7 Mobile Units

MUs represent the materials that flow from object to object within the Frame. After creating new MUs, they move through the model and remain at the end until they are destroyed.

7.1 Standard Methods of Mobile Units

7.1.1 Create

Syntax:

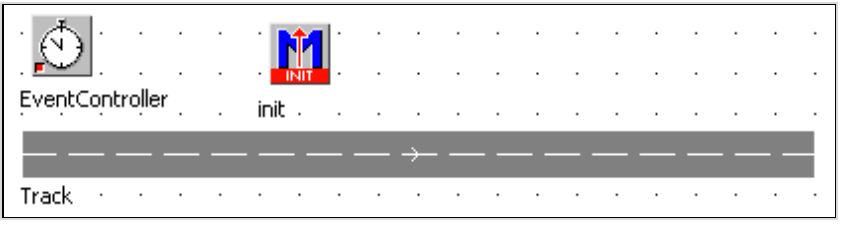
```
<MU_path>.create(<object>[,length]);
```

The method creates an instance of MU on <object>; MU-path is the path to the MU (e.g., class library). Optionally, you can specify a length on a length-oriented object (e.g., track, line). If you do not specify a length, then the MU is generated at the end of the object (ready to exit).

Example 72: Create MUs

At 200 m, 500 m, and 700 m, a transporter is to be created on a track which is 1000 m long.

Create the following Frame:



Note:
If you want to create track, which is a 1000 m long by dragging, then you first have to change the scaling in the Frame window. The default setting is a grid of 20 x 20 pixels and a scale of 1 m per grid point. Therefore, you have to change the scale so that it shows 50 m per grid point (Frame window, **TOOLS – SCALING FACTOR** …). Another possibility is the following: On the tab **CURVE** (track) turn off the option **TRANSFER LENGTH**.

Then you scale a length of 1000 m on the tab **ATTRIBUTES**. Then the track is no longer shown to scale.

S. Bangsow: Manufacturing Simulation with Plant Simulation, Simtalk, pp. 139 – 181, 2010.

The init method is to first delete all MUs and then to create the three transporters:

```
is
do
   deleteMovables;
   .MUs.Transporter.create(track,200);
   .MUs.Transporter.create(track,500);
   .MUs.Transporter.create(track,700);
 end;
```

After creating the MU on a place-oriented object, it can exit immediately. On a buffer, MUs are created on the first free place, if no place has been specified. On a length-oriented, object the MU is created as close as possible to the exit if no position was specified. Creation will fail if the capacity of the object is exhausted and the length-oriented object is shorter than the MU to be created (Return value: VOID).

7.1.2 MU-Related Attributes and Methods

Method	**Description**
<MU-path>.delete;	This method destroys the specified MUs. It does delete MUs in the class library.
<MU-path>.move; < MU-path>. move(<target>); <MU-path>.move(<index>);	Move the front of the transferred MU. If no argument is specified, then it will be transferred to each successor alternately. If should be moved to a particular successor, then you can use an index (index). The return value (boolean) is TRUE when moving was successful and FALSE if moving failed (successor is occupied, paused, failed). MUs cannot be moved to a source.
<MU-path>.transfer; <MU-path>.transfer (<target>); <MU-path>.transfer (<index>);	Transfer moves a MU from one object to another. It moves the entire length to the next object (not just the forefront like the method Move).
<MU-path>.numMU;	NumMu returns the number of MU, which are booked on the MU (integer).

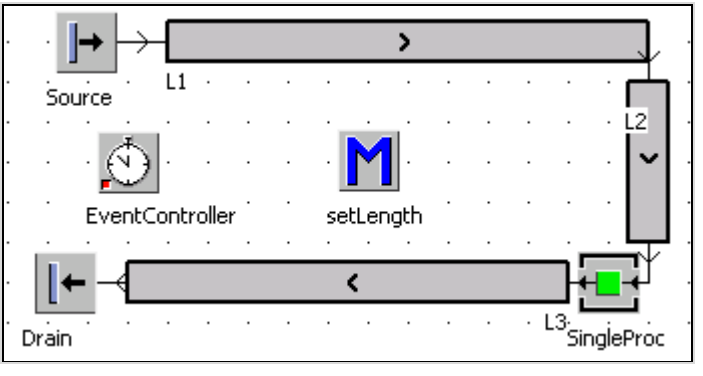
Within an entrance or exit control, you can access the triggering MU with the anonymous identifier "@".

7.2 Length, Width, and Booking Point

All MUs have the properties length, width, and related booking points. The booking point determines the position of the MU, from this it is booked on the next object and can be accessed from there. The position of the booking point must be always less than or equal to the length or width. Otherwise, you get an error message. In some cases it is necessary to dynamically adjust the length during the simulation. This may be when the processing is changing the length or if you repeatedly change the direction of transport during the simulation.

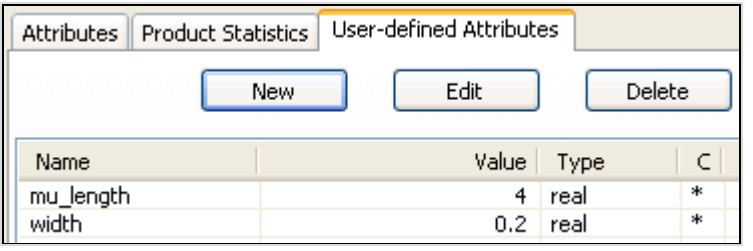
Example 73: Change MU Length

We want to simulate a small part of a production. There are steel beams processed. The beams are initially transported lengthwise, then crosswise, then processed, and then transported again lengthwise. The cross conveyor serves as a buffer. Create the following frame:



Create an entity ("beam") in the class library. The beam is 4 meters long and 20 cm broad. Set the booking points for each length and width to 0.1 meters. Redesign the icon of the beam: width 80 pixels, height 5 pixels. Delete the old icon and fill the icon with a new color. Set the reference point to the position 40.3. Change the icon "waiting" analogous. Make the following settings:

The source creates each 3 minutes one beam. The lines L1 and L3 are 12 meters long and transport the beams with a speed of 0.05 meters per second. Uncheck the option Rotate MUs in the tab Curve in L2 so that the line L2 transports the beams "cross". The SingleProc has 2 minutes processing time, 75% availability and 30 minutes MTTR. Start the simulation. If a failure occurs in SingleProc, the beams at L2 do not jam as intended. In the original setting, L2 promotes the cross transport, but the capacity is calculated on the length of conveyor line and length of the beam. If the beam is 4 meters long and L2 also exactly one beam fits on the line. To correct this, you have to temporarily reduce the length to the width of the beam (from 4 meters to 20 centimeters). After leaving the cross conveyor, you have to reset the length again to 4 meters. Therefore, insert two user-defined attributes into the class beam (class library):



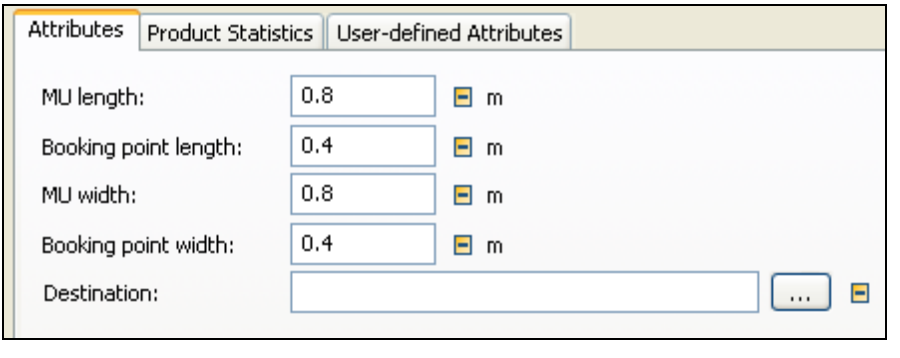
The method setLength must be called twice, once at the entry of MUs in the line L2 (reduce length) and once when MUs enter the SingleProc (set length back to the original value). The method could look as follows:

```
is
do
    if ?=L2 then
      @.muLength:=@.width;
    elseif ?=SingleProc  then
      @.muLength:=@.mu_length;
    end;
end;
```

The beams jam now at L2 if a failure occurs at the SingleProc.

7.3 The Entity

The entity does not have a basic behavior of its own. It is passed along from object to object. The main attributes are length and width. Booking points determine at which position the entity will be booked while being transported to the succeeding object (mainly track and line).



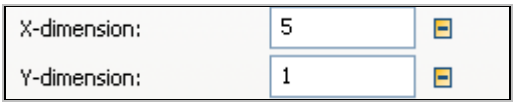
The destination can be used to store the destination station during transport operations (especially during transportation by a worker).

7.4 The Container

The container can load and transport other MUs. It has no active basic behavior of its own. The storage area is organized as a two-dimensional matrix. Each space can hold one MU. The container is transported from object to object along the connectors or by methods. With containers you can, for example, model boxes and palettes.

7.4.1 Attributes of the Container

The attributes of the container are mostly identical to those of the entity. In addition, the container has the following attributes:



The capacity of the container is calculated by multiplying the X-dimension with the Y-dimension. The access to the MUs, which are transported by the container, is analogous to the material flow objects.

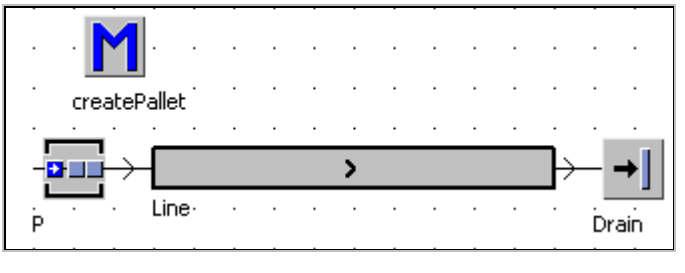
Method	Description
<MU-path>.cont	The method "cont" returns a MU, which is booked on the container (with the longest length of stay).
<MU-path>.pe(x,y)	With "pe" you get access to a place in the container.

7.4.2 Loading Containers

The approach is somewhat complicated. A container cannot exist just by itself. That is why you need another object, which can transport the container, e.g., to load a box. In addition, transporters can easily transport containers. For loading containers, you can use the assembly station (see chapter “AssemblyStation”) or the transfer station. You can also load the container with SimTalk methods.

Example 74: Loading Containers

We want to develop a method that creates a palette that is loaded with 50 parts and to pass these parts onto the simulation. Create the following Frame:



The method createPallet must first create the palette on P and then create parts on the palette until the palette is full. Create duplicates of container (Pallet, X-dimension: 25, Y-Dimension: 2) and entity (part) in the folder MUs.

Method createPallet:

```
is
do
  --create palette
  .MUs.pallet.create(p);
   -- create parts
  while not p.cont.full loop
     .MUs.part.create(p.cont);
  end;
  -- pass the palette
  p.cont.move;
end;
```

Digression: Working with Arguments 1
You will also need the facts described above in the exercises below. Therefore, it would be useful if the method could be applied to all similar cases. For this purpose, you need to remove all direct references to this particular simulation from the method (mark bold, italic).

```
is
do
  -- create palette
  .MUs.pallet.create(p);
   -- create parts
  while not p.cont.full loop
     .MUs.part.create(p.cont);
  end;
  -- pass the palette
  p.cont.move;
end;
```

The method only contains three specific references:

- Location of the palette (.MUs.pallet)
- Location of the part (.MUs.part)
- Target of creation (p)

These three items are declared as arguments.

```
(pallet,part,place:object)
```

Next, replace the specific details with the arguments (using find and replace in larger methods).

```
(pallet,part,place:object)
is
do
  -- create palette
  pallet.create(place);
   -- create parts
  while not place.cont.full loop
     part.create(place.cont);
  end;
  -- pass the palette
  place.cont.move;
end;
```

The method call now just has to include the arguments. Create an init method. The call of the method createPallet in the init method could look like this:

```
is
do
   createPallet(.MUs.pallet,.MUs.part,p);
end;
```

Start the init method. To check whether the method has really produced 50 parts, check the statistics of the palette (double-click the filled palette) on the tab **STATISTICS**.

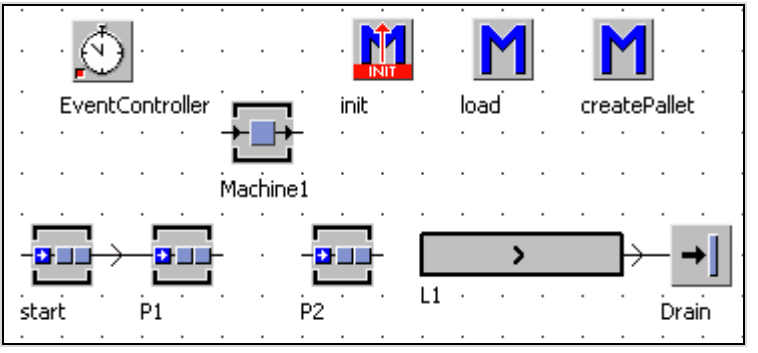
Contents:	50
Minimum contents:	0
Maximum contents:	50
Entries:	50

7.4.3 Unloading Containers

You can easily simulate unloading processes using the dismantle station. For some simulation tasks, this approach might prove to be too cumbersome, and the number of objects would be growing enormously. You can easily program unloading the palettes with SimTalk. Accessing the palette and the parts on the palette takes place by using the "underlying" object.

Example 75: Batch Production

You are to simulate batch production. Parts are delivered in containers and placed close to the machines. The machine operators remove the parts from the container and place the finished parts onto another container (finished parts). Once the batch is processed, the container with the finished parts will be transported to the next workplace and the empty palette will be transferred. Create a Frame with a single machine:



Settings: Start, P1, P2 each one place, processing time 0 seconds, L1 4 meters length, speed 1 m/s, Machine1 processing time one minute, no failures.

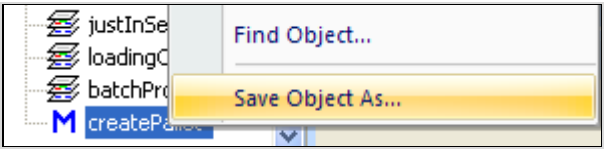
Digression: Reusing Source Code from Other Models

We want to use the same method (source code) as in the example "loading of containers" from the previous chapter. You can do this in two ways:

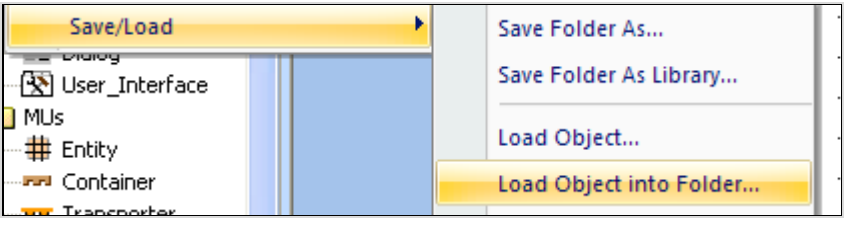
1. Save and import the method as an object file.
2. Export and import the methods as text.

Saving and importing the method as an object file:

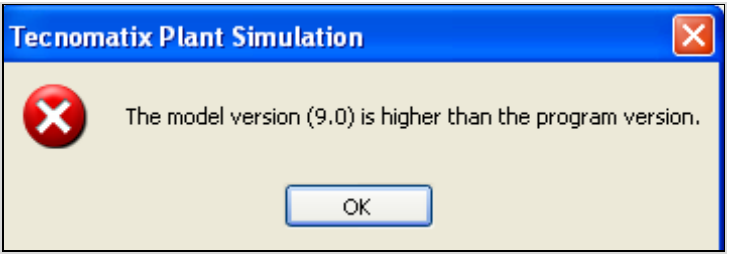
You can store classes and Frames as objects at any time. The functionality is located in the class library. Therefore, you first have to create a "class" out of the method in the Frame. Hold down the Ctrl key and drag the method from the Frame to the class library (if you do not hold down the Ctrl key, the method will be moved!). If needed, rename the method in the class library. From here, you can save the method as an object. Select the context menu (right mouse button) and select **SAVE OBJECT AS**.



Now you can load the method into another file as an object file. Click a folder icon with the right mouse button. Select **SAVE/LOAD-LOAD OBJECT INTO FOLDER…** from the context menu**:**

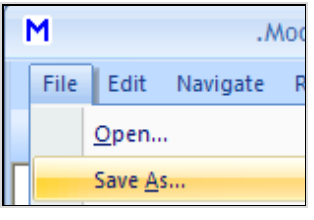


You can now select the file and insert the object into the class library. Caution: When you import the objects, the compatibility of versions/licenses will be considered. You cannot use objects from newer versions of Plant Simulation in older versions (or in eM-Plant).



Exporting and importing a method as text:

You can export and import the source code of the method as a text file. Open the method createPallet. Select **FILE – SAVE AS** in the method editor**:**



Select a location and a name for the file. Insert a method in the new Frame (without content, the contents will be completely replaced when you import a text file). Select **FILE – OPEN** in the method editor: Select the text file with the source code, and confirm. The text of the method is completely overwritten with the contents of the file.

Continuation Example Batch Production

If not already available, create the palette container (capacity 50 parts), entity: part. The init method should produce a full palette (50 parts) at the station P1 and an empty palette at the station P2.

```
is
do
  deleteMovables;
  createPallet(.MUs.pallet,.MUs.part,start);
  .MUs.pallet.create(p2);
end;
```

The simulation model (without a dismantle station) could look as follows. The palette itself transfers the first part of the palette onto the machine (exit control of P1). If a part on the machine is ready, it triggers the exit sensor of the machine. A

method transports the part to the palette on the station P2, and a new part from the palette on P1 to the machine. If the finished parts palette is full (p2), they will be transported to F1, the empty palette is moved from P1 to P2, and a new palette will be generated at the beginning.

Method load (exit control front P1, exit control front machine):

```
is
do
  if ? = P1 then
  -- load the first part onto machine1
    @.cont.move(machine1);
  elseif ?=machine1 then
  -- load the part onto the palette
    @.move(p2.cont);
    if p2.cont.full then
      p2.cont.move(L1);
      p1.cont.move(p2);
      -- create a new palette
       createPallet(.MUs.pallet,.MUs.part,start);
    else
    -- already parts on p1
      p1.cont.cont.move(machine1);
    end;
  end;
end;
```

You can access the part on the palette with

```
p1.cont.cont.
```

In our case p1.cont is a palette and p1.cont.cont is a part.

Digression: Working with Arguments 2 – User-Defined Attributes

Imagine that the simulation of the production contains 50 machines (each with two buffer places). You need to extend the method "load" for each machine. There is an easier option though, namely to separate event handling of the buffer and the machine.

Method: loadFirstPart (exit control front P1):

```
is
do
  -- move the first part to the machine
  @.cont.move(machine1);
end;
```

Method load (without branching after objects):

```
is
do
    -- load the part onto the palette
  @.move(p2.cont);
  if p2.cont.full then
    p2.cont.move(L1);
    p1.cont.move(p2);
    -- create a new palette
    createPallet(.MUs.pallet,.MUs.part,start);
  else
   -- already parts on p1
   p1.cont.cont.move(machine1);
  end;
end;
```

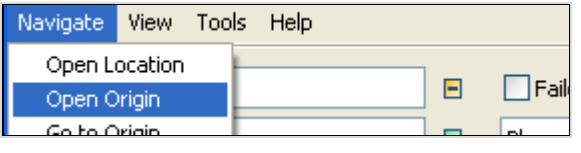
First step: Which object in the method has a reference to a specific case?

- P1, P2
- L1
- Machine1
- createPallet (only machine1)

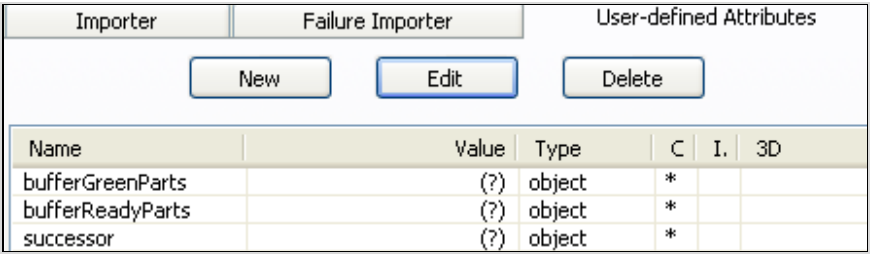
The anonymous identifier "?"can replace Machine1. P1 and P2 are buffers which belong to the machine. If you are using many similar machines, it is worth your while to define the respective attributes in the class SingleProc (in the class library). Name the attributes:

- bufferGreenParts (object)
- bufferReadyParts (object)
- successor (object)

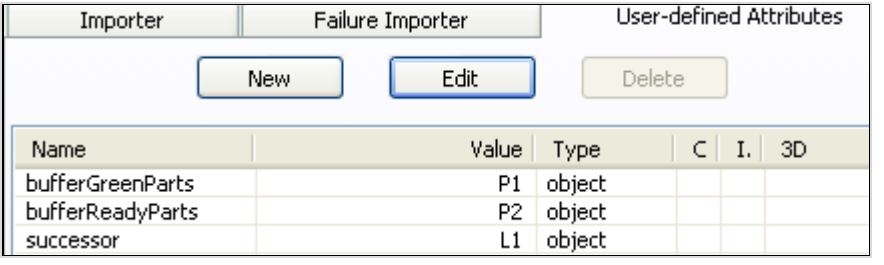
Open Machine1 in the Frame. In the dialog of the object select:



It will open the class of the SingleProc in the class library: Click the tab **USER-DEFINED ATTRIBUTES**, and define the two buffers and the successor:



Save your changes by clicking OK. You can now assign the two buffers (bufferGreenParts → P1, bufferReadyParts → P2) to Machine1 and F1 as successor. To do this, click the tab **USER-DEFINED ATTRIBUTE** in the object Machine1, double-click the relevant line and select the buffer or L1.



This makes programming the method easier for many applications:

- machine1 will be replaced by ?
- p1 will be replaced by ?.bufferGreenParts
- p2 will be replaced by ?.bufferReadyParts
- L1 will be replaced by ?.successor

Specific instructions for a machine are written into an "if then else …" statement.

Method load:

```
is
do
   -- load the part into the palette
  @.move(?.bufferReadyParts.cont);
  if ?.bufferReadyParts.cont.full then
    ?.bufferReadyParts.cont.move(?.successor);
    ?.bufferGreenParts.cont.move(
    ?.bufferReadyParts);
    -- create a new palette
   if ? = machine1 then
        createPallet(.MUs.pallet,.MUs.part,start);
   end;
  else
   -- already parts on p1
   ?.bufferGreenParts.cont.cont.move(?);
  end;
end;
```

Likewise, you can convert the method loadFirstPart. Define an attribute "machine" in the class of the PlaceBuffer (type object). Set Machine1 as the value for the attribute machine in the buffer P1. The method should look like this:

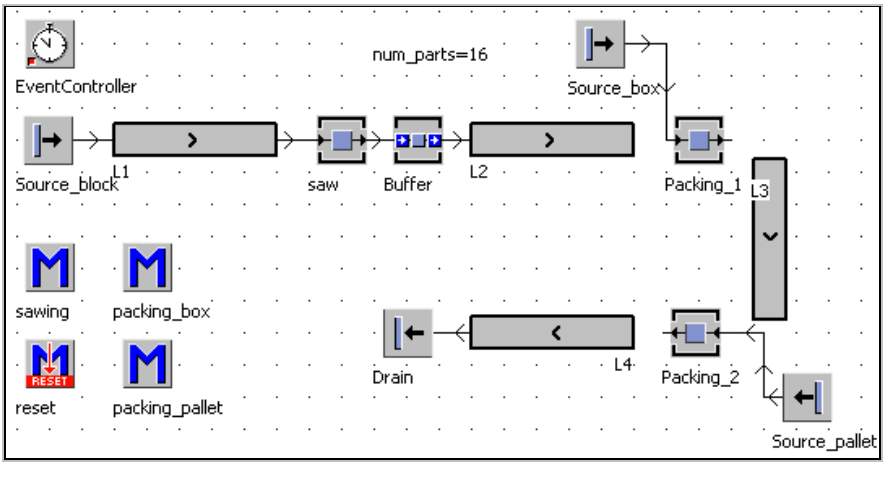
```
is
do
  -- move the first part to the machine
  @.cont.move(?.machine);
end;
```

Now you can easily extend the simulation without the need to write new methods. You have only to assign the methods to the sensors and set the attributes of the objects.

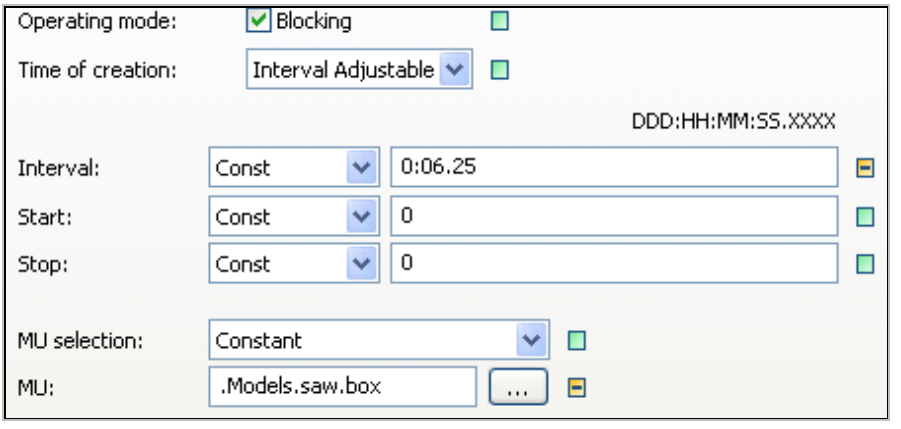
Example 76: Saw

You are to simulate the following process. A block with an edge length of 40 cm is to be sawed into 16 parts. Ten parts each will then be packed into a box. Ten boxes are packed in a carton. Between the saw and the individual packing stations, lines with a length of 5 meters will be set up. Create the folder "Saw" below models. Duplicate all required classes in this folder.

Create the following Frame:



Settings: L1, L2, L3, and L4 length 5 meters, speed 1 m/s; saw: processing time 10 seconds, packing_1 and packing_2 two seconds processing time, drain 0 sec. processing time. Create two entities (block, part) and two containers (box, palette). Block: 0.4 meter length, part 0.1 meter length. Change the icon of the part to a size of 7 x 7 pixels. Box: container, capacity 10 parts, palette: container, capacity 16 boxes. Arrange your sources so that they produce the correct type of MU (e.g., source_box).



Interval palette: 1:40, interval block: 0:10

Method sawing (exit control front of SingleProc saw): The method must destroy the block and create a certain number of parts. Creating the parts works best with a buffer object. The processing time of the sawing can be considered as processing time of the SingleProc. To make the simulation more flexible, define the number of parts outside of the method (e.g., in a global variable in the example: num_parts).

Method sawing (exit control front saw):

```
is
  i:integer;
do
  -- destroy block
  @.delete;
  -- create num_parts
  for i:=1 to num_parts loop
     .Models.saw.part.create(buffer);
  next
end;
```

At the end of line L2, the parts are to be packed into boxes. If a box is placed on the station Packing_1, the incoming part is transferred to the box. If the box is full, it will be transferred to line L3.

Method packing_box (exit control L2):

```
is
  box:object;
do
  -- wait for a box
  waituntil packing_1.occupied prio 1;
  box:=packing_1.cont;
  -- pack parts into the box
```

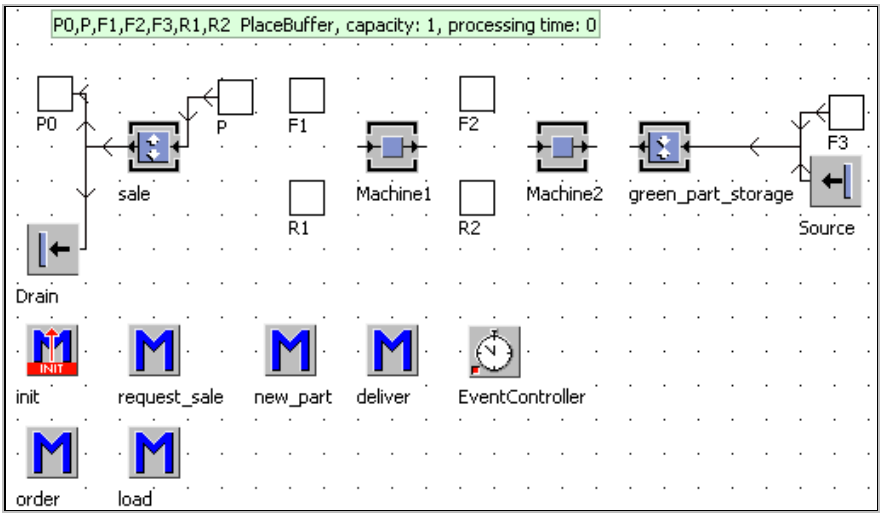
```
    @.move(box);
    if box.full then
      box.move(L3);
    end;
  end;
```

Similarly, you need to program the method packing_palette. To ensure a smooth start of the simulation, destroy the MUs when resetting the simulation.

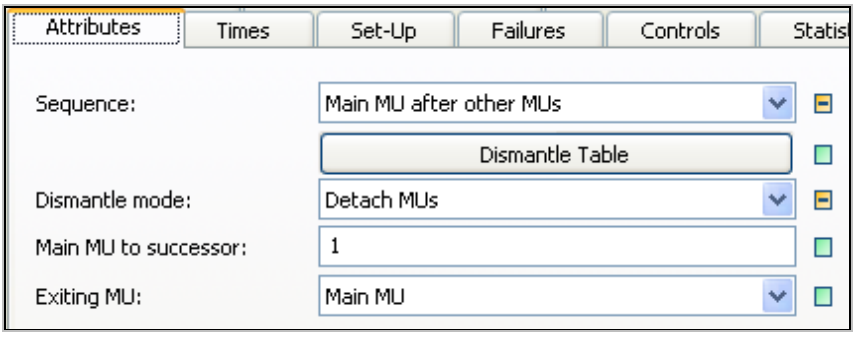
Note:
The task can also be solved with dismantle and assembly stations.

Example 77: Kanban Control

Simulations regarding the flow of materials have a fixed direction. The sources produce parts according to a fixed schedule (e.g., batch). The parts move from the source to the drain and trigger the production at the machines (push-control). Many companies (especially in Japan) use the opposite control concept. There, the succeeding stations trigger the production of the preceding stations. Trigger and main information carrier within this system is the Kanban card. We will simulate a Kanban container system. Create the following Frame. Also create the methods:



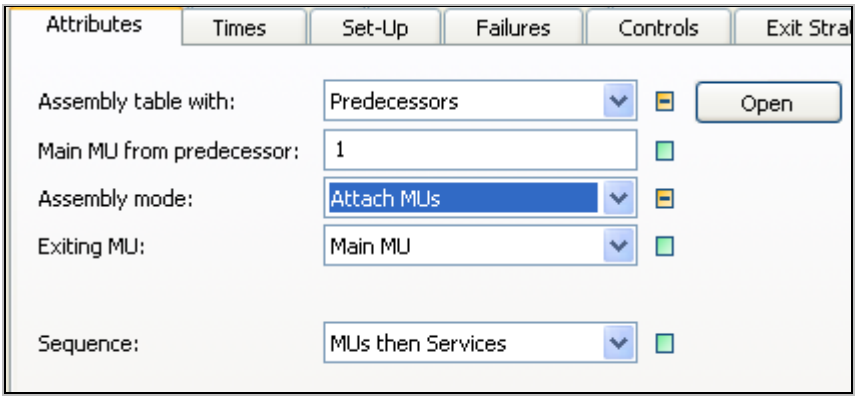
Settings: Machine1 and Machine2 processing time: 1 minute, 100% availability; sale: DismantleStation, successor number 1: P0 (for containers), successor 2: Drain 1 minute processing time. Create an entity .mus.part. The source produces .mus.part, interval 1 minute. Select the following settings in the DismantleStation sale:



Enter the following information into the dismantle table:

	MU	Number	Successor
1	.MUs.part	20	2

The assembly station green_part_storage loads 20 parts produced by the source onto a container that is transferred from F3. First, connect F3 with green_part_storage, then to the source. Select the following settings in the station green_part_storage:



Assembly table:

	Predecessor	Number
1	2	20

The production flow in this model should be like this: A container is located on the station sale and is unloaded gradually. If the container is empty, it will be transferred to the station F1 (finished part place of Machine1). The arrival of the container triggers the order of the unfinished parts. This takes place by transferring an empty container from the place R1 (unfinished part place of Machine1) to the place F2 (finished part place of Machine2). Machine2 sends a container from R2

to F3 and in this way triggers the delivery of the unfinished parts. The station green_part_storage loads the container with parts and sends it back to R2. The arrival of the container initiates the production at Machine2. If the finished part container of Machine2 (F2) is full, it will be transferred to the unfinished part place of Machine1 (R1). The finished parts of Machine1 will be transferred to the station sale (P1). Trigger and main control tool of the production are the kanban containers. They contain all information required for controlling the production. In your simulation model you can accomplish this with user-defined attributes. Create the container Kanban_container (capacity: 20 parts) in the class library. Create the following user-defined attributes:

Name	Value	Type	C	I.	3D
number	20	integer	*		
part	*.MUs.part	object	*		
processing_time	1:00.0000	time	*		
target_empty	(?)	object	*		
target_full	(?)	object	*		
workplace	(?)	object	*		

Step 1:
Create a filled container on P and empty containers on R1 and R2. Set the necessary information in the kanban containers. A kanban system represents a system of self-regulating control loops. For the present simulation, this means that a container shuttles between two places. The container, which is located on the station sale, shuttles between Machine1 and sale. Empty containers will be transported to F1, full containers always to P. The unfinished parts container of Machine1 (place R1) shuttles between Machine1 and Machine2 (place F2). In other words, if the container is full, it is transported to R1, if it is empty always to F2, etc. For each container, this information has been stored in user-defined attributes. The required init method should look like this:

```
is
  container:object;
do
   deleteMovables;
  -- initialisieren
  -- create a container at sale, load it
  -- target_empty: F1
  -- target_full: sale
  -- workplace: machine1
  container:=.MUs.kanban_container.create(p);
  container.target_empty:=F1;
  container.target_full:=P;
  container.workplace:=machine1;
```

```
    while not container.full loop
       .MUs.part.create(container);
    end;
    -- create kanban_container at r1 and r2
    container:=.MUs.kanban_container.create(r1);
    container.target_empty:=F2;
    container.target_full:=R1;
    container.workplace:=machine2;
    container:=.MUs.kanban_container.create(r2);
    container.target_empty:=F3;
    container.target_full:=R2;
    container.workplace:=green_part_storage;
end;
```

Step 2:
If the container on the station sale is empty, it orders new parts from Machine1 by sending the container to the station F1. Method request_sale, exit control front P0:

```
is
do
  -- empty container
  -- move to target_empty
  @.move(@.target_empty);
  -- request finished parts
end;
```

Step 3:
On arrival, a container on the finished part station, the machine has to send an unfinished part container as a request to the preceding workplace. To enable more convenient programming of this function, define two user-defined attributes in the class of the SingleProc in the class library:

Name	Value	Type	C	I.	3D
bufferGreenParts	(?)	object	*		
bufferReadyParts	(?)	object	*		

Type the machines into the respective buffers, for example Machine1:

Name	Value	Type	C	I.	3D
bufferGreenParts	R1	object			
bufferReadyParts	F1	object			

Method order, exit control front F1 and F2:

```
is
  container:object;
do
  -- get a reference to the unfinished parts container
```

```
    container:=@.workplace.bufferGreenParts.cont;
    -- send unfinished parts container
    container.move(container.target_empty);
  end;
```

Step 4:
After loading the container with unfinished parts, the container has to be transferred to the first unfinished part place (R2).Method deliver exit control front green_part_storage

```
  is
  do
    @.move(@.target_full);
  end;
```

Step 5:
After the arrival of the unfinished parts in the unfinished parts buffer, the first part is transferred to the machine. Create the user-defined attribute machine in the buffer class in the class library. Set the attribute machine of R2 to Machine2 and of R1 to Machine1.

Name	Value	Type	C	I.	3
machine	Machine2	object			

Method load exit control front R1 and R2:

```
  is
  do
    if @.occupied then
      @.cont.move(?.machine);
    end;
  end;
```

Step 6:
After completing processing of the parts on the machine, the machine transfers the part to the container, which is located on the finished part place. If the container is full, then it will be transferred to the station target_full. If the container is not yet full, a new part is loaded onto the machine. Method new_part exit control front Machine1 and Machine2:

```
  is
  do
    -- load part into the finished part container
    @.move(?.bufferReadyParts.cont);
    if ?.bufferReadyParts.cont.full then
      -- move container
      ?.bufferReadyParts.cont.move(
      ?.bufferReadyParts.cont.target_full);
```

```
    else
    -- load next part
      ?.bufferGreenParts.cont.cont.move(?);
    end;
  end;
```

The simulation now works according to the just-in-time principle.

7.5 The Transporter

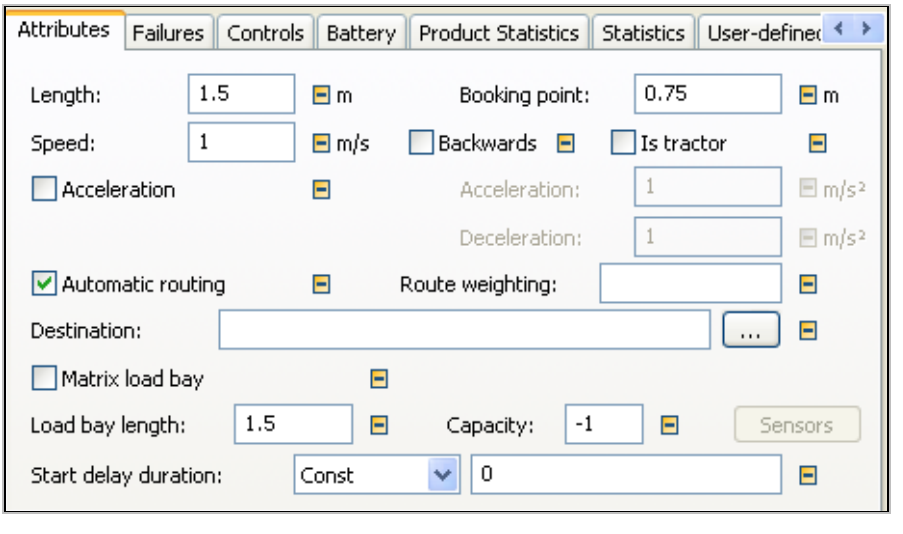
7.5.1 Basic Behavior

The Transporter moves on the track with a set speed forward or in reverse. Using the length of the track and the speed of the transporter, the time the Transporter spends on the track is calculated. At the exit, the track transfers the transporter to a successor. Transporter cannot pass each other on a track. If a faster transporter moves up close to a slower one, then it automatically adjusts its speed to the slower transporter. When the obstacle is no longer located in front of the Transporter, the Transporter accelerates to its previous speed. Transporters can have two types of load area:

- Matrix loading space
- Length-oriented loading space

7.5.2 Attributes of the Transporter

Create the object forklift (duplicate a transporter, speed 1 m/s) in the class library. Open the object by double-click it.



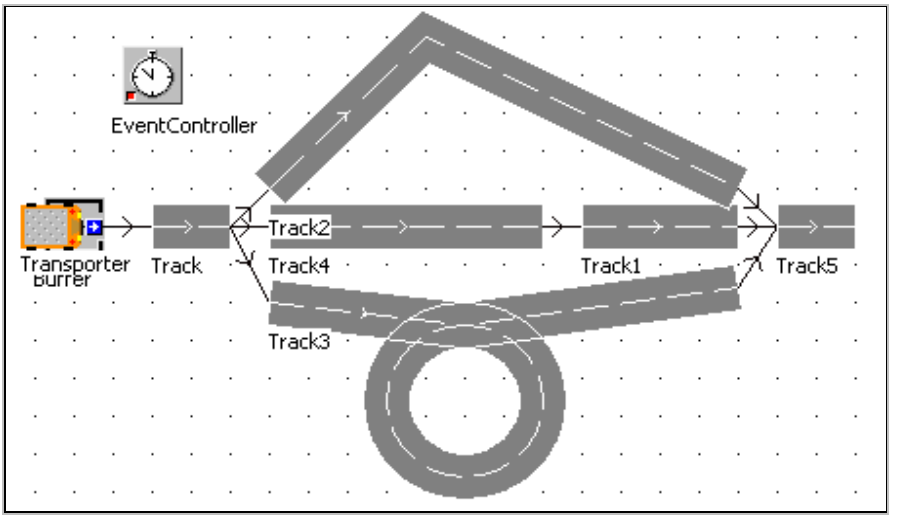
Length: The length of the transporter must be smaller than the length of the track, if you want to create a transporter on a track. The capacity of the tracks (setting capacity = -1) is calculated as the length of the tracks divided by the length of the transporter.

Speed: Enter the speed with which the transporter moves on the object track. The speed is a positive value (data type real). If you set the speed to 0, the Transporter stops. You can also simulate acceleration and deceleration of the transporter (option **ACCELERATION**).

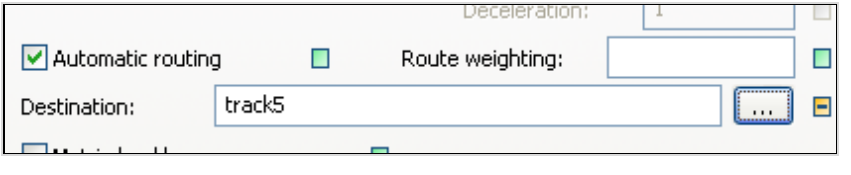
Backwards: This option activates moving of the transporter in reverse on the track (it also can be called by a method, for example, to drive back the transporter after unloading).

Automatic routing (+destination): If you select this option, then Plant Simulation searches along the connectors for the shortest route to the destination. All objects to the destination must be connected.

Example 78: Automatic Routing

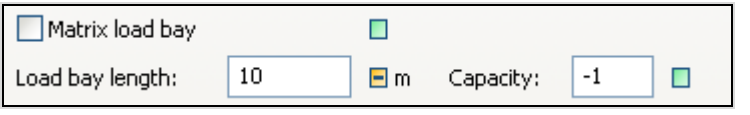


Drag a transporter from the class library to the buffer. Open the dialog of the transporter by double-clicking it. The destination of the transporter is track5.



Start the simulation and reduce the simulation speed. The transporter finds the shortest way.

Matrix load bay: If the option "matrix load bay" is selected, the xy coordinates then indicate the position of MUs on the load bay. If the box is cleared, Plant Simulation uses a length-oriented load bay.



Load bay length: Enter the length of the load bay. You can insert sensors and use the length-oriented load bay like a track, e.g., for representing panel carts, automobile transporters, loaders, cranes, etc.

Capacity: Enter the number of MUs, which can be located on the transporter, whole or in part, at any one time. -1 means that no limitations apply. This means that the loading bay of the transporter is full if all MUs are touching each other.

7.5.3 Routing

If a track has various successors, four different types for routing are available:

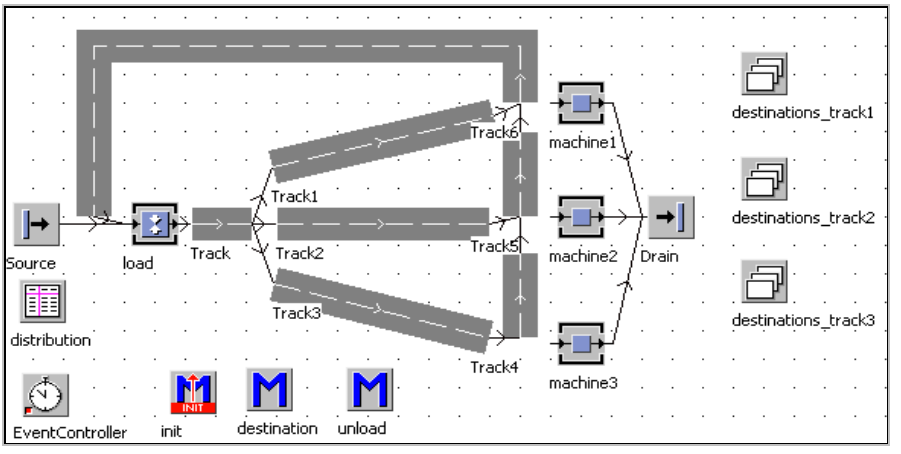
- Automatic routing
- Drive control
- Exit control
- Basic behavior

7.5.3.1 Automatic Routing

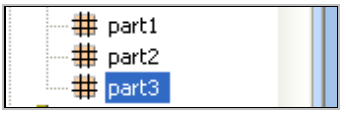
To use automatic routing, you need to supply the transporter with destination information, and assign information to the track about which destinations are to be reached on the track (target list). Plant Simulation searches the target lists of the successors for the destination of the track. Then Plant Simulation transfers the transporter to the first track whose target list contains the destination.

Example 79: Automatic Routing

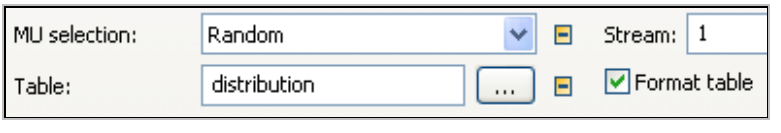
A source randomly produces three parts. A transporter loads the part and transports it to the relevant machine. Then the transporter drives back to the source. Each machine can only process one kind of part. A special track leads to each machine. Create the following Frame:



Duplicate the entity three times. Name the parts Part1, Part2, and Part3. Color the parts differently.



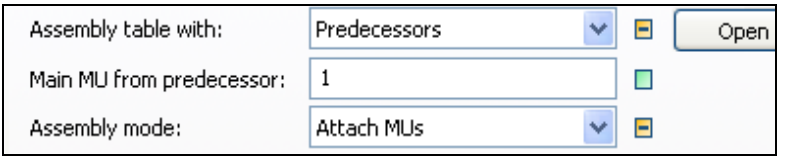
Go to the source. Enter an interval of 2 minutes. Select MU Selection random. Enter the table distribution into the text box Table.



Plant Simulation formats the table distribution. Open the table. Drag the parts from the class library to the table (this will enter the absolute path into the table). You can also enter the absolute path yourself. Next to the addresses of the parts, type in the distribution of the parts in relation to the total amount.

	object 1	real 2	string 3
string	MU	Frequencies	Name
1	.MUs.part1	0.10	
2	.MUs.part2	0.20	
3	.MUs.part3	0.70	

The source now produces part1, part2, and part3 in a random sequence.
An assembly station then loads the transporter. First, connect track6 with the assembly station, then with the source. Insert track6 so that the exit is located close to the assembly station. Select the following settings in the assembly station.



One part from the predecessor 2 is to be mounted.

	Predecessor	Number
1	2	1

The processing time of the assembly station is 10 seconds. The init-method inserts the transporter close to the exit of track6.

Method init:

```
is
do
   deleteMovables;
   .MUs.Transporter.create(track6,15);
end;
```

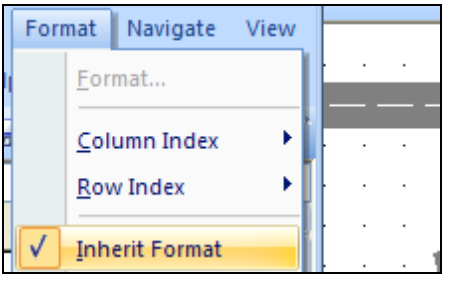
Determine the destination of the transporter depending on the name of the part. Set the value of the attribute destination of the transporter with a method. The output sensor (rear) of the assembly station is to trigger the method.

The method destination sets the attribute depending on the MU names.

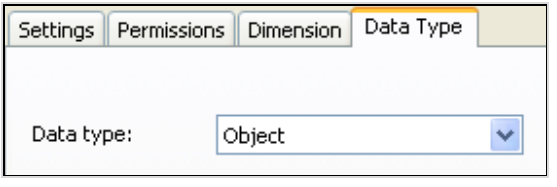
```
is
do
  -- @ denotes the transporter
  -- @.cont is the part on the transporter
  if @.cont.name="part1" then
    @.destination:=machine1;
  elseif @.cont.name="part2" then
    @.destination:=machine2;
  elseif @.cont.name="part3" then
    @.zielort:=machine3;
  end;
end;
```

Create and assign the destination list of the tracks. For creating the destination lists, use objects of type CardFile.

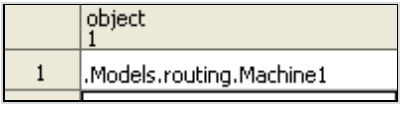
*The required data type is object. First, turn off inheritance (**FORMAT – INHERIT FORMAT**).*



Then click the list header (gray, string) with the right mouse button. Select ***FORMAT****. Select the data type object on the tab data type.*



Now enter the objects, which can be reached via the track. You can also enter the destinations by dragging the objects onto the list and dropping them onto the respective line. This inserts the absolute path. Insert Machine1 into the destinations_list1 and so on.



Enter the destination list on the tab Attributes of the track (forward destination list).

Move the parts at the end of the tracks.The parts are loaded onto the machines at the end of the tracks 1, 2, 3. To accomplish this, we use the destination addresses of the transporters. Enter the method into the exit controls of the tracks 1, 2, and 3(rear).

Method unload:

```
is
do
   -- @ is the transporter
   @.cont.move(@.destination);
end;
```

At the end, the transporter drives by itself to Machine3 on track3 to Machine2 on track2, etc., depending on which part is loaded.

Note:
Plant Simulation transfers the transporter onto the first object in whose destination list the destination of the transporter is registered. If the following track is failed, the transporter stops and waits until the failure is removed. While routing, Plant Simulation does not take the status of the tracks into account.

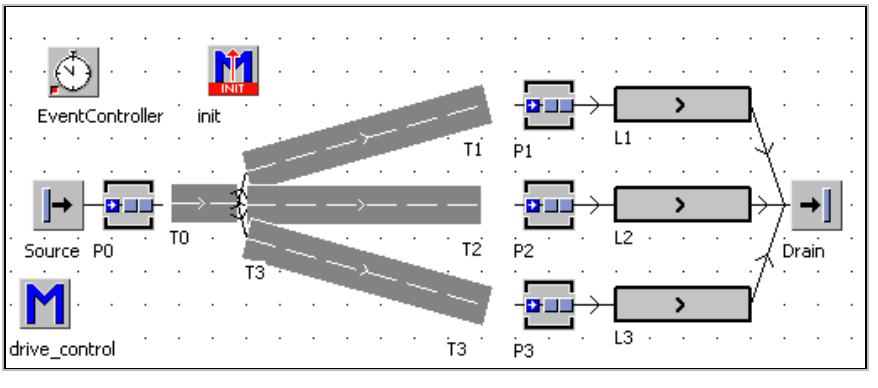
7.5.3.2 Driving Control

At a junction, you can determine the destination of the transporter with SimTalk, for example, depending on the availability and the load of the target station, and transfer the transporter on the correct track.

Example 80: Driving Control

We want to simulate a manipulation robot (e.g., FlexPicker by ABB). The robot can freely transport parts within a restricted area at high speed. You are to simulate the following problem. The robot takes parts from one place and distributes them onto three lines. The lines have an availability of 98% and an MTTR of 25 minutes. The robot itself has an availability of 99% and an MTTR of 30 minutes. The robot can reach an acceleration/deceleration of 100 m/s² and has a maximum speed of 10 m/s. The cycle time is 1.05 seconds (source interval). The speed of the lines is 0.1 m/s. The part has a length of 0.3 m. The robot has a work area with a diameter of 1.2 meters. Set the scaling factor in the Frame to 0.005.

Create the following Frame:



Length of the tracks T0: 0.2 m, W1: 0.75 m, W2: 0.7 m, W3: 0.75 m, processing time P1, P2, P3 3 seconds (to secure a distance between the parts), the capacity of all buffers is one part. The length of the transporter is 0.1 meter (booking point 0). The transporter must drive backwards after being inserted into the frame. Therefore, select Backwards in the dialog of the transporter in the class library. Proceed as follows.

1. Program the Init method. It creates a transporter on the track T0. When creating, a length is passed so that the transporter can trigger a backward exit sensor.

Method init:

```
is
do
   .MUs.Transporter.create(T0,0.1);
end;
```

2. Program the backward exit control drive_control: The transporter waits until a part is located on P0 and loads it. The transporter drives forward until the end of track T0. A single method is to be used for all controls. For that reason, the object and possibly the direction of the transporter will be queried in the method.

Method drive_control:

```
is
do
  if ? = T0 and @.backwards then
  ----------------------------------
  -- T0 exit backwards
   @.stopped:=true;
    waituntil P0.occupied prio 1;
    P0.cont.move(@);
    @.backwards:=false;
    @.stopped:=false;
  ----------------------------------
  end;
end;
```

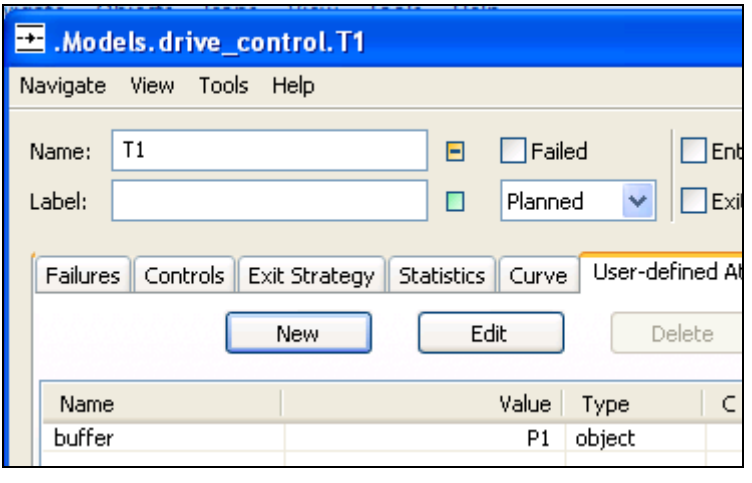
Assign the method drive_control to the track T0 as the exit control and the backward exit control.

3. Program the Exit control T0: At the end of the track T0, you have to be deciding to which place the transporter is to drive. The transporter waits until P1, P2, or P3 is empty. Starting with the station P1, the method queries whether the place is empty. The transporter will be transferred onto the track to the first empty place. Method drive_control, a new branch in the query if ? = T0 and @.backwards then ...:

```
is
do
  if ? = T0 and @.backwards then
  ----------------------------------
-- see above
  ----------------------------------
  elseif ?=T0 and @.backwards= false then
    -- T0 exit
   @.stopped:=true;
    waituntil P1.empty or P2.empty or P3.empty prio 1;
   --drive to the empty place
```

```
      if P1.empty then
            @.move(T1);
      elseif P2.empty then
            @.move(T2);
      elseif P3.empty then
            @.move(T3);
      end;
       @.stopped:=false;
    end;
  end;
```

4. Program the Exit control of the tracks T1, T2, and T3: The transporter loads the part into the buffer. After this, the transporter moves backwards to load a new part. To simplify matters, define the attribute buffer (type object) in the class track in the class library and assign the buffer P1 to the track T1, etc.



One control only is required for unloading. Therefore, you can program it as an else-block in the query of the objects.

```
is
do
  if ? = T0 and @.backwards then
  ------------------------------------
  -- T0 backwards exit
  -- see above
  ------------------------------------
  elseif ?=T0 and @.backwards= false then
    -- T0 exit
   -- see above
  else
   --unload onto buffer
```

```
      @.stopped:=true;
      @.cont.move(?.buffer);
      @.backwards:=true;
      @.stopped:=false;
    end;
  end;
```

7.5.4 Methods and Attributes of the Transporter

7.5.4.1 Creating a Transporter

You can use the method create for creating transporters.

Syntax:

```
<object>.create(target object) or
<object>.create(target object, length)
```

On length-oriented objects, you can determine the initial position at which the transporter will be inserted on the target object.

7.5.4.2 Unloading a Transporter

Unloading of transporters is accomplished analogous to unloading containers, for example, initiated by an exit control of the tracks.

Example 81: Unloading a Transporter

The content of the transporter will be transferred to the machine M2.

```
is
do
  @.cont.move(M2);
end;
```

Explanation: @ denotes the transporter in this case. You can access the part using the method cont of the transporter (@.cont). The method cont returns a reference to the part. You can then transfer the part to the machine M2 with …move(M2).

7.5.4.3 Driving Forward and Backward

Transporters often shuttle between objects. You need to change direction, so that the transporter can move in the opposite direction of the connectors.

Syntax:

```
@.backwards:=true/false;
```

The attribute backwards returns true, if the direction of the transporter is backward, and false, if the transporter is moving forward. You can set and get the value of the attribute backwards.

7.5.4.4 Stopping and Continuing

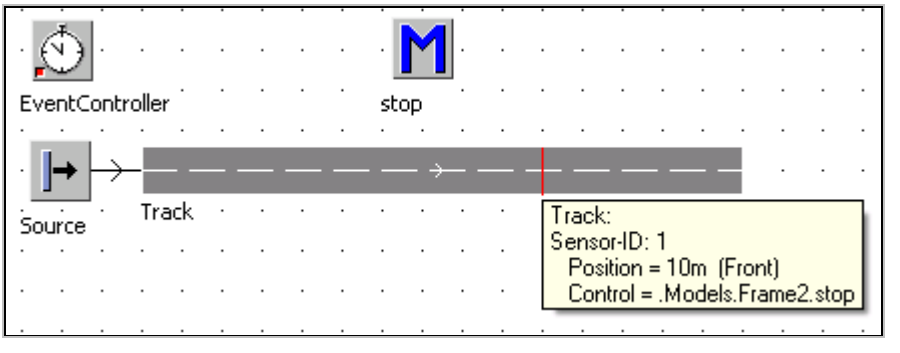
To stop and to continue after a certain time is the normal behavior of the transporter. While the transporter waits, you can, e.g., load and unload the transporter or recharge its battery. In SimTalk, you use the attribute stopped to stop the transporter and make it continue on its way.
Syntax:

```
@.stopped:=true; --stop the transporter
@.stopped:=false; --the transporter drives again
```

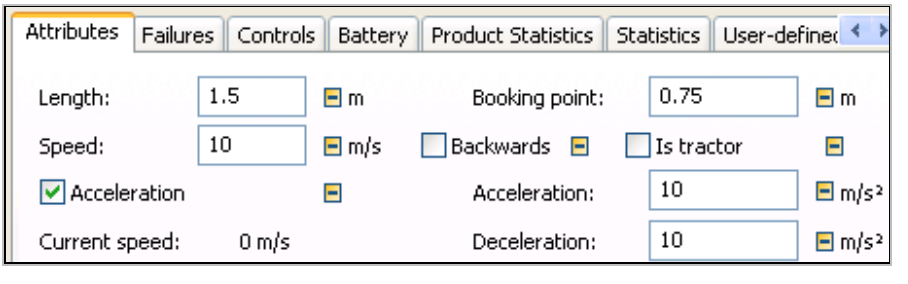
Another possibility to stop the vehicle is to set the speed to 0. The vehicle then slows down with the set acceleration and stops. You can start the vehicle again by setting the speed to its original value. In this way, you can take into account acceleration and slow down in the simulation. To demonstrate these two options, you can use the following example:

Example 82: Stopping Transporters

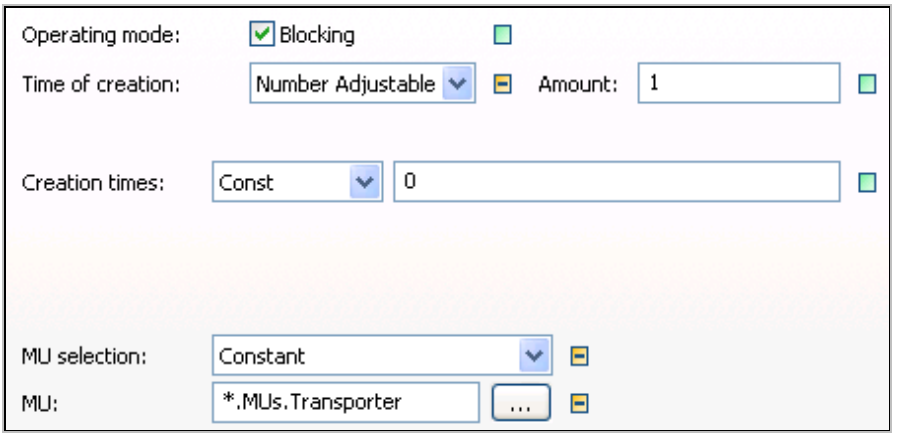
Create the following frame:



Make the following settings in the class transporter in the class library:



The source should produce only one transporter. This works with the following setting:



The transporter should stop after 10 meters.

***Variant 1**: You insert a sensor in the holding position and trigger at this position a method which stops the vehicle. The slow down is not taken into account. The method stop should look as follows:*

```
(SensorID : integer)
is
do
   @.stopped:=true;
end;
```

If you set the attribute stopped back to false, the transporter moves again.

***Variant 2**: The transporter slows down and stops at 10 meters. For this variant, you need a second sensor (approx. 5.1 meters), on wich you start the slow down. The method has in the second variant the following content:*

```
(SensorID : integer)
is
do
   @.speed:=0;
end;
```

The transporter starts again, if you set the speed to a value greater than 0.

7.5.4.5 Drive after a Certain Time

Often the transporter stands for a while before it starts again, for example, for loading and unloading. There are different ways for modeling the standing times of the transporter. Basically, you need to start the transporter with the same manner with which you have stopped it (either with the attribute stopped or speed). Even if the transporter runs at the end of the track and stops by itself, you need to

set the attribute stopped: = true to be able to start it later again (with change in direction) without problems.

You can pause the transporter using the method <path>.startPause(<integer>). After <integer> seconds ends the pause. If the transporter is paused, it stops and drives again, if the pause ends.

Example: Transporter Starts after a Certain Time

The transporter from the example above should start again after 10 seconds. The 10 seconds should be taken into account with the method startPause.

Variant 1: transporter stopped with stopped:=true

```
(SensorID : integer)
is
do
   @.stopped:=true;
-- Start again after 10 seconds
   @.startPause(10);
   @.stopped:=false;
end;
```

Variant 2: transporter stopped with speed:=0
It is easiest to use a second sensor to start (and unloading) the transporter again (e.g., SensorID 1 stop, SensorID 2 go). You should make sure that the vehicle, the second sensor also triggers and not stops before. The method could look as follows:

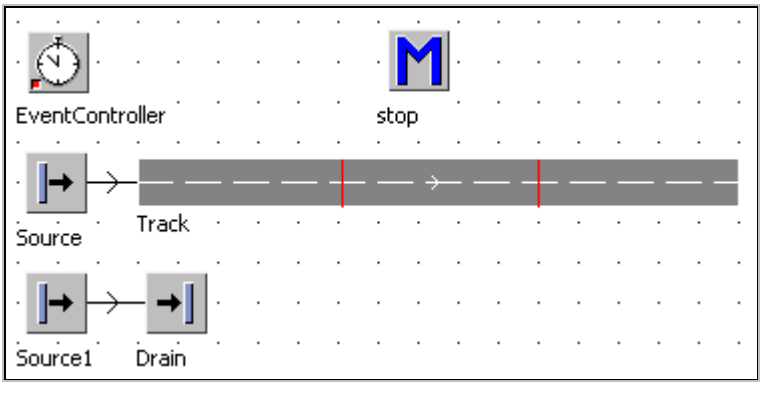
```
(SensorID : integer)
is
do
   if sensorID = 1 then
         @.speed:=0;
   elseif sensorID=2 then
         --start after 10 seconds
         @.startPause(10);
         @.speed:=10;
   end;
end;
```

A second method to start the transporter after a certain time is to use the time of the event controller in combination with a waituntil statement. Therefore, you take at a certain time the simulation time of the event controller and add a certain amount of time to it (e.g., 10 seconds). The simulation time (simTime) of the event controller is observable. Then you can interrupt the processing of the method until your set time is reached. An appropriate method could look as follows:

```
(SensorID : integer)
is
```

```
    simulTime:time;
do
    if sensorID = 1 then
          @.speed:=0;
    elseif sensorID=2 then
          simulTime:=
          eventController.simTime+num_to_time(10);
          --start again after 10 seconds
          waituntil eventController.simTime >=
          simulTime prio 1;
          @.speed:=10;
    end;
end;
```

Note: If no further event is there to process, the event controller stops the simulation. You may have to make a small bypass, so the simulation continues running. In the example above, for example, the following extension reaches (Source1 and Drain).



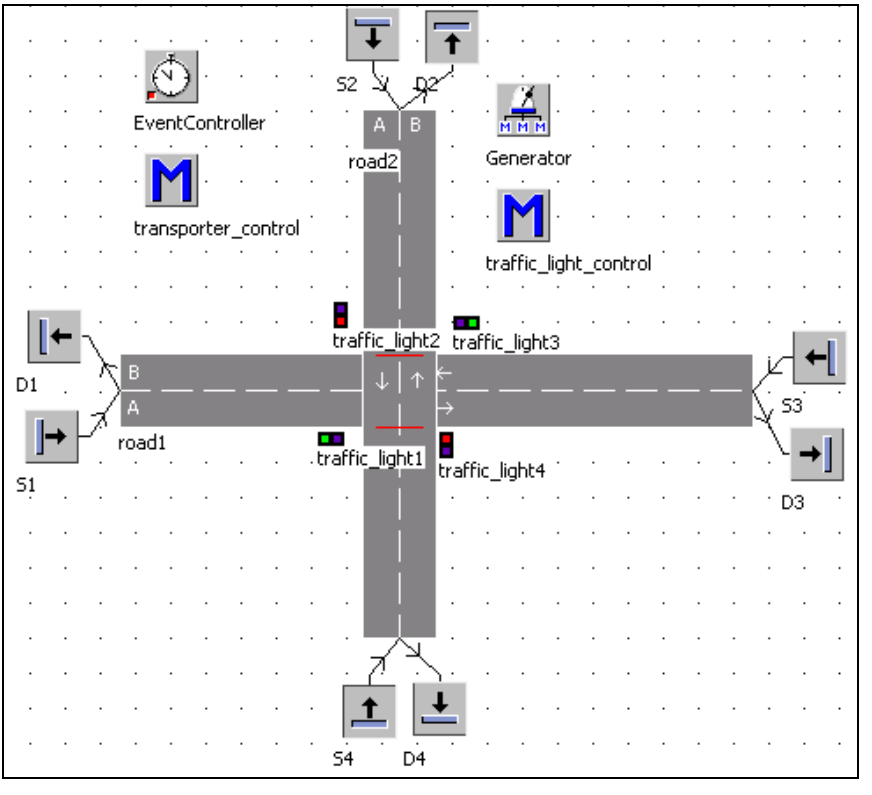
Set the Source1 so that each second one part is produced. This will always generate new events and the simulation continues.

7.5.4.6 Start Delay Duration

A transporter stops automatically when it collides with a standing transporter on the same track. Starts the first transporter again, then all collided transporters automatically start again. To model this behavior more realistic, you can use the attribute start delay duration (e.g., 0.5 seconds). The following transporter will start with a lag of 0.5 seconds, after start of the transporter in front. You can set the start delay duration on the dialog of the transporter (class library):

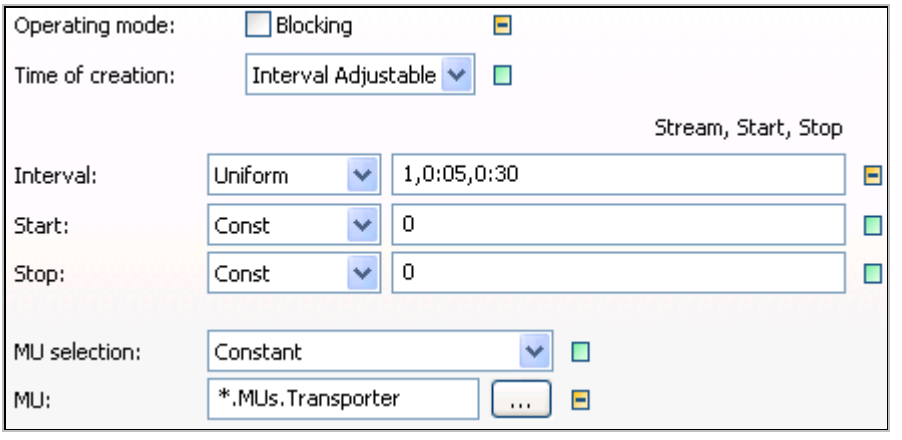
Example 83: Start Delay Duration, Crossroads

You are to simulate a simple crossroads. The crossroads is regulated by traffic lights. In this example, the transporter decides directly at the crossroads, if it stops or goes on (without slow down). All transporters behind it drive against it. Insert a traffic_light in the class library (duplicate a class SingleProc and rename it). Create two icons in the class traffic_light (icon1: green, icon 2: red). Insert in the class traffic_light a user-definded attribute "go" (data type boolean). Create a new frame. Set the scaling factor to 0.25 (Frame window – Tools – Scaling factor). Set up the following frame:

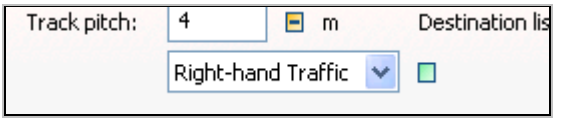


In this example, we also show how the TwoLaneTrack works.

Settings: The sources S1, S2, and S3 create each transporter. The interval of the creation should be randomly distributed. Make the following setting in the source S1:



Make this setting also for S2, S3, and S4. Set the stream (first number in field interval9 for each source to another value). The transporters move at a speed of 10 meters per second and accelerate with 10 m/s². To ensure that the transporter can pass each other, set in the ways a track pitch of 4 meters.



Create at the crossroads sensors on the lanes. You can specify for each lane its own sensors. You must uncheck for the other lane the checkboxes for the front and rear. For every track, you need to specify two sensors. One sensor in lane A and one sensor in lane B. All sensors will trigger the method transporter_control. In case of road1, it could look as follows:

ID	Position	Front	Rear	Path
1	35m / 55m	A		transporter_control
2	45m / 45m	B		transporter_control

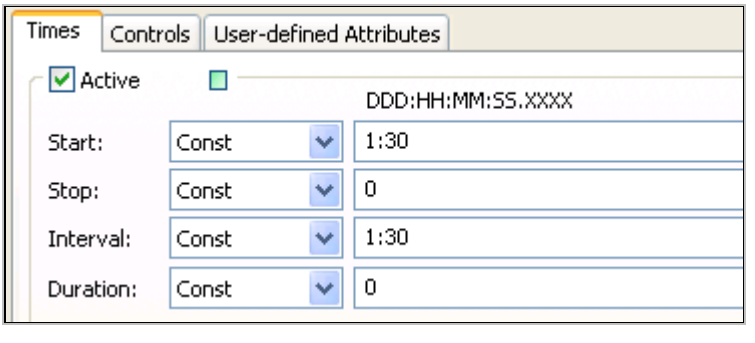
Initialize the simulation, so that two traffic lights show the icon 2 (red) (right mouse button – next icon) and the attribute go is false; the other two traffic lights show the green icon and the attribute go has the value true.

Traffic light control

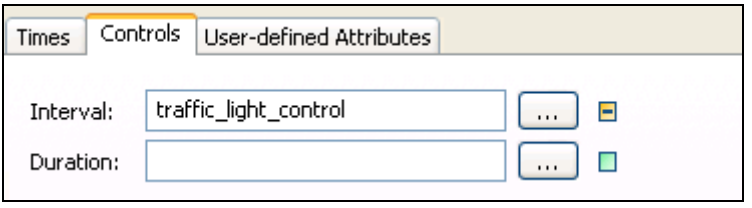
For the traffic light control, we use in this example a method (traffic_light_control) and a generator. The method switches the lights and the generator repeatedly calls the method with an interval of 1:30 minutes.
Method traffic_light_control:

```
is
do
   --switchs the traffic light
   -- icon1 green, icon2 red
   if traffic_light1.go then
         traffic_light1.go:=false;
         traffic_light1.CurrIconNo:=2;
         traffic_light3.go:=false;
         traffic_light3.CurrIconNo:=2;

         traffic_light2.go:=true;
         traffic_light2.CurrIconNo:=1;
         traffic_light4.go:=true;
         traffic_light4.CurrIconNo:=1;
   else
         traffic_light1.go:=true;
         traffic_light1.CurrIconNo:=1;
         traffic_light3.go:=true;
         traffic_light3.CurrIconNo:=1;

         traffic_light2.go:=false;
         traffic_light2.CurrIconNo:=2;
         traffic_light4.go:=false;
         traffic_light4.CurrIconNo:=2;
   end;
end;
```

You can test the method by starting this repeatedly. The lights should "switch". The method is called by the generator. Open the generator by double-click and make the following settings on the tab times:



Enter the method traffic_light_control in the field Interval on the tab Controls (more in chapter information flow objects).



The transporter should stop when the relevant traffic light has the value go=false and wait until go=true. Then the transporter should start again. If the lane A has sensor number 1 and the lane B has sensor number 2, the method transporter_control should look as follows:

```
(sensorID : integer)
is
do
   if ?=road1 and sensorID=1 then
         if traffic_light1.go=false then
               --traffic_light is red - stop
               @.stopped:=true;
               waituntil traffic_light1.go prio 1;
               @.stopped:=false;
         end;
   elseif ?=road1 and sensorID=2 then
         if traffic_light3.go=false then
               @.stopped:=true;
               waituntil traffic_light3.go prio 1;
               @.stopped:=false;
         end;
   elseif ?=road2 and sensorID=1 then
         if traffic_light2.go=false then
               @.stopped:=true;
               waituntil traffic_light2.go prio 1;
               @.stopped:=false;
         end;
   elseif ?=road2 and sensorID=2 then
         if traffic_light4.go=false then
               @.stopped:=true;
               waituntil traffic_light4.go prio 1;
               @.stopped:=false;
         end;
   end;
end;
```

Change the start delay duration in the class transporter (class library) to 0.5 seconds and watch what happens.

Note:
If the transporter does not stop exactly on the line of the sensor, then the reference point of the vehicle is in the wrong position. The reference point is in the default icon in the middle of the symbol. If the transporter is to stop exactly on the line, then you need to set the reference point to the right edge of the icon.

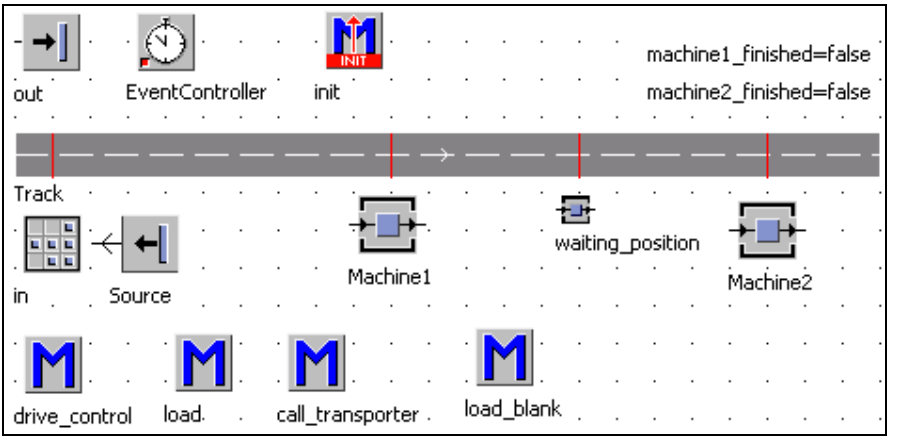
7.5.4.7 Important Methods and Attributes of the Transporter

Method/attribute	*Description*
`<MU-path>.startPause;` `<MU-path>.startPause (<time>);`	The method startPause immediately pauses the Transporter and sets the attribute pause to the value true. When a parameter is passed (integer greater than 0), it determines after which time (in seconds) the transporter changed back to the non-paused state.
`<MU-path>.startPauseIn (<time>)`	Pauses the transporter after the period defined in <time> has passed.
`<MU-path>.collided;`	Collided returns true if the transporter is collided with another transporter.
`<MU-path>.XDim;` `<MU-path>.YDim;`	Sets/gets the dimension of the matrix load bay
`<MU-path>.speed;`	Specifies the speed with which the transporter moves on the track. The speed must be equal to or greater than 0. If you set the speed to 0, then the transporter stops.
`<MU-path>.destination;`	Sets/gets the destination of the transporter

The Transporter also provides a number of methods and properties, which deal with the battery operation and related problems.

Example 84: Portal Loader Parallel Processing

You are to simulate a portal loader which loads two machines. The machines simultaneously process the same kind of part. The loader picks up parts at a place at the beginning of the track parts (capacity one part) and distributes them to the machines. If both machines are loaded and working, the loader waits empty between the two machines. When the first machine has finished processing, the loader drives to the machine and unloads it. A time of 5 seconds for the handling by the loader is considered (the movement in the z-axis is not simulated). Create the following Frame:



Settings: The source produces parts at an interval of 3:30 minutes. The processing time of Machine1 is 7:50 minutes, of Machine2 7:40 minutes. Both machines have an availability of 90% and an MTTR of 45 minutes. The track has a length of 25 meters; the transporter has a length of 1.5 meters, a speed of 1 m/s and a capacity of one part. The capacity of the object in is one part. Insert the following sensors into the track (as the drive_control):

ID	Position	Front	Rear	Path
1	1m		x	drive_control
2	10m	x		drive_control
3	15m	x		drive_control
4	20m	x		drive_control

The global variables machine1_finished and machine2_finished have the data type boolean and the initial value false.

1. Insert the transporter: The init method creates the transporter. Prior to that, all MUs will be destroyed.

Method init:

```
is
do
   deleteMovables;
   .MUs.Transporter.create(track,12);
   track.cont.destination:=in;
   track.cont.backwards:=true;
end;
```

2. Program the driving control: The transporter will be addressed for each trip. For this, the attribute destination is used. Destinations are assigned to certain sensor IDs. Once the transporter arrives at the destination (target sensor ID), the

transporter stops. The method drive_control needs a parameter for the sensor_ID. First, the transporter is to stop at the sensor_ID 1.

Method drive_control:

```
(sensorID : integer)
is
do
    if sensorID=1 and (@.destination=in
    or @.destination=out) then
        @.stopped:=true;
    end;
end;
```

3. Program the method for loading the unfinished part: The method load_blank uses the transporter as the parameter and should have the following functionality: The method waits until the place "in" is occupied. If Machine1 or Machine2 is empty, the transporter loads the part and drives to the empty machine. If both machines are occupied, the transporter drives to the waiting position. The handling time is taken into account by pausing the transporter.

Method load_blank:

```
(transporter:object)
is
do
   --search an empty and operational machine
   --wait for an unfinished part
   waituntil in.occupied prio 1;
   if machine1.empty and machine1.operational then
         in.cont.move(transporter);
         transporter.destination:=machine1;
   elseif machine2.empty and machine2.operational
   then
         in.cont.move(transporter);
         transporter.destination:=machine2;
   else
         transporter.destination:=waiting_position;

   end;
   --drive forward
   transporter.backwards:=false;
   -- 5 seconds handling time
   transporter.startPause(5);
   transporter.stopped:=false;
end;
```

Call within the method drive_control: If a loaded transporter arrives at sensor 1, first the part is transferred to the drain "out", then the method load_blank is called.

Method drive_control:

```
(sensorID : integer)
is
do
   if sensorID=1 and (@.destination=in or
       @.destination=out) then
         @.stopped:=true;
         if @.empty then
               load_blank(@);
         else
               -- move parts to the drain
               @.cont.move(out);
               --load new parts
               load_blank(@);
         end;
   end;
end;
```

4. Program the method load: The loaded transporter is to stop at the machine and load the part onto the machine (if the machine is operational). Thereafter, the transporter moves to the waiting position. If the transporter is empty, the transporter unloads the machine and drives to the drain "out". The method load requires three parameters: machine, transporter, and the direction to the waiting position (backwards true/false). The method load could look like this:

```
(transporter:object;machine:object;
directionWaitingPosition:boolean)
is
do
   transporter.stopped:=true;
   if transporter.occupied then
   -- transporter is loaded
         waituntil machine.operational prio 1;
         transporter.cont.move(machine);
         transporter.destination:=waiting_position;
         transporter.backwards:=
            directionWaitingPosition;
   else
   -- transporter is empty
         machine.cont.move(transporter);
         transporter.destination:=out;
         transporter.backwards:=true;
   end;
```

```
      -- start after 5 seconds
      transporter.startPause(5);
      transporter.stopped:=false;
end;
```

5. Program the method call_transporter: If the machines are ready, they must send a signal. The class SingleProc provides the method ready that returns true if the station has finished processing parts. This value, however, is not observable. One solution would be the following: If the machine is ready, the processed part then triggers a control that sets a global variable to true (e.g., machine1_finished). Global variables are observable and an appropriate action can be triggered. Within the frame, the ready variables consist of the machine name and "_finished". A universal method for registering the finished machines could look like this:

Method call_transporter:

```
is
do
  -- ? object, that calls
  str_to_obj(?.name+"_finished"):=true;
end;
```

str_to_object converts a string (object name) to an object reference. Assign the method call_transporter to the exit control (front) of Machine1 and Machine2.

6. Complete the method drive_control: Within the drive_control, the method load must be called at the positions of Machine1 and Machine2. A control for the waiting position is established at the position of the waiting position: If Machine1 or Machine2 is operational and empty, the transporter drives into the station and delivers a new part. Otherwise, the transporter waits until Machine1 or Machine2 is ready. The transporter might change its direction, set a new destination, and set the finished variable of the machine to false. Finally, the transporter drives to the machine. The completed method drive_control should look as follows:

```
(sensorID : integer)
is
do
  if sensorID=1 and (@.destination=in or
  @.destination=out)
  then
    @.stopped:=true;
    if @.empty then
      load_blank(@);
    else
      -- move parts to the drain
      @.cont.move(out);
      -- load new parts
```

```
      load_blank(@);
    end;
  elseif sensorID=2 and @.destination = machine1
  then
    load(@,machine1,false);
  elseif sensorID=4 and @.destination = machine2
  then
    load(@,machine2,true);
  elseif sensorID=3 and @.destination =
         waiting_position then
    stopped:=true;
    --new part if one machine is empty and
    --operational
    if (machine1.empty and machine1.operational)  or
       (machine2.empty and machine2.operational)
    then
      @.destination:=in;
      @.backwards:=true;
    else
      -- wait until one machine has finished
      waituntil machine1_finished or
      machine2_finished
      prio 1;
      if machine1_finished then
        @.destination:=machine1;
        @.backwards:=true;
        machine1_finished:=false;
      elseif machine2_finished  then
        @.destination:=machine2;
        @.backwards:=false;
        machine2_finished:=false;
      end;
    end;
    @.stopped:=false;
  end;
end;
```