moving towards the end of the track on which the given side is located, **false** if not.
- **Route getRoute()** – returns the route the train is currently following, if any. The route only exists when the train is handled by the **TrainMoveTo** block.

- **Agent getHeadCar()** – returns the car that is at the head of a moving train (either first or last). If the train is not moving, returns **null**.**
- **Agent getTailCar()** – returns the car that is at the tail of a moving train (either first or last). If c the train is not moving, returns **null**.**
- **double getSpeed(SpeedUnits units)** – returns the speed of the train in specified speed units. If the train has no cars, returns 0.*
- **double setSpeed(double v, SpeedUnits units)** – sets the new speed of the train in specified speed units. It applies immediately even if the train is moving. If the train is not moving, just remembers the speed but does not start the train. While the train is moving, speed cannot be set to 0.*
- **accelerateTo(double speed, SpeedUnits units)** – accelerates or decelerates the train to achieve a given speed in the specified speed units. Can only be called while the train is moving. It doesn't change the cruise speed and uses the current settings for acceleration/deceleration. You can't achieve a complete halt of the train movement using this function, since the stop is controlled by **TrainMoveTo** block.*
- **RailwayTrack getTrack(boolean front)** – returns the track where a given side of the train is currently located (if **front** is **true**, this is front side, otherwise – rear).
- **List<RailwayTrack> getAllOccupiedTracks** – returns all occupied railway tracks.
- **double getOffset(boolean front, LengthUnits units)** - returns the offset of a given side of the train relative to the track start point (in specified length units), 0 is the beginning of the track (if **front** is **true**, this is front side, otherwise – rear).*
- **RailwayTrack getTargetTrack()** – returns the target track of a moving train, **null** if target is not set.
- **double getTargetOffset(LengthUnits units)** – returns the target offset (on the target track) of a moving train in specified length units, assumes the target is set.*
- **double getDistanceToTarget(LengthUnits units)** – returns the distance from the current position to the target point in specified length units. Assumes the train is moving along a route (specified manually or calculated automatically) and the target is set.*
- **double getDistanceDriven()** – returns the distance driven by the train (in meters) since its creation or since the last call of the **resetDistanceDriven()** function.
- **resetDistanceDriven()** – resets the distance driven by the train to 0.

* - The **units** argument takes one of AnyLogic constants defining the corresponding measurement units. There is another function notation with the **units** argument

being omitted. It returns/sets the requested value in meters, meters per second, or meters per second^2, depending on the function.
** - If the rail car is of a custom type, object of this type will be returned.

Rail car

In the AnyLogic Rail Library, a rail car has dimensions (length and width), can move along a track, and can be coupled and decoupled with another car. A car can have 2D and 3D animation.

Rail cars are created by **TrainSource** blocks as a part of a train and are fully controlled by trains during their whole lifetime in the rail system.

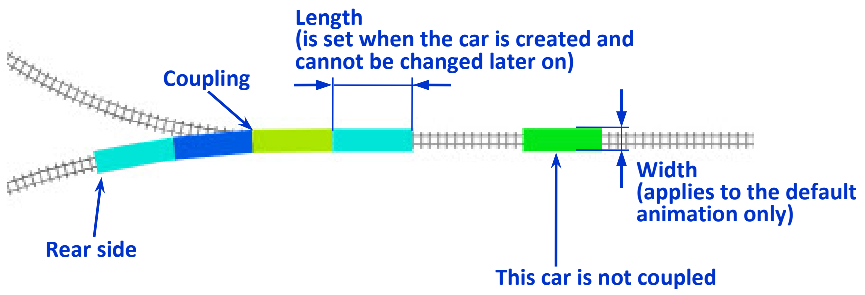


Figure 9.33 Rail car details explained on default 2D animation

Rail car has length and two sides: front and rear. The length of the rail car can only be set up at the time of its creation and cannot be changed later.

The default 2D animation of the car is a rectangle, see Figure 9.33. If the car is highlighted, a red border will be drawn around it. The default 3D animation is a parallelepiped.

Custom rail car types

From the Rail Library viewpoint there is no difference between a car and a locomotive or between freight and passenger cars, but you can make such a distinction in your model. You can create different rail car types with different properties and functions, assign different animation shapes and optionally specify different dimensions. And of course, you can associate different behavior logic. The custom rail car creation is described in Example 9.1: “Train stop”.

All custom rail car types implement interface **IRailCar**, which provides the low-level interface to car creation, movement and coupling/decoupling. Rail cars are fully controlled by trains, so the **IRailCar** API is still supported by the Rail Library mostly for compatibility with earlier models.

You can get the location (track, orientation on track, and offset) of the either side of the car. You can ask the car to *callback* and execute a custom action at the specified point of a given track. Arbitrary information can be passed to the callback code.

Here are some functions of interface **IRailCar**, see *AnyLogic Help. Library Reference Guides: Rail Library* (The AnyLogic Company, 2020) for the full list:

Car dimensions and appearance

- **setLength(double lengthInMeters)** – sets the length of the car in meters. Can only be done before the car is added to the rail yard, e.g. in the **Car setup** code of **TrainSource** block.
- **double getLength(LengthUnits units)** – returns the length of the car in specified length units.*
- **setWidth(double width, LengthUnits units)** - sets the width of the car in specified length units. Applies to the default animation only.*
- **double getWidth(LengthUnits units)** – returns the width of the car in specified length units.*
- **setColor(Color color)** – sets the color of the car. Although the color is always stored in a rail car, it only applies to the default animation.
- **Color getColor()** – returns the color of the car.
- **setShape(Shape shape)** – sets a custom 2D or 3D shape that will be used to animate this car.
- **Shape getShape()** – returns a custom shape that is used to animate the car, or **null**.

Information about car

- **Train getTrain()** – returns the train the car belongs to, or **null**.
- **RailwayTrack getTrack(boolean infront)** – returns the track on which a given side of car is currently located (if **infront** is **true**, this is front side, otherwise rear side).
- **Agent getCoupledCar(boolean infront)** – returns the car coupled with this one at a given side, and **null** if there is no coupled car. If **infront** is **true**, this is the front side, and if **false** – the rear side. If the rail car is of a custom type, object of this type will be returned.
- **boolean isFirst()** – returns **true** if this car is the first in the sequence of coupled cars, i.e. if there is no coupled car moving before this car.
- **boolean isLast()** – returns **true** if this car is the last in the sequence of coupled cars.
- **double getX(boolean infront)** – returns the X-coordinate of a given side of the car (if **infront** is **true**, this is front side, otherwise rear side).

- **double getY(boolean infront)** – same for the Y-coordinate.
- **double getOffset(boolean infront)** – returns the distance from a given side of car to the track start point in meters: 0 is the beginning of the track (if **infront** is **true**, this is front side, otherwise rear side).
- **double getDistanceDriven()** – returns the distance (in meters) driven by the rail car since its creation or since the last call of **resetDistanceDriven()**.
- **resetDistanceDriven()** – resets the distance driven by the rail car to 0.

Movement

- **boolean isMoving()** – returns **true** if the car is currently moving, **false** if not.
- **boolean getDirection()** – returns **true** if the car is moving (or was moving last time) forward, i.e. front before rear. Returns **false** if not.
- **boolean getOrientation(boolean infront)** – returns **true** if the car has the same orientation as the track on which the given side of the car is located. Returns **false**, if not.
- **boolean isForwardOnTrack(boolean infront)** – returns **true** if the direction of the car movement is the same as the track on which one (or both) sides of car are located. Returns **false** if not.

Callbacks

- **callbackAt(RailwayTrack track, double offset, Object info)** – requests the car to execute a callback (the code in the field **On at callback** of the **RailSettings** block) at the specified point, namely when its side that moves in front reaches a given offset on a given track. You can pass arbitrary information (**Object**) to the callback code to identify what kind of event it is.
- **callbackAt(RailwayTrack track, PositionOnTrack pointOnTrack, Object info)** – same as **callbackAt(track, offset, info)** where **offset** is the offset of the given position on track.

* - The **units** argument takes one of AnyLogic constants defining the corresponding measurement units. There is another function notation with the **units** argument being omitted. It returns/sets the requested value in meters, meters per second, or meters per second^2, depending on the function.