

AnyLogic V

User's Manual



© 1992-2004 XJ Technologies Company Ltd.

www.xjtek.com

Copyright © 1992-2004 XJ Technologies. All rights reserved.

XJ Technologies Company Ltd

AnyLogic@xjtek.com

<http://www.xjtek.com/products/anylogic>

Contents

1. CREATING ANYLOGIC MODEL	1
1.1 ANYLOGIC MODELING LANGUAGE.....	1
1.2 STARTING ANYLOGIC	2
1.2.1 <i>Working with projects</i>	4
1.3 EDITING THE PROJECT	8
1.3.1 <i>Project window</i>	9
1.3.2 <i>Properties window</i>	12
1.3.3 <i>Arranging windows</i>	14
1.4 MODEL ELEMENTS.....	16
1.4.1 <i>Project</i>	16
1.4.2 <i>Package</i>	17
1.4.3 <i>Library</i>	17
1.4.4 <i>Experiment</i>	18
1.5 ACTIVE OBJECT	20
1.5.1 <i>Structure diagram</i>	21
1.5.2 <i>Diagram editors. Generic operations</i>	23
1.5.3 <i>Active object icon</i>	29
1.5.4 <i>Encapsulated objects</i>	34
1.5.5 <i>Root object</i>	36
1.5.6 <i>Active object data</i>	37
1.5.7 <i>Active object behavior</i>	38
1.5.8 <i>Active objects interaction</i>	40
1.5.9 <i>Writing code for an active object</i>	43
1.5.10 <i>Active object inheritance</i>	46
2. REPLICATED OBJECTS	48

2.1	CREATING A REPLICATED OBJECT	48
2.1.1	<i>Replication factor parameterization</i>	<i>49</i>
2.2	ACCESSING AND MODIFYING A REPLICATED OBJECT AT RUNTIME	49
2.2.1	<i>Accessing replicated objects from code.....</i>	<i>50</i>
2.2.2	<i>Adding/removing objects to/from a replicated object.....</i>	<i>50</i>
2.3	CONNECTING REPLICATED OBJECTS.....	51
2.3.1	<i>Connecting interface elements of replicated objects graphically.....</i>	<i>51</i>
2.3.2	<i>Connecting replicated objects with other objects</i>	<i>52</i>
2.3.3	<i>Connecting individual elements of a replicated object.....</i>	<i>55</i>
2.3.4	<i>Connecting replicated objects programmatically.....</i>	<i>56</i>
2.3.5	<i>Creating a ring of cells.....</i>	<i>57</i>
2.3.6	<i>Creating a torus of cells.....</i>	<i>58</i>
2.4	VIEWING AND MODIFYING REPLICATED OBJECTS AT RUNTIME	60
2.5	ANIMATING A REPLICATED OBJECT	61
3.	PARAMETERS.....	63
3.1	PARAMETER TYPES.....	63
3.2	DEFINING PARAMETERS	65
3.3	SETTING UP ACTUAL PARAMETER VALUES.....	68
3.4	MODIFYING PARAMETERS AT RUNTIME.....	70
3.4.1	<i>Modifying parameters from Model Explorer</i>	<i>70</i>
3.4.2	<i>Modifying parameters from AnyLogic animation.....</i>	<i>70</i>
3.4.3	<i>Accessing and modifying parameters programmatically.....</i>	<i>71</i>
3.4.4	<i>Defining parameter change handlers</i>	<i>71</i>
3.5	PARAMETER PROPAGATION.....	72
3.6	DYNAMIC PARAMETERS.....	75
3.7	OBSERVING MODEL BEHAVIOR WITH DIFFERENT MODEL PARAMETERS.....	75
3.8	OPTIMIZING MODEL PARAMETERS.....	76

4. VARIABLES.....	77
4.1 DEFINING A VARIABLE.....	77
4.1.1 <i>Initializing a scalar variable</i>	80
4.1.2 <i>Viewing and modifying scalar variables at runtime</i>	81
4.2 EQUATIONS.....	87
4.3 VARIABLE SHARING.....	87
4.3.1 <i>Variable sharing rules</i>	88
4.3.2 <i>Connecting variables</i>	90
4.4 CHANGING VARIABLES AND REACTING TO THEIR CHANGES	95
5. EQUATIONS.....	98
5.1 EQUATION TYPES	98
5.2 DEFINING AN EQUATION	103
5.3 FUNCTIONS.....	105
5.3.1 <i>Predefined functions</i>	106
5.3.2 <i>Using intelli-sense</i>	108
5.3.3 <i>Lookup tables</i>	109
5.3.4 <i>Mathematical functions</i>	114
5.3.5 <i>Algorithmic functions</i>	115
5.4 ALGEBRAIC LOOPS	117
5.5 RUNTIME ERRORS CAUSED BY EQUATIONS.....	118
5.6 NUMERICAL METHODS	118
6. MATRICES AND HYPER-ARRAYS.....	121
6.1 MATRICES	121
6.1.1 <i>Defining a matrix</i>	121
6.1.2 <i>Setting an initial value for a matrix</i>	121
6.1.3 <i>Accessing and modifying a matrix from Model Viewer</i>	123
6.1.4 <i>Accessing and modifying a matrix from code</i>	124

6.1.5	<i>Using matrices in equations</i>	128
6.1.6	<i>Working with matrices</i>	130
6.2	HYPER-ARRAYS	132
6.2.1	<i>Enumerations</i>	133
6.2.2	<i>Defining variables of hyper-array type</i>	134
6.2.3	<i>Setting an initial value for a hyper-array</i>	135
6.2.4	<i>Aggregation functions</i>	139
6.2.5	<i>Using hyper-arrays in equations</i>	140
7.	MESSAGE PASSING	144
7.1	PORTS	145
7.1.1	<i>Adding a port to an active object class</i>	145
7.1.2	<i>Port queue</i>	149
7.1.3	<i>Connecting ports</i>	150
7.1.4	<i>Connecting ports to statecharts</i>	153
7.1.5	<i>Message routing rules</i>	155
7.1.6	<i>Sending and processing messages</i>	156
7.1.7	<i>Receiving messages</i>	159
7.1.8	<i>Inspecting port state at runtime</i>	161
7.2	MESSAGES	163
7.2.1	<i>Defining message classes</i>	163
7.2.2	<i>Cloning messages to avoid sharing violation</i>	166
7.2.3	<i>Messages encapsulating/ inheriting other messages</i>	166
7.3	DEFINING CUSTOM PORT CLASSES	167
7.3.1	<i>Predefined port classes</i>	167
7.4	MESSAGE PASSING USE CASES	172
7.4.1	<i>Filtering messages by message contents</i>	172
7.4.2	<i>Filtering messages by message type</i>	173

7.4.3	<i>Sending a message with a delay</i>	175
7.4.4	<i>Sending a message with a delay dependent on the message field</i>	177
7.4.5	<i>Getting information on connected objects</i>	178
7.4.6	<i>Sending messages to all recipients</i>	181
7.4.7	<i>Sending a message to a specific recipient / a group of recipients</i>	182
7.4.8	<i>Verifying port connections at runtime</i>	192
7.4.9	<i>Defining message class constructors</i>	195
7.4.10	<i>Modeling a LIFO queue</i>	196
7.4.11	<i>Modeling a priority queue</i>	199
7.4.12	<i>Connecting ports at runtime</i>	200
7.4.13	<i>Modeling a system with dynamically changing structure</i>	204
7.4.14	<i>Implementing instantaneous data feedback</i>	205
7.4.15	<i>Implementing instantaneous message exchange</i>	206
7.4.16	<i>Message passing in Enterprise Library</i>	211
8.	TIMERS	218
8.1	DYNAMIC TIMERS.....	219
8.2	STATIC AND CHART TIMERS.....	220
9.	STATECHARTS	222
9.1	CREATING A STATECHART.....	222
9.2	STATECHART DIAGRAM.....	224
9.3	EXECUTION ORDER.....	236
9.4	TRIGGERING A TRANSITION.....	238
9.4.1	<i>Immediate triggering</i>	239
9.4.2	<i>Triggering after a timeout</i>	240
9.4.3	<i>Change event trigger</i>	240
9.4.4	<i>Signal event</i>	243
9.5	OBSERVING STATECHART AT RUNTIME.....	247

9.5.1	<i>Animated statechart diagram</i>	247
9.5.2	<i>Debugging a statechart</i>	248
10.	STOCHASTIC MODELING	250
10.1	RANDOM NUMBER GENERATOR.....	250
10.2	PROBABILITY DISTRIBUTIONS.....	251
11.	RUNNING AND OBSERVING A MODEL	258
11.1	RUNNING THE MODEL WITH ANYLOGIC.....	259
11.1.1	<i>Creating and destroying the model</i>	259
11.1.2	<i>Controlling the model execution</i>	260
11.1.3	<i>Setting up model simulation speed</i>	264
11.2	VIEWING AND CONTROLLING THE MODEL.....	265
11.2.1	<i>Model Explorer</i>	266
11.2.2	<i>Animated structure diagram</i>	270
11.2.3	<i>Animated statechart diagram</i>	273
11.2.4	<i>Inspect window</i>	275
11.2.5	<i>Log window</i>	276
11.2.6	<i>Chart window</i>	278
11.3	DEBUGGING THE MODEL	280
11.4	SETTING UP MODEL EXECUTION PARAMETERS	281
11.5	OPTIMIZING A MODEL.....	281
12.	ANIMATION	282
12.1	ANIMATION CONCEPTS	282
12.1.1	<i>Reflecting the state of an object in animation</i>	283
12.1.2	<i>Animating hierarchical models</i>	285
12.1.3	<i>Interactive control of animation</i>	287
12.2	ANIMATION DIAGRAM	287

12.2.1	<i>Animation editor</i>	287
12.2.2	<i>Animation shapes</i>	292
12.2.3	<i>Indicators</i>	299
12.2.4	<i>Controls</i>	302
12.2.5	<i>Writing code for an animation</i>	306
12.3	3D ANIMATION DIAGRAM	307
12.3.1	<i>3D animation editor</i>	307
12.3.2	<i>3D animation shapes</i>	312
12.3.3	<i>3D animation rendering principles</i>	322
12.3.4	<i>Managing a camera</i>	322
12.3.5	<i>Writing code for 3D animation</i>	325
12.4	RUNNING ANIMATION	326
12.5	RUNNING 3D ANIMATION	327
12.5.1	<i>Moving and rotating the animation</i>	328
12.6	CONFIGURING AN ANIMATION RUN	328
12.6.1	<i>Setting up animation update rate</i>	329
12.6.2	<i>Setting up animation anti-aliasing</i>	330
13.	SIMULATION SETTINGS	331
13.1	SIMULATION SPEED	331
13.2	MODEL REPLICATIONS	333
13.3	SIMULATION STOP CONDITIONS	333
13.3.1	<i>Defining a model time stop condition</i>	334
13.3.2	<i>Defining additional stop conditions</i>	334
13.4	CONTROLLING MODEL REPLICATIONS	337
13.4.1	<i>Writing code to be executed between model replications</i>	337
13.4.2	<i>API to control replications</i>	337
14.	DEBUGGING A MODEL	340

14.1	CHECKING MODEL SYNTAX.....	340
14.2	VIEWING AND MODIFYING ANYLOGIC EVENTS.....	342
14.2.1	<i>Event processing at the simulation engine</i>	<i>342</i>
14.2.2	<i>Events window.....</i>	<i>346</i>
14.3	BREAKPOINTS.....	348
14.4	LOGGING A MODEL.....	349
14.4.1	<i>Log window</i>	<i>349</i>
14.4.2	<i>Writing to logs.....</i>	<i>351</i>
14.5	RUNTIME ERRORS	351
14.5.1	<i>Java exceptions.....</i>	<i>352</i>
14.5.2	<i>Throwing runtime errors</i>	<i>352</i>
14.5.3	<i>Simulation errors.....</i>	<i>353</i>
14.6	DEBUGGING JAVA CODE	353
15.	CREATING A MODEL WITH DYNAMICALLY CHANGING STRUCTURE	
	354	
15.1	MANUAL CREATION AND DESTRUCTION OF ENCAPSULATED OBJECTS.....	354
15.1.1	<i>Writing code executed on object creation and destruction.....</i>	<i>355</i>
15.2	DYNAMICALLY CHANGING CONNECTIONS	357
16.	OPTIMIZATION	358
16.1	SETTING UP AN OPTIMIZATION.....	359
16.1.1	<i>Creating an optimization experiment.....</i>	<i>359</i>
16.1.2	<i>Defining the objective.....</i>	<i>361</i>
16.1.3	<i>Defining optimization parameters</i>	<i>362</i>
16.2	CONSTRAINTS	365
16.2.1	<i>Defining a constraint.....</i>	<i>365</i>
16.2.2	<i>Feasible and infeasible solutions.....</i>	<i>366</i>
16.3	OPTIMIZATION SETTINGS	367

16.4	RUNNING THE OPTIMIZATION.....	369
16.5	HOW TO INCREASE OPTIMIZATION PERFORMANCE	372
16.6	TIPS AND NOTES.....	374
17.	COLLECTING DATA AND PERFORMING STATISTICAL ANALYSIS	375
17.1	COLLECTING DATA IN DATASETS.....	375
17.1.1	<i>Defining a dataset</i>	377
17.1.2	<i>Dataset types</i>	378
17.1.3	<i>Intervals associated with datasets</i>	380
17.1.4	<i>Statistics block</i>	380
17.2	COLLECTING DATASETS FOR VARIABLES	381
17.3	VISUALIZING COLLECTED DATA.....	382
17.3.1	<i>Visualizing collected data in Model Viewer</i>	382
17.3.2	<i>Visualizing collected data in AnyLogic animation</i>	389
17.4	EXPORTING STATISTICAL DATA TO OTHER APPLICATIONS	390
18.	STANDALONE MODEL RUNNING.....	391
18.1	RUNNING A MODEL FROM THE COMMAND LINE.....	391
18.2	RUNNING A MODEL AS AN APPLLET	392
18.2.1	<i>Configuring your Web browser to view Java applets</i>	394
18.3	CONTROLLING THE MODEL SIMULATION	394
18.3.1	<i>Controlling the model simulation</i>	395
18.3.2	<i>Setting up model speed</i>	395
18.3.3	<i>Configuring animation</i>	396
19.	LIBRARIES AND EXTERNAL FILES	399
19.1	EXTERNAL FILES	399
19.2	LIBRARIES.....	400
19.2.1	<i>Creating a library</i>	400
19.2.2	<i>Working with libraries</i>	401

20. DATABASE SUPPORT	404
20.1 INTRODUCTION	404
20.2 CREATING A DATA SOURCE.....	405
20.2.1 <i>Associating with a database file</i>	406
20.2.2 <i>Associating with an ODBC data source</i>	407
20.2.3 <i>Viewing data source content</i>	411
20.3 WORKING WITH DATA SOURCES	412
20.4 CUSTOM QUERIES	413
20.5 TIPS AND NOTES.....	414
21. CUSTOMIZING ANYLOGIC UI	415
21.1 CUSTOMIZING TOOLBARS AND MENUS	415
21.1.1 <i>Toolbars page</i>	416
21.1.2 <i>Commands page</i>	417
21.1.3 <i>Keyboard page</i>	419
21.1.4 <i>Menu page</i>	420
21.1.5 <i>Options page</i>	421
21.2 CUSTOMIZING COLORS.....	422
22. PROJECT BUILDING. MODEL INITIALIZATION AND TERMINATION ORDER	424
22.1 BUILDING A PROJECT	424
22.1.1 <i>Project building stages</i>	424
22.1.2 <i>Building a project</i>	426
22.1.3 <i>Project building options</i>	426
22.2 MODEL INITIALIZATION AND TERMINATION ORDER.....	428
22.2.1 <i>Model initialization order</i>	428
22.2.2 <i>Model termination order</i>	429
23. THREADS	430

1. Creating AnyLogic model

1.1 AnyLogic modeling language

The modeling language of AnyLogic is an extension of UML-RT - a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.

The main building block of AnyLogic model is the *active object*. Active objects can be used to model very diverse objects of the real world: processing stations, resources, people, hardware, physical objects, controllers, etc.

An active object is an instance of an active object class. When you develop an AnyLogic model, you actually develop classes of active objects and define their relationships. You can also use ready to use active object classes from AnyLogic libraries.

Active object classes map to Java classes. Therefore, they allow inheritance, virtual methods, polymorphism, etc. Object-oriented modeling brings evident benefits. Modeling with classes provides for structural decomposition and active objects reuse. Once an active object class with the required structure is defined, you can create multiple active object instances in your model. Class hierarchies allow further expansion of these ideas.

Active objects inheritance

Being Java classes, active object classes may inherit one from another. The subclass inherits the interface of the superclass and may add specific structure elements and methods. Inheritance permits the reuse of code and supports easy model modification. Once you have defined the fundamental class, representing an automobile, its general characteristics can be inherited by subclasses, representing sport cars and trucks.

Actually all objects of the real world have complex structure. Decomposition is an essential principle in order to manage and master the complexity of large systems. Breaking up a whole system into parts, which may then be further decomposed, helps to overcome the limitations of the human cognition.

Hierarchical decomposition

AnyLogic models are hierarchically decomposed, since active objects may encapsulate other active objects to any desired depth. This enables you to decompose a model into as many levels of detail as required, since each active object typically represents a logical section of the model. Each AnyLogic model has a root active object which contains encapsulated objects which, in turn, contain their encapsulated objects, and so on. This way, the hierarchical tree of active objects is constructed. Encapsulation also enables you to hide all the complexities of a modeled object.

Structural decomposition

AnyLogic models are structurally decomposed since they have well-defined interaction interfaces. Active objects interact with their surroundings solely through boundary objects. Connections between objects are conveniently described by defining connectors which model physical coupling. This de-coupling of the internal object implementation from any direct knowledge about the environment makes active objects reusable.

Active objects reuse

Besides the use of inheritance, reuse of modeling knowledge is supported by use of libraries containing model classes. AnyLogic lets you create reusable libraries of active object classes developed for some particular application area or modeling task. Libraries provide for better reuse of classes across multiple models. A class can be developed and stored once and used in several projects.

1.2 Starting AnyLogic

You develop and run models using AnyLogic development environment – hereinafter AnyLogic. First, you start AnyLogic and create a new project or open an existing one. Second, you build up the model using constructs provided by AnyLogic. Third, you run the simulation.

This section provides the reference on how to start AnyLogic.

► To start AnyLogic

1. Invoke the *AnyLogic* shortcut from Windows *Start* menu (by default it is located on the following menu path: *Start | Programs | AnyLogic 5.0 | AnyLogic*).
2. Registration wizard appears. To build models with AnyLogic you should have an evaluation (limited time) or fully-functional permanent key.

► To evaluate AnyLogic


1. Obtain an evaluation key. If you have downloaded AnyLogic for evaluation, the key is sent to you by e-mail. Otherwise, open the web-page <http://www.xjtek.com/products/anylogic/evaluate/> and fill out the form. The key will be sent to you by e-mail.
2. With the registration wizard opened, enter the evaluation key received by e-mail by selecting the *Enter permanent or evaluation key* option of the wizard.
3. For subsequent runs of AnyLogic, you can select the *Continue with evaluation version* option.

► To obtain a fully functional AnyLogic key

1. With the registration wizard opened, select the *Send request for permanent key* option and follow the wizard instructions. You will be prompted to send the request information to the request processing center by e-mail.
2. When you receive your personal unlock key by e-mail, open the registration wizard again (if necessary, point to the *Help* menu of AnyLogic and select *Register product ...*), then select the *Enter permanent or evaluation key* option and enter your personal unlock key.

Once you complete the product registration wizard, AnyLogic is started (see Figure 1).

► To start AnyLogic and open a particular project

Click on the AnyLogic project file (extension `.alp`, icon ) in the Windows Explorer. If you already completed the project registration wizard, AnyLogic should be running. If not, complete the product registration wizard and click the icon again.

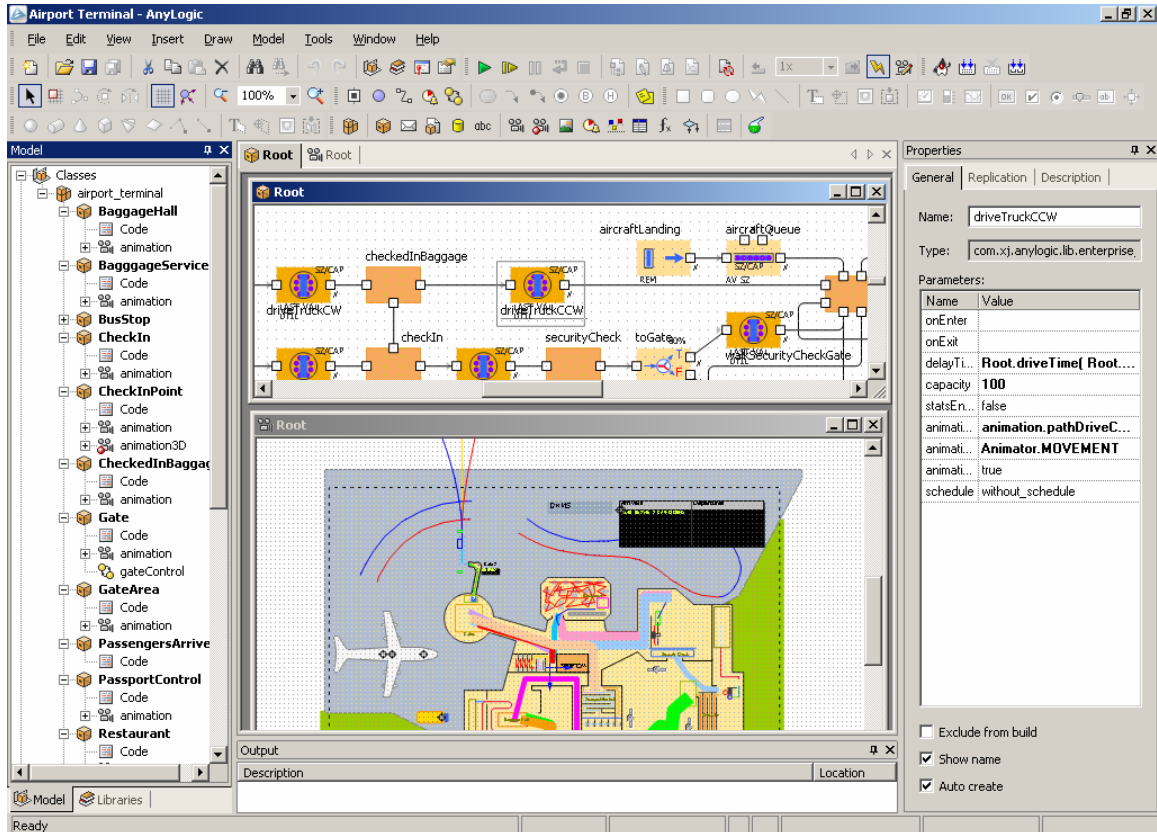


Figure 1. AnyLogic

Note that clicking on a model file does not result in starting another instance of AnyLogic. Instead, that file is opened in the same AnyLogic instance (you are prompted to save the already opened file if necessary).

1.2.1 Working with projects

When opened, AnyLogic displays the *Start Page* (see Figure 2). The *Start Page* prompts you to create a new project, open an existing one, or open one of the state-of-the-art AnyLogic examples.

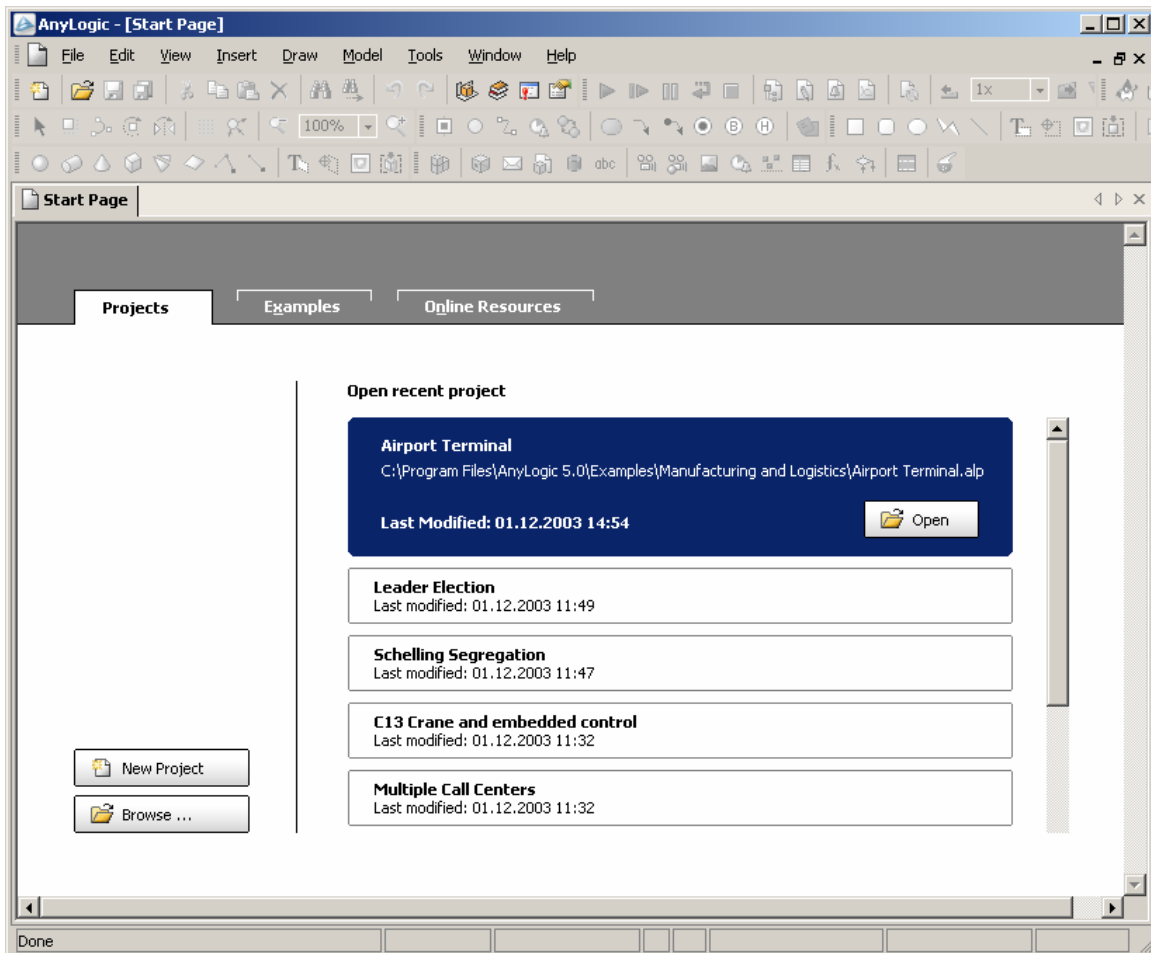



Figure 2. AnyLogic Start Page


A project is a workspace for developing your model. To manage AnyLogic projects, use both the *Start Page* and AnyLogic *File* menu.

► To create a new project

1. On the *Projects* page of the *Start Page*, click the *New project* button, or
 Click the *New*  toolbar button, or
 Choose *File|New...* from the main menu, or
 Press **Ctrl+N**.
 The *New project* dialog box is displayed.
2. Specify the the name and location of the new project.

3. Specify whether you want to create a folder for the project.
4. Click *OK*.

► **To open an existing project**

1. On the *Projects* page of the *Start Page*, click *Browse* button, or
Click the *Open*  toolbar button, or
Choose *File | Open...* from the main menu, or
Press *Ctrl+O*.
The *Open* dialog box is displayed.
2. Browse for the project file you want to open,
Double-click this file, or
Click it and click *Open* button.

AnyLogic provides easy access to recently opened projects.

► **To open a recently opened project**

1. Choose the project you want to open from the project list in the bottom of the
AnyLogic *File* menu, or
Click the *Projects* tab of the *Start Page*, click the project you want to open in the list of
recently opened projects and click *Open* button.

If you are working with one particular project, you can tell AnyLogic not to display the *Start Page* on AnyLogic startup, but instead to open the last project you were working with.

► **To open last used project/the Start Page on AnyLogic startup**

1. Choose *Tools | Options...* from the main menu.
The *Options* dialog box is displayed.
2. On the *Miscellaneous* page, select/clear the *Reload last project on startup* check box.
3. Click *OK*.

AnyLogic standard distribution includes several state-of-the-art examples, organized by categories (education, logistics, mechanics, traffic, etc.). Have a look at these examples to get an idea of how to develop your own models.

► To open an AnyLogic example


1. Click the *Examples* tab of the *Start Page*.
The list of AnyLogic examples appears in the right panel of the *Start Page*.
2. If needed, choose *By category* option to sort examples by AnyLogic categories.
The list of example categories appears in the right panel. To show all examples in the category, click the plus icon to the left of the category item.
If needed, choose *Sorted alphabetically* option to sort examples alphabetically.
3. Click the example item.
The example description is displayed.
4. Click *Open* button.

From the *Start Page*, you can examine AnyLogic online resources and documentation.

► To view online resources

1. Click the *Online Resources* tab of the *Start Page*.
The list of AnyLogic online resources and documentation appears in the right panel of the *Start Page*.
2. Click the online resource you want to examine.
3. Click the *Open* button.


► To save the current project

1. Click the *Save*  toolbar button, or
Choose *File | Save* from the main menu, or
Press Ctrl+S.

► **To save the current project with a new name**

1. Choose *File* | *Save As...* from the main menu.
The *Save As* dialog box is displayed.
2. Specify the new name and location of the project.
3. Click the *Save* button.

► **To save all projects**

1. Click the *Save All*  toolbar button, or
Choose *File* | *Save All* from the main menu.

► **To close the current project**

1. Choose *File* | *Close* from the main menu.

AnyLogic can open one project at a time. However, it is possible to run several instances of AnyLogic and open different projects in different instances.

► **To run two or more AnyLogic instances simultaneously**

1. While running AnyLogic, invoke *AnyLogic* shortcut from Windows *Start* menu.
2. In a newly opened instance of AnyLogic, open the desired project.

This is the way to work on several projects simultaneously and to copy classes from one project to another.

1.3 Editing the project

AnyLogic development environment is built up to the state-of-the-art Windows UI level. It features:

- Customizable windows, toolbars, colors, images
- Drag and drop editing

- Diagram zooming
- Easy navigation throughout the project using the classes tree
- On-the-fly checking of types, parameters, and diagram syntax
- Graphical errors highlighting

You edit the project using several AnyLogic editor windows. When a new project is created, the Project window and the Properties window are displayed (see Figure 3). The Project window is used to create, view and manipulate model elements. The Properties window is used to view and modify the properties of model elements. This section gives the description of these windows.

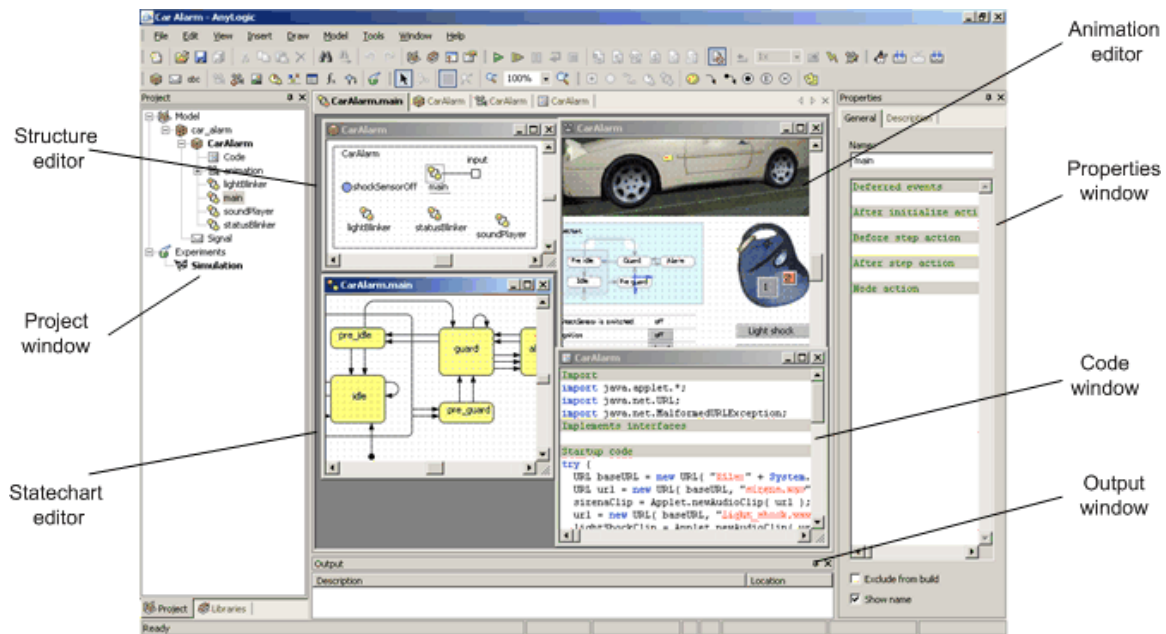


Figure 3. Windows used to edit the project

1.3.1 Project window

The Project window (a page in the Workspace window - see Figure 4) provides access to various project elements such as packages, classes, etc. As the project is organized hierarchically, it is displayed as a tree: the project itself forms the top level, packages are the

next level items, active object and message classes – one level down, etc. The workspace tree provides easy navigation throughout the project.

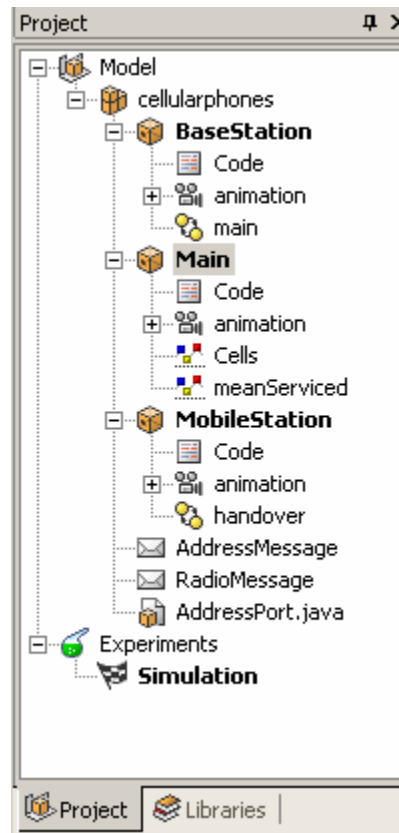


Figure 4. Project window

► To show the Project window


1. Click the *Project*  toolbar button, or
Choose *View | Project* from the main menu, or
Press Alt+0.

There are some common operations you can perform with items in the Project window. You can copy, move and delete items. Thus, you can easily manage your project. When copying or deleting, you should select the item.


► **To select an item**

1. Click on the item.
The Properties window will display the properties of the selection.


► **To delete the item**

1. Click the *Delete*  toolbar button, or
Choose *Edit|Delete* from the main menu, or
Right-click the item and choose *Delete* from the popup menu, or
Press Del.
2. Confirm the deletion by clicking *Yes* button.


► **To copy the item**

1. Click the *Copy*  toolbar button, or
Choose *Edit|Copy* from the main menu, or
Right-click the item and choose *Copy* from the popup menu, or
Press Ctrl+Ins.

► **To cut the item**

1. Click the *Cut*  toolbar button, or
Choose *Edit|Cut* from the main menu, or
Right-click the item and choose *Cut* from the popup menu, or
Press Shift+Del.
2. Confirm cutting by clicking *Yes* button.

► **To paste the item**

1. Select the parent item you want to paste into.
For example, you can paste an active object class into a package.
2. Click the *Paste*  toolbar button, or
Choose *Edit|Paste* from the main menu, or
Right-click the parent item and choose *Paste* from the popup menu, or
Press Shift+Ins.

If needed, you can exclude the project element from the model. Thus, you can adjust your model structure by excluding one element and including other ones at the design time.

► **To exclude/include an item from/to a model**

1. Right-click the item and choose *Exclude from Build* from the popup menu.
The item image becomes diffuse/sharp.

1.3.2 Properties window

The Properties window is used to view and modify the properties of a currently selected object. When you select something – e.g., in the Project window or in a diagram editor window (see section 1.5.2, “Diagram editors. Generic operations”) – the Properties window (see Figure 5) displays the properties of the selection.

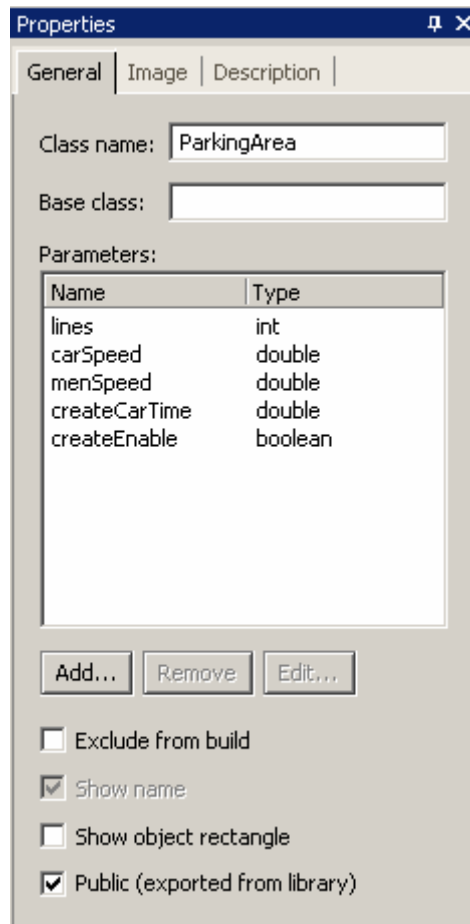



Figure 5. Properties window

The Properties window consists of several pages. Every page contains controls such as edit boxes, check boxes, buttons etc., used to view and modify properties. The number of pages and their appearance depend on the type of a selected object.

You can directly drag an item from the model tree to a field in the Properties window.

► To show/hide the Properties window

1. Click the *Properties*  toolbar button, or
Choose *View | Properties* from the main menu, or
Press Alt+Enter.

► **To display a particular page of the Properties window**

1. Click the corresponding tab in the top of the Properties window.

Each model element can have a descriptive text associated with it to make the model easier to understand.

► **To set an element description**

1. Select the model element.
2. Enter your text on the *Description* page of the Properties window.

AnyLogic displays tooltips with the detailed description of properties of a currently selected element.

► **To get an information about a property**

1. Point the mouse cursor to the required control on the Properties window and wait for tooltip to appear.

If needed, you can tell AnyLogic not to show tooltips for properties.

► **To show/hide tooltips for properties**

1. Choose *Tools | Options...* from the main menu.
The *Options* dialog box is displayed.
2. On the *Miscellaneous* page, select/clear the *Enable property tips* check box.
3. Click *OK*.

1.3.3 Arranging windows

When editing your project you actually work with several windows. Names of open windows are listed at the foot of the AnyLogic *Window* menu. Use the *Window* menu options to arrange open windows.

► **To display model windows in a cascading style**

1. Choose *Window* | *Cascade* from the main menu.

► **To display model windows vertically across the width of the AnyLogic window**

1. Choose *Window* | *Tile Vertically* from the main menu.

► **To display model windows horizontally down the length of the AnyLogic window**

1. Choose *Window* | *Tile Horizontally* from the main menu.

► **To close all the windows**

1. Choose *Window* | *Close All* from the main menu.

► **To make a window active**

1. Choose the window name from the *Window* menu.
2. If too many windows are opened, the windows list displays only some of them. Choose *Window* | *More Windows...* from the main menu, select the window you want to make active in the *Select window* dialog and click *OK*.

► **To make the next window from the list active**

1. Choose *Window* | *Next* from the main menu, or Press **Ctrl+F6**.

► **To make the previous window from the list active**

1. Choose *Window* | *Previous* from the main menu, or Press **Ctrl+Shift+F6**.

1.4 Model elements

AnyLogic models are hierarchically organized. An AnyLogic *project* consists of *packages* used for better structuring of a project. A project may use other projects as *libraries* — collections of classes developed for some particular application area. To facilitate your work, AnyLogic enables you to specify several *experiments* with different model execution parameters depending on experiments you need to carry out with your model.

This section describes these AnyLogic model elements.

1.4.1 Project

A working unit of AnyLogic is called a project. A project completely defines a model or a library. A project is a root item in the Project window.

The following project properties are specified on the *General* page of the Properties window.

General properties

Name – name of the project, usually the same as the model file name.

Loaded from – [read only] the project (.alp) file location.

Target file – [optional] name of the file to which the generated code is packed. You have to specify this property if you wish to use a project as a library, see Chapter 19, “Libraries”.

Additional library files – [optional] semicolon-separated list of Java archives you wish to add to the project at the compile time.

Folder for generated files – [optional] path to the folder where AnyLogic should store generated files. If not specified, AnyLogic puts generated files into Windows temporary directory.

1.4.2 Package

A project consists of packages. There may be one or more packages in the project. Packages contain active objects, messages, other classes, and external files. Packages may be used for better structuring of a project.

► To add a new package to the project

1. Choose *Insert | New Package...* from the main menu, or
In the Project window, right-click the project and choose *New Package...* from the popup menu.
The *New Package* dialog box is displayed.
2. Specify the name of the new package and click *OK*.

Properties

Name – name of the package.

Exclude from build – if set, the package is excluded from the model.

When AnyLogic generates code, it maps every AnyLogic package to a Java package with the same name. As a result, classes in different AnyLogic packages are put into different Java packages. The rules of using AnyLogic packages are the same as rules of using Java packages. To use a class from another package, you have to import that package or prefix the name of the class with the name of the package. Package import is described in section 1.5.9.1, “Importing packages”.

If you do not want to deal with namespaces, you may use just one package in your project — e.g., `mypackage`. By default, when you create a new project, AnyLogic creates one package with the same name as the project name.

1.4.3 Library

A project may use other projects as libraries. Libraries are collections of classes developed for some particular application area or modeling task. Several libraries come with the tool, and you can easily create your own.

Libraries have several benefits:

- Provide for better reuse of classes across multiple models. A class can be developed and stored once and referenced from several projects.
- Libraries enable you to organize teamwork in AnyLogic projects: a part of the model developed by a team member may be put into a library, and others use consistent versions of the library in their work.
- By developing the right library you can convert AnyLogic into a high-level modeling tool with point-and-click interface for a specific domain.

AnyLogic shows available libraries in the Libraries window. AnyLogic standard distribution includes several libraries located in the `Lib` directory. Have a look at these libraries to get an idea of how to develop your own. The detailed information on creating libraries and using AnyLogic libraries classes is given in Chapter 19, “Libraries and external files”.

1.4.4 Experiment

An AnyLogic *experiment* defines the type of experiment you want to carry out with your model. AnyLogic supports the following types of experiments:

- Simulation experiment

Simulation experiment is used in most cases. It runs model simulation with animation displayed and model debugging enabled. Running a model simulation is described in Chapter 11, “Running and observing a model”. Other AnyLogic experiments are used only when the model parameters play a significant role and you need to analyze how they affect the model behavior, or when you want to find optimal parameters of your model.

- Optimization experiment

If you need to run a simulation and observe system behavior under certain conditions, as well as improve system performance - for example, by making decisions about system parameters and/or structure - you can use the optimization capability of AnyLogic. Optimization is the process of finding the optimal combination of conditions that results in the best possible solution. Optimization is described in details in Chapter 16, “Optimization”.

- Custom experiment


Allows defining a custom experiment script using Java language.

Experiment has a set of parameters you can adjust to configure model simulation (simulation time mode, number of model runs, simulation stop conditions, etc.). Setting up simulation parameters is discussed in Chapter 13, “Simulation settings”.

When a new project is created, a simulation experiment is created and set as the *current experiment*. The current experiment defines simulation parameters actual for the current simulation.

You can simply customize model simulation by creating several experiments with different simulation parameters set up and simulating your model with different current experiments. For instance, you may need to execute your model with different values of the root object parameters. Thus you create several simulation experiments with different parameter values specified and simulate your model with different experiments set as current.

► To create a new experiment

1. Click the *New Experiment*  toolbar button, or Choose *Insert | New Experiment...* from the main menu, or In the Project window, right-click the *Experiments* item and choose *New Experiment...* from the popup menu.
The *Create a new experiment* dialog box is displayed.
2. Choose the desired type of the experiment in the group box.
3. Type the experiment name in the *Name* edit box.
4. Select the root object of the experiment from the *Root object* drop-down list.
5. Click *OK*.

In the Project window, experiments are stored in the *Experiments* subtree of the workspace tree, located under the project item.

► To set an experiment to be a current experiment


1. Right-click the experiment in the Project window and choose *Set as Current* from the popup menu.
The current experiment name is emphasized in the Project window with bold.

1.5 Active object

Active objects are the main building blocks of AnyLogic model. Active objects can be used to model very diverse objects of the real world: processing stations, resources, people, hardware, physical objects, controllers, etc.

An active object is an instance of an active object class. Active objects classes are developed by the user, or they can be taken from libraries.

► To add a new active object class to a package

1. Click the *New Active Object Class*  toolbar button, or Choose *Insert | New Active Object Class...* from the main menu. The *New Active Object Class* dialog box is displayed. Specify the name of the new active object class, choose the package, which will contain the active object class, and click *OK*.
2. Alternatively, in the Project window, right-click the package, which will contain the active object class, and choose *New Active Object Class...* from the popup menu. The *New Active Object Class* dialog box is displayed. Specify the name of the new active object class and click *OK*.

Each active object class has the following properties:

Properties

Class name – name of the class.

Base class – [optional] name of the base class. This can be `ActiveObject` or its subclass. If not specified, `ActiveObject` is assumed. See section 1.5.10, “Active object inheritance” for information on active object inheritance.

Parameters – [optional] a set of formal parameters of the active object class (see Chapter 3, “Parameters”).

Exclude from build – if set, the class is excluded from the project.

Show name – if set, the name of the class is shown on its structure diagram.

Show object rectangle – if set, the rectangle representing the border of this object is shown on its structure diagram.

Public (exported from library) – if reset, in case this project is used as a library, this class is not accessible from other projects (you may need this to hide some auxiliary library classes).

1.5.1 Structure diagram

Each active object class has a structure diagram associated with it. The structure diagram plays several roles. It:

- Defines the interface of the active object class
- Defines the encapsulated objects and their interconnection
- Defines behavior elements, such as timers and statecharts.

The structure diagram is constructed of various shapes, namely: this object, encapsulated object, port, variable, connector, chart timer, statechart, and text box (see Figure 6).

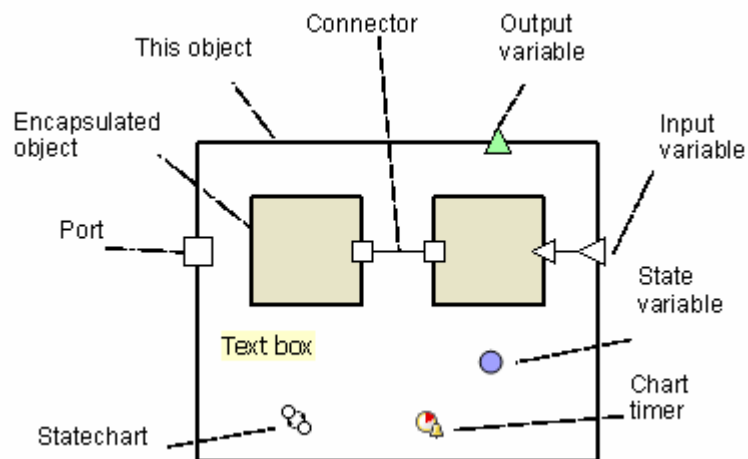


Figure 6. Structure diagram

The structure diagram of an active object is edited in the structure editor using the structure toolbar (see Figure 7).

Structure editor

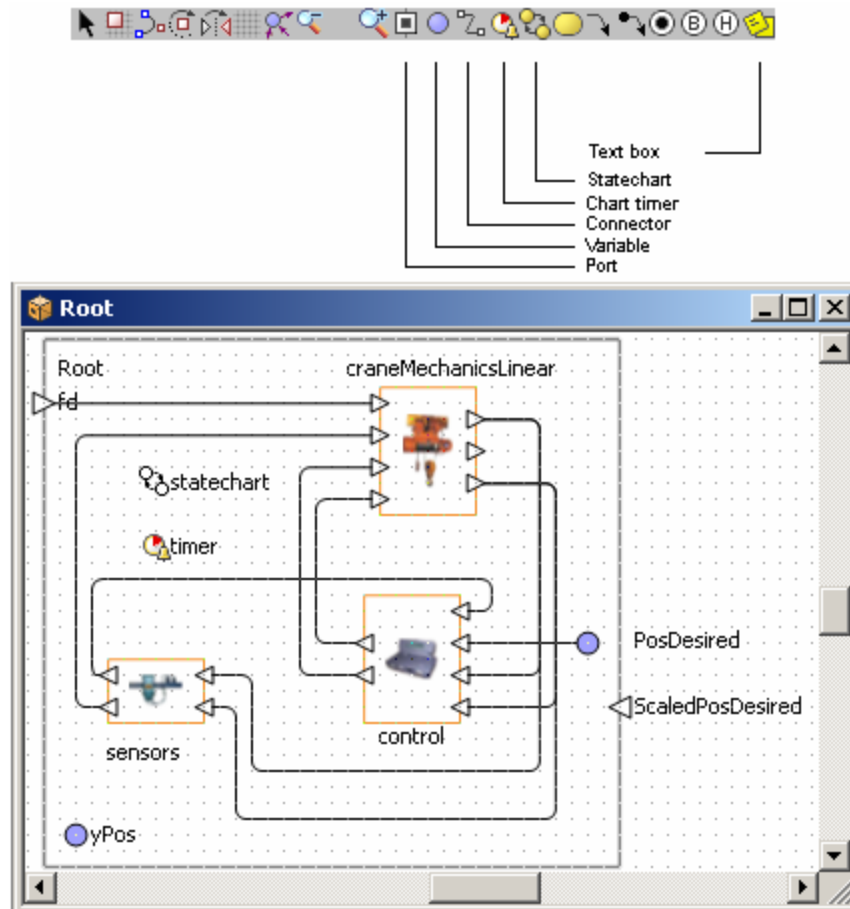


Figure 7. Structure editor and toolbar


► To open the structure diagram of an active object class

1. Right-click the active object class in the Project window and choose *Open Structure* from the popup menu, or
Double-click the active object class in the Project window.

This active object is displayed as a bold frame. It denotes the “border” of the active object on its structure diagram. The semantics of this shape is that all ports and variables placed on it become interface elements of the active object class. This shape is optional on the diagram. The properties of this shape are actually the properties of the active object class.

You can put a comment on a diagram using a text box. This does not affect the model behavior.

► **To add a text box**

1. Click the *Text Box*  toolbar button, or Choose *Draw | Text Box* from the main menu.
2. Click the place on the diagram where you want to put the text box. Then drag to choose the size of the shape.

► **To modify the content of a text box**

1. Double-click the text box.
2. Edit the content of the text box.
3. Click the empty area of the diagram or press Esc to store the modified text.

You can also modify the text of the text box using its Properties window.

1.5.2 Diagram editors. Generic operations

AnyLogic includes four diagram editors: structure editor, statechart editor, animation editor and 3D animation editor. They all are based on the same technology and therefore share a large set of generic editing operations described in this section. The specific operations are described in section 1.5.1, “Structure diagram”, section 9.2, “Statechart diagram”, section 12.2, “Animation diagram” and section 12.3, “3D animation diagram” correspondingly.

AnyLogic diagrams are constructed of graphical objects – shapes. To draw a shape, you click the corresponding toolbar button and place the shape on the diagram. Each editor has an associated toolbar with shapes specific for its diagram type.

Selecting shapes

You can select any shape on a diagram. When you select a shape, the Properties window displays its properties.

► **To select a shape**

1. Click the shape.

► **To select more than one shape**

1. Drag the selection rectangle around the shapes.

► **To add/remove a shape to/from the selection**

1. Shift-click the shape.


► **To select all shapes on the diagram**

1. Choose *Edit|Select All* from the main menu, or
Press Ctrl+A.


Copying, moving and deleting shapes

You can copy, move and delete shapes.


► **To copy the selection on the Clipboard**

1. Click the *Copy*  toolbar button, or
Choose *Edit|Copy* from the main menu, or
Right-click the selection and choose *Copy* from the popup menu, or
Press Ctrl+Ins.

► **To cut the selection**

1. Click the *Cut*  toolbar button, or
Choose *Edit|Cut* from the main menu, or
Right-click the selection and choose *Cut* from the popup menu, or
Press Shift+Del.

► **To paste the content of the Clipboard**

1. Click the *Paste*  toolbar button, or
Choose *Edit|Paste* from the main menu, or
Right-click the empty area of the diagram and choose *Paste* from the popup menu,
or
Press Shift+Ins.
The pasted shapes appear in blue outline.
2. Move the pasted shapes to the desired position.


► **To copy the selection**

1. Ctrl-drag the selection.

► **To move the selection**

1. Drag the selection, or
Use arrow keys.

► **To delete the selection**

1. Click the *Delete*  toolbar button, or
Choose *Edit|Delete* from the main menu, or
Right-click the selection and choose *Delete* from the popup menu, or
Press Del.

► **To hide the selection**

1. Choose *Draw|Hide* from the main menu.

► **To unhide all hidden shapes**


1. Choose *Draw|Unhide All* from the main menu.

► **To get the image of the whole diagram on the Clipboard**


1. Choose *Draw | Copy Image* from the main menu, or Right-click the empty area of the diagram and choose *Copy Image* from the popup menu.

You can undo previously performed actions.

► **To undo the previous action**

1. Click the *Undo*  toolbar button, or Choose *Edit | Undo* from the main menu, or Press Alt+Backspace, or Ctrl+Z.

► **To redo the previously undone action**

1. Click the *Redo*  toolbar button, or Choose *Edit | Redo* from the main menu, or Press Ctrl+Y.

You can move, center, or zoom in on the diagram to take a good look at some particular parts of it.

► **To move the diagram**

1. Click on the diagram with the right mouse button and, while holding the right mouse button down, move the mouse.

► **To center the diagram**

1. Choose *Draw | Go to Center* from the main menu, or Right-click the the empty area of the diagram and choose *Go to Center* from the popup menu.
The diagram will be centered.

► **To zoom in on the diagram to fit all shapes**

1. Choose *Draw | Zoom | Zoom to Fit* from the main menu, or Right-click the the empty area of the diagram and choose *Zoom to Fit* from the popup menu.
The diagram will be centered and zoomed to fit all the shapes.

► **To zoom in**

1. Click the *Zoom In*  toolbar button, or Choose *Draw | Zoom | Zoom In* from the main menu.

► **To zoom out**

1. Click the *Zoom Out*  toolbar button, or Choose *Draw | Zoom | Zoom Out* from the main menu.

► **To zoom to the specified rectangle**


1. Choose *Draw | Zoom | Zoom to Rectangle* from the main menu.
2. Drag the selection rectangle around the area you want to zoom to.

► **To zoom to the default scale**

1. Choose *Draw | Zoom | Zoom to Default* from the main menu.

You can control the diagram grid appearance.

► **To enable/disable the grid**

1. Click the *Enable Grid*  toolbar button, or Choose *Draw | Grid | Enable Grid* from the main menu.

► **To show/hide the grid**

1. Choose *Draw | Grid | Show Grid* from the main menu.

► To snap a shape to the grid

1. Select the shape.
2. Click the *Snap to Grid*  toolbar button, or Choose *Draw | Grid | Snap to Grid* from the main menu.

By default, when you resize, drag, or move a shape, other shapes logically related to it also move. For example, connectors move with ports, encapsulated objects move with this object, simple states reflect changes of the composite state, etc. Sometimes this is undesirable, so you always can switch smart dragging off.

► To switch smart dragging off during the operation


1. Hold Shift while finishing the operation.

► To edit the name of a shape

1. Double-click the name of the shape, or Right-click the shape and choose *Edit Name* from the popup menu, or Press F2.
2. Type the name of the shape.
3. Press Enter or click the empty area of the diagram to store the modified name, or Press Esc to cancel editing.

The name of a shape can also be edited in the Properties window.

► To rotate a shape

1. Click the *Rotate*  toolbar button, or Choose *Draw | Rotate* from the main menu.
2. Use rotation handles to rotate a structure element.

► To get compact information on shape properties

1. Point mouse cursor at the shape and wait for tooltip to appear.

1.5.3 Active object icon

Each active object class may have specific icon associated with it. Each time an instance appears as an encapsulated object on a structure diagram, or on animated structure diagram (see section 11.2.2, “Animated structure diagram”), this icon is displayed. Note that this image has nothing to do with AnyLogic animation and is not displayed on the structure diagram of this active object class itself.

1.5.3.1 Icon diagram

The active object icon is defined on the icon diagram. An icon diagram is edited in the icon editor (Figure 8) using the animation toolbar.

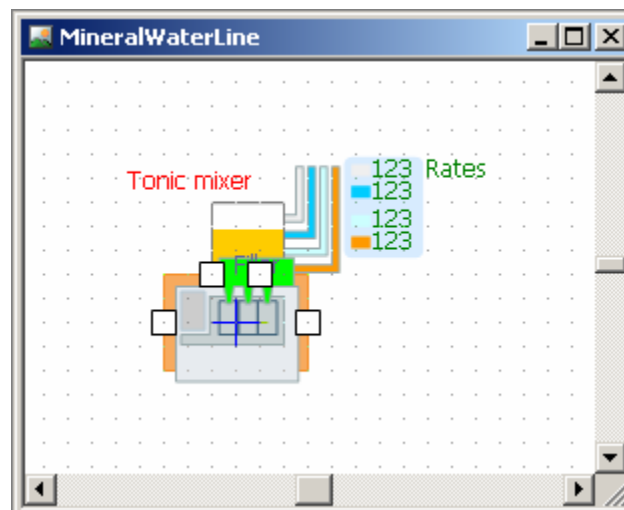



Figure 8. Icon editor

► To create an icon for an active object class

1. Click the *New Icon*  toolbar button, or Choose *Insert | New Icon...* from the main menu. The *New Icon* dialog box is displayed. Specify the name of the new icon, choose the active object class, which will contain the icon, and click *OK*.

2. Alternatively, in the Project window, right-click the active object class, which will contain the icon, and choose *New Icon...* from the popup menu.
The *New Icon* dialog box is displayed.
Specify the name of the new icon and click *OK*.
3. The icon editor window is displayed.

Each active object class may have only one associated icon.

► **To open the existing icon of an active object class**

1. Right-click the *Icon* item of the active object class in the Project window and choose *Open Icon* from the popup menu.
The icon editor window is displayed.

Icon editor shares a set of generic editing operations described in section 1.5.2, “Diagram editors. Generic operations”.

The blue cross is the origin point (0, 0) of the icon diagram.

An icon is a drawing composed of various shapes: circles, rectangles, lines, etc., and also indicators. You can construct your icon from any animation shapes and animation indicators. See section 12.2.2, “Animation shapes” and section 12.2.3, “Indicators” for the detailed description of these shapes. The icon size is set automatically to fit all the shapes.

Each shape has a number of properties defining its visual appearance: position, height, width, color, and so on. The generic properties of the shapes are described in section 12.2.1.2, “Generic properties of animation shapes”.

1.5.3.2 Icon animation concepts

Each shape has a number of properties defining its visual appearance: position, height, width, color, and so on. These properties are typically organized as shown in Figure 9:

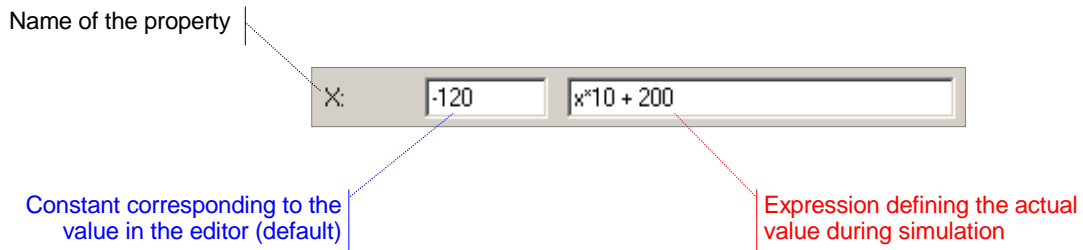


Figure 9. Property of an icon shape

The concepts are similar to the AnyLogic animation concepts – icon diagram links shape properties to active object data.

The static value on the left shows the value of the property as defined while drawing in the editor. It is also treated as a default value. The expression on the right defines the actual value during simulation. This is the place where you can link the appearance of a shape to any data of the active object. The data may change and it will be reflected in the picture. In case the expression is empty, the property retains the default static value throughout the whole simulation.

An example of associating graphical properties of icon shapes with active object data is shown in Figure 10. Here the coordinates of the circle are dynamically defined by the variables x and y of the active object, and the rotation angle of the rectangle is defined by the object member variable α .

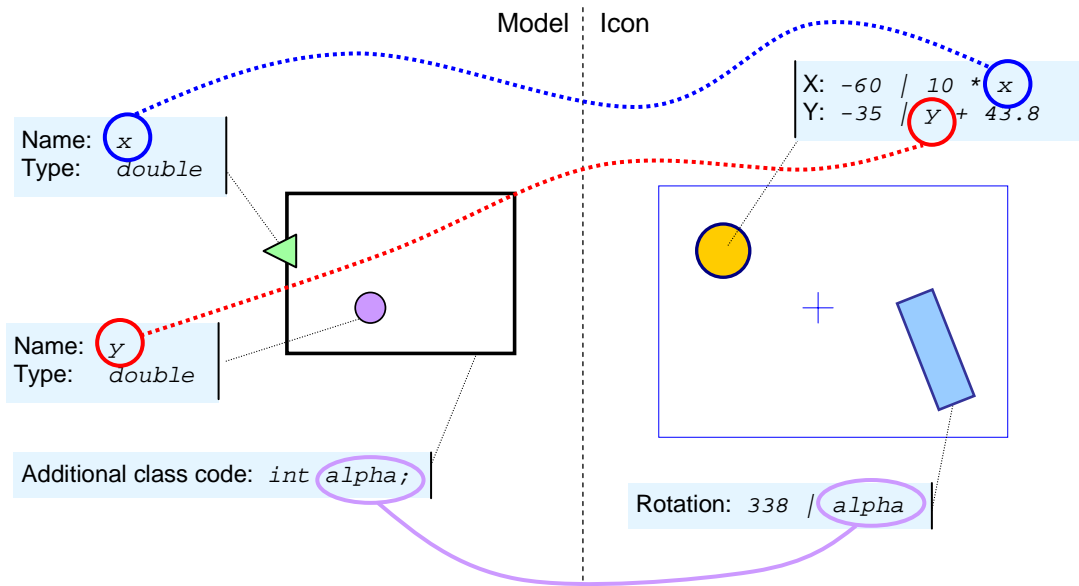


Figure 10. Associating graphical properties with model data

Thus, by linking shape properties to active object data, you can animate active objects on the animated structure diagram at the model runtime.

1.5.3.3 Active object image

Active object icon is the new feature of AnyLogic V. In the previous versions of AnyLogic, you could only associate a static image loaded from file, with an active object class.

In fact, active object image feature is left for compatibility with the previous versions only. It is recommended that you create an icon for an active object class and add an image shape to this icon instead of using active object image. Note that the image of an active object class is suppressed with the icon of this active object, if the latter is defined.

Active object image is defined on the *Image* page of the active object's Properties window (see Figure 11).

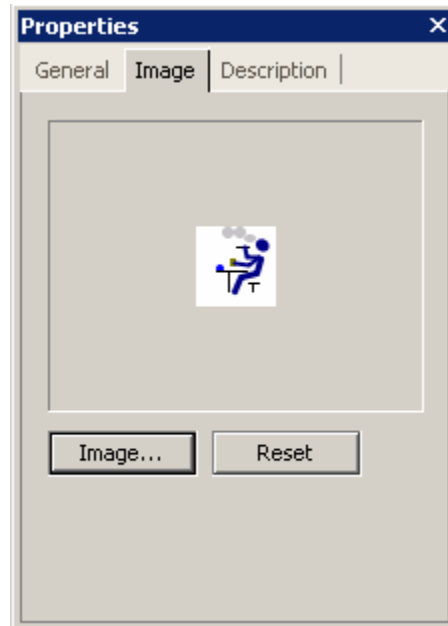


Figure 11. Image page of active object's Properties window

► **To define an image for an active object class**

1. Click the active object class in the Project window.
2. Click the *Image* button on the *Image* page of the Properties window.
The *Choose Image* dialog box is displayed. The dialog displays the predefined images.
3. To use one of predefined images,
Double-click the image, or
Click the image and click *OK*.
4. Otherwise, to load the image from file,
Click the *Browse* button.
The *Open* dialog box is displayed.
Browse for the image file you want to use.
Double-click the file, or
Click the file and click *Open* button to select the file.

The specified image is shown on the *Image* page of the Properties window.

► **To remove the specified active object image**

1. Click the active object class in the Project window.
2. Click the *Reset* button on the *Image* page of the Properties window.

1.5.4 Encapsulated objects

Active objects may encapsulate other active objects to any desired depth. Encapsulated objects are instances of other active object classes, encapsulated by each instance of this active object class. Encapsulating a class is the step to create the model hierarchy.

► **To add an encapsulated object to an active object class**

1. Drag the active object class from the Project window onto the structure diagram of the parent active object class.

All encapsulated objects should be placed inside this object shape if the latter is present.

Encapsulated objects are displayed as filled rectangles (see Figure 12). In case AnyLogic cannot find the referred class, the encapsulated object is displayed red.

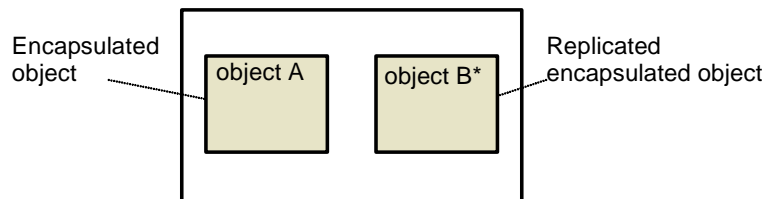


Figure 12. Encapsulated objects

Encapsulated objects have the following properties.

Properties

Name – name of the encapsulated object.

Type – [read only] class of the encapsulated object.

Parameters – [optional] set of actual parameters of the encapsulated object. Every parameter should be given in form: *Name Value*, where *Name* is the name of the parameter, *Value* is the value of the parameter.

Exclude from build – if set, the encapsulated object is excluded from the model.

Show name – if set, the name of the encapsulated object is shown on the structure diagram.

Auto create – if set, AnyLogic creates the encapsulated object automatically. Otherwise the object should be created manually.


You can open the structure diagram of an encapsulated object class.

► To open the structure diagram of an encapsulated object

1. Double-click the encapsulated object, or
Right-click the encapsulated object and choose *Open Structure* from the popup menu.

If needed, you can flip an encapsulated object image on the structure diagram.

► To flip an encapsulated object horizontally

1. Select the encapsulated object.
2. Click the *Flip Horizontal*  toolbar button, or
Right-click the encapsulated object on the structure diagram and choose *Flip Horizontal* from the popup menu, or
Choose *Draw | Flip Horizontal* from the main menu.

Encapsulated objects can be single or replicated (do not confuse this with model replications). A replicated object represents a collection of active objects of the same class. Object replication provides for very economical representation of complex structures of objects with arbitrary interconnections. Replicated object name displayed on the structure diagram of the parent class is followed with an asterisk (see Figure 12). See Chapter 2, “Replicated objects” for the detailed description of replicated objects.

1.5.5 Root object

An AnyLogic model is a tree of active objects encapsulating each other (see Figure 13). Thus a model is decomposed into several levels of detail, since each active object typically represents a logical section of the model. The root of that tree is called the root object. The root object represents the highest abstraction level of your model. When you specify the class of the root object, you tell AnyLogic where to start the model creation.

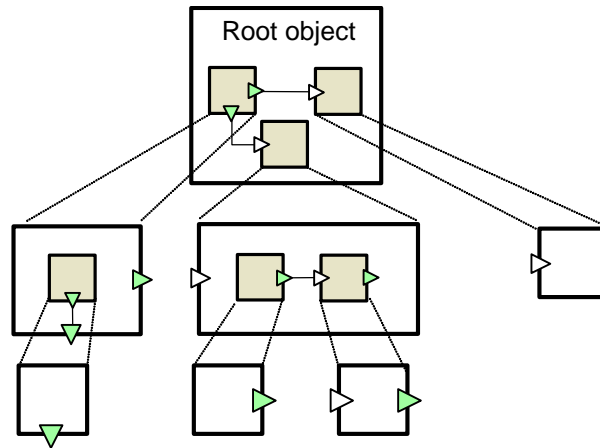


Figure 13. Tree of active objects

AnyLogic supports easy model modification since you can change the root object of a model. You can create several experiments with different root objects in the same project. Thus you can simply adjust your model structure by changing current experiments.

► To set a root object for an experiment

1. Click the experiment in the Project window.
2. In the Properties window, select the root object of the experiment from the *Root object* drop-down list.

For example, you model an automobile. You define the `Automobile` class and set this class to be the root object class since it represents the highest abstraction level of your model. This class encapsulates objects representing parts of the automobile: wheels, engines, carburetors, etc. They, in turn, may encapsulate objects representing their parts, and so on. However, you may need to focus on the automobile engine. Therefore you can simply set

Engine class to be the root object class of your model. Or vice versa, you may want to model a garage. In this case you need to create the new abstraction level in your model and make it the top of the model hierarchical tree. Unlike some other simulation tools with well-defined model root objects, in AnyLogic you can change the model structure in a very simple manner. For example, for the case described above, do the following:

1. Define the Garage class, representing the garage.
2. Encapsulate developed classes representing automobiles into the class Garage.
3. Create a new simulation experiment and make it current.
4. Make the class Garage the root object class of the created experiment.

1.5.6 Active object data

You can define active object data by specifying parameters and variables. You can also specify class member variables by writing your own Java code.

1.5.6.1 Parameters

Active objects may have parameters. Commonly, parameters are used to parameterize the object. It is needed when object instances have the same behavior described in class, but differ in parameter values. All parameters are visible and changeable throughout the model execution. Thus you can simply adjust your model by changing parameters at runtime. Active object parameters can be linked to parameters of encapsulated objects. In this case, parameter changes are propagated down the active object tree along the parameter dependencies. See Chapter 3, "Parameters" for the detailed description of parameters.

Use a variable instead of a parameter if you need to model some data unit continuously changing over time.

1.5.6.2 Variables

If you need to define some data unit (maybe continuously changing over time), you can define a variable. A variable can be of an arbitrary scalar type, a matrix, or a hyper-array.

Variables may be either internal (state variables) or public (interface variables). The latter ones can be shared with other active objects. Variables can appear in differential and algebraic equations and model values changing continuously over time. During the model execution, variables are observable and changeable from AnyLogic UI. See Chapter 4, “Variables” for the detailed description of variables.

1.5.6.3 Class member variables

You can declare a Java member variable in the *Additional class code* section of the Code window of an active object class. You can access these data members within this object. See section 1.5.9, “Writing code for an active object” for details.

Declare a member variable if you just need a data item to be accessed only within an active object and only at discrete steps (i.e., it is neither shared with other objects nor changed continuously), and you do not want to observe or change that item during model execution. Otherwise, declare AnyLogic variable.

1.5.7 Active object behavior

Active objects may have internal behavior. AnyLogic enables you to define discrete time, continuous and hybrid behavior.

- The continuous processes are described with differential and algebraic equations over continuously changing variables.
- The discrete activities within the object can be defined using timers in very simple cases or statecharts (extended state machines) in cases when event and time ordering becomes more complicated.
- When discrete and continuous time behaviors are interdependent, this needs hybrid modeling. You can define hybrid behavior with hybrid statecharts.

1.5.7.1 Equations

Continuous time behavior can be defined by equations. You can define a set of differential equations, algebraic equations, and formulas to describe continuous changes of variables over time. See Chapter 5, "Equations" for the detailed description of equations.

1.5.7.2 Timers

The activities within the object can be defined using timers. Timers are used to schedule some user-defined actions. There are static and dynamic timers in AnyLogic. The latter are used to schedule multiple events. See Chapter 8, "Timers" for the detailed description of timer usage.

1.5.7.3 Statecharts

During its lifetime, an active object performs operations in response to external or internal events and conditions. Existence of a state within an active object means that the order in which operations are invoked is important. For some objects, this event- and time-ordering of operations is so pervasive that you can best characterize the behavior of such objects in terms of a state transition diagram – a statechart. A statechart is used to show the state space of a given algorithm, the events that cause a transition from one state to another, and the actions that result from state change. AnyLogic supports hybrid statecharts – the most natural and powerful way to integrate discrete logic and continuous time behavior. In hybrid statecharts you can associate a set of equations with a statechart state. Then state transitions will alter the continuous behavior. Also, you can specify a condition over continuously changing variables as a trigger of a transition. Then continuous process will drive the discrete logic. See Chapter 9, "Statecharts" for the detailed description of statecharts.

1.5.7.4 Writing code for an active object

You can write your own Java code for an active object class in the Code window of the active object class. Namely, you can define arbitrary member variables, nested classes and methods. Class data members can be accessed anywhere within this object, and methods can be called on some object activity; e.g., on timer expiration or on triggering statechart transition. Writing your code, you can also specify arbitrary actions to be executed at the

model startup. See section 1.5.9, “Writing code for an active object” to get the detailed information on writing code for an active object.

1.5.8 Active objects interaction

AnyLogic supports continuous and discrete time active object interaction mechanisms. Combining these mechanisms you can create sophisticated interfaces of active objects.

1.5.8.1 Variable sharing

Variable sharing is the continuous time interaction mechanism in AnyLogic. An active object can have variables, modeling values changing continuously over time. Those variables may be exposed at the active object interface and shared with other active objects. When two interface variables of different objects are linked, changes of one variable (set up as the output one) are immediately propagated to another variable (considered as input). This provides for continuous and/or discrete time object interaction. See Chapter 4, “Variables” for the detailed description of variables and variable sharing.

1.5.8.2 Message passing

Message passing is the discrete time interaction mechanism in AnyLogic. The mechanism concept is passing data units – messages – between active objects. Messages are sent and received at the special elements of active objects – ports – and routed along connection lines, which links ports. Message passing may model a notification or signaling mechanism – in this case, messages represent commands or signals being passed in a control system; or an entity flow – in this case messages model various objects of the real world – e.g., products, people, trucks, etc., or data packets being passed in a network. See Chapter 7, “Message passing” for the detailed description of ports and message passing.

The subsequent section describes how to establish active object interaction in systems constructed solely from AnyLogic library objects.

1.5.8.3 Establishing inter-object interaction

To establish inter-object interaction you need to connect the respective interface elements of active objects with connectors. A connector is a line connecting two ports or two variables. In AnyLogic, connectors are used to construct topologies. Connecting two ports means that messages will be passed between them. Connecting variables means that they will have the same value at any moment of time (the output variable value will be passed to input variable).

Interface elements are displayed on the structure diagram as small shapes on the borders of encapsulated objects (see Figure 14). Ports are displayed as squares and variables as triangles (triangles pointing inside the object denote input variables whereas pointing outside – output ones). You cannot add, delete, or move interface elements, since they reflect the interface of an encapsulated object class.

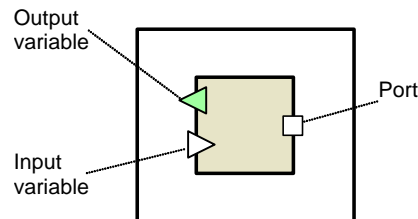


Figure 14. Interface elements of an encapsulated object


Interface elements of encapsulated objects have the following properties:

Properties

Name – [read only] the name of the interface element.

Show name – if set, the name of the interface element is shown on the structure diagram.

► To connect interface elements of encapsulated objects

1. Drag the interface element of one encapsulated object onto the interface element of another encapsulated object, or
Click the *Connector*  toolbar button, click the first interface element and then click the second interface element, or
Choose *Draw | Structure | Connector* from the main menu, click the first interface

element and then click the second interface element.

The connector linking two interface elements appears (see Figure 15).

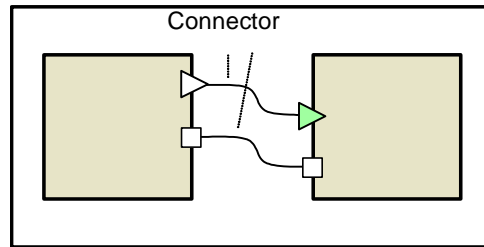


Figure 15. Interface elements of encapsulated objects connected

A connector has the following properties:

Properties


Name – [optional] name of the connector.

Exclude from build – if set, the connector is excluded from the project.

Show name – if set, the name of the connector is shown on the structure diagram.

You can edit a connector appearance by editing its salient points.


► To add a salient point

1. Select the connector.
2. Click the *Edit Points*  toolbar button, or
Choose *Draw | Edit Points* from the main menu, or
Right-click the connector and choose *Edit Points* from the popup menu.
The points of the connector should turn yellow.
3. Drag a segment of the connector to create a salient point, or
Right-click the segment and choose *Add Point* from the popup menu.

► To move a salient point

1. Drag the point.

► **To remove a salient point**

1. Select the connector.
2. Click the *Edit Points*  toolbar button, or
Choose *Draw | Edit Points* from the main menu, or
Right-click the connector and choose *Edit Points* from the popup menu.
3. Right-click the point and choose *Delete Point* from the popup menu, or
Drag the point to an adjacent point of the connector.
The dragged point disappears.

Interface elements of encapsulated objects can be exported to the interface of the parent object class. It means that the corresponding interface element is added to the parent object interface and connected to the exported element. Thus the encapsulated object can interact with its parent object by passing messages via connected ports or propagating connected variables changes.

► **To export an interface element of an encapsulated object to an interface of this object**

1. Right-click the interface element and choose *Export to Parent* from the popup menu.
The new class interface element is created and connected to the exported interface element of an encapsulated object.

1.5.9 Writing code for an active object

You can write arbitrary Java code for an active object to be executed on different occurrences. The code for an active object class is specified in the Code window associated with this active object.

► **To open the Code window of an active object class**

1. In the Project window, right-click the *Code* item in the object class' subtree of the workspace tree, and choose *Open Code* from the popup menu, or
Double-click the *Code* item in the object class' subtree of the workspace tree.
The Code window of the active object class is displayed (see Figure 16).

```

Root
Import

Implements interfaces

Startup code
//add rocks
rocks.add( animation.rectRockA );
rocks.add( animation.rectRockB );
rocks.add( animation.rectRockC );
rocks.add( animation.rectRockD );

Equations
d(y)/dt = vy
d(x)/dt = vx

Additional class code
//all rectangular rocks
Vector rocks = new Vector();
//simple distance
double distance( double x0, double y0, double x1, double y1 ) {
    return Math.sqrt( (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) );
}

//point hit detection
boolean xLTxt;
boolean yLTyt;

void setupTraget( double xt, double yt ) {
    xLTxt = x < xt;
    yLTyt = y < yt;
}

```

Figure 16. Code window of an active object class

The Code window of active object class has the following sections:

- Import* – import statements needed for correct compilation of the class code. When Java code is generated, these statements are inserted before definition of the Java class.
- Implements interfaces* – comma-separated list of interfaces implemented by the class.
- Startup code* – the sequence of Java statements to be executed after all objects throughout the whole model are constructed, connected, and initialized, and before anything else is done. This is a place for starting object's activities such as statecharts, threads and timers. The order of execution of *Startup* code of different objects is not guaranteed.

Equations – the set of equations associated with the active object. See Chapter 5, “Equations” for details.

Additional class code – arbitrary member variables, nested classes, constants and methods are defined. This code will be inserted into the class definition. You can access these class data members anywhere within this object. The methods can be called on some object activity; e.g., on timer expiration or on triggering statechart transition.

1.5.9.1 Importing packages

If you use a class from another package, you have to import that package using `import` statement or prefix the name of the class with the name of the package.

► To add the “import” statement to an active object class

1. In the *Import* section of the active object class' Code window, type the sequence of Java import statements.

► To add the “import” statement to all classes in a project

1. In the Project window, click the project to which you want to import a package.
2. On the *Code* page of the Properties window, type the sequence of Java `import` statements in the *Import (applies to all classes)* section.

1.5.9.2 Accessing active objects from code

A non-replicated encapsulated object `anObject` can be accessed simply by its name, `anObject`. AnyLogic active objects are instances of the class `ActiveObject`. Please consult AnyLogic Class Reference for more information on properties and methods of `ActiveObject`.

► To view AnyLogic Class Reference

1. Choose *Help | Class Reference* from the main menu.
The browser window opens with AnyLogic Class Reference displayed.

The method of the active object class `getOwner()` returns the parent active object, i.e. the one that encapsulates this object, or `null` if called from the root object. This method returns an object of type `ActiveObject` and you have to cast it to a particular known object type if needed.

1.5.10 Active object inheritance

Being Java classes, active object classes may inherit one from another. Any user-defined active object class inherits from the predefined class `ActiveObject` directly or transitively. The class `ActiveObject` has properties common to all active objects. Please consult AnyLogic Class Reference for more information on properties and methods of `ActiveObject`.

There is one restriction, however. Among an active object class and its superclasses, there should be exactly one active object class generated by AnyLogic and all other classes should be defined manually; e.g., in external files.

► To set a base class for an active object class

1. Click the active object class in the Project window.
2. In the Properties window, specify the base class name in the *Base class* edit box.

Figure 17 shows examples of inheritance between active object classes. Classes generated by AnyLogic – i.e., classes that have structure diagrams – are framed.

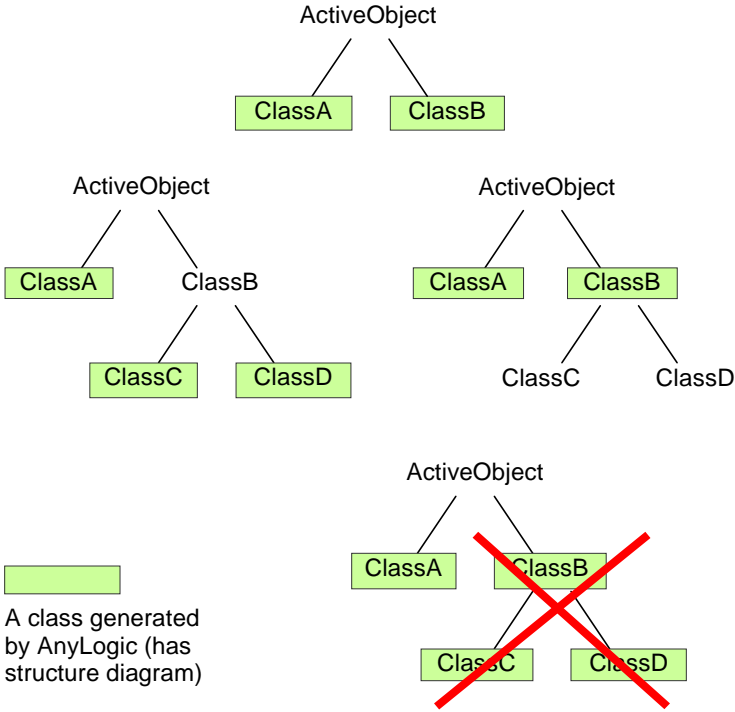


Figure 17. Inheritance among active object classes

2. Replicated objects

AnyLogic supports object *replication* - very easy and convenient way of modeling regularly organized structures of objects of arbitrary size and topology, such as vector, mesh, torus, hypercube, chain, ring, etc. AnyLogic saves you from the boring process of manual creation of the specified number of objects and establishing connections between them as this approach is limited, tedious, and forms the system with a predefined and constant number of objects.

In AnyLogic you simply declare a replicated object and specify the number of instances as a parameter. A replicated object represents a collection of active objects of the same class. Object replication provides very economical representation of complex structures of objects with arbitrary interconnections.

Object replication enables you to:

- Build scalable systems by creating replicated objects with the number of elements defined by a parameter of the model.
- Create complex structures of objects with sophisticated topologies by establishing arbitrary interconnections between replicated objects either from AnyLogic UI or programmatically.
- Model systems with dynamically changing structures by adding and removing objects to the group of replicated objects at runtime.

Do not confuse object replication with model replications. Model replication is a single model run and has nothing to do with object replication.

2.1 Creating a replicated object

In AnyLogic you can create a collection of objects of the same type simply by creating a replicated object. First, encapsulate an object you want to create a structure from, and then declare it replicated.

► To declare a replicated object

1. On the structure diagram, click the encapsulated object you want to make replicated.
2. On the *Replication* page of the Properties window, specify the number of individual elements of the replicated object (the replication factor) in the *Number of objects* edit box.
The name of the replicated object is displayed on the structure diagram with the trailing asterisk.

2.1.1 Replication factor parameterization

You can specify any expression evaluating to `integer` or `double` type as a number of elements of the encapsulated object. Thus, you can parameterize the number of object instances in a replicated object. You may need this to facilitate modifying the number of objects of several replicated objects. See Chapter 3, “Parameters” for more information on parameters.

If you use a parameter as a replication factor of an encapsulated object, AnyLogic creates a number of instances equal to the initial value of the parameter. However, AnyLogic does not automatically adjust the number of instances in case you change the parameter at runtime. If such behavior is needed, you should use the handler method `onChange_myParam()` to take care of creation and destruction of instances (see Chapter 15, “Creating a model with dynamically changing structure”).

2.2 Accessing and modifying a replicated object at runtime

This section describes accessing individual elements of a replicated object from code and creation/removing objects to/from a group of replicated objects at runtime. See section 15.1, “Manual creation and destruction of encapsulated objects” for general information about creating and destroying encapsulated objects at runtime.

2.2.1 Accessing replicated objects from code

The number of active objects in a replicated object `anObject` can be retrieved by calling `anObject.size()`.

In order to get any individual element from `anObject` vector of replicated objects, you can use `anObject.item(index)` method, which takes an index of the object in the vector, from 0 to `anObject.size() - 1`.

Individual elements of a replicated object can find out their indexes by calling `getIndex()`.

2.2.2 Adding/removing objects to/from a replicated object

To add a new object to an `anObject` replicated object, use the `setup_anObject()` method generated by AnyLogic. The method adds an object to the array of replicated objects and registers the object within the simulation framework. If there are any connectors to other objects designed in the structure diagram, those connections are established for the new object. The method takes two arguments: an object and an object identifier. The object identifier must be unique within the replicated object vector to distinguish objects in the Model Explorer at runtime. For the reasons of simplicity, you can use the `hashCode()` method, returning the unique integer value.

For example, write the following code to add `c` object to `clients` replicated object.

```
setup_clients(c, c.hashCode());
```

Use generated by AnyLogic method `dispose_anObject()` to remove the object from the `anObject` vector of objects.

Related method of ActiveObject

```
final void setReplication(int index) – the framework method. You can call it to place the element of a replicated object in the place specified by index parameter.
```



2.3 Connecting replicated objects

AnyLogic provides a convenient mechanism of creating structures of objects with sophisticated topologies. It supports arbitrary types of interconnections between individual elements of replicated objects and between a replicated object and other objects (probably, also replicated).

First, this section describes how to connect ports and variables of replicated objects graphically. Then the information on establishing arbitrary types of interconnections between replicated objects is given. As connection rules are the same for variables and ports, they are described generally for interface elements of replicated objects.

2.3.1 Connecting interface elements of replicated objects graphically

► To connect interface elements of replicated objects

1. Drag the interface element of one object onto the interface element of another object, or
Click the *Connector*  toolbar button, click the first interface element and then click the second interface element, or
Choose *Draw | Structure | Connector* from the main menu, click the first interface element and then click the second interface element.
The connector linking two interface elements appears (see Figure 18).

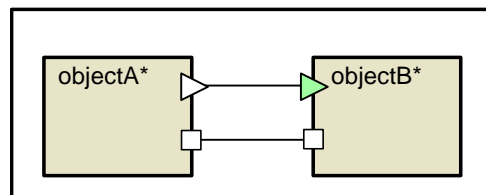


Figure 18. Interface elements of replicated objects connected

When you connect two replicated objects using a connector, you get configurations shown in Figure 19, i.e. each element in the `client` vector is connected with the `server` object.

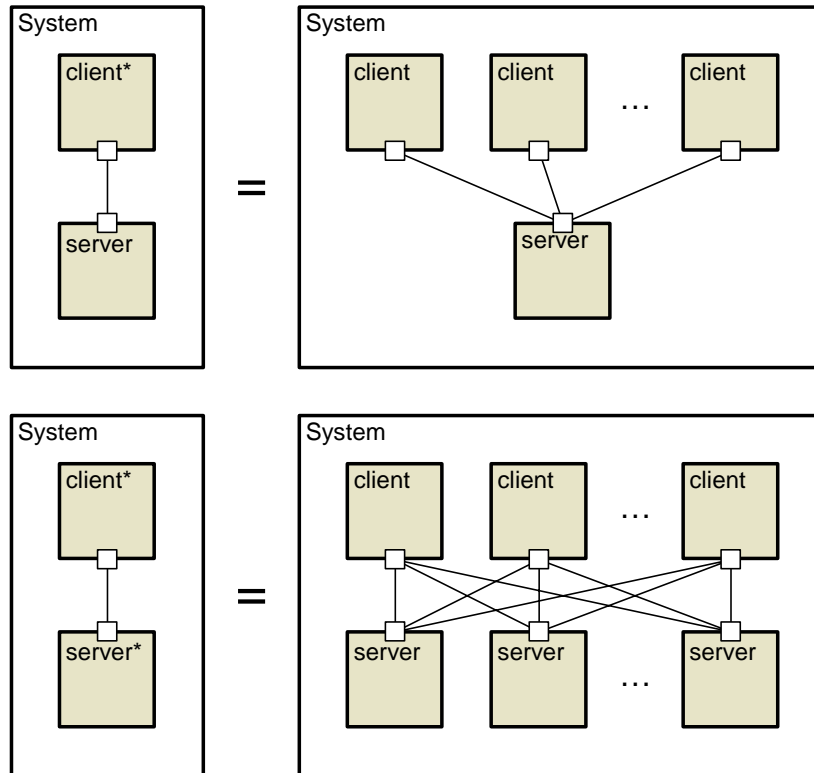


Figure 19. Connection of replicated objects

2.3.2 Connecting replicated objects with other objects

A replicated object represents a collection of object copies. When you graphically connect an interface element of some encapsulated object to an interface element of a replicated object, it is connected to interface elements of all object copies of the replicated object (see Figure 19). In some cases you may want to connect an interface element of one object to the interface elements of the certain individual elements of the replicated object only. In AnyLogic you can establish arbitrary types of connections by adjusting connector properties.

► **To set a connection type between interface elements of a replicated object and another object**

1. In the structure diagram, select the connector, linking the interface element of the replicated object with the interface element of the other object.

2. In the Properties window, specify individual elements of the replicated object to be connected to the interface element of the other object from the *Connect from 'interface element'* of drop-down list:
 - All objects* – the default value. Interface elements of all individual elements of the objects replicated object are connected to the interface element of the other object.
 - First of objects* – the interface element of the first individual element only is connected.
 - Last of objects* – the interface element of the last individual element only is connected.

Figure 20 illustrates all these connection types.

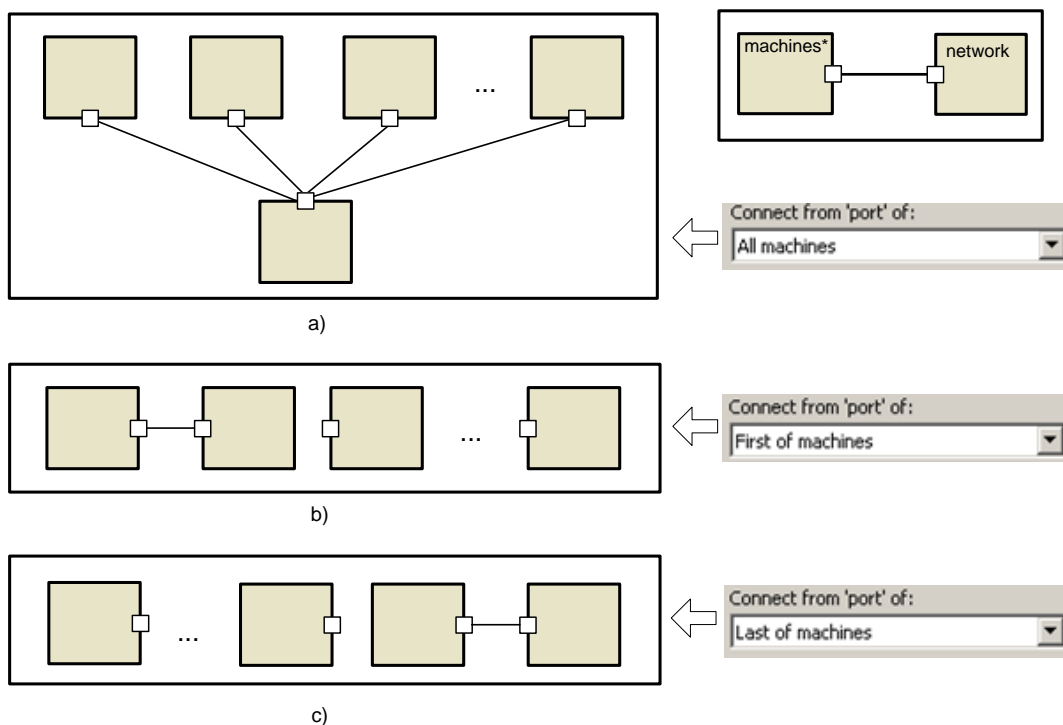


Figure 20.

You may connect interface elements of two replicated objects as well. By default, interface elements of all object copies of one replicated object are connected to interface elements of all object copies of another replicated object (see Figure 19).

However, you can connect interface elements of certain object copies only. Therefore you need to specify the object copies of replicated objects you want to connect. This is done in

Connect from 'interface element' of and To 'interface element' of connector's properties in the same way as described above for the connection of a replicated object with an encapsulated object.

Figure 21 illustrates some possible cases of connecting interface elements of two replicated objects.

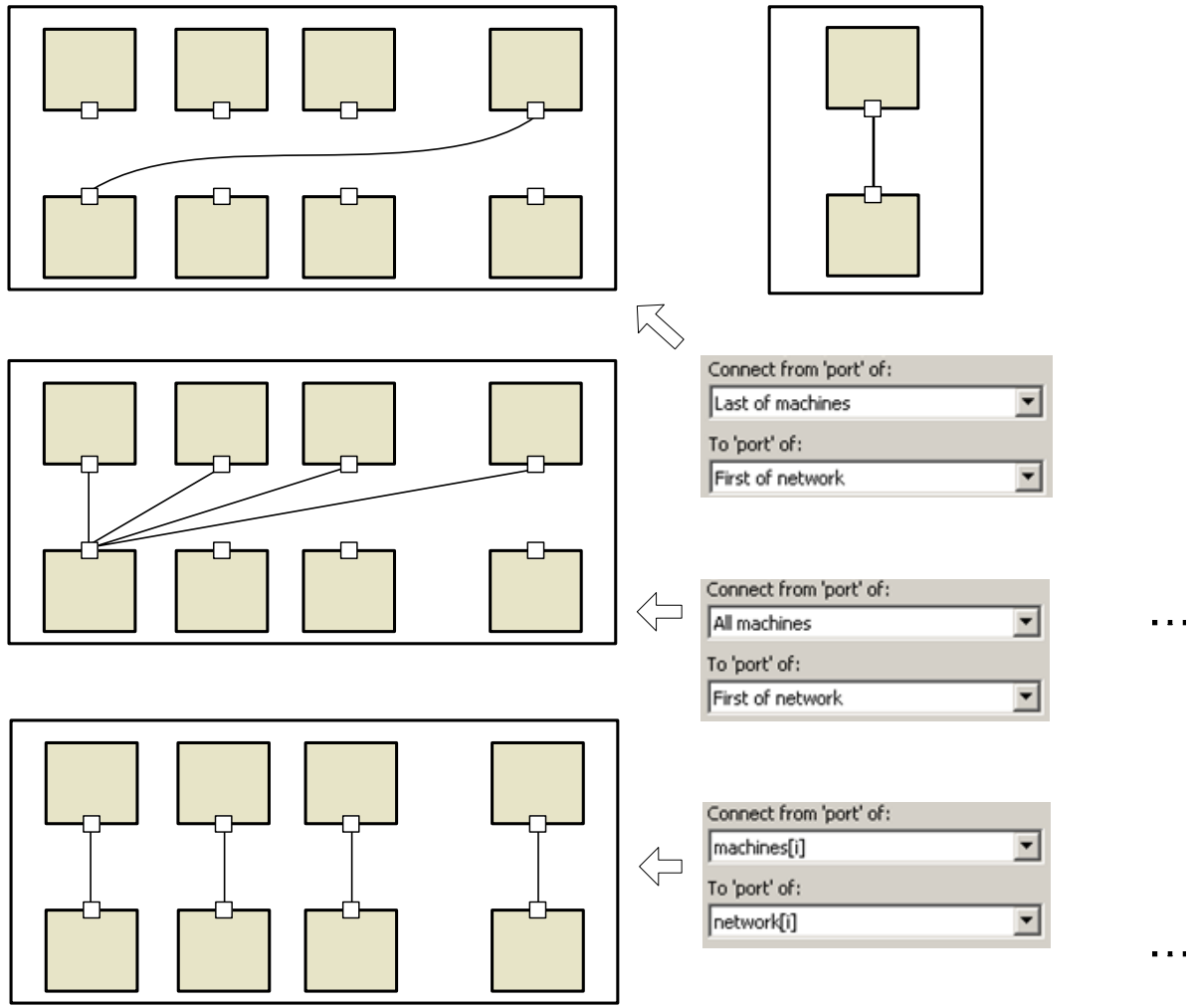


Figure 21. Replicated objects connection cases a)

In the case two replicated objects have the same number of individual elements, you can connect the i -th element of one replicated object with the i -th element of another replicated object, choosing $objects(i)$ from both drop-down lists – see c case in Figure 21.

2.3.3 Connecting individual elements of a replicated object

You may specify arbitrary interconnections between individual object copies of a replicated object. You can connect the same or different interface elements of some individual elements of a replicated object. Connect them in the same manner as you connect individual elements of a replicated object with another object.

► To connect individual elements of a replicated object

1. In the structure diagram, connect the required interface elements of the replicated object. You can connect an interface element to itself.
2. Select the created connector in the structure diagram.
3. In the Properties window, specify individual elements of the replicated object you want to connect for each connected interface element in *Connect from 'interface element' of* and *To 'interface element' of* drop-down lists.

Choose *All objects*, *First of objects*, or *Last of objects* to connect all individual elements, the first individual element, or the last individual element of the objects replicated object only.

You can connect interface elements of the same object copies in a vector of replicated objects by choosing *objects(i)* from both drop-down lists.

Also, you can connect interface elements of the adjacent object copies by choosing *objects(i)* and *objects(i+1)*.

Some connection cases are shown in Figure 22.

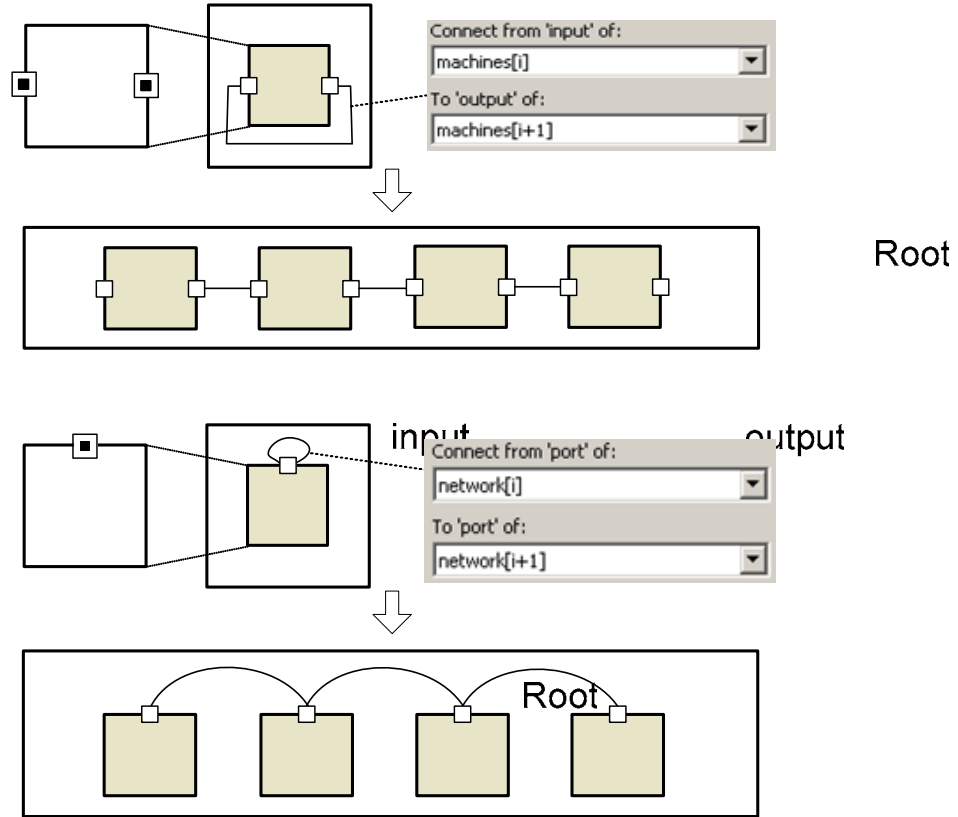


Figure 22. Regular interconnections in a vector of replicated objects

Thus, in AnyLogic you can easily create a regularly structured system.

port Root

2.3.4 Connecting replicated objects programmatically

Sometimes you may need to create a complex structure of objects with sophisticated topologies that cannot be established graphically. Or you may need to model a system with dynamically changing connections. Use the methods `connect()` and `disconnect()` to establish connections between replicated objects and their individual elements at runtime.

Use the `connect()/disconnect()` methods of the `Port` or `VariableRef` classes to connect/disconnect ports or variables of two objects. The methods take two arguments: two port objects, or two variable objects. The name of a port object is the name of the port in

Root

the structure diagram. However, the name of a variable object is the name of the variable in the structure diagram with the *_ref_* prefix.

- Use `Port.connect(objectA.portA, objectB.portB)`, or `objectA.portA.connect(objectB.portB)` code for ports
- Use `VariableRef.connect(objectA._ref_varA, objectB._ref_varB)`, or `objectA.ref_varA.connect(objectB.ref_varB)` code for variables.

For more information on connecting ports and variables refer to section 7.4.12, “Connecting ports at runtime” and section 4.3.2.2, “Connecting variables at runtime”. For more information on the `Port` and `VariableRef` classes, refer to AnyLogic Class Reference.

2.3.5 Creating a ring of cells

Suppose you want to build a ring of cells.

Create the `Ring` replicated object with the number of individual elements defined by `numberOfCells` parameter of `int` type. Then connect the output port with the input port and define the interconnection as shown in Figure 22 (the first case).

Connect ports of the leftmost and the rightmost objects manually to form a ring. Type the `onCreate()` method in the *Additional class code* section of the Code window of the `Ring` active object class to connect the ports on object creation.

```
public void onCreate() {
    ring.item(0).input.connect(ring.item(ring.size()-1).output);
}
```

The resulting structure is shown in Figure 23.

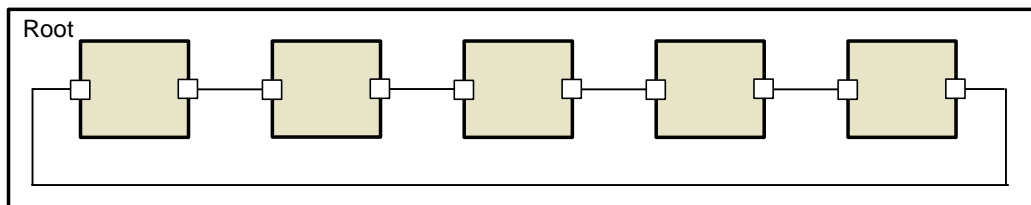


Figure 23. The ring of cells

2.3.6 Creating a torus of cells

Suppose you want to build a torus of cells. Let N be the radix of the torus. You first declare $N*N$ replicated cells.

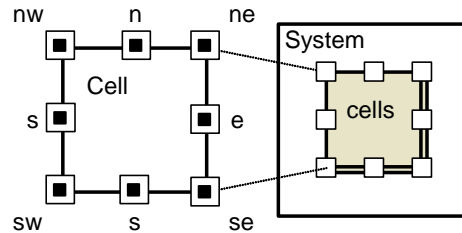


Figure 24. A cell for torus

Now you need to connect the opposite interface elements of adjacent elements of a replicated object, namely up interface element with down, up-right with down-left, right with left and down-right with up-left.

You connect `cells` manually in the method `onCreate()` in the *Additional class code* code section of the class `System` (this method is called on the active object creation):

```
...
public void onCreate() {
    for( int i = 0; i < N; i ++ ) {
        for( int j = 0; j < N; j ++ ) {
            cells.item(index(i,j)).nw.connect(cells.item(index(i-1,j-1)).se);
            cells.item(index(i,j)).n.connect(cells.item(index(i-1,j)).s );
            cells.item(index(i,j)).ne.connect(cells.item(index(i-1,j+1)).sw);
            cells.item(index(i,j)).e.connect(cells.item(index(i,j+1)).w);
        }
    }
}
...
```

Here the method `index()` maps the two-dimensional index to the one-dimensional index, e.g.:

```
...
int index( int r, int c ) {
    r += N; r = r % N;
    c += N; c = c % N;
    return r * N + c;
}
```


}
...

The resulting configuration of encapsulated objects is shown in Figure 25.

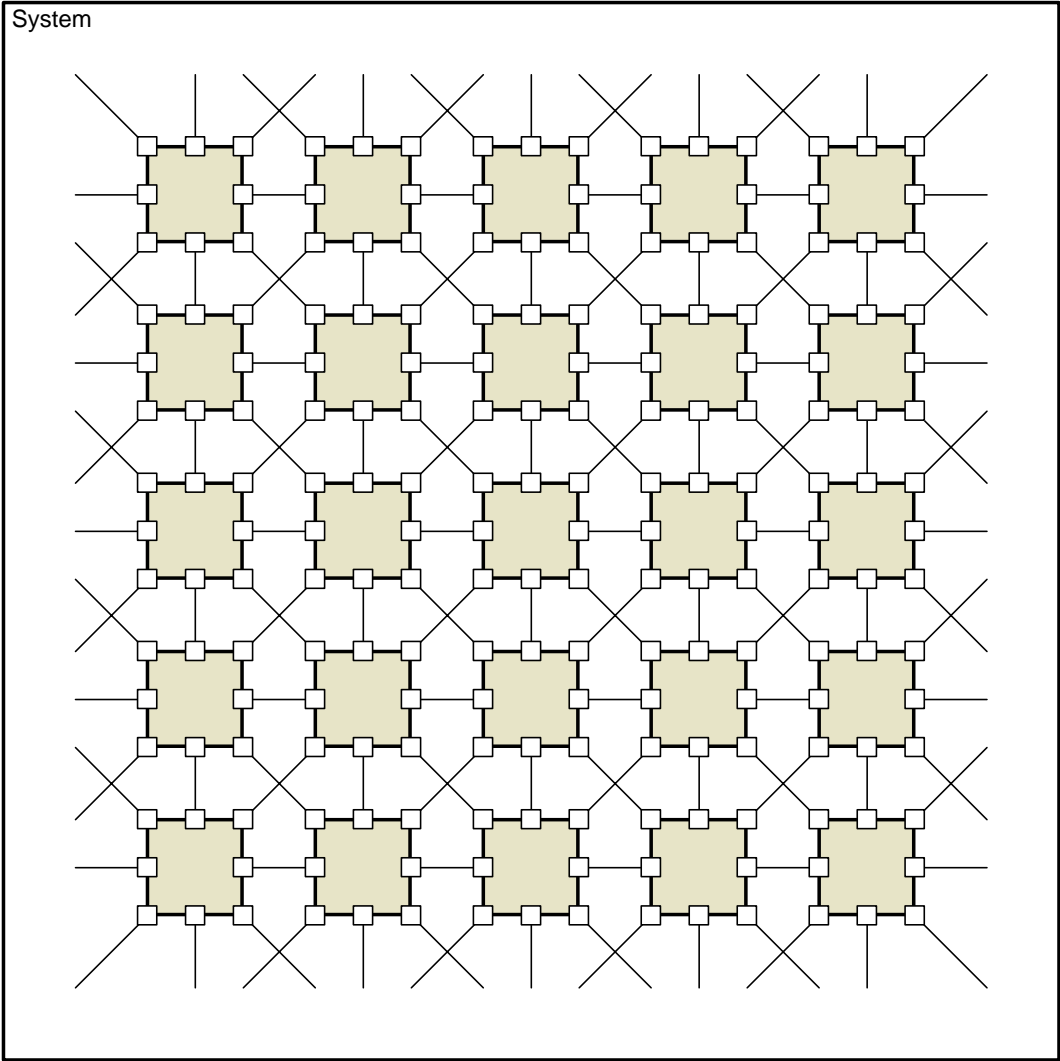


Figure 25. Pattern of cell configuration

The technique presented here provides the ability to create periodically organized structures of objects of arbitrary size and topology.

2.4 Viewing and modifying replicated objects at runtime

You can access and modify a replicated object and its individual elements at runtime like any other active objects either programmatically (see section 2.2.1, “Accessing replicated objects from code”) or from the AnyLogic UI.

In the Model Explorer, a replicated object is displayed as a branch of the model tree containing individual elements (see Figure 26). Along with the name of a replicated object, the number of its elements is displayed. Elements of the machine replicated object are named machine-0, machine-1, etc. You can expand and collapse object contents by clicking the plus and minus icons correspondingly.

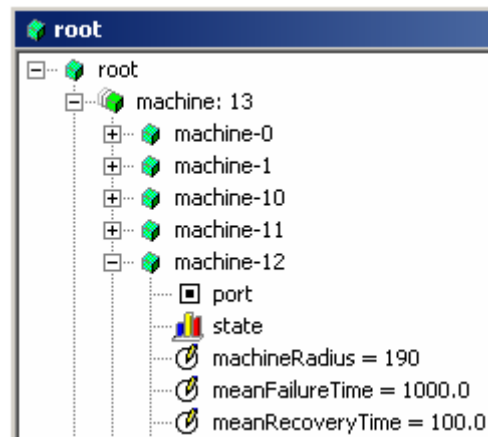


Figure 26. Replicated object in the Model Explorer

You can examine structure and current state of individual elements of a replicated object like any other active objects: open animated structure diagram window for an object, open inspect and log windows for object elements, etc. Running and debugging a model is described in Chapter 11, “Running and observing a model” and Chapter 14, “Debugging a model” correspondingly.

If you dynamically add or remove some elements to/from a vector of replicated objects, these changes are automatically displayed in the Model Explorer.

2.5 Animating a replicated object

AnyLogic animation is constructed from animations defined for active objects and composed according to the model hierarchy.

You can display an animation of a replicated object on an animation of a container object, displaying

- Animations defined for all the elements of a replicated object, or
- Animation(s) of the certain element(s) only.

To animate a replicated object, you define an animation for a replicated object class and place it on the animation of a container object. It is drawn as a rectangle showing the content of the animation of either all the individual elements or only of the certain object copies of a replicated object. You can move, scale, and rotate an encapsulated animation shape in the animation editor, or you can assign expressions to necessary properties to allow a model to move, scale, and rotate an encapsulated animation shape at runtime. The detailed information about animating encapsulated objects is given in section 12.1.2, “Animating hierarchical models”.

If an animation for an encapsulated object class is already defined when you create an instance of this class, the encapsulated animation shape automatically appears on the animation of a container object.

If you create a replicated object, and the replicated object class does not have animation defined for it, then the encapsulated animation shape is not created. If you define animation for the encapsulated object class after that, you have to manually create an encapsulated animation shape.

You may also need to manually place an encapsulated animation shape on the animation of the container object if you wish to display only the certain elements of the replicated object on the animation.


There are two options to place animations of a replicated object on the animation of a container:

- To display all elements of the replicated object, you draw an encapsulated animation shape with the encapsulated object name set to the whole vector of objects, e.g. cars,

server. In that case positions of different encapsulated animations are usually specified in the properties of the animation of the replicated object class.

- To display only the selected elements of the replicated object, you draw as many encapsulated animation shapes as necessary and specify the encapsulated object name for each shape in the form of <encapsulated object name>-<number>, e.g. cars-5, server-0, etc. Position of such encapsulated animation is usually specified directly in the properties of the encapsulated animation shape.

► To draw an encapsulated animation of a replicated object

1. Click the *Encapsulated Animation*  toolbar button, or Choose *Draw | Animation | Encapsulated Animation* from the main menu.
2. Click or drag a rectangle area on the animation diagram of a container object.
3. In the Properties window, specify the name of a replicated object (e.g., cars) or of a particular element (e.g., cars-1) this shape refers to in the *Object* edit box. The shape shows the content of the animation of the specified object.

In case a replicated object is created and destroyed dynamically, its animation appears and disappears synchronously with the object.

3. Parameters

Active object may have parameters. Using parameters, you can parameterize active objects in your model. It is needed when object instances have the same behavior described in class, but differ in parameter values.

All parameters are visible and changeable throughout the model execution. Thus, you can simply adjust your model by changing parameters at runtime. If you want, you can define a handler method that is invoked on a parameter change.

A parameter of an active object class can be associated with a parameter of its encapsulated object. In this case, the class parameter change propagates to the associated object parameter.

There is a clear difference between variables (see Chapter 4, “Variables”) and parameters. A variable represents a model state, and may change during simulation. A parameter is commonly used to describe objects statically. A parameter is normally a constant in a single simulation, and is changed only when you need to adjust your model behavior.

Use a variable instead of a parameter if you need to model some data unit changing over time.

3.1 Parameter types

Alike other simulation tools AnyLogic supports parameters of primitive types: `real`, `integer`, `boolean`. But only AnyLogic gives you infinite possibilities in parameterizing your objects by supporting parameters of any Java classes.

You can define parameters of common Java classes – e.g., a parameter of the `String` class to represent character strings, or of the `Vector` class to create a parameter representing a dynamic array of objects. You can create a parameter of the `Object` class (the base class for all Java classes) and assign an instance of any Java class to this parameter. Later on you will need to check the actual type of this parameter and cast it explicitly to the original Java class. For details on Java classes, see Java SDK documentation available at <http://java.sun.com/docs>.

You can define parameters of your own classes, defined in the active object class code.

Since all AnyLogic objects are instances of Java classes, you can define parameters of these classes and thus use AnyLogic objects as parameters. Table 1 lists some of AnyLogic objects along with their class names:

Object	Class	Description
Matrix	<code>Matrix</code>	See section 6.1, “Matrices”. In contrary to matrix variables, automatically initialized with zeros, always initialize matrix parameters on your own.
Enumeration	<code>Enumeration</code>	See section 6.2.1, “Enumerations”.
Hyper-array	<code>HyperArray</code>	See section 6.2, “Hyper-arrays”.
Lookup table function	<code>LookupTable</code>	Lookup tables are used for defining some complicated variable dependencies that cannot be described by composition of standard functions; e.g., time schedule. See section 5.3.3, “Lookup tables”.
Animation shape	<code>ShapeBase</code>	You can associate a custom animation shape with an object. Later on, when accessing this parameter in code, you will need to check the parameter type and cast it explicitly from the base animation shape class <code>ShapeBase</code> to the original animation shape class.
3D animation shape	<code>Shape3DBase</code>	You can associate a custom 3D animation shape with an object. Later on, when accessing this parameter in code, you will need to check the parameter type and cast it explicitly from the base 3D animation shape class <code>Shape3DBase</code> to the original 3D animation shape class.


Table 1. AnyLogic types parameters

See AnyLogic Class Reference for more information on AnyLogic classes.

3.2 Defining parameters

Parameters of active object classes are defined in the *Parameter* dialog box.

► To add the parameter to an active object class

1. In the Project window, click the active object class.
2. In the Properties window, click the *New Parameter*  button. The *Parameter* dialog box is displayed (see Figure 27).
3. In the *Parameter* dialog box, set up the parameter properties.

Parameter

Name:

Type:

Default value:

Simple
 Dynamic
 Global
 Separator

Predefined symbols:

Symbol	Value
MIN_PERIOD	0.1
MAX_PERIOD	0.85

Hide if:

Parameter	=	Value
yMax	==	30

Description:

OK Cancel

Figure 27. Parameter dialog box

The following properties of the parameter are set up in the *Parameter* dialog box:

Name – the name of the parameter.

Type – the type of the parameter.

Default value – the default value of the parameter. If any predefined symbols are defined, they appear in the combo box.

Simple – if set, this parameter is neither *global* nor *dynamic*.

Dynamic – if set, the parameter is dynamic. See section 3.6, “Dynamic parameters” for information on dynamic parameters.

Global – if set, the parameter is global. The global parameter of the active object class has the same value for all the instances of this class in the model.

Separator – if set, the parameter is used as a separator. Such a parameter is shown as the blank row in the *Parameters* table of actual parameters of the encapsulated object. It is used for visual separation of some groups of parameters.

Predefined symbols – the set of predefined symbols defined in the *Symbol Value* form, where *Symbol* is the name of the predefined symbol and *Value* is its value. Thus you can define some useful constants and refer to them when specifying the default value or the actual value of the parameter.

Hide if – the list of conditions defining when the actual parameter of the encapsulated object is hidden. Specify the *Parameter*, the $==$ or \neq comparison operation and the *Value*.


Description – the description of the parameter.

When you've finished defining the parameter, the new row appears in the *Parameters* table of the active object properties.

You can use predefined symbol “index” as the root object parameter value to refer the current run number (see section 13.2, “Model replications”).

Later, you can change the properties of the parameter using the *Parameter* dialog box.

► To change parameter properties


1. In the Project window, click the active object class.
2. In the Properties window, select the parameter in the *Parameters* table.
3. Double-click the parameter in the *Parameters* table, or
Click the *Edit*  button.
The *Parameter* dialog box is displayed.

► To delete a parameter


1. In the Project window, click the active object class.
2. In the Properties window, select the parameter in the *Parameters* table.

3. Click the *Delete*  button.


► **To move a parameter up**

1. In the Project window, click the active object class.
2. In the Properties window, select the parameter in the *Parameters* table.
3. Click the *Move Up*  button.

► **To move a parameter down**

1. In the Project window, click the active object class.
2. In the Properties window, select the parameter in the *Parameters* table.
3. Click the *Move Down*  button.

► **To duplicate a parameter**

1. In the Project window, click the active object class.
2. In the Properties window, select the parameter in the *Parameters* table.
3. Click the *Duplicate*  button.

3.3 Setting up actual parameter values

An active object class has a set of formal parameters. When the object is encapsulated in another object, actual parameters may be assigned to its formal parameters; otherwise, the default values are assumed. Thus, you can set various actual parameter values for different instances of the same active object class.

► **To set actual parameter value of an object instance**

1. Select the encapsulated object on the structure diagram.

- In the Properties window, specify the parameter value in the *Value* cell of the *Parameters* table (see Figure 28).

The changed parameter value is emphasized with bold.

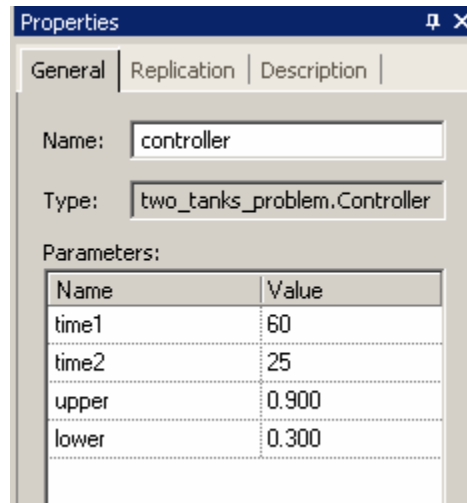


Figure 28. Encapsulated object properties page. Parameters table

If you change parameters of an active object class after creating instances of that class, check parameters of those instances afterwards.

You can assign actual parameters to parameters of the root object of experiment. Thus, you can create several experiments with the same root object set up but with differing parameters. Simulating your model with different current experiments, you can observe and compare model behavior with different parameters.

► To set actual parameter value for a root object of an experiment

- In the Project window, click the experiment.
- In the Properties window, specify the parameter value in the *Value* cell of the *Parameters* table.

The changed parameter value is emphasized with bold.

3.4 Modifying parameters at runtime

You can change parameters at runtime programmatically as well as from the AnyLogic viewer UI or from the AnyLogic animation. If a parameter is associated with any parameters of encapsulated objects, the change is propagated along the parameter dependencies, see section 3.5, “Parameter propagation”. You can define a handler method that is invoked on a parameter change.

3.4.1 Modifying parameters from Model Explorer

► To modify a parameter

1. In the Model Explorer, double-click the parameter, or Right-click the parameter and choose *Modify* from the popup menu. The *Modify* dialog box is displayed.
2. Type new value in the *Enter new value* edit box.
3. Click *OK*.

3.4.2 Modifying parameters from AnyLogic animation

AnyLogic offers a set of controls (buttons, text inputs, checkboxes, sliders) for creating interactive animations. You can modify a parameter of an active object by associating it with an animation control and changing control at runtime.

You can associate:

- a parameter of `boolean` type with a button or a check box
- a parameter of `String` type with an edit box
- a parameter of `double` type with a slider.

► To associate a parameter with an animation control

1. Select the control on the animation diagram.
2. In the Properties window, choose the parameter from the *Variable name* drop-down list.
3. If needed, specify the code to be executed when the user changes the control in the *Event handling code* edit box.

See Chapter 12, “Animation” for more information on AnyLogic animation.

3.4.3 Accessing and modifying parameters programmatically

You can change a parameter programmatically. A parameter `myParam` of an active object can be accessed simply as a member variable `myParam`. Parameter may appear in equations and code within an active object.

If, however, you are modifying the parameter and want this modification to be propagated down along the parameter dependencies (see section 3.5, “Parameter propagation”), you have to call the method – e.g., `set_myParam()` generated by AnyLogic, passing the value you want to assign as a method parameter.

The parameters of the root object can be setup from the command line, see section 18.1, “Running a model from the command line”.

Parameters of an encapsulated object are passed to it right after it has been created by the new operator. During execution of the constructor, parameters are not yet initialized. Thus, if you want to use a parameter to initialize, e.g., a member variable of an object, you cannot do it in the member variable declaration. You should do it somewhere else; for example, in the method `onCreate()` of the object. By the time `onCreate()` is called, all parameters are already initialized.

3.4.4 Defining parameter change handlers

If you need to perform some action on parameter change, you can define a handler method. For example, for the `myParam` parameter AnyLogic calls the method `onChange_myParam()`.

If you use a parameter as a replication factor of an encapsulated object, AnyLogic creates a number of instances equal to the initial value of the parameter. However, AnyLogic does not automatically adjust the number of instances in case you change the parameter at runtime. If such behavior is needed, you should use the handler method `onChange_myParam()` to take care of creation and destruction of instances (see section 15.1, “Manual creation and destruction of encapsulated objects” for information on dynamic object creation).

3.5 Parameter propagation

You can associate a parameter of an active object class with a parameter of its encapsulated object. In this case if you change a class parameter during the model execution, the associated object parameter depending on it also changes. This holds generally for all parameter dependencies down the active object tree from the modification point.

Propagate values of parameters down the objects hierarchy when:

- You need to change parameters of several encapsulated objects (perhaps of different classes). You can simply do this by creating single parameter of the root object and propagating its value to several parameters you need to change.
- You need to optimize the model by changing the parameter of a non-root object. In this case, you also need parameter propagation since you can optimize model by changing only the root object parameters.

You can associate only parameters of the same type.

3.5.1.1 Setting up parameter propagation manually

► To set a propagation manually

1. Select the encapsulated object on the structure diagram.
2. In the Properties window, select the parameter you want to link with the parent object parameter, in the *Parameters* table.

3. In the *Value* field, type the name of the parent class parameter you want to propagate.

3.5.1.2 Setting up parameter propagation using Export Parameter to Owner dialog

You can also set parameter propagation using the *Export parameter to Owner* dialog.

► To set a propagation link

1. Select the encapsulated object.
2. In the Properties window, right-click the row holding the parameter you want to propagate in the *Parameters* table.
3. Choose *Export to Owner* from the popup menu.
The *Export Parameter to Owner* dialog is displayed.
4. If you want to link object parameter with a new class parameter, choose the *Create a new parameter* option, click the *Next* button, and specify the parameter name, type and default value using the *Parameter*, *Type*, *Default value* controls.
5. Otherwise, if you want to link object parameter with an existing class parameter, choose the *Select an existing parameter* option, click the *Next* button and select a parent class parameter you want to propagate by clicking it in the parameters table.
6. Click the *Finish* button.

3.5.1.3 Removing propagation link

► To remove a propagation link

1. Select the encapsulated object.
2. In the Properties window, right-click the row holding the linked parameter in the *Parameters* table.
3. Choose *Reset to Default* from the popup menu.

Parameter change is propagated only in one direction, down the active object tree along the parameter dependencies.

Figure 29 illustrates parameter propagation.

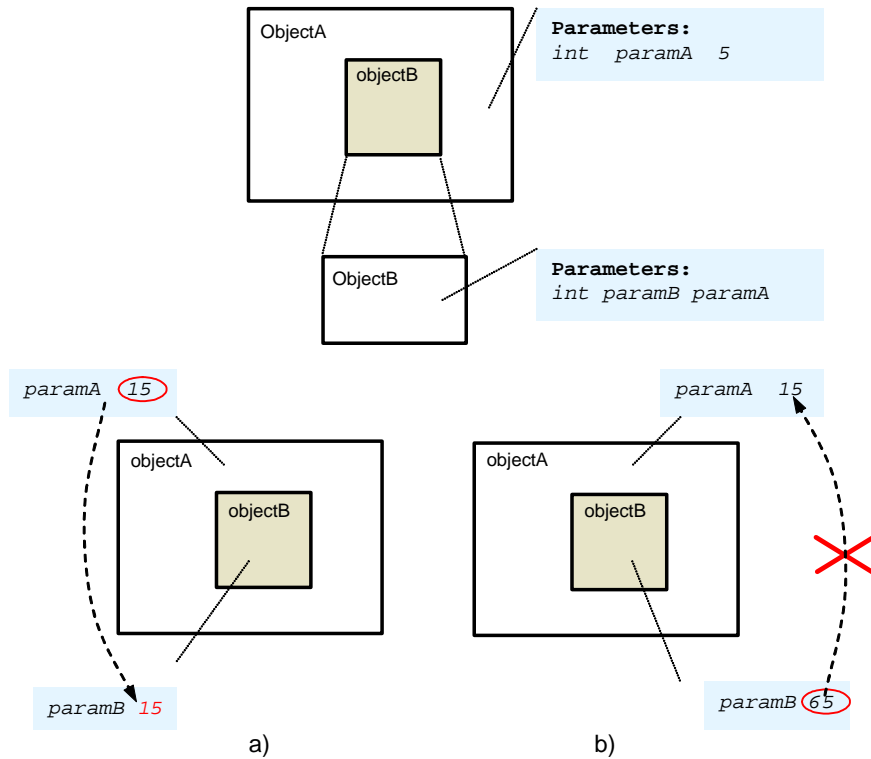


Figure 29.

The objectB is encapsulated in the ObjectA class. Its paramB parameter is associated with the paramA parameter of the parent class.

When you change the paramA parameter of the capsule class, the paramB parameter of the encapsulated object is changed as well. But changing the paramB parameter does not affect the paramA parameter.

3.6 Dynamic parameters

Dynamic parameters are special types of parameters. Dynamic parameter value is recalculated each time you assess the parameter; i.e., this parameter acts as a function.

As a value for a dynamic parameter you can type any expression evaluating to the parameter type. The example of a boolean expression is:

```
msg.weight + 5 > 75
```

where `msg` is the predefined variable or Java variable of the active object class. Providing different codes, you can decide what packs are big for each active class instance individually.

Note that the parameter expression will be reevaluated each time you assess a parameter value. You obtain the parameter value using function-call notation – e.g., `myParameter()`, not `myParameter` – because dynamic parameters become functions in the generated Java code.

Using dynamic parameters, you can parameterize active object instances with some code strings and thus greatly improve the flexibility and reuse of your active object class.

3.7 Observing model behavior with different model parameters

You may need to carry out several simulation experiments to observe and compare model behavior with different model parameters.

A model simulation comprises one or several single model runs - replications. You can use several replications in one model simulation and vary model parameters after each model replication. Thus, you can run your model with different parameters and analyze its behavior. This can be done by specifying a number of model runs and writing your own code to be executed between model replications on the *Code* page of the project's properties window.

For example, you need to observe the model behavior with the integer `rateParameter` parameter of the root object changing from 1 to 10 over 10 model replications. Thus your model will be run 10 times, each time with increased `rateParameter` parameter value.

First, specify a number of model runs for the current experiment.

► **To set a number of model runs for an experiment**

1. In the Project window, click the experiment.
2. In the Properties window, set the number of model replications in the *Number of runs* edit box.

Second, specify code you want to be executed before or after each model replication in the *Before replication* or *After replication* section of the project properties correspondingly.

Type the following code in the *After replication* section of the project code properties:

```
rateParameter ++;
```

3.8 Optimizing model parameters

If you need to run a simulation and observe system behavior under certain conditions, as well as improve system performance, for example, by making decisions about system parameters and/or structure, you can use the optimization capability of AnyLogic. Using sophisticated algorithms, AnyLogic automatically finds the optimal values of model parameters, with respect to certain constraints. The optimization process consists of repetitive simulations of a model with different parameters.

Only a parameter of the root object can be optimized. If you need to optimize parameters of encapsulated objects, you must use parameter propagation.

For detailed information see Chapter 16, “Optimization”.

4. Variables

An active object can contain variables – entities that model data items (may be continuously changing over time). Variables are generally used to store the results of model simulation or to model some characteristics and parameters of objects, changing over time. During the model execution, variables are observable and changeable from the AnyLogic UI.

A variable can be of an arbitrary scalar type, a matrix of an arbitrary dimension, or a hyper-array – a multi-dimensional array of real numbers. This chapter describes scalar variables only. AnyLogic matrices and hyper-arrays are described in detail in Chapter 6, “Matrices and hyper-arrays”.

You can define a set of differential equations, algebraic equations, and formulas to describe continuous changes of variables over time. Thus you can define a continuous time object behavior. Modifying variables at runtime, you can adjust the model behavior.

Variables can be exposed to the active object interface and shared with other active objects. In this case, variable changes are immediately propagated to the dependent variable of another object. This provides for continuous and/or discrete time object interaction.

You can declare a Java member variable in the *Additional class code* code section of an active object class. Use member variables instead of variables if you just need a data item to be accessed only within an active object and only at discrete steps (i.e., it is neither shared with other objects nor changed continuously), and you do not want to observe or change that item during model execution.


Use a parameter (see Chapter 3, “Parameters”) instead of a variable if you just need to model some static parameter of an object changed only at particular moments of time (e.g., between model runs).

4.1 Defining a variable

Variables can be either internal (or state variables) or public (or interface variables). A state variable is only accessible from within the active object. An interface variable can be shared with other active objects.

Declare interface variable if you need to establish a continuous time object interaction by sharing it with other objects. Otherwise, if you need just to model some data unit within the active object, declare a state variable.

► **To define a state variable**

1. Click the *Variable*  toolbar button, or
Choose *Draw | Structure | Variable* from the main menu.
2. Click inside the class border on the structure diagram.
A new variable appears, displayed as the small circle, see Figure 30.

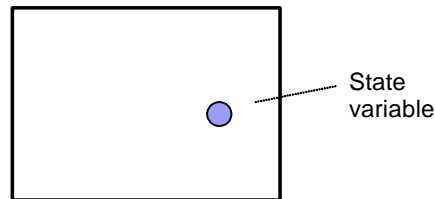


Figure 30. State variable

► **To define an interface variable**


1. Click the *Variable*  toolbar button, or
Choose *Draw | Structure | Variable* from the main menu.
2. Click the class border on the structure diagram.
A new variable appears, displayed as the small triangle, see Figure 31.



Figure 31. Interface variables

Placing a variable on the class border makes it interface.

You can simply change the accessibility status of the created variable.

► **To make variable interface**

1. On the structure diagram, drag the variable on the class border.

► **To make variable state**

1. On the structure diagram, drag the variable into the class border.

Interface variable is displayed on the structure diagram as a triangle. It can be either input (a triangle pointing inside the object) or output (a triangle pointing outside). The variable “direction” is significant in variable sharing – output variable changes will be propagated to the dependent input variable (see section 4.3, “Variable sharing” for the detailed description of variable sharing mechanism).

Once the new variable is created, you can specify the variable name in the text line editor opened on the right of the variable in the structure diagram.

Moving, copying and deleting variables

There are some common operations you can perform with variables on the structure diagram. You can copy, move and delete variables just as any other class elements. First, you should select the variable by clicking it.


► **To move a variable**

1. Drag the variable with the mouse or use arrow keys.

► **To copy a variable**

1. Ctrl-drag the variable.

► **To delete a variable**

1. Click the *Delete*  toolbar button, or
Choose *Edit | Delete* from the main menu, or
Right-click the variable and choose *Delete* from the popup menu, or
Press Del.

A variable has the following properties:

Properties

Name – name of the variable.

Variable type – type of the variable. Variable can be scalar of an arbitrary type (*Scalar*), a matrix (*Matrix*), or a hyper-array (*Array*).

Direction – *No direction* | *Input* | *Output* direction of the variable.

Equation – [optional] equation associated with this variable.

Auto collect dataset – if set, AnyLogic collects variable samples from the beginning of the simulation. Otherwise data collecting starts only when and if the variable is added to a chart window.

Exclude from build – if set, the variable is excluded from the model.

Show name – if set, the name of the variable is shown on the structure diagram.

You can declare scalar variables of any Java types, namely:

- variables of primitive types (real, integer, boolean),
- instances of any Java classes (String, Vector, Object, etc.),
- instances of your own classes, defined in the active object class code.

► To define a scalar variable

1. Select the variable on the structure diagram.
2. Go to the *Variable type* section of the Properties window.
3. Select the *Scalar* option.
4. Specify the type of variable in the combo box on the right of the *Scalar* option.

4.1.1 Initializing a scalar variable

You can define an initial value for a variable. This value can be changed afterwards at runtime (see section 4.1.2, “Viewing and modifying scalar variables at runtime”).

► To initialize a scalar variable

1. Select the variable on the structure diagram.
2. Go to the *Equation* section of the Properties window.
3. Choose *No equation* from the *Form* drop-down list.
4. Specify the initial value of the variable in the *Initial value* edit box.

If an initial value is not specified, Java rules apply, for example a variable of type `double` is set to 0.

If a formula is defined for a variable, its initial value is calculated by this formula.

You cannot initialize input variable, since it commonly plays the role of dependent variable and its value depends on the value of connected output variable.

4.1.2 Viewing and modifying scalar variables at runtime

AnyLogic supports variable modifying during the model simulation. You can change variables at runtime programmatically as well as from the AnyLogic viewer UI or from AnyLogic animation. If a variable is connected with another variable, the change is propagated in the shown “direction” of dependency, see section 4.3, “Variable sharing”. You can define a handler method that is invoked on a variable change.

4.1.2.1 Modifying variables from Model Viewer

► To modify a variable

1. Double-click the variable in the Model Explorer or on the animated structure diagram, or
Right-click the variable in the Model Explorer or on the animated structure diagram and choose *Modify* from the popup menu.
The *Modify* dialog box is displayed.
2. Type a new value in the *Enter new value* edit box.

3. Click *OK*.

In the Model Viewer you can modify scalar variables only of primitive types: integer, real, boolean.

4.1.2.2 Inspecting variable

You can inspect the current value of a variable at the model runtime from the variable's inspect window.

► To open the inspect window

1. Right-click the variable in the Model Explorer or in the animated structure diagram and choose *Inspect* from the popup menu.
Variable's inspect window is displayed.

4.1.2.3 Modifying variables from AnyLogic animation

AnyLogic offers a set of controls for creating interactive animations. You can modify a variable by associating it with an animation control and changing control at runtime. See Chapter 12, "Animation" for information on AnyLogic animation.

You can associate:

- a variable of `boolean` type with a button or a check box
- a variable of `String` type with an edit box
- a variable of `double` type with a slider.

► To associate a variable with an animation control

1. Select the control on the animation diagram.
2. In the Properties window, choose the variable from the *Variable name* drop-down list.

3. If needed, specify the code to be executed when the user changes the control in the *Event handling code* edit box.

4.1.2.4 Accessing and modifying variables from code

You can treat a variable as a regular Java object and modify, test, and use it at discrete event steps in any expressions — in actions of states, transitions, ports, timers, from threads, etc. A variable `var1` of an active object can be accessed simply as the member variable `var1`.

For example, write the following code to modify and check `varA` variable value:

```
varA = 7;  
  
traceln("VarA = " + varA);
```

4.1.2.5 Collecting dataset on a variable

AnyLogic supports collecting data on a variable during the model simulation in a dataset. You can visualize and export data collected in a dataset (see Chapter 17, “Collecting data and performing statistical analysis”).

Samples of variables can be collected automatically from the beginning of the simulation.

► To collect dataset automatically

1. Select the variable on the structure diagram.
2. In the Properties window, select the *Auto collect dataset* check box.

If auto collecting data is not set, data collecting starts only when and if the variable is added to a chart window.

4.1.2.6 Displaying plots of variables in Viewer

You can visualize variable changes during model simulation using AnyLogic chart windows. You can visualize data collected for variables in either explicitly created or automatically collected by AnyLogic datasets. See Chapter 17, “Collecting data and performing statistical analysis” for details on collecting and analyzing data.

► **To display a plot of a variable in a chart window**

1. Right-click the variable in the Model Explorer and choose *Chart* from the popup menu.

You can open a blank chart window and add a variable later.

► **To open a blank chart window**

1. Click the *New Chart*  toolbar button, or Choose *View|New Chart* from the main menu.

Chart window looks as shown in Figure 32.

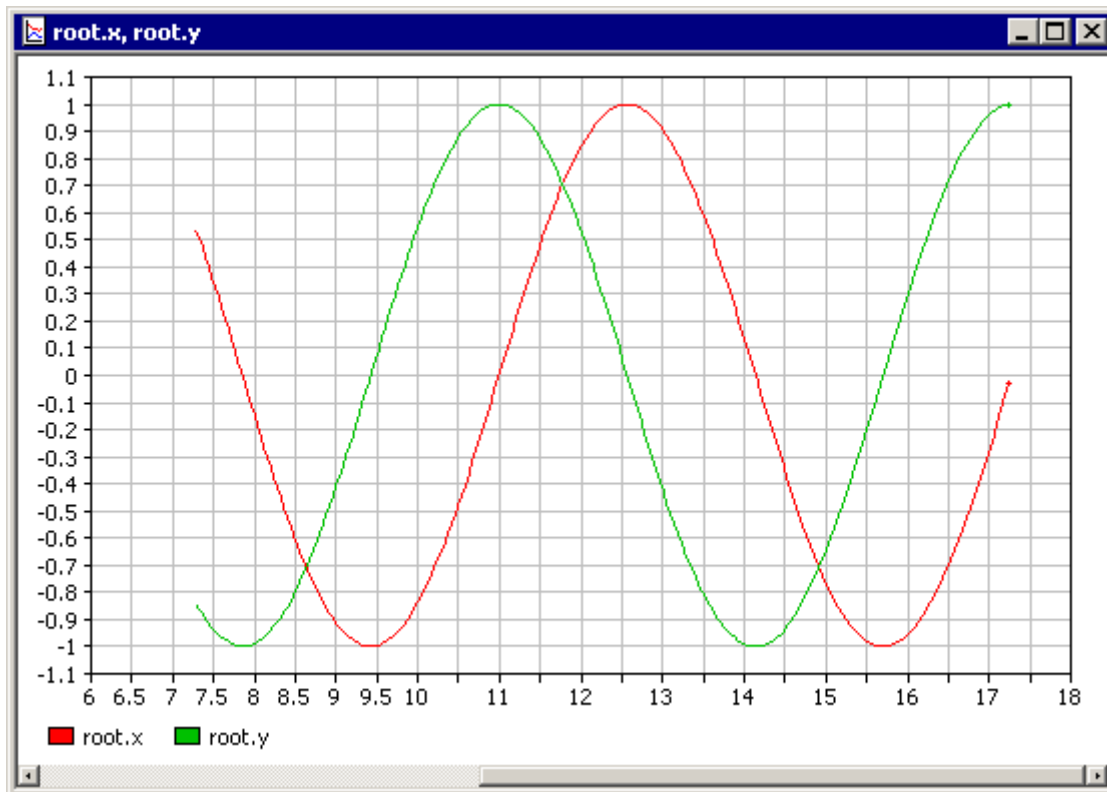



Figure 32. Chart window

Variables can be added to a chart window using drag and drop or using the *Chart Setup* dialog box.

► **To add a variable to a chart window**

1. Drag the variable from the Model Explorer onto the chart window.

► **To set up the chart**

1. Right-click the chart window and choose *Chart Setup...* from the popup menu. The *Chart Setup* dialog box is displayed (see Figure 33).
2. To add a variable to the chart, double-click the corresponding item in the *Variables, parameters and datasets* list.
3. To remove a variable from the chart, double-click the corresponding item in the *Axis Y* list.
4. By default, *Time* is chosen in the *Axis X* list, that is the chart is timed. If you need to plot one variable against another variable, dataset, or parameter, make the chart phased. Set up the variable/dataset/parameter to be displayed on the x-axis by clicking the corresponding item in the *Variables, parameters and datasets* list, and then clicking the  button to the left of the *Axis X* list. To make plot timed again, remove the variable/dataset/parameter item from the *Axis Y* list by double-clicking.
5. Click *OK*.

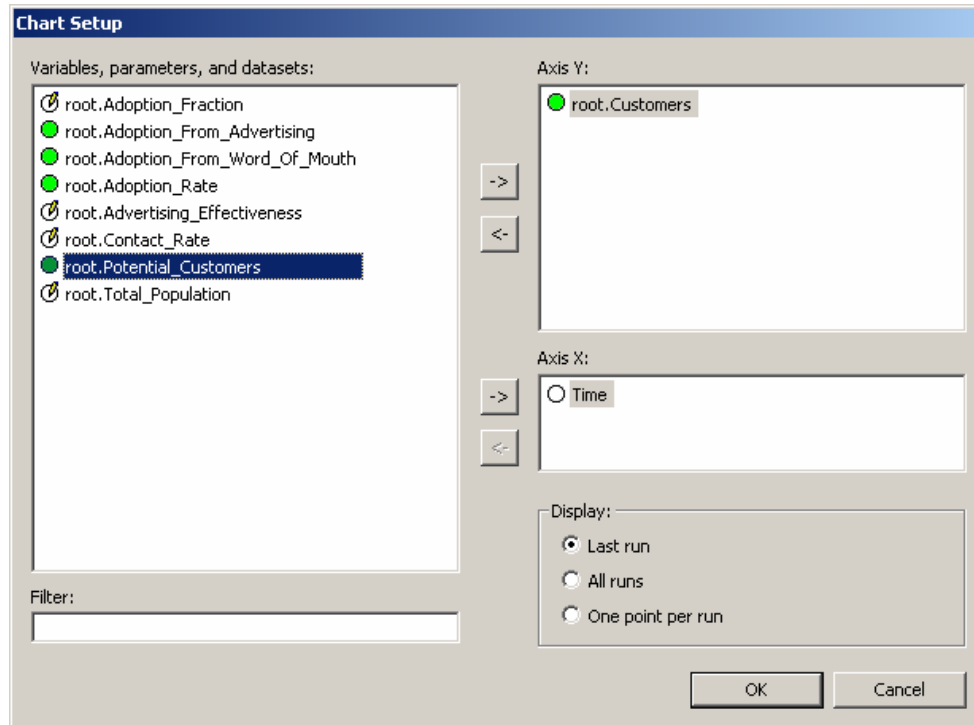



Figure 33. Chart Setup dialog box

4.1.2.7 Displaying plots of variables in animation

Variable plots can also be displayed in AnyLogic animation using a special control - chart indicator. Chart indicator displays a variable in one of the following forms: scatter, Gantt, pie chart, or bar chart.

► To create a chart indicator

1. Click the *Chart Indicator*  toolbar button, or Choose *Draw | Animation | Chart Indicator* from the main menu.
2. Click or drag the indicator area on the animation diagram.
3. In the Properties window, specify the variable you want to indicate in the *Value to indicate* combo box.

See section 12.2.3.3, “Chart indicator” for the detailed description of chart indicator.

4.2 Equations

You can define a set of equations to describe continuous changes of variables over time. Thus you can define continuous time object behavior. See Chapter 5, “Equations” to know how to define equations.

An equation can be of three types:

- Differential equation (see section 5.1.1.1, “Differential equations”)
- Algebraic equation (see section 5.1.1.2, “Algebraic equations”)
- Formula (see section 5.1.1.3, “Formulas”)

You can define equations for a set of variables, maybe across several active objects.

AnyLogic supports matrices and hyper-arrays as well as primitive types in equations, see Chapter 6, “Matrices and hyper-arrays”.

4.3 Variable sharing

Variables can be shared with other active objects. Shared variables will have the same value at any moment of time; i.e., changes of one variable will be immediately propagated to another variable. This provides for continuous and/or discrete time object interaction.

Only the variables of compatible types can be connected.

To establish variable sharing, you should connect the respective variables (see section 4.3.2, “Connecting variables”). You can connect:

- interface variables of two encapsulated objects;
- either state or interface variable of a container object with an interface variable of an encapsulated object.

Connect variable of encapsulated object with an interface variable of the container object if you need to share it with container object neighborhood. Otherwise, if you need to share it only with a container object, use state variable.

You cannot connect two variables of the same object.

4.3.1 Variable sharing rules

One of connected variables always acts as the dependent variable and its value depends on the value of another variable. The variable role in the connection is defined by the direction type of connected variables. As mentioned above, interface variables have the explicitly specified direction type: input and output. Output variable changes are propagated to the connected input variable.

The dependency direction of interface variable is specified in the variable's properties window.

► To change the dependency direction of interface variable

1. Select the variable on the structure diagram.
2. In the Properties window, specify the dependency direction in the *Direction* drop-down list.

Internal variable does not have the explicit dependency direction. The role of internal variable depends on the particular connection case.

AnyLogic permits only some well-defined configurations of variable connections to prevent collisions. The valid configuration cases are shown in Figure 34. The general rule is that for connecting variables of two encapsulated objects, one variable is always output and another one is input; while for connecting a variable of an encapsulated object with a variable of a container object, both variables should be of the same "direction."

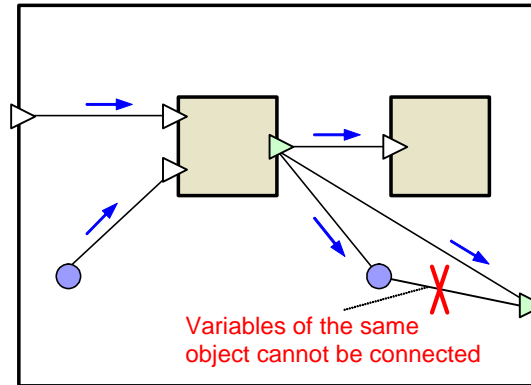


Figure 34. Connection of variables

Variable change is propagated according to the well-defined set of rules (in Figure 34 directions of variable propagation are shown with arrows).

- When variables of two encapsulated objects are connected, output variable change is propagated to input variable.
- When variables of encapsulated and container objects are connected, the direction of variable change propagation depends on the dependency direction type of variable of encapsulated object:
 - When it is the output variable, this variable change is propagated outside from the encapsulated object to the connected variable of the container object.
 - Otherwise, if it is the input variable, the change is propagated from the container to the encapsulated object.

AnyLogic allows connection of several dependent variables to one output variable. But to prevent collisions, only one output variable can be connected to a dependent variable; otherwise, AnyLogic raises a compilation error. Some possible cases of invalid connection configurations are shown in Figure 35.

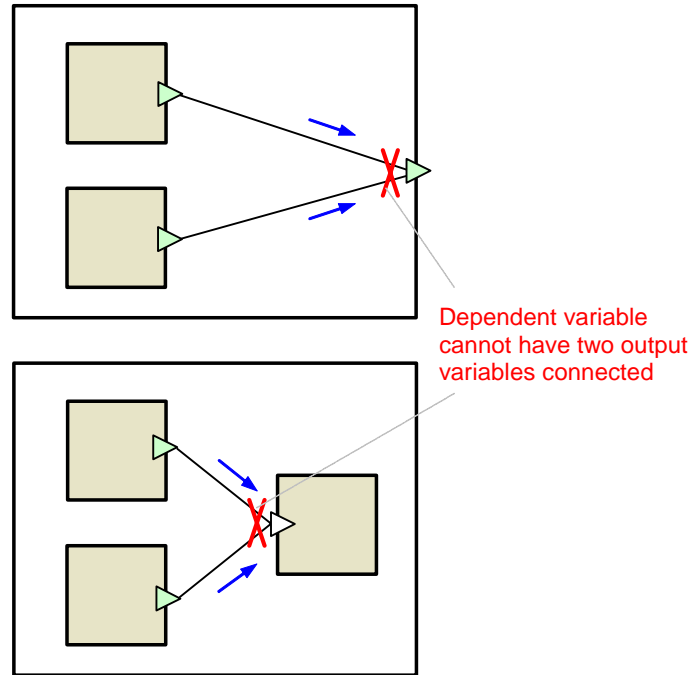


Figure 35. Invalid configuration case of variable connection

4.3.2 Connecting variables

You can connect variables either graphically with connectors (see section 1.5.8, “Active objects interaction” for information about connectors) or programmatically.


4.3.2.1 Connecting variables graphically

Variables can be connected graphically in the structure diagram.

Connecting variables of encapsulated objects

You can establish continuous time interaction between two encapsulated objects by connecting their interface variables. You can connect only an input variable with an output variable. The output variable value will be passed to the connected input variable.

► To connect variables of encapsulated objects

1. Drag the variable of one encapsulated object onto the variable of another encapsulated object, or
Click the *Connector*  toolbar button, click the first variable and then click the second variable, or
Choose *Draw | Structure | Connector* from the main menu, click the first variable and then click the second variable.
The connector linking two variables appears (see Figure 36).

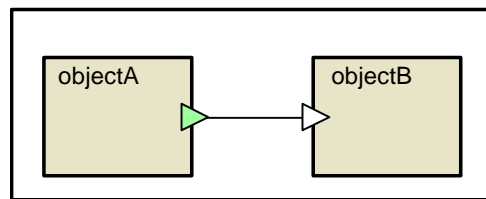



Figure 36. Variables of encapsulated objects connected

Connecting variable of an encapsulated object with a variable of a container object

To establish continuous time interaction between the encapsulated object and the container active object you should connect a variable of an encapsulated object with a variable of the parent object.

Connect a variable of an encapsulated object with an interface variable if you want to establish interaction with parent object neighborhood. However, if you need only to propagate value of the variable of container object and encapsulated object, use a state variable.

► To connect a variable of an encapsulated object with a variable of a container object

1. Drag the variable of the encapsulated object onto the variable of the container object, or
Click the *Connector*  toolbar button, click the first variable and then click the second variable, or
Choose *Draw | Structure | Connector* from the main menu, click the first variable and then click the second variable.
The connector linking two variables appears (see Figure 37).

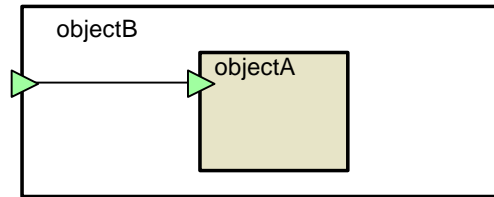


Figure 37. Variables of container and encapsulated objects connected

If connector is shown red on the structure diagram, it means that you have established an invalid connection (see section 4.3.1, “Variable sharing rules” to check the connection rules).

Exporting variables to the parent active object

A variable of an encapsulated object can be exported to the parent object class. It means that the variable is added to the parent object class and connected to the exported variable of the encapsulated object (see Figure 37). Thus the encapsulated object and the parent object can interact with each other by propagating variable changes.

► To export a variable of an encapsulated object to the parent active object

1. Right-click the variable on the structure diagram and choose *Export to Parent* from the popup menu.

4.3.2.2 Connecting variables at runtime

Variables can be connected and disconnected at runtime using the methods of the `VariableRef` class. Call the methods `connect()` and `map()` to connect variables and the methods `disconnect()` and `unmap()` to disconnect them. For more information on the `VariableRef` class refer to AnyLogic Class Reference.

The methods take the connected variable object as argument. The name of a variable object is the name of the variable in the structure diagram with the `_ref_` prefix.

Use the methods `connect()/disconnect()` to connect or disconnect variables, located on the same level of containment hierarchy and the `map()/unmap()` methods otherwise.

In other words, use the `connect()`/`disconnect()` methods to connect or disconnect:

- Variables of encapsulated objects

Use the `map()`/`unmap()` methods to connect or disconnect:

- A variable of an encapsulated object with a variable of a container object

4.3.2.3 Using the methods `connect()/disconnect()`

Figure 38 shows the situation when the `connect()`/`disconnect()` methods are used.

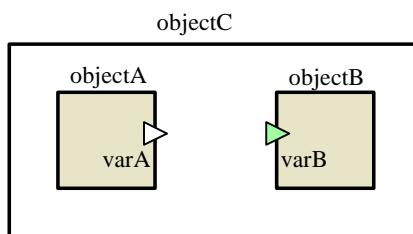


Figure 38. Variables of encapsulated objects

Call the `connect()` method to connect `varA` with `varB`:

```
objectA._ref_varA.connect(objectB._ref_varB);
```

Call the `disconnect()` method to disconnect variables in the similar manner:

```
_ref_varA.disconnect(_ref_varB);
```

You can write this code anywhere you like in the parent active object class (`objectC` in the first case and `objectA` in the second one) code.

4.3.2.4 Using the methods `map()/unmap()`

The method `map()` is used for connecting variables located on the different levels of the containment hierarchy, namely a variable of an encapsulated object with a variable of a container object. Variables are disconnected by calling the `unmap()` method.

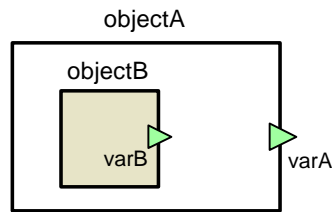


Figure 39. An interface variable and a variable of an encapsulated object

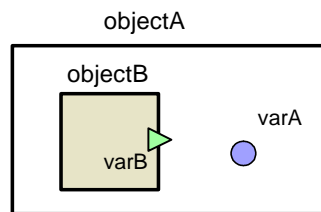


Figure 40. A state variable and a variable of an encapsulated object

Figure 39 and Figure 40 show the situations when the methods `map()`/`unmap()` are used.

To connect these variables, call the method `map()` of one variable with another variable, specified as a parameter anywhere you like in the parent active object class (`objectA`) code:

```
_ref_varA.map(objectB._ref_varB);
```

To disconnect these variables, call the method `unmap()`:

```
_ref_varA.unmap(objectB._ref_varB);
```

AnyLogic also provides you the ability to programmatically disconnect a variable from all connected variables and to check whether the input variable is connected to the output one. The related methods of the class `VariableRef` are listed below. Refer to AnyLogic Class Reference for details.

Related methods of `VariableRef`

`void breakLinks()` – the method disconnects a variable from all connected variables.

`boolean hasInputConnection()` – the method returns `true` if the input variable has an incoming connection and `false` otherwise.

4.4 Changing variables and reacting to their changes

You can also test variables continuously by specifying a Boolean expression as a trigger of a transition. As soon as the expression evaluates to `true` (even if this happens in the middle of a time step), the transition becomes enabled.

We recommend you to specify awaited conditions as triggers and not as guards. This is because guards are not tested during time steps.

In the example in Figure 41, the transition in object `myObj2` fires as soon as the object `myObj1` changes the value of `x`.

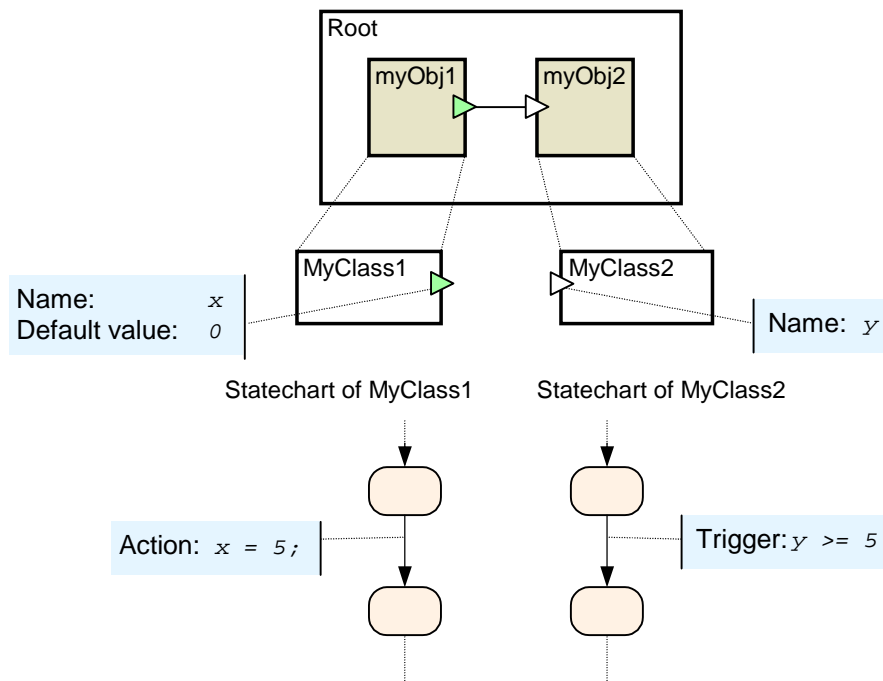


Figure 41. Responding to variable change (I)

In the example above both objects are discrete. In the example below, continuous behavior of one object affects discrete behavior of another. The transition in myObj2 fires when the value of x in myObj1 exceeds 5.

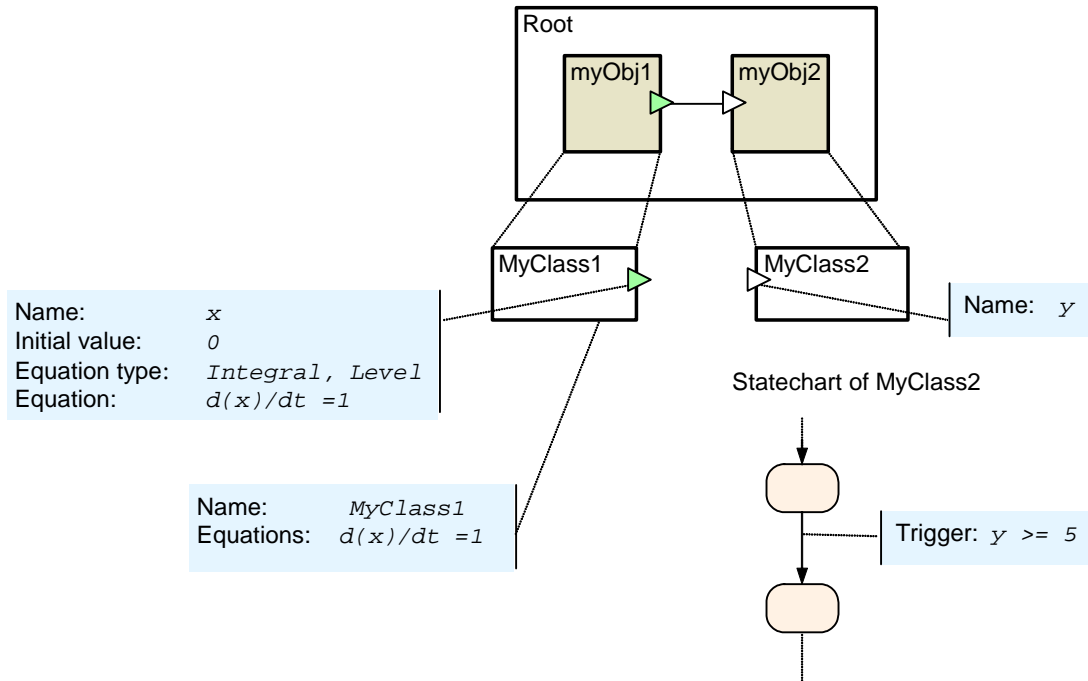


Figure 42. Responding to variable change (II)

In case a variable is used in a set of differential and algebraic equations, the connection of variables means they are treated as single variable by the equation solver. Let us consider the example. Suppose we have objects connected as shown in Figure 43, and the formulas $x = -y$ and $v = w - 2$ are currently active in objects myObj1 and myObj2 respectively.

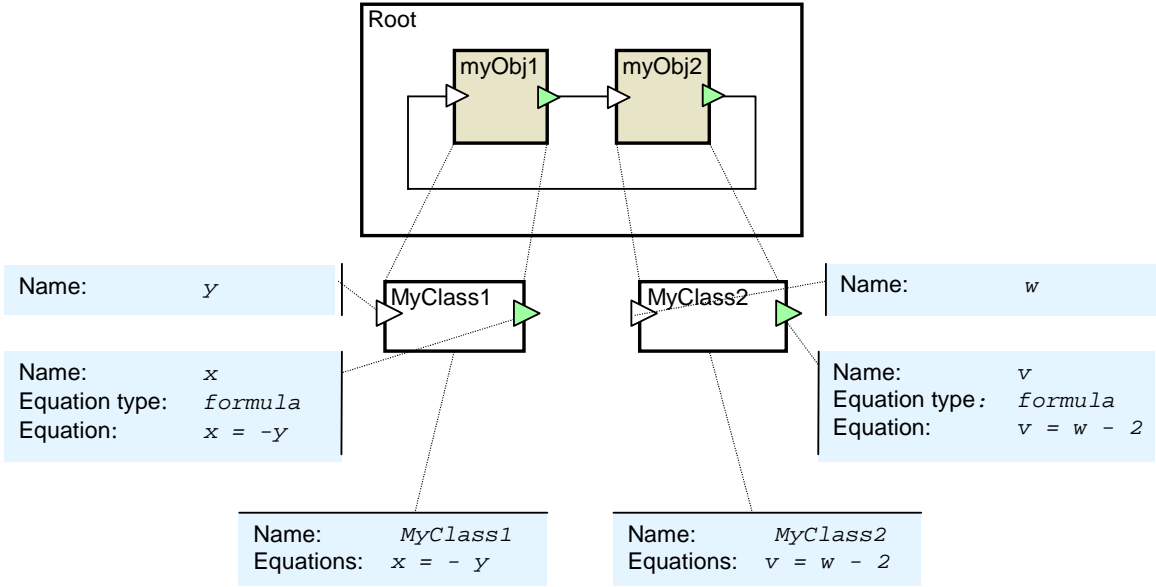


Figure 43. Equations over variables

This leads to the following set of algebraic equations:

$$\begin{cases} x = -y \\ v = w - 2 \\ v = y \\ w = x \end{cases}$$

and we can easily solve it:

$$\begin{cases} x = -y \\ v = w - 2 \\ v = y \\ w = x \end{cases} \Rightarrow \begin{cases} x = -y \\ y = x - 2 \\ v = y \\ w = x \end{cases} \Rightarrow \begin{cases} x = -x + 2 \\ y = x - 2 \\ v = y \\ w = x \end{cases} \Rightarrow \begin{cases} x = 1 \\ y = -1 \\ v = y \\ w = x \end{cases}$$

The values of the variables are set exactly to these values irrespective of their initial values.

5. Equations

You can define a set of differential equations, algebraic equations, and formulas to describe continuous changes of variables over time. You may associate equations (hereinafter we use the term equations for both equations and formulas) with:

- An active object;
- A particular state of a statechart.

At any given moment of time there exists the global active set of equations in a model. This set is constructed of all currently active equations of all objects and all statechart states. Equations associated with an active object are active during the whole object lifetime. Equations associated with a statechart state are active while the statechart is at that state. A statechart contributes to the global set not only the equations of the current simple state, but also of all its container states.

AnyLogic supports matrices and hyper-arrays as well as primitive types in equations, see Chapter 6, “Matrices and hyper-arrays”.

5.1 Equation types

An equation can be of three types:

- Differential equation (see section 5.1.1.1, “Differential equations”)
- Algebraic equation (see section 5.1.1.2, “Algebraic equations”)
- Formula (see section 5.1.1.3, “Formulas”)

5.1.1.1 Differential equations

Differential equation  defined in the form

$$d(x)/dt = F(x, y, t, \dots)$$

where x is an output or state variable of type `double` or of matrix type, and $F(x, y, t, \dots)$ is a well-formed arithmetic expression which may contain any variables of the active object and the special symbol t denoting time. The expression $F(x, y, t, \dots)$ can contain any arithmetic operations and method calls, such as, e.g. `sin()`, `cos()`, `sqrt()`, etc., or calls to your own methods.

$F(x, y, t, \dots)$ should not call those methods that indirectly use variables of the active object; the methods should take variables as parameters. The best way to ensure this is to use static methods only.

$F(x, y, t, \dots)$ should never call those methods that change variables of the active object.

$F(x, y, t, \dots)$ should not contain conditional operators e.g. `b>0 ? c:d`.

You can define any number of differential equations. It is an error if a variable occurs more than once in the left-hand side in the set of equations. An input variable is not allowed to appear in the left-hand side.

Example

In the example shown in Figure 44, the behavior of the variable x is defined by the differential equation $d(x)/dt=t$.

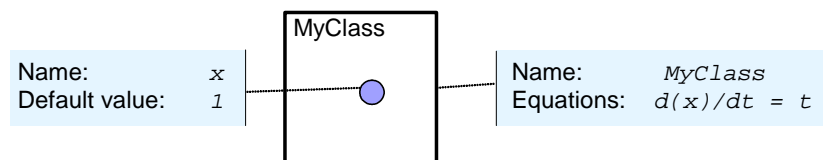


Figure 44. Differential equations (I)

If you solve this equation analytically, you get $x(t)=t^2/2 + C$, which, if you take into account the initial value $x(0)=1$, gives $x(t)=t^2/2+1$. This is a parabola. Figure 45 shows this parabola calculated and drawn by AnyLogic.

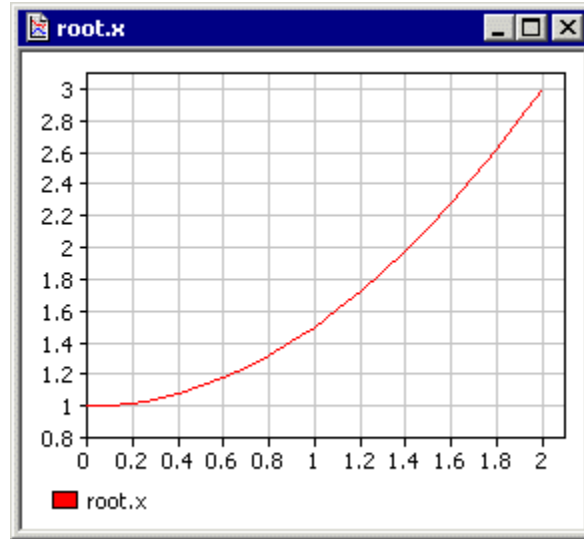


Figure 45. Parabola drawn by AnyLogic

Example

The differential equations presented in Figure 46 define a mathematical pendulum; x is the pendulum coordinate, y is its velocity. The solution calculated by AnyLogic is shown in Figure 47.

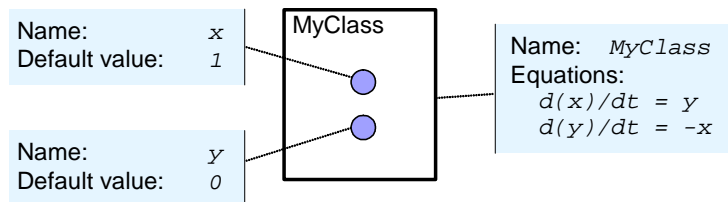


Figure 46. Differential equations (II)

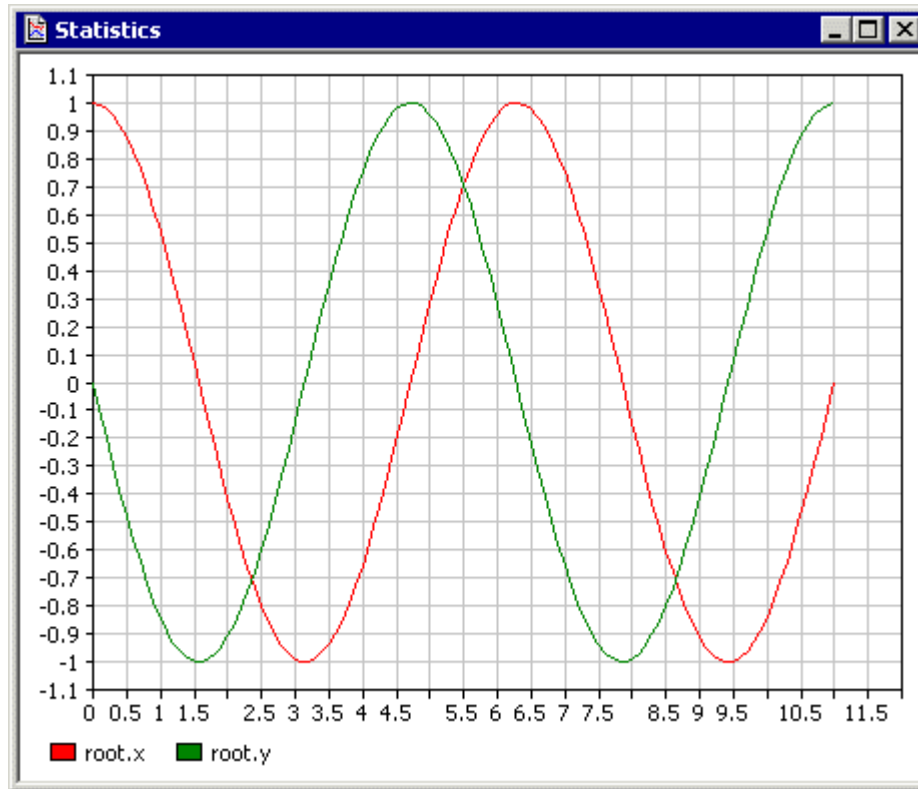


Figure 47. Coordinate and velocity of the mathematical pendulum

5.1.1.2 Algebraic equations

Algebraic equations are defined in the form

$$x = F(x, y, t, \dots)$$

where x is an output or state variable of type `double` or of a matrix type, and $F(x, y, t, \dots)$ is a well-formed arithmetic expression. For the rules that apply to $F(x, y, t, \dots)$ see section 5.1.1.1, “Differential equations”.

Also, you specify the set of unknown variables, assuring that the number of unknown variables equals the number of algebraic equations:

$$\text{find}(x, y, \dots)$$

You can define any number of algebraic equations. You cannot specify an input variable as an unknown one.

Example

Figure 48 shows an example of a set of two equations with two unknown variables.

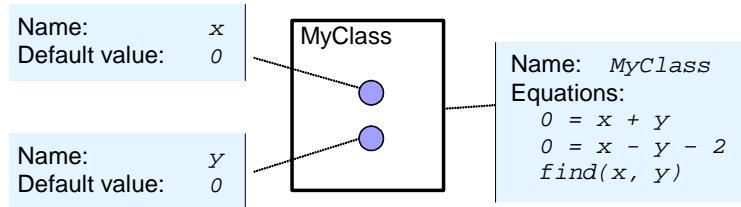


Figure 48. Algebraic equations

Obviously, the solution is: $x = 1, y = -1$.

5.1.1.3 Formulas

Formulas are defined in the form

$$x = F(y, t, \dots)$$

where x is output or state variable of type `double` or of vector/matrix type, and $F(y, t, \dots)$ is a well-formed arithmetic expression. For the rules that apply to $F(y, t, \dots)$ see section 5.1.1.1, “Differential equations”. The only additional restriction is that $F(y, t, \dots)$ cannot not contain x because then it would be an algebraic equation, which must be defined using another form.

You can define any number of formulas. It is an error if a variable occurs more than one time in the left-hand side in the set of equations. An input variable cannot appear in the left-hand side.

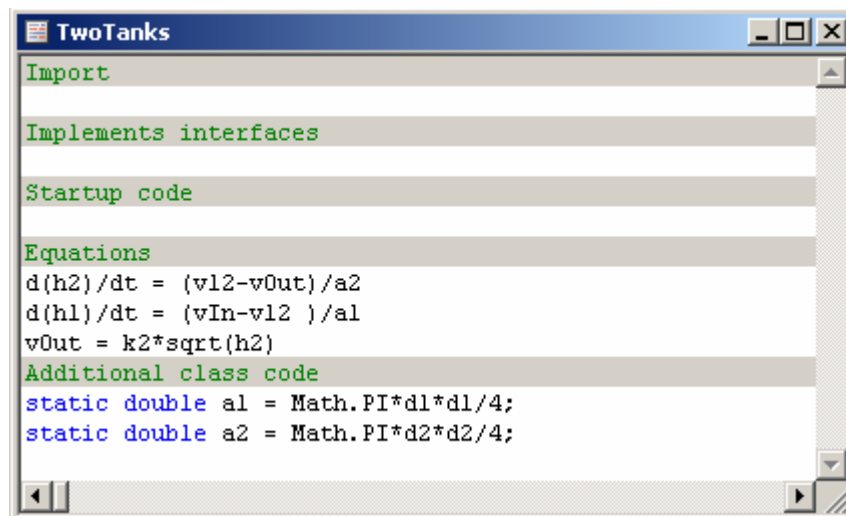
A formula defines a direct dependency between variables. It is a special case of algebraic equation, but it does not need a time-consuming numerical solution. Therefore it is recommended to use formulas whenever possible to increase simulation performance.

5.2 Defining an equation

You specify an equation as a string. For convenience you can specify equations in the *Equations* code section of an active object class or in the *Equation* section of a variable. Those two places are self-synchronized; that is, if you write or remove an equation in one place, the equation is added or removed in the other place automatically. In the *Equations* code section of an active object class, you can specify all kinds of equations. However, in variable's properties, you can define only differential equations and formulas for this particular variable.

► To define an equation in active object class code

1. In the Project window, right-click the *Code* item of the active object class and choose *Open Code* from the popup menu, or
Double-click the *Code* item of the active object class.
The Code window of the active object is displayed, see Figure 49.
2. Type the equation in the *Equations* section.



```

TwoTanks
Import
Implements interfaces
Startup code
Equations
d(h2)/dt = (v12-v0ut)/a2
d(h1)/dt = (vIn-v12)/a1
v0ut = k2*sqrt(h2)
Additional class code
static double a1 = Math.PI*d1*d1/4;
static double a2 = Math.PI*d2*d2/4;

```

Figure 49. Code window of the active object class

► To define a differential equation for a variable

1. Select the variable on the structure diagram.


2. In the *Equation* section of the Properties window, choose *Integral or Stock* from the *Form* drop-down list.
3. Type the right hand of the equation in the $d(\text{variable})/dt =$ edit box.
4. Type the variable initial value in the *Initial value* edit box.

► **To define a formula for a variable**

1. Select the variable on the structure diagram.
2. In the *Equation* section of the Properties window, choose *Formula* from the *Form* drop-down list.
3. Type the formula for a variable in the *variable =* edit box.

AnyLogic enables you to visualize the resulting dependencies between variables in your model with arrows (see Figure 50). An arrow pointing from variable A to variable B means that variable A is mentioned in the equation of variable B.

► **To show/hide the dependencies between variables**

1. Click the *Show/Hide Variable Dependencies*  toolbar button, or Choose *Draw | Variable Dependencies* from the main menu.
If dependencies are shown, the toolbar button looks blue.

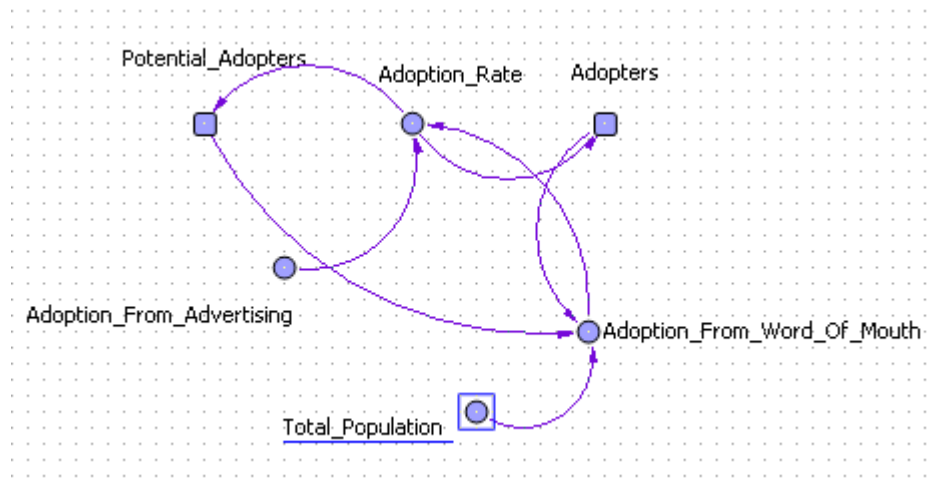


Figure 50. Dependencies displayed on the structure diagram

You may associate equations with a particular state of a statechart (composite as well as simple). You define equations for the statechart state into the property *Equations* of the state.

► To define equations for a statechart state

1. Click the state on the statechart diagram.
2. In the Properties window, type equations in the *Equations* section.

5.3 Functions

AnyLogic enables using custom functions in equations, namely:

- AnyLogic provides a set of *predefined functions* – the most frequently used mathematic functions.
- AnyLogic supports a special type of function – a *lookup table*. A lookup table is a continuous function defined in the table form. You may need it to define a complex non-linear relationship which cannot be described as a composition of standard functions, or to bring data defined as a table function to a continuous mode.
- AnyLogic enables you to define custom functions. This is frequently needed when there is a standard composition of functions used in multiple equations. In this case

you can simply define a function once and reuse it. You can define a *mathematical function* if you need to define a mathematical expression or an *algorithmic function* (a function written in Java), if you need to code some calculation algorithm.

This section describes using all these types of functions in equations.

5.3.1 Predefined functions


AnyLogic provides a set of *predefined functions* – the most frequently used mathematic functions. You can use any composition of the predefined functions in a right-hand side of an equation and in initial  values.

Table 2 lists the predefined functions.

Predefined functions	Description
<code>sin, cos, tan</code>	Trigonometrical functions.
<code>asin, acos, atan, atan2</code>	Inverse trigonometrical functions.
<code>pow, sqrt</code>	Power and square root functions.
<code>exp</code>	Exponent function.
<code>log</code>	Natural logarithm.
<code>abs</code>	Absolute value.
<code>min, max</code>	Minimal (maximal) value of the two arguments.
<code>round, rint, floor, ceil</code>	Real-to-integer conversions.
<code>random</code>	Random number uniformly distributed in the interval [0,1).
<code>delay</code>	Time shift.
<code>xidz, zidz</code>	Division functions.

Table 2. Predefined functions

There are two useful constants, which you can use in equations:

Constant	Description
<code>Math.E</code>	The e number – the base of the natural logarithms and exponent function.
<code>Math.PI</code>	The π number – the ratio of the circumference of a circle to its diameter.

Table 3. Constants

The constants and the functions except `delay`, `xidz` and `zidz` are defined in the `java.lang.Math` class. So you may use accustomed `Math.<function name>()` notation as well. Consult Java documentation available at <http://java.sun.com/docs>.

Predefined functions are added using AnyLogic function wizard (see section 5.3.2, “Using intelli-sense”).

5.3.1.1 Simulation time

You can refer to the current simulation time in equations. Get the current simulation time value by typing the symbol ‘`t`’ or calling the function `getTime()`.

5.3.1.2 Time shift

The `delay` function implements the time shifting.

There is a simple example. Variable `y` should be a time function `x` shifted by two model time units. For instance, let `x` represents an exponent decay process. We specify the formulas for the variables in the following way:

```
x = exp(-t)
y = delay(x, 2.0)
```

The variable `y` follows `x` with two time units delay.

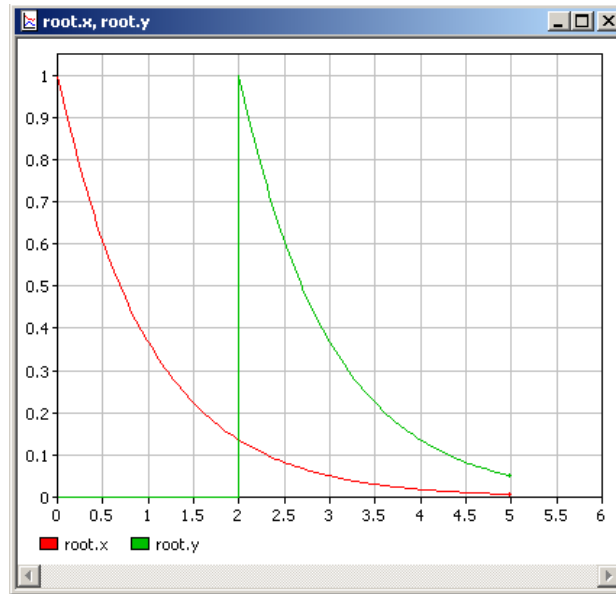


Figure 51. Example of a time shift

The call of the function is:

```
delay ( <variable>, <time shift value>, <initial value> )
```


The first argument should be an AnyLogic variable, but not an expression. The second one could be either a constant or a numeric expression (e.g. function call). Thus, the time shift value can change during the simulation. The delay function with zero or negative time shift returns the variable value without shifting.

5.3.2 Using intelli-sense

AnyLogic supports intelli-sense mechanism. This significantly simplifies typing equations since you do not need to type the whole names of functions, variables and parameters. You can use the intelli-sense wizard to insert a variable name or a call of predefined function.

The wizard looks as a list, containing variables (V icon), parameters (P icon), and functions (f icon) ordered alphabetically (Figure 52). You can simply select the name in the list, and it will be inserted in the expression automatically.

► **To insert an object name into an equation using the intelli-sense wizard**

1. Move cursor at the position in the *Equation* edit box where you want to place the object name.
2. Click the  button or press Ctrl+space.
The wizard listing all model variables and predefined functions appears.

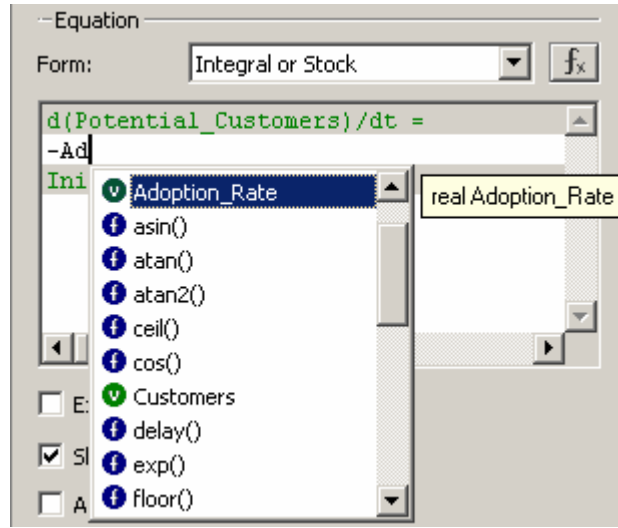


Figure 52. Intelli-sense function wizard

3. Scroll to the name you want to add, or type the first letters of the name until it becomes visible in the list.
4. Select the name by clicking. The wizard displays the detailed description of the selected object in the popup text box.
5. Double-click the name to insert it into the equation expression.


5.3.3 Lookup tables

Sometimes you may need to define a complex non-linear relationship, which cannot be described as a composition of standard functions. Or you may need to bring experimental data defined as a table function to a continuous mode. That needs the function being interpolated somehow.

AnyLogic supports special type of functions - lookup tables. A lookup table is a function defined in the table form. You can simply make it continuous by interpolating and/or extrapolating.

You can call lookup table function in equations as any other function (see section 5.3.3.3, “Accessing lookup values” for details).

► **To create a new lookup table**

1. Click the *New Lookup Table*  toolbar button, or Choose *Insert | New Lookup Table...* from the main menu.
The *New Lookup Table* dialog box is displayed.
Specify the name of the new lookup table, choose the active object class, which will contain the lookup table, and click *OK*.
2. Alternatively, in the Project window, right-click the active object class, which will contain the lookup table, and choose *New Lookup Table...* from the popup menu.
The *New Lookup Table* dialog box is displayed.
Specify the name of the new lookup table and click *OK*.

► **To define data for a lookup table**

1. Click the *Show Scatter...* button on the *General* page of the Properties window.
The *Lookup Table* dialog box is displayed (Figure 53).
2. Enter the points into the *Data* grid, namely:
Type the argument value in the *Argument* cell,
Type the function value in the *Function* cell.
3. Choose interpolation type in the *Interpolation/Approximation* section.
4. Select the required reaction on out-of-range argument values.

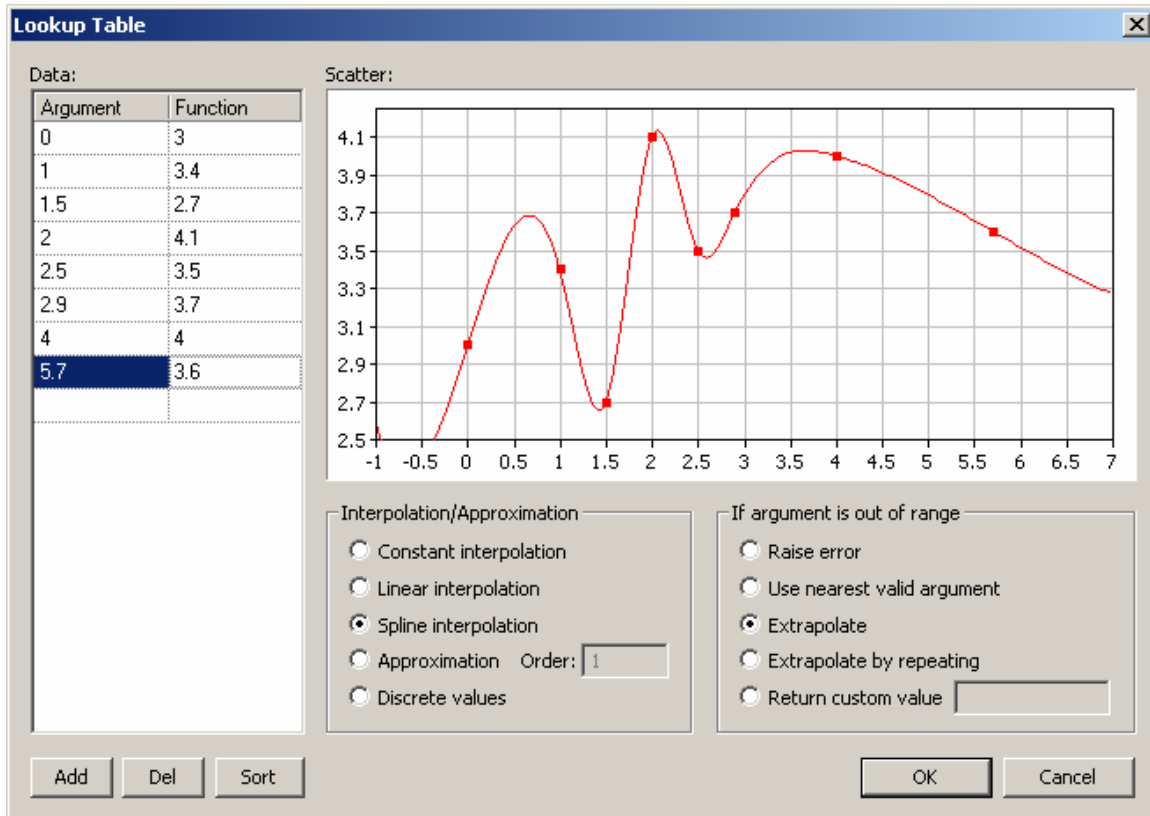


Figure 53. Lookup Table dialog box

The *Lookup Table* dialog box has the following controls:

Scatter – the graph displays the resulting function.

Add – the button adds new data point with the last argument value incremented.

Sort – the button sorts data points in the lookup table. In fact, points are automatically sorted by the argument value in ascending order.

Del – the button deletes the point selected in the table.

Interpolation/Approximation – specifies the interpolation type for a lookup table function (see section 5.3.3.1, “Function interpolation”).

If argument is out of range – specifies how the function behaves when the argument is out of range (see section 5.3.3.2, “Function behavior in infeasible area”).

You can access the table’s data at the *General* page of the lookup table’s Properties window too.

5.3.3.1 Function interpolation

Lookup table function can be interpolated. The possible interpolation types are listed in Table 4.

Interpolation type	Description
Constant interpolation	The function value between two points is the same as in the point with less argument value.
Linear interpolation	The points are connected with straight-line segments.
Spline interpolation	4 th order spline. The points are connected with 4 th order polynomial segments. For each point 0 th , 1 st , and 2 nd derivatives of right and left segments are equal. The 2 nd derivative in the ending points equals zero.
Approximation	The resulting function is a polynomial of <i>Order</i> you specify in the edit-box on the right, formed in order the sum of the root-mean-square error in the points is minimal.
Discrete values	No interpolation applied.

Table 4. Interpolation types

If you want to get a smooth curve, the spline interpolation is the best. However, it takes more time to calculate a spline interpolation than a linear one. So, if a discontinuous function is acceptable, use the linear interpolation.

5.3.3.2 Function behavior in infeasible area

The feasible area of a lookup table function is the function's range, if the function is interpolated; or it is the defined set of points only, if no interpolation is set. You should define what should happen if a lookup argument lies out of feasible area. Therefore, in the *If argument is out of range* section of the *Lookup Table* dialog box, choose one of the options listed in Table 5.

Function behavior	Description
Raise error	If argument lies out of feasible area, runtime error is raised and a message box reporting about the error is displayed.
Use nearest valid argument	For all arguments to the left (right) of the range, the function takes the value the function has in the leftmost (rightmost) point.
Extrapolate	The function is extrapolated outside the range in accordance to the interpolation type.
Extrapolate by repeating	The function is made periodic with the function range as a period.
Return custom value	If argument lies out of feasible area, lookup returns a custom value, defined in the edit box on the right of the option.

Table 5. Function behavior in infeasible area

A spline is extrapolated with linear functions. At the left (right) side, the extrapolation function is a ray which starts in the leftmost (rightmost) point and has the same first derivative as the spline has in that point. A function with linear interpolation is extrapolated with the rays which continue the outmost linear segments.

5.3.3.3 Accessing lookup values

To access lookup value with the specified argument use one of the related methods of the class `LookupTable` (access lookup simply by its name). Refer to AnyLogic Class Reference for details.

Related methods of `LookupTable`

`double lookup(double x)` – returns the lookup value corresponding to the argument passed as a parameter.

`double getDouble(double x)` – returns the lookup value corresponding to the argument passed as a parameter.

`double getDouble(Object x)` – returns the lookup value corresponding to the argument passed as a parameter.


`Object getObject(double x)` – returns the lookup value corresponding to the argument passed as a parameter.

`Object getObject(Object x)` – returns the lookup value corresponding to the argument passed as a parameter.

5.3.4 Mathematical functions

Mathematical functions are useful if there is a standard composition of functions which you use in multiple equations and you want to define it once and reuse.

► To define a mathematical function

1. Click the *New Mathematical Function*  toolbar button, or Choose *Insert | New Mathematical Function...* from the main menu. The *New Mathematical Function* dialog box is displayed.



Specify the name of the new function, choose the active object class, which will contain the function, and click *OK*.

2. Alternatively, in the Project window, right-click the active object class, which will contain the function, and choose *New Mathematical Function...* from the popup menu. The *New Mathematical Function* dialog box is displayed. Specify the name of the new function and click *OK*.

A mathematical function has the following properties:

Properties

Name – name of the mathematical function.

Function type - type of the function return value.

Arguments – a set of arguments of the function. Every argument should be declared in form: *Type Name*, where *Type* is the type of the argument, *Name* is the name of the argument. Function arguments may appear in the expression.

Expression – the function expression.

Function wizard – the button opens the function wizard.


Static – if set, the function is static.

Exclude from build – if set, the function is excluded from the model.

Rules that apply to the mathematical function expression are similar to the rules for differential equation expression (see section 5.1.1.1, “Differential equations”). You can use model time symbol, predefined functions, lookups, and non-global parameters. Note that you cannot use variables there.

The usage of variables in mathematical functions may cause an error. If you hide reference to B in a custom mathematical function, and call this function in the equation of A, AnyLogic cannot discern the dependency. In this case, there is no guarantee that the equations will be solved properly.

If you want to refer to variable B in a function, create a new function argument and pass variable B as the argument when you call the function in an equation of A.

Entering the expression, you can use the function wizard. The function wizard displays the list of function arguments ( icon), predefined functions and lookups. You can simply select the object name in the list, and it will be inserted in expression automatically. See section 5.3.2, “Using intelli-sense” to know how to work with function wizard.

► To open function wizard


1. In the Project window, click the function item.
2. In the Properties window, click the place in the *Expression*, where you want to place an object name.
3. Click the *Function wizard* button, or Press Ctrl+space.

5.3.5 Algorithmic functions

The purpose of an *algorithmic function* is similar to mathematical, but it is more powerful and allows you to implement functions that are more than a single mathematical expression but are an algorithm of calculations. You write algorithmic functions in Java, so you have all the

advantages of this language, such as conditional execution (if-then-else), cyclic execution (while, for), branches (switch) and more. An algorithmic function is commonly used when you cannot build your function as a composition of predefined functions or with a lookup table.

► To define an algorithmic function

1. Click the *New Algorithmic Function*  toolbar button, or Choose *Insert | New Algorithmic Function...* from the main menu. The *New Algorithmic Function* dialog box is displayed. Specify the name of the new function, choose the active object class, which will contain the function, and click *OK*.
2. Alternatively, in the Project window, right-click the active object class, which will contain the function, and choose *New Algorithmic Function...* from the popup menu. The *New Algorithmic Function* dialog box is displayed. Specify the name of the new function and click *OK*.

An algorithmic function has the following properties:

Properties

Name – name of the algorithmic function.


Function type - type of the function return value.

Arguments – a set of arguments of the function. Every argument should be declared in form: *Type Name*, where *Type* is the type of the argument, *Name* is the name of the argument. Function arguments may appear in the function body.

Function body – body of the function.

Function wizard – the button opens the function wizard.

Exclude from build – if set, the function is excluded from the model.

Function body is not the same as the equation expressions. Here you cannot use operators on hyper-arrays or model time symbol. Entering the function body, you can use the function wizard, where among the arguments and predefined functions some very useful Java operators are provided ( icon).

Operator	Description
<code>for(){ },</code> <code>while(){ }</code>	Cyclic operators.
<code>if(){ },</code> <code>if(){ }else{ }</code>	Conditional operators.
<code>println()</code>	Print operator.

Table 6. Java language operators

5.4 Algebraic loops

Sometimes formulas form an algebraic loop. In the example shown in Figure 54 the formula in object `myObj1` defines dependency of `x` on `y`, and the formula in the object `myObj2` – dependency of `y` from `x`.

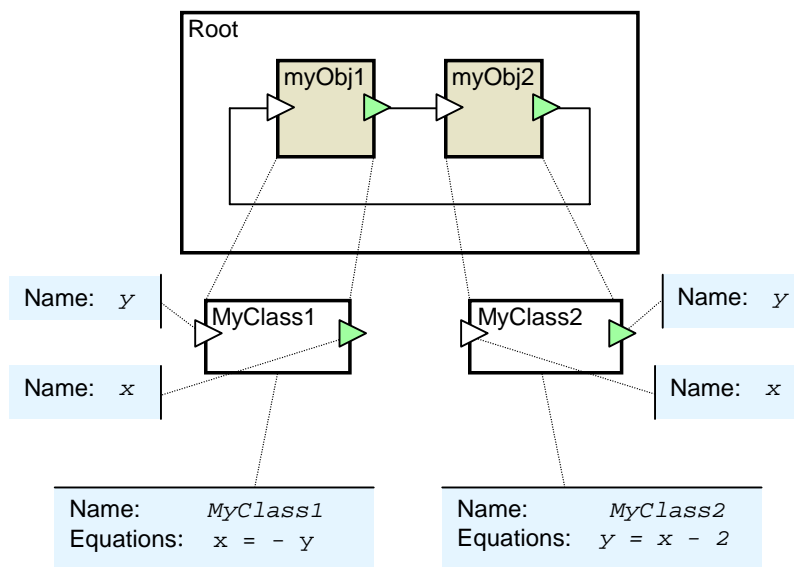


Figure 54. Algebraic loop

Clearly, these two formulas form a set of two algebraic equations with two unknown variables. The user, however, should not care about this. AnyLogic automatically detects

algebraic loops and substitutes formulas with necessary equations. In our example it looks like:

$$\begin{cases} 0 = x + y \\ 0 = x - y - 2 \end{cases}$$

The solution is: $x = 1, y = -1$.

An algebraic loop also happens when initial values of two variables refer to each other. For example, when `var1` initializes `var2`, `var2` initializes `var3`, and `var3` initializes `var1`. AnyLogic substitutes these expressions into an algebraic equation system, solves this system numerically, and initializes the variables with the obtained solution.

5.5 Runtime errors caused by equations

During time steps (in between discrete event steps), the model engine numerically solves the global set of equations. This process may cause errors of three types:

- A computational error in an expression specified by the user, e.g. division by zero or square root of a negative value.
- Incorrectly constructed set of equations, e.g. a variable appears more than once on the left-hand side.
- The set of equations cannot be solved with the given initial values.

In case of error of the third type, you should analyze the set of algebraic equations to find out if it has a solution. In case it has, it is recommended to give a good initial approximation to the solution. You should set up the initial approximation just before the set of equations is activated; e.g., in the entry action of the state containing equations.

5.6 Numerical methods

When a model starts, the equations are assembled into the main differential equation system. During the simulation, this DES is solved by one of the numerical methods built in AnyLogic. AnyLogic provides the rich collection of numerical methods for solving ordinal

differential equations (ODE), algebraic-differential equations (DAE), or algebraic equations (NAE).

By default, the numerical solver is chosen automatically by AnyLogic at runtime. You can set up numerical methods used for solving equations in the *Numerical methods* section on the *Additional* page of the experiment's properties window (see Figure 55).

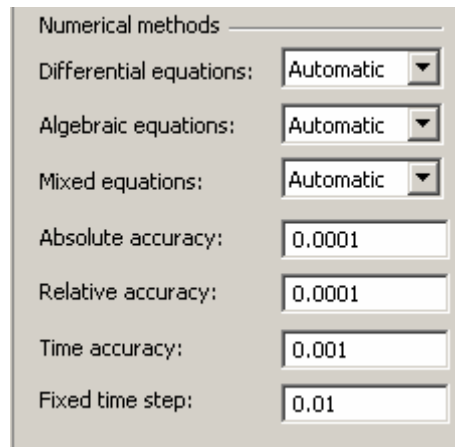


Figure 55. Additional page of experiment's properties window. Numerical methods section

Differential equations – preferred method used to solve ordinary differential equations.

By default, *automatic* method is selected, which chooses the fastest possible method for the current equation set.

Algebraic equations – preferred method used to solve algebraic equations. By default, *automatic* method is selected, which chooses the fastest possible method for the current equation set.

Mixed equations – preferred method used to solve algebraic-differential equations. By default, *automatic* method is selected, which chooses the fastest possible method for the current equation set.

Absolute accuracy – the desired absolute value accuracy for solving equations. Absolute accuracy is used when it is impossible to use relative accuracy – e.g., when the value is close to zero.

Relative accuracy – the desired relative value accuracy for solving equations with methods that change the integration step (e.g. RK45, RK853, RADAU5). Used by default.

Time accuracy – the desired time accuracy for finding change events (switch points) when solving equations.

Fixed time step – fixed time step for methods using the fixed integration step (e.g. Euler, RK4).

If you do not specify particular solver, AnyLogic chooses the numerical solver automatically at runtime in accordance to the behavior of the system. For instance, when solving differential equations, it starts integration with forth-fifth-order Runge-Kutta method. If the system is stiff, AnyLogic plugs in another solver – RADAU5, designed for stiff differential equation systems. Both these methods change the integration step to achieve the given accuracy. You can set up methods with fixed step (Euler and forth-order Runge-Kutta), which are not as accurate as the first pair of methods, but perform numerical integration more rapidly. However, if you need a precise solution, use the *Automatic* solver.

6. Matrices and hyper-arrays

6.1 Matrices

AnyLogic supports variables of vector and matrix type. Since a vector is a special case of a matrix, the term matrix will refer to both matrices and vectors. In AnyLogic you can define equations over matrix variables.

AnyLogic class `Matrix` provides means for definition, initialization, access, and major operations with matrices, including inverse, transpose, trace, LU decomposition, QR decomposition, Singular value decomposition, Eigenvalue Decomposition, Cholesky Decomposition, Matrix condition (2 norm), One norm, Two norm, Frobenius norm, Infinity norm, and others. This class has been adapted from the JAMA set (A Java Matrix Package, see <http://math.nist.gov/javanumerics/jama/>). Please consult AnyLogic Class Reference for more details.

6.1.1 Defining a matrix

► To define a matrix variable

1. Select the variable on a structure diagram.
2. In the *Variable type* section of the Properties window, select the *Matrix* option.
3. Specify the number of rows in the *Rows* edit box.
4. Specify the number of columns in the *Columns* edit box.

6.1.2 Setting an initial value for a matrix

By default all elements of a matrix get zeroes. You can set up initial values for elements of a matrix.

► **To set initial value for a matrix**

1. Select the variable on the structure diagram.
2. In the *Equations* section of the Properties window, choose *No equation* from the *Form* drop-down list.
3. Specify the *Initial Value* expression.

Provide an initial value expression using such notation: delimit matrix with braces, enter value row-by-row, wrapping each row with braces and separating rows and elements in a row with commas. Be sure all the rows have the same number of elements. For example, a matrix is the following:

$$\begin{vmatrix} 5 & 7.83 & -1 \\ 0.2 & -3.4 & 0 \end{vmatrix}$$

For two-dimensional matrices these values are assigned in row-first order: Into the *Initial value* property, you type:

```
{{ 5, 7.83, -1}, { 0.2, -3.4, 0 }}
```

, or

```
[5, 7.83, -1 ; 0.2, -3.4, 0]
```

To increase readability you can split the string into multiple lines:

```
{{ 5, 7.83, -1 },
 { 0.2, -3.4, 0 }}
```

You can initialize identity matrices, i.e. matrices with ones on the main diagonal and zeros elsewhere, by typing `identity` keyword.

6.1.3 Accessing and modifying a matrix from Model Viewer

6.1.3.1 Inspecting a matrix variable

You can inspect the current value of a variable at the model runtime from the variable's inspect window.

► To open the inspect window

1. Right-click the variable in the Model Explorer or in the animated structure diagram and choose *Inspect* from the popup menu.

The *Select Item* dialog box is displayed, see Figure 56.

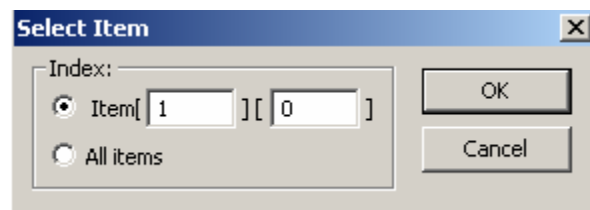


Figure 56. Select Item dialog box

2. Choose whether you want to inspect some specific element of a matrix or all items.
3. Click *OK*.

The Inspect window is displayed (Figure 57 shows the inspect window of a matrix).

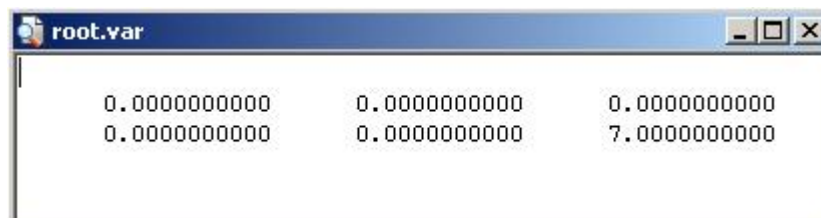


Figure 57. Inspect window of a matrix

6.1.3.2 Modifying a matrix variable

Elements of a matrix can be modified during the model simulation.

► To modify an element of a matrix

1. Double-click the variable in the Model Explorer or on the animated structure diagram, or
Right-click the variable in the Model Explorer or on the animated structure diagram and choose *Modify* from the popup menu.
The *Modify* dialog box is displayed (see Figure 58).
2. In the *Apply to* section, specify whether you want to modify some specific item of a matrix, or all items once.
3. Type a new value in the *New value* edit box.
4. Click *OK*.

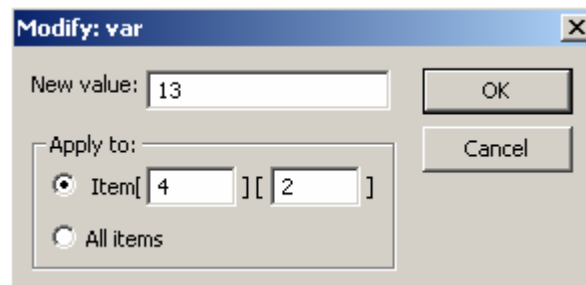


Figure 58. Modifying a matrix variable

6.1.4 Accessing and modifying a matrix from code

6.1.4.1 Defining and initializing a matrix from code

To define a variable of matrix type in Java code (commonly in the *Startup code* section of the active object's Code window), you use one of the following forms:

`Matrix A = new Matrix(<number of rows>, <number of columns>);`
 to create a two-dimensional matrix the specified number of rows and columns

`Matrix A = new Matrix(<integer number>);`
 to create a vector with the specified number of rows

`Matrix A = new Matrix(<real number>);`
 to create a 1x1 matrix with the single element with the specified value

`Matrix A = new Matrix (<number of rows>, <number of columns>, <real number>)`
 to create a matrix initialized with a real number

`Matrix A = Matrix.random(<number of rows>, <number of columns>)`
 to create a matrix with elements uniformly distributed in interval [0,1]

`Matrix A = Matrix.identity(<number of rows>, <number of columns>)`
 to create an identity matrix with ones on the main diagonal and zeros elsewhere

`Matrix A = new Matrix (new double[][] { { 5, 7.83, -1 }, { 0.2, -3.4, 0 } });`
 to create a 3x4 matrix initialized as shown in the previous section. Use the same notation as used for defining matrix variables in the *Initial value* property: delimit matrix with braces, enter value row-by-row, wrapping row with braces and separating rows and elements in a row with commas. Be sure all the rows hold the same number of elements.

To initialize matrix in code at once, write the following code:

```

double a = new double[2][3]; // creating a source array of real numbers
a[0][0] = 5; a[0][1] = 7.83; a[0][2] = -1;
a[1] = new double[] { 0.2, -3.4, 0 }
A = new Matrix ( 2, 3, a );

```

If there is a simple algorithm evaluating the elements, you can initialize it using cycles. See the example below:

$$\begin{vmatrix} 1 & 2 & 3 & 4 \\ -5 & 1 & 2 & 3 \\ -5 & -5 & 1 & 2 \end{vmatrix}$$

Write the following code to initialize this matrix:

```
int m = 3; // number of rows
```

```

int n = 4; // number of columns
A = new Matrix ( m, n ); // create zero matrix
for (int i=0; i<m; i++){
    for (int j=0; j<n; j++){
        if (j >= i) A.set( i, j, 1-i+j );
        else A.set( i, j, -5 );
    }
}

```

6.1.4.2 Modifying and accessing elements of a matrix from code

If you want to operate with a matrix itself, you refer to it simply by its name.

If you need a submatrix (e.g., one of the columns) of the matrix A taking part in an equation, use the following function:

```
A.getMatrix(i0, i1, j0, j1)
```

where the points (i0, j0) and (i1, j1) specify the upper left and the lower right elements of the submatrix to select from the matrix A.

Please note that indexes in matrices are zero-based – i.e., start with “0”.

You can access the individual elements of matrices in the code by calling:

```

A.get(i,j) // if A is a matrix
A.get(i)   // if A is a vector

```

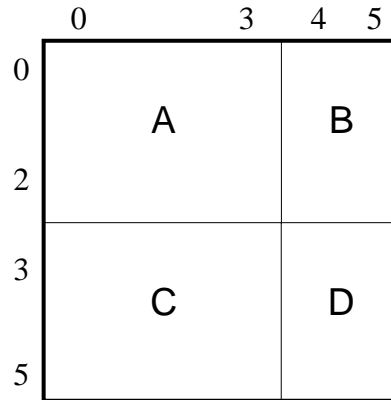
To modify an element of a matrix, use the `set(i, j, value)` method, e.g.:

```

A = Matrix.identity(4,4);
A.set(3,0,7); // set left lower element to 7
A.set(0,3,1); // set right upper element to 1

```

Sometimes you may need to modify a submatrix. For instance, you have a block matrix X used in some equations. Blocks A, B, C, and D should be calculated in different ways (by different formulas).



The idea is to calculate the blocks separately and then combine them into one matrix using transforming matrices. To deal the problem with A and B blocks:

1. Define a matrix variable **A** with dimensions [3,4] and set a necessary formula to it.
2. Define a matrix variable **Aleft** with dimensions [6,3].
3. Type in the *Initial Value* property of **Aleft** the key word *identity*.
4. Define a matrix variable **Aright** with dimensions [4,6].
5. Type in the *Initial Value* property of **Aright** the keyword *identity*.
6. Set the formula for **X**: $Aright * A * Aleft$.
7. Now block **A** of **X** matrix gets the value from variable **A** and other blocks hold zeros.
Block **B** does not lie on the main diagonal, so the right transforming matrix is not an identity one. We will initialize it in startup code.
8. Define a matrix variable **B** with dimensions [3,2] and set a necessary formula to it.
9. Define a matrix variable **Bleft** with dimensions [6,3].
10. Type in the *Initial Value* property of **Bleft** the keyword *identity*.

11. Define a matrix variable `Bright` with dimensions [2,6]. Leave *Initial Value* property empty.

12. In the startup code type:

```
Bright[0,4] = 1;    Bright[1,5] = 1;
```

13. Change the formula for `x`: `Aright*A*Aleft + Bleft*B*Bright`.

Now blocks A and B are calculated properly. To complete the task, perform analogous steps for blocks C and D.

Working with the example, you noticed, perhaps, that matrix `Aleft` equals `Bleft`, `Aright` equals `Crigh`t and so on. Actually, you need only four matrices instead of eight as it may have appeared.

6.1.5 Using matrices in equations

AnyLogic supports matrices as well as primitive types in equations. You can define a set of differential equations, algebraic equations, and formulas to describe continuous changes of variables over time. Thus you can define continuous time object behavior. See Chapter 5, “Equations” to know how to define equations.

If you use matrices in equations, in the right-hand side you can use the operations listed in Table 7. Variables `A` and `B` are assumed to be matrices, `x` is a real variable.

Operation	Description
$A+B$, $A-B$, $-A$	Addition, subtraction, unary minus.
$x*A$, $A*x$	Scalar multiplication.
$A*B$	Matrix multiplication.
$1/B$	Inverse matrix.
A/B	Multiply with inversed matrix, equals to $A*(1/B)$.

<code>A.transpose()</code>	Transpose matrix.
<code>A[i, j]</code>	Access to a matrix element.
<code>A[i]</code>	Access to a matrix row (element, in case of vector).
<code>f(..., A, ...)</code>	A call to a method accepting <code>Matrix</code> type arguments.

Table 7. Available operations for matrices

While generating code for a right-hand side, AnyLogic follows these rules:

- If all variables are scalars, the simple code like `x*y+z` is generated, and the result is treated as scalar.
- Otherwise AnyLogic assumes matrices and generates code like `Matrix.add(Matrix.mult(x,y), z)`.

To change the order of operations, use the round brackets. For instance, there is a matrix differential equation:

$$\frac{dX}{dt} = -D \cdot (5A + B^{-1}) \cdot C^T \cdot e_0,$$

where A, B, X are square matrices $n \times n$; C, D, E – vectors $n \times 1$.

The appropriate right-hand side expression of the integral equation is:

```
- D * ( 5*A + 1/B ) * C.transpose() * E[0]
```

You can explicitly notify AnyLogic that the equation is of scalar type by placing the '#' symbol at the beginning of expression, e.g. `a=#x*y+z`. This may result in simulation performance improvement.

The right-hand side of a scalar variable may refer to matrices, if the result of the expression is scalar. In this example, real variable x could be calculated through the formula:

```
# - D * ( 5*A + 1/B ) * C.transpose()
```

6.1.6 Working with matrices

The `Matrix` class supports many of linear algebra operations you can use in equations, active object class code and in discrete event handlers (timer's *Expiry action*, or statechart transition's *Action* code). Table 8 gives a quick reference to matrix operations.

Operation	Description
<code>A.plus(B)</code>	Matrix addition: $A+B$.
<code>A.minus(B)</code>	Matrix subtraction: $A-B$.
<code>A.uminus()</code>	Unary minus: $(-A)$.
<code>A.times(x)</code>	Scalar multiplication: $x*A$.
<code>A.times(B)</code>	Matrix multiplication: $A*B$.
<code>A.inverse()</code>	Inverse matrix: A^{-1} .
<code>A.transpose()</code>	Transpose matrix: A^T .
<code>A.norm1()</code>	Maximum column sum.
<code>A.norm2()</code>	Maximum singular value.
<code>A.normInf()</code>	Maximum row sum.
<code>A.normF()</code>	Frobenius norm; square root of sum of squares of all elements.
<code>A.solve(B)</code>	Matrix X , which satisfies $A*X = B$, if A is not square, X is the least squares solution.
<code>A.solveTranspose(B)</code>	Matrix X , which satisfies both $A*X = B$, $A'*X' = B'$, if A is not square, X is the least squares solution.
<code>A.inverse()</code>	Inverse or pseudo-inverse matrix.
<code>A.det()</code>	Matrix determinant.

A.rank()	Effective numerical rank obtained from singular value decomposition.
A.cond ()	Matrix condition; ratio of largest to smallest singular value.

Table 8. Matrix operations

There is an example. You should get a matrix that is the result of an expression such as:

$$Z = -D \cdot (5A + B^{-1}) \cdot C^T \cdot e_0$$

where A, B, X are square matrices n*n; C, D, E – vectors n*1.

Type the following code to create such a matrix:

```
Matrix Z = D.uminus().times(((A.times(5)).plus(B.inverse()))).times(
C.transpose()).times(E.getMatrix(0,0,0,E.getRowDimension()));
```

Note that if you write code like A=B, you only let A refer to the same matrix object that B does. If you change matrix B, A changes too, and otherwise. To avoid this dependency you must assign matrix A a copy of matrix B. To make a copy of a matrix, type

```
A=B.copy();
```

If you need checking the matrix content at some moment, you can print it out to the global log. Just type in the appropriate place of the event handler the following:

```
A.print(Engine.log, <number of characters in column>, <number of digits after
the decimal>);
```

To solve a linear equation, write:

```
Matrix x = A.solve(B)
```

To get the eigenvalues of a matrix, write:

```
EigenvalueDecomposition e = A.eig();
double[] eigRe = e.getRealEigenvalues(); // vector of real parts
double[] eigIm = e.getImagEigenvalues(); // vector of image parts
Matrix V = e.getV(); // matrix of eigenvectors
```

6.2 Hyper-arrays

Hyper-array is storage of real numbers that has N dimensions. Each dimension has finite number of indexes - *subscripts*. There are many operations and functions defined for hyper-arrays. You can define equations for hyper-arrays in the same way you do for scalar variables.

Hyper-arrays are used when:

- It is necessary to store a large set of coefficients and access them.
- There are multiple model layers.

The second case is useful when you have defined a model for some subsystem and there are other subsystems, which have the same structure, as the first one, but other numerical parameters. It is a model with multiple layers (Figure 59). The layers can be independent or have some dependencies between their variables.

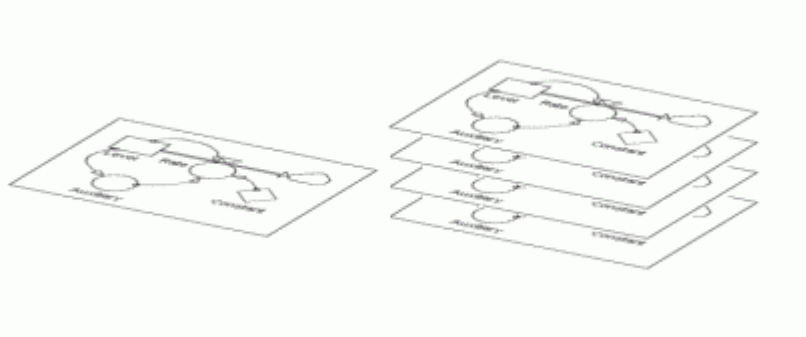


Figure 59. A scalar model and a model with multiple layers

One can implement such multi-dimensional models making copies of the default diagram and changing the parameters. Such approach has one great disadvantage: if you want to change the model, you need do it so much times, as many layers you have; the diagram grows and becomes incomprehensible.

Hyper-array allows you to create a single diagram for all the layers. Therefore, the model remains compact, and changes you make will affect the whole model, but not a single layer.

Before studying hyper-arrays, you should know what an enumeration is.


6.2.1 Enumerations

Sometimes you need a parameter representing an attribute which can take a finite set of values. For example, you work with colors. You can create an integer parameter representing a color and, e.g., assign number 1 to red color, 2 to green color, 3 to blue one. However, you see only integer numbers in code, but not the names of colors. To increase the readability of code, you should use *enumerations*. Each enumeration has named elements, each of them having an integer counterpart that is hidden from you. For example, you can define `Color` enumeration with `Red`, `Green`, and `Blue` elements and use these self-descriptive names instead of integer numbers in code.

```
Color myColor = Red;
...
myColor = Green;
...
if (myColor == Green) traceln("I'm green");
...
```

6.2.1.1 Defining an enumeration

► To define an enumeration

1. Click the *New Enumeration*  toolbar button, or Choose *Insert | New Enumeration...* from the main menu. The *New Enumeration* dialog box is displayed. Specify the name of the new enumeration, choose the package, which will contain the enumeration, and click *OK*.
2. Alternatively, in the Project window, right-click the package, which will contain the enumeration, and choose *New Enumeration...* from the popup menu. The *New Enumeration* dialog box is displayed. Specify the name of the new enumeration and click *OK*.

An enumeration has the following set of properties:

Properties

Name – name of the enumeration.

Elements – the set of enumeration elements.

Exclude from build – if set, the enumeration is excluded from the model.

The enumeration elements are specified in the *Elements* table on the enumeration's properties page. Each element is specified in an individual row.

► To add an enumeration element

1. In the Project window, click the enumeration.
2. In the Properties window, go to the last row of the *Elements* table.
3. Type the enumeration element name.

The order of enumeration elements in the table is very significant.

► To move an element up/down in the table

1. In the Properties window, select the element in the *Elements* table.
2. Press Ctrl+Up/Ctrl+Down.

When the enumeration is defined, you can:

- Define model parameters of enumeration type.
- Use the enumeration as a dimension of a hyper-array.

6.2.2 Defining variables of hyper-array type

AnyLogic supports both variables and parameters of hyper-array type.

► To define a hyper-array parameter

1. In the *Parameter* dialog box, specify `HyperArray` as a parameter type.

2. Initialize the hyper-array in the *Default value* combo box.

For example, you can initialize the hyper-array with another one, defined and initialized in the *Additional class code* code section of the active object class.

► To define a hyper-array variable

1. Select the variable on a structure diagram.
2. In the Properties window, choose the variable type *Array*.
3. Specify as much dimensions as you need in the table on the right of the *Array* option. Each row of the table defines one dimension.
To define a new dimension, go to the last row of the table and click the *Dimension* field. In the combo box, choose an enumeration to be a hyper-array's subscript.

A hyper-array cannot have a single enumeration as two or more dimensions.

The order of dimensions is also important. You can reorder the dimensions pressing Ctrl+Up and Ctrl+Down.

6.2.3 Setting an initial value for a hyper-array

► To set the initial value for a hyper-array

1. Select the variable on the structure diagram.
2. In the *Equations* section of the Properties window, choose *No equation* from the *Form* drop-down list.
3. Specify the *Initial Value* expression.

There are two different notations for hyper-array constants. We illustrate both notations by hyper-array constants used as initial values of some variables. The subscripts are listed before each constant in the order of dimension declarations in these variables.

Suppose you create a model of a nation's health, describing some social or health processes in respect to different groups of population. Separate people by three characteristics: gender,

age group, and social group. The characteristics fit well in the enumeration concept. There are such enumerations: Gender(male, female), Age(child, teenager, adult, aged), and SocialGroup(wealthy, middleclass, deprived).

- Matrix-style. This notation can be used for 1D and 2D arrays.

Age

```
[ 1, 0.8, 0.62, 0.2 ]
```

Age, Gender

```
[      1 ,   1.2 ;
      0.8 ,   0.75 ;
      0.62 ,   0.5 ;
      0.2 ,   0.1   ]
```

- Java-style. This notation can be used for hyper-arrays with arbitrary dimension number.

Gender

```
{      1 /*male*/, 1.2 /*female*/ }
```

Age, Gender

```
{ {      1 , 1.2 },      // Age: child
  { 0.8 , 0.75 },      // Age: teenager
  { 0.62 , 0.5 },      // Age: adult
  { 0.2 , 0.1 }        // Age: aged
}
```

SocialGroup, Age, Gender

```

{
  {
    // SocialGroup: wealthy
    { 1 , 1.2 }, // Age: child
    { 0.8 , 0.75 }, // Age: teenager
    { 0.62 , 0.5 }, // Age: adult
    { 0.2 , 0.1 } // Age: aged
  } ,
  {
    // SocialGroup: middleclass
    { 1.3 , 1.4 }, // Age: child
    { 0.3 , 0.96 }, // Age: teenager
    { 0.6 , 0.52 }, // Age: adult
    { 0.2 , 0.05 } // Age: aged
  } ,
  {
    // SocialGroup: deprived
    { 1.6 , 1.5 }, // Age: child
    { 0.3 , 0.81 }, // Age: teenager
    { 0.35 , 0.4 }, // Age: adult
    { 0.05 , 0.15 } // Age: aged
  }
}

```

To initialize hyper-array elements with a single scalar constant, just type the desired scalar number in the *Initial value* property.

If you need an array with elements distributed by some law, you create an appropriate distribution object (see Chapter 10, “Stochastic modeling”) and pass it to the method `random()` of the hyper-array.

For instance, if you need array A initialized with elements, uniformly distributed in interval $[\min, \max)$, you type the following string in its *Initial value* property:

```
random( new DistrUniform(min,max) )
```

Or you can initialize it in the *Startup code* code section of the active object class with the following string:

```
A.random( new DistrUniform(min,max) );
```

6.2.3.1 Setting specified elements of a hyper-array

Sometimes, you need setting some group of elements of a hyper-array separately.

Therefore call method `set()` in the *Startup code* or in the event handler:

```
<hyper-array>.set([<list of subscripts>,] <value>);
```

The `<list of subscripts>` identifies, which element(s) of the `<hyper-array>` should get the `<value>`. For example, hyper-array variable `SmokingRate` has the dimensions: `SocialGroup`, `Age`, `Gender`. The examples of possible subscript lists for this array are listed in Table 9.

Subscript lists	Description
<code>wealthy, teenagers, male</code>	One element gets the <code><value></code> - smoking rate for wealthy male teenagers.
<code>wealthy, teenagers</code>	Vector of two elements gets the <code><value></code> - smoking rate for wealthy teenagers of both genders.
<code>teenagers</code>	A sub-array of elements gets the <code><value></code> - smoking rate for teenagers of both genders and of all social groups.
<code><empty list></code>	All array elements get the <code><value></code> .

Table 9.

6.2.4 Aggregation functions

Sometimes you need performing aggregation operations on elements of hyper-arrays. Aggregation functions are listed in Table 10. 'N' is the size of the set of the aggregated elements $\{x_i\}$.

Function	Description
Sum	The sum of the aggregated elements: $S = \sum_{i=1}^N x_i$
Avg	Average value: $\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$
max, min	Maximum/minimum value of the aggregated elements.
Prod	The product of the elements: $P = \prod_{i=1}^N x_i$
Stddev	Standard deviation: $\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$

Table 10. Aggregation functions for hyper-arrays

The function syntax is:

```
<hyper-array>.<function name>()
```

Sometimes it is necessary to perform aggregation on a sub-array. You can do it with the following call:

```
<hyper-array>.get(<list of subscripts>).<function name>()
```

The `<list of subscripts>` defines the sub-array to aggregate in the same manner as for `set()` method (see section 6.2.3.1, "Setting specified elements of a hyper-array").

6.2.5 Using hyper-arrays in equations

AnyLogic supports hyper-arrays as well as primitive types in equations. See Chapter 5, “Equations” to know how to define equations. You can define differential equations or formulas for hyper-array variables in a way similar to scalar variables. However, there are some specifics that we describe in the current section.

6.2.5.1 Arithmetic operations with hyper-arrays

In counterpart to matrices, operations with hyper-arrays do not go by the linear algebra rules, but are applied to array elements one-by-one. The operands may be of different dimensions. The result of the operation is a hyper-array with the dimensions that the operand subscripts union has. The following figures illustrate some cases [symbol # is the operation sign placeholder – it can be addition (+), subtraction (-), multiplication (*), or division (/) symbol].

If both operands have the same dimensions, the resulting hyper-array is of the same dimensions also. Each element of the resulting hyper-array is the result of operation with two co-indexed elements of A and B.

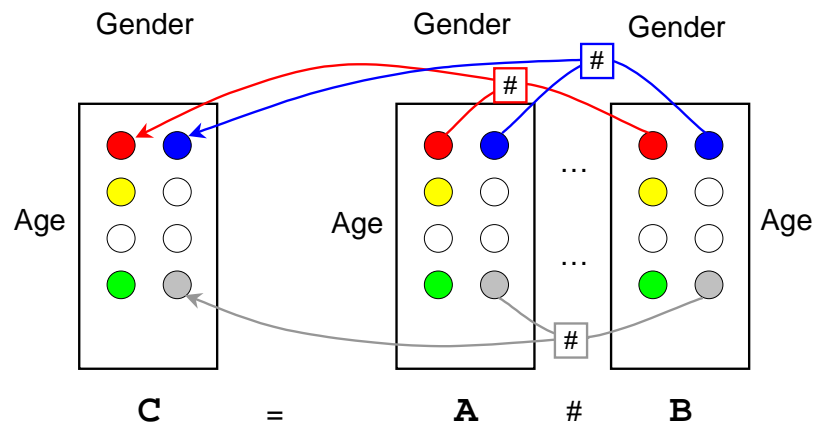


Figure 60. Operation with hyper-arrays having the same dimensions

Another case: the dimension set of one operand (A) contains all the dimensions of the other one (B). In other words, the intersection of the dimension sets equals the dimension set of B and their union equals the set of A.

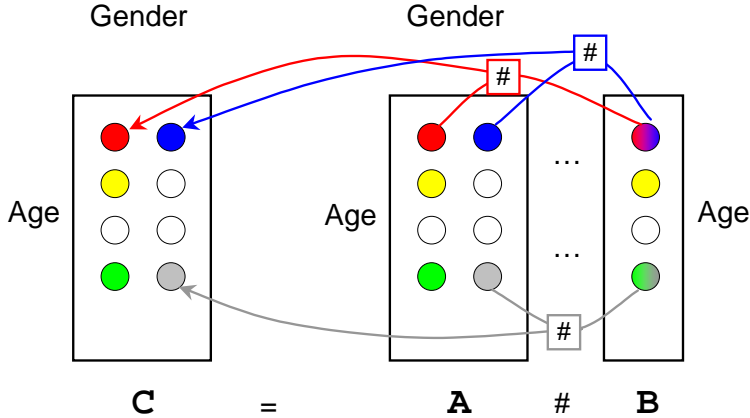


Figure 61. Operation with hyper-arrays, where one has less dimensions

The operation with the integer or real scalar value (x) is applied to each element of the hyper-array A (Figure 62).

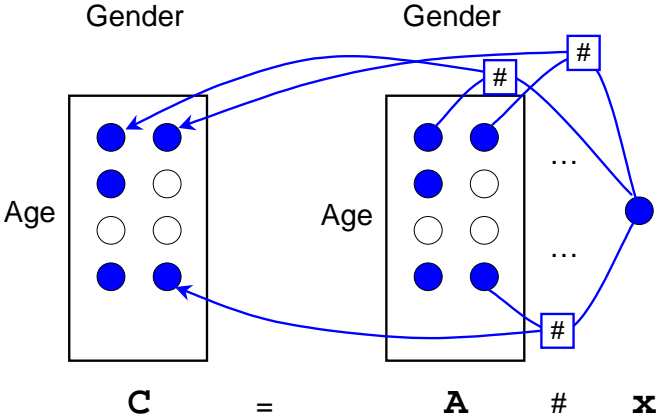


Figure 62. Operation with a hyper-array and a scalar

In the case shown in Figure 63, the resulting dimension set is the union of subscripts. One can describe the operation as a sequence of operations with the first array and vectors extracted from the second one (as shown in the Figure 61). Regarding the figure example, “wealthy” sub-array of resulting hyper-array (where ‘social group’ subscript is ‘wealthy’, marked with red color) is calculated in the same way, as by operation with hyper-array A and vector extracted from B where ‘social group’ is ‘wealthy.’

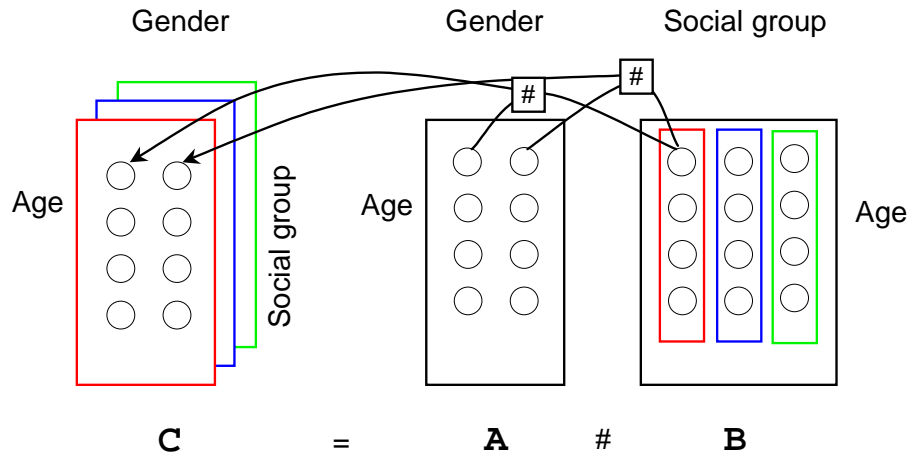


Figure 63. Operation with arrays having different dimensions

6.2.5.2 Array functions

All the predefined functions and all the lookup tables you define within your project accept both scalar and hyper-array arguments. In case of hyper-array, unary function is applied to all the elements of the array argument, similar to arithmetic operations with a scalar. Binary functions are applied to two hyper-arrays in the same way as operation with hyper-arrays having the same subscripts.

6.2.5.3 Getting array elements and sub-arrays

Sometimes you need to access only some specific element or a sub-array. In equations you get the desired element or sub-array by the following expression:

```
<hyper-array> '[' <list of subscripts> ']'
```

The `<list of subscripts>` identifies the array to get in the same manner as in `set()` method (see section 6.2.3.1, “Setting specified elements of a hyper-array”).

For instance, there is a hyper-array `Smoking_Rate`, which holds the average smoking rate of people in respect to gender, age, and social group.

To get the value of an element of a hyper-array in an equation, you type the following code in an equation expression. You should specify subscripts for all the dimensions.

```
SmokingRate[female, teenagers, middleclass]
```

To get a sub-array in an equation, specify subscripts not for all dimensions. For example, you are interested in smoking rate for all people, pertained to the deprived social group. You get the corresponding sub-array having (gender, age) dimensions this way:

```
SmokingRate[deprived]
```

You can perform conditioned extraction. Therefore you define a parameter whose type is an enumeration. For instance, it is of type `Age`, is named `ageParam`, and has `child` as default value.

```
SmokingRate[Female, ageParam]
```

Then you can change the value of this parameter on some event (e.g. on timer expiry action):

```
ageParam = Teenagers;
```

7. Message passing

Commonly you may need to send some information from one active object to another. In AnyLogic you can establish active object interaction by passing *messages* – data units, carrying some useful information. Messages can model various objects of the real world. You may implement notification or signaling mechanism in your system – in this case, messages may represent commands or signals passed in a control system. Or you may need to model entity flow in your system, where messages represent entities – the items that are being served, produced, or otherwise acted on by your process: documents in business processes, customers in service systems, parts and products in manufacturing models. You may have different types of entities in the same model.

Messages are sent and received via the special elements of active objects – *ports*. Message passing is enabled only between ports connected by *connectors* – paths used by messages to flow through the model.

This chapter gives the detailed description of the message passing mechanism. It is organized as follows:

- Section 7.1, “Ports”, describes how to create ports and add them to your own active object classes. This section also contains the detailed explanation of message routing rules.
- Section 7.2, “Messages”, describes how to define your own message classes. You should read it if you need to carry some data in messages being passed between active objects in your model.
- Section 7.3, “Defining custom port classes”, gives the detailed description of the predefined AnyLogic port classes. This section describes how to create your own port classes with customized behavior to change the semantics of message passing.

Consider the chapter organization to study sections meeting your requirements.


If you create your model from AnyLogic library objects, please refer to section 1.5.8, “Active objects interaction” to know how to connect objects and establish message passing.

7.1 Ports

Ports play the central role in the message passing mechanism. Messages are sent and received through ports. Ports are bi-directional and can serve both for input and output. Ports may be public – i.e., accessible from outside the class; or private – accessible only inside an active object. To communicate with other objects, you need to add a public port to the active object; and to establish a communication inside the object, you can use a private port.

7.1.1 Adding a port to an active object class

► To add a public port to an active object class

1. Click the *Port*  toolbar button, or Choose *Draw | Structure | Port* from the main menu.
2. Click the class border on the structure diagram.
A new port appears, displayed as the small square, see Figure 64.

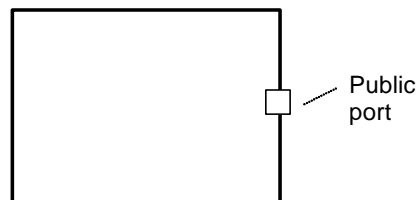



Figure 64. A public port

Placing a port on the class border makes it public.

► To add a private port to an active object class

1. Click the *Port*  toolbar button, or Choose *Draw | Structure | Port* from the main menu.
2. Click inside the class border on the structure diagram.
A new port appears, displayed as the small square, see Figure 65.

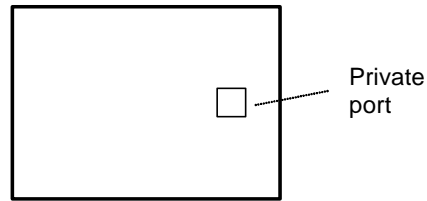


Figure 65. A private port

Once the new port is created, you can specify the port name in the text line editor opened on the right of the port in the structure diagram.

7.1.1.1 Moving, copying and deleting ports

There are some common operations you can perform with ports on the class structure diagram. You can copy, move and delete ports just as any other class elements. First, you should select the port by clicking it.


► To move a port

1. Drag the port with the mouse or use arrow keys.

► To copy a port

1. Ctrl-drag the port.

► To delete a port

1. Click the *Delete*  toolbar button, or
Choose *Edit | Delete* from the main menu, or
Right-click the port and choose *Delete* from the popup menu, or
Press Del.

7.1.1.2 Port properties

A port has a set of customizable properties. To set the port properties, first select the port on the structure diagram by clicking, and next set the property values in the Properties window.

The following properties can be set on the *General* page of the Properties window (Figure 66):

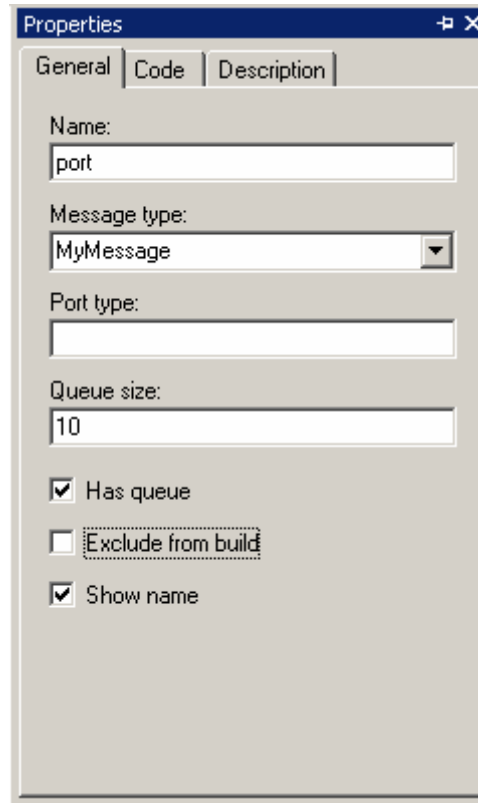


Figure 66. General page of the port's Properties window

General properties

Name – the name of the port.

Message type – [optional] every message received by the port is cast to this type. If the message cannot be cast, it is discarded. Using combo box you can choose from all known message classes of this project. Message class definition is described in section 7.2, “Messages”.

Port type – [optional] the type of the port. Port class definition is described in section 7.3, “Defining custom port classes”.

Has queue – if set, incoming messages will be placed in the port queue. See section 7.1.2, “Port queue” for details.

Queue size – [optional] the size of the queue. Leaving it blank makes the size of the queue “infinite” to store all incoming messages.

Exclude from build – if set, the port is excluded from the model.

Show name – if set, the name of the port is shown on the structure diagram.

The properties listed below can be set on the *Code* page of the Properties window (Figure 67):

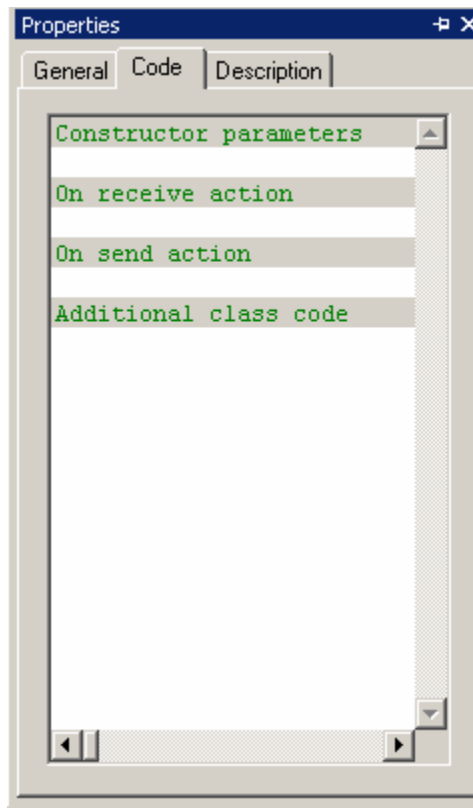


Figure 67. Code page of the port's Properties window

You can change the default port behavior by writing code on the *Code* page of the port's Properties window. AnyLogic provides you an ability to specify port reactions to different occurrences by writing your own Java code.

Code properties

Constructor parameters – [optional] parameters to be passed to the port's constructor, e.g.: `/*count*/ 20, /*load*/ load.`

On receive action – [optional] a sequence of Java statements to be executed on every message reception.

On send action – [optional] a sequence of Java statements to be executed on every message sending.

Additional class code – [optional] Java code to be inserted into the port class definition. Arbitrary constants, variables, and methods can be defined here.

An active object – owner of the port can be accessed by the `getOwner()` method of the port. The detailed description of AnyLogic port classes methods is given in section 7.3.1, “Predefined port classes”.

When writing your code for an active object – e.g., in its *Additional class code* – you can access a port `myPort` of an active object as its member variable `myPort`.

7.1.2 Port queue

You can specify that a port has a queue to store incoming messages. Stored messages can be extracted afterwards. A port queue is commonly used to store port incoming messages while the active object is busy with processing another message – e.g., while a server processes client request in the client-server system. A port queue can also model a data storage.

The presence of a port queue and its capacity is specified on the *General* page of the Properties window.

You may have noticed that ports with queues and ports without them appear somewhat differently on the structure diagram. A port with a queue is displayed on the structure diagram with a dot inside it, see Figure 68.

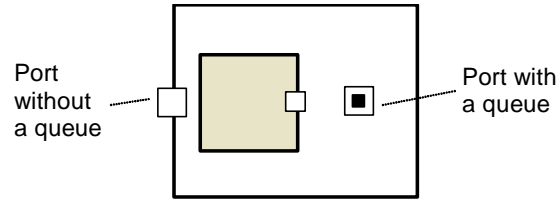


Figure 68.

Port queue is FIFO queue –i.e., the first placed in the queue message is extracted first. If the specified queue capacity is reached, an incoming message is lost. Queue can be made infinite to store all incoming messages.

You can access messages stored in the port queue using the methods `get()` and `peek()` automatically generated by AnyLogic for each port with a queue (the `get()` method extracts the message from the queue as well). These methods return the message of type specified in the *Message type* property of the port.

You can check the queue size using the `size()` method. You may check `portA` queue e.g. on timer expiration. Create timer and type the following code in its *Expiry action* property:

```
if ( portA.size() > 0 )
    Object msg = portA.get();
```

First, the method `size()` checks if there are any messages stored in the queue. If any, the `get()` method extracts the first one.

7.1.3 Connecting ports

To establish message passing you should connect the respective ports by *connectors*. Connectors are paths used by messages to flow through the model.

You can establish message passing between:

- Ports of encapsulated objects
- A public port and a private port
- A public port and a port of an encapsulated object
- A private port and a port of an encapsulated object.


You can also connect a port to a statechart, see section 7.1.4, “Connecting ports to statecharts”.

This section describes the technique of establishing connections between ports.

7.1.3.1 Connecting ports of encapsulated objects

To establish message passing between two encapsulated objects, you should connect ports of encapsulated objects.

► To connect ports of encapsulated objects

1. Drag the port of one encapsulated object onto the port of another encapsulated object, or
Click the *Connector*  toolbar button, click one port, and then click another port, or
Choose *Draw | Structure | Connector* from the main menu, click one port, and then click another port.
The connector linking two ports appears.

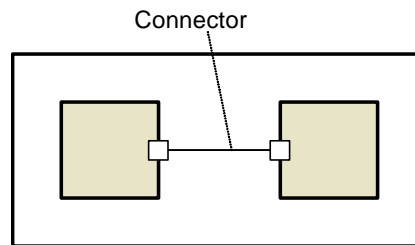



Figure 69. Ports of encapsulated objects connected

7.1.3.2 Connecting a public port with a port of an encapsulated object

To establish communication between the encapsulated object and the container object neighborhood, you should connect a port of an encapsulated object with a public port of the container active object.

► **To connect a public port with a port of an encapsulated object**

1. Drag the port of the encapsulated object onto the public port, or
Click the *Connector*  toolbar button, click one port, and then click another port, or
Choose *Draw | Structure | Connector* from the main menu, click one port, and then click another port.
The connector linking two ports appears.

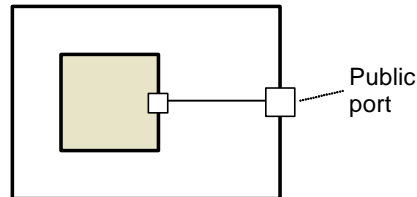



Figure 70. A public port connected to a port of an encapsulated object

7.1.3.3 Connecting a private port with a port of an encapsulated object

To establish communication between the encapsulated object and the container active object you should connect a port of an encapsulated object with a private port of the container object.

► **To connect a private port with a port of an encapsulated object**

1. Drag the port of the encapsulated object onto the private port, or
Click the *Connector*  toolbar button, click one port, and then click another port, or
Choose *Draw | Structure | Connector* from the main menu, click one port, and then click another port.
The connector linking two ports appears.

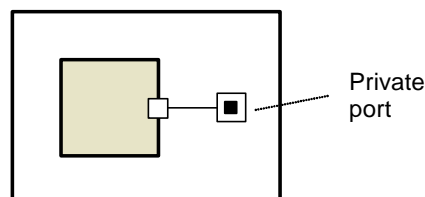



Figure 71. A private port connected with a port of an encapsulated object

7.1.3.4 Connecting a public port with a private port

To establish a communication between a private port and other active objects you should connect the private port with a public port. However, if you need only to send messages via the private port to other active objects, it can be made public.

► To connect a public port with a private port

1. Click the *Connector*  toolbar button, click one port, and then click another port, or Choose *Draw | Structure | Connector* from the main menu, click one port, and then click another port.

The connector linking two ports appears.

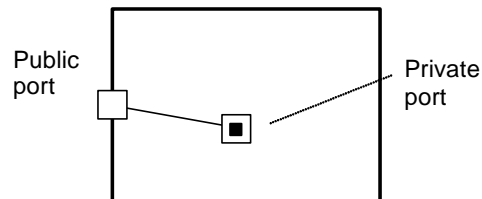


Figure 72. Public and private ports connected

7.1.4 Connecting ports to statecharts

Ports can also be connected to statecharts. In that case messages received in a port are forwarded to a connected statechart, where they are processed as signal events. Using such a connection, you can react to the message arrival in the statechart. For detailed information on statecharts see Chapter 9, “Statecharts”.

You can connect a statechart to:

- A public port,
- A private port.


You cannot connect active object's statechart to a port of its encapsulated object.

This section describes the technique of establishing connections between ports and statecharts.

7.1.4.1 Connecting a public port to a statechart

You connect a public port to a statechart if you need to react in the statechart to messages received in the port from other active objects.

► To connect a public port to a statechart

1. Click the *Connector*  toolbar button, click the port, and then click the statechart, or Choose *Draw | Structure | Connector* from the main menu, click the port, and then click the statechart.

The connector linking the port and the statechart appears.

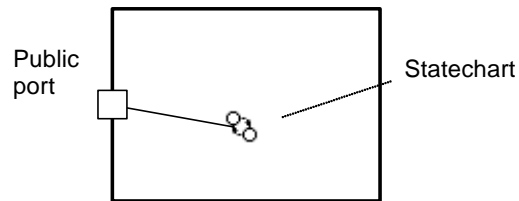



Figure 73. A public port connected to a statechart

7.1.4.2 Connecting a private port to a statechart

You can connect a private port to a statechart if you need to react in the statechart to messages received in the port. It is commonly used when the private port is connected to a port of an encapsulated object and you want to react in the statechart to messages received from the encapsulated object.

► To connect a private port to a statechart

1. Click the *Connector*  toolbar button, click the port, and then click the statechart, or Choose *Draw | Structure | Connector* from the main menu, click the port, and then click the statechart.

The connector linking the port and the statechart appears.

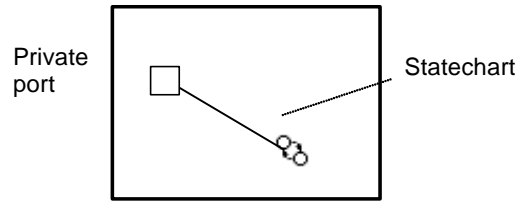


Figure 74. A private port connected to a statechart

7.1.5 Message routing rules

Message processing at a port depends on the direction the message is going and on the type of a port.

Public ports act as relay ports, forwarding messages depending on the direction the message is going. If the message arrives at a public port from inside the active object, it is forwarded along all external connections of the port. Otherwise, if it is received from outside, it is forwarded along all port connections inside the active object.

A message received at a private port is forwarded only to connected statecharts.

Message routing rules are illustrated in Figure 75.

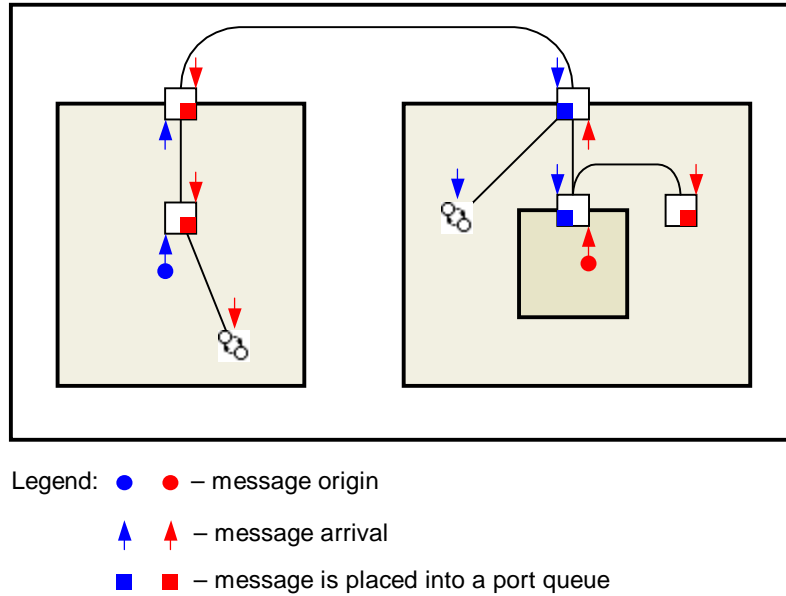


Figure 75. Messages routing rules

Note that in case a port has a queue and is also connected to inner ports and statecharts, then an incoming message is both placed in the port queue and is sent to connected ports and statecharts. Therefore, you have to take care of processing all of them.

7.1.6 Sending and processing messages

7.1.6.1 Sending messages

To send a message, you should simply call the method `send()` of a port, providing the instance of a message class as a parameter. Defining your own message classes is described in section 7.2, "Messages". If you need to just signal an object, you can send an instance of class `Object` that does not carry any data by calling the method `send()` with omitted parameter.

For example, you can type the following line of code in the *Startup code* code section of an active object to send a message of `Message` type via its `portA` port at the model startup.

```
portA.send( new Message() );
```

7.1.6.2 Processing sent messages

When you send a message via a port, it is processed further. The way the sent message is processed further depends on the type of the port:

- In the case of a public port, the message is forwarded along all the port connections outside the active object.
- In the case of a private port it is forwarded to all connected ports.

Arrived message processing at a port also depends on the type of a port.

If a message arrives at a public port from an internal port (from a private port or a port of an encapsulated object), the public port acts as a relay port and sends the message further along the external connections of the public port. In this case the method `send()` of the port is called and the code specified in the *On send action* code property of the port is executed (see section 7.1.6.3, “Defining message sending handler” for more details).

Otherwise, if a message arrives to a public port from the outside or if it arrives to a private port, the message is received by a port and processed further according to the message routing rules, described in section 7.1.5, “Message routing rules”. The message received at a port with a queue is placed in the queue as well. The method `receive()` of the port is called and the code specified in the port *On receive action* code property is executed (see section 7.1.7.1, “Defining message reception handler” for more details).

These rules are illustrated graphically in Figure 76.

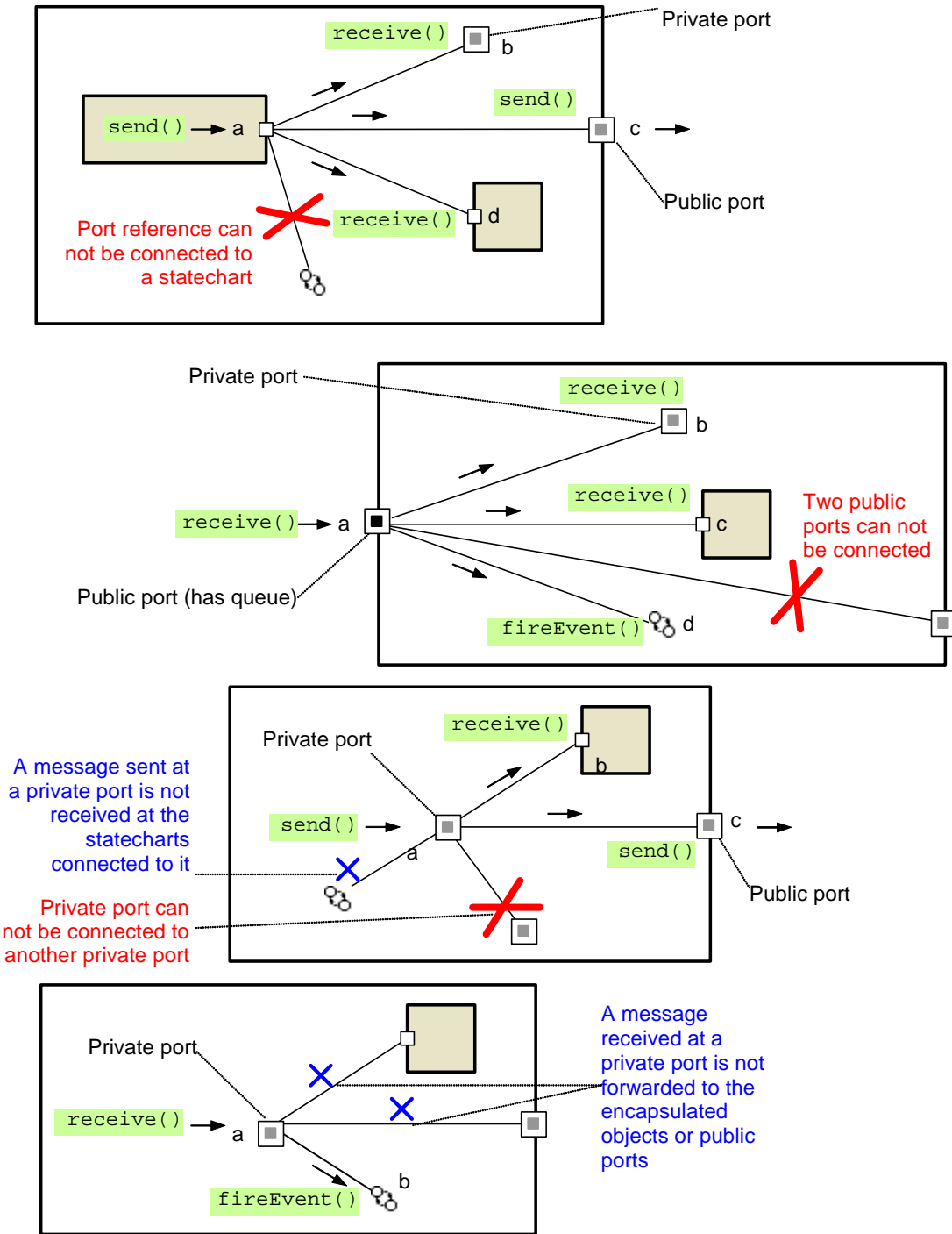


Figure 76. Message processing rules

7.1.6.3 Defining message sending handler

You can define the message reception handler in the *On receive action* code property of the port. This code is executed each time a message is sent. In that code you can use a local variable `msg`, which is a reference of type `Object` to a just sent message. If `true` is returned, the message is processed further as defined by message sending rules, see section 7.1.6.2, “Processing sent messages”. The same happens by default when *On send action* is left blank. If `false` is returned or if you write any code and return nothing, the default processing is omitted.

7.1.7 Receiving messages

To react to a received message, you can choose one of the ways described below.

7.1.7.1 Defining message reception handler

You can define the message reception handler in the *On receive action* code property of the port. This code is executed each time a message is received. In that code you can use a local variable `msg`, which is a reference to a just received message. The code may contain `return` statement returning `true` or `false`. If `true` is returned, the message is forwarded further as defined by message routing rules, see section 7.1.5, “Message routing rules”. If the port has a queue, the message is placed in the queue as well. The same happens by default when *On receive action* is left blank. If `false` is returned or if you write any code and return nothing, the default processing is omitted – i.e., the received message is neither forwarded further nor placed in the queue.

7.1.7.2 Generating signal events for a statechart

You can connect a port to a statechart. Then all messages accepted by the port are placed into the statechart queue as signal events. Note that they are also placed into the port queue. The detailed information on statechart's signal events is given in section 9.4.4, “Signal event”.

7.1.7.3 Triggering a transition of a statechart by a non-empty port queue

You can trigger a statechart transition if the port has a nonempty queue of messages. If by the time the statechart comes to the source state of such transition, the port queue already has messages, the transition becomes enabled immediately. Otherwise the transition becomes enabled on a message arrival to that port. As usual, the transition is executed only if the guard condition for this transition is `true` as well. The detailed information on statecharts is given in Chapter 9, “Statecharts”.

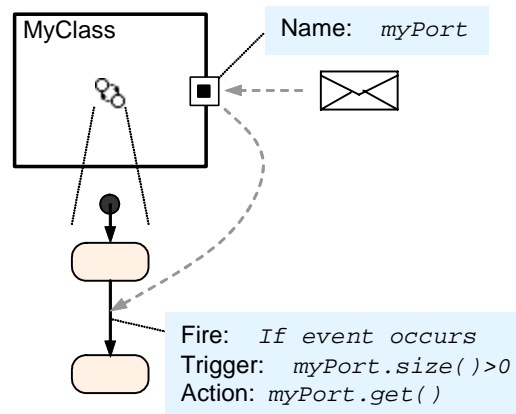


Figure 77. Transition triggered by port event

Please note that triggering of such a transition does not delete the message from the port queue. If you do not need to keep that message anymore, call the method `get()` of the port in the *Action* section of the transition properties to consume the message from the port queue, as shown in Figure 77.

7.1.7.4 Receiving messages in a thread

You can react to the message arrival in a thread. Thread is an activity implemented in a Java method and runs in a separate thread concurrent with all other activities. Threads are less visual than statecharts or timers, but in rare cases, they may provide for more natural representation of some algorithms. In general, however, if the activity has a set of states with different reactions to events, make it a statechart.

You can call the method `waitForMessage()` of the class `PortQueuing` from the thread. If there are messages in the queue, the first one is extracted. If not, the thread suspends until a message arrives to this port.

In the example shown in Figure 78 the thread performs an infinite loop: it waits for the message arrival, then makes 10 time units delay.

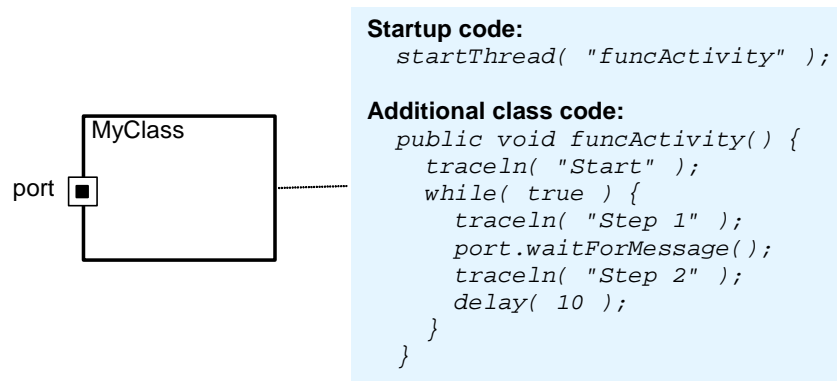


Figure 78. Thread

7.1.8 Inspecting port state at runtime

You may want to get some information about the current state of a port at the model runtime. AnyLogic provides several ways of inspecting port state at runtime. These are: setting breakpoints on ports and displaying inspect strings associated with the ports in AnyLogic Inspect window.

7.1.8.1 Setting a breakpoint on a port

You may need to trace all message arrivals to the specific port. Therefore, you can set a breakpoint on a port. The model execution will be stopped on every message passing through the port with a set breakpoint.

A breakpoint can be set from the Model Explorer or from an animated structure diagram. Breakpoints are displayed in animated diagrams dashed red.

You can manage (remove, enable, and disable) breakpoints using the *Edit Breakpoints* dialog box. See section 14.3, “Breakpoints” for details.

► **To set/clear breakpoint on a port**

1. Right-click the port and choose *Breakpoint* from the popup menu.

7.1.8.2 Inspecting port state

You can inspect the current state of the port at the model runtime from the port’s inspect window. The inspect window displays the inspect string associated with the port.

► **To open the inspect window of a port**

1. Double-click the port, or
Right-click the port and choose *Inspect* from the popup menu.

AnyLogic allows the user to define custom inspect strings for a port. This is done using the following method of the `Port` class.

Related method of `Port`

```
void setPortInspect( String s ) - sets the inspect string for the port.
```

For example, you can display a number of messages sent via a port. Define the `sentCount` member variable to count sent messages in the port’s *Additional class code*:

```
int sentCount = 0;
```

Enter the following code in the *On receive action* property of the port:

```
sentCount++;  
setPortInspect(sentCount + " messages were sent via the port.");  
return true;
```

Thus, the inspect window will display the number of messages sent via the port, as shown in Figure 79.

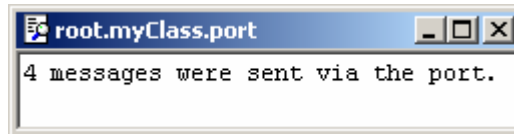


Figure 79. Inspect window

7.2 Messages

A message is a data packet that is passed between active objects. Messages can model various objects of the real world. For example, they can represent entities – parts, products, people, trucks, etc., or data packets being passed in a network, or commands and signals in a control system.


Messages are instances of arbitrary Java classes. Usually, a message carries some data. To define such a message, you need to define a message class with necessary member variables and, may be methods.

In case you need just to signal an object, you can use an instance of the class `Object` as a message not carrying any data.

7.2.1 Defining message classes

The easiest way to define a message class is to use the Project window.

► To add a new message class to a package

1. Click the *New Message Class*  toolbar button, or Choose *Insert | New Message Class...* from the main menu. The *New Message Class* dialog box is displayed. Specify the name of the new message class, choose the package, which will contain the message class, and click *OK*.
2. Alternatively, in the Project window, right-click the package, which will contain the message class, and choose *New Message Class...* from the popup menu.

The *New Message Class* dialog box is displayed.
Specify the name of the new message class and click *OK*.

Since a message class is a standard Java class you can define it anywhere in the code; e.g., in the *Additional class code* property or in an external file if you need to reuse it in other models as well.

In AnyLogic a message class has the following properties:

Properties

Name – name of the message class.

Base class – [optional] name of the base class. This can be any Java class. If omitted, *Object* is assumed.

Fields – [optional] set of fields of the class. Every field should be declared in form: *Type Name Default*, where *Type* is the type of the field, *Name* is the name of the field, *Default* is the optional default value of the field. When instantiating the class, field values may be specified or default ones may be left. During the lifetime of a message fields may be accessed as member variables of the message.

Exclude from build – if set, the class is excluded from the model.

In the Code window of a message class you can define arbitrary code for a message class.

► To open the Code window of a message class

1. In the Project window, right-click the message class item of workspace tree and choose *Open Code* from the popup menu, or Double-click the message class.
The Code window of the message class is displayed (see Figure 80)

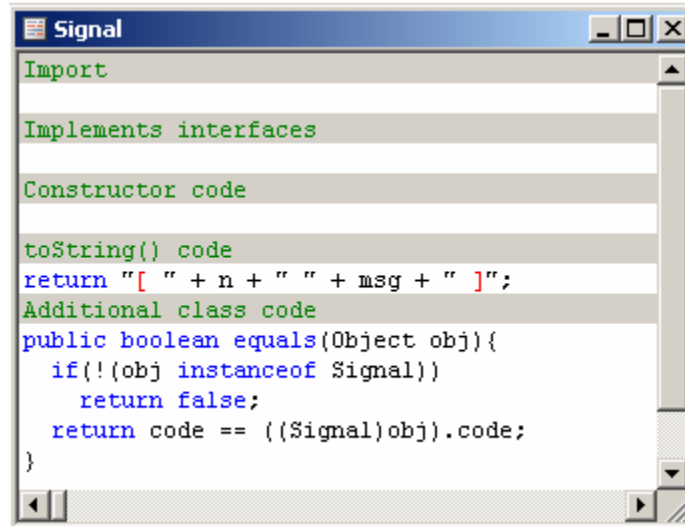


Figure 80. Code window of a message class

Code window has the following sections:

- Import* – [optional] set of `import` statements needed for correct compilation of the class code. When Java code is generated, *Import* property is inserted before definition of the corresponding Java class.
- Implements interfaces* – [optional] comma-separated list of interfaces implemented by the class.
- Constructor code* – [optional] sequence of Java statements to be executed on message construction.
- toString() code* – [optional] code of the method `toString()` of the message class (including the `return` statement). This method is used by Java to convert the message into a character string, which can be used, e.g., to print messages in inspect windows and logs. If no code is specified, a default `toString()` method is generated printing the values of all the message fields.
- Additional class code* – [optional] Java code to be inserted into the class definition. Constants, variables, and methods can be defined here.

7.2.2 Cloning messages to avoid sharing violation

According to message routing rules, a message is broadcasted to all the recipients. It is important to understand that all the recipients get references to the same Java object, representing the message. Therefore, if any of them modifies the message, this affects all other recipients as well.

If you need to modify the message received in multiple locations, you have to clone it explicitly and work with a clone to avoid sharing problems with other recipients.

To clone a message, use the method `clone()` of the message class. The method implements field-by-field copy of a message.

In fact, each message class automatically implements the standard Java `Cloneable` interface, and the `clone()` method generated by AnyLogic just delegates the actual cloning to the `Object.clone()` method doing some wrapping to catch any exception. Thus, the resulting message clone is the `Object` class instance, and you may need to cast it explicitly to the original message class.

For example, we send messages of the `Message` type. Type the following code in the port's *On receive action* property to clone the received message and cast the message clone to the `Message` class:

```
Message message = (Message)msg.clone();  
  
//modify the message clone  
  
return true;
```

7.2.3 Messages encapsulating/inheriting other messages

Sometimes – e.g., while modeling communication protocols – it is necessary to pass one message containing another one. There are two ways to implement this:

- You can encapsulate the message in another one by passing it as a message field. In this case you can use some generic class for the field type (e.g., `Object`), so that the container message does not need to know anything about the encapsulated message – i.e., it does not need to know its class.

- You can inherit your message class from another base message class. Thus your message class inherits all fields and methods of the base class. In this case access to base member fields is straightforward. The message inheritance is set by specifying the base message class name in the *Base class* property of your message class. Note that the derived class members will override the class members of the base class.

The choice among these techniques depends on what you want to achieve. If you want to have a reusable message that can contain a message of any type, you use the first technique. If it is vital for your model to have simple access to message parameters, the second solution is preferable.

7.3 Defining custom port classes

You can customize the port behavior by writing your own code in the *On send action*, *On receive action* and *Additional class code* sections of the port properties. However, if the defined port functionality is frequently needed, defining your own port class is preferable. Thus, instead of writing the same code in the properties of all the port instances in your model, you need just to create port class once and specify the created port class name in the *Port type* properties of the ports in the model. You can create your own port class in code, external file, or a library. Defining port class in an external file or in a library has an advantage of future reuse of your custom ports in other models as well (see Chapter 19, “Libraries and external files” for details).

This section gives the detailed description of AnyLogic port classes and describes how to create your own port classes to customize the default behavior of ports and change the semantics of message passing.

7.3.1 Predefined port classes

AnyLogic has two predefined port classes: `Port` for a port without a queue and `PortQueueing` for a port with a queue. If you want to customize the default behavior of ports, you need to define your own port class, derived from one of these base classes.

7.3.1.1 Port class

The `Port` class is the base class for all port classes in `AnyLogic`. This class provides basic functionality for message sending and receiving and also supports port connection and disconnection.

The methods of the `Port` class are listed in the table below.

Method	Description
<code>void send()</code>	The method creates the message of type <code>Object</code> and sends it via this port. The <code>onSend()</code> method of the port is called to check whether the port actually should process the message.
<code>void send(Object msg)</code>	The method sends the <code>msg</code> message via this port. The <code>onSend()</code> method of the port is called to check whether the port actually should process the message. According to time semantics of message passing, the method <code>send()</code> finishes only after the message has been delivered to all final destinations and <code>onReceive()</code> methods of the message recipients have finished their work.
<code>boolean onSend(Object msg)</code>	The method is called before the <code>msg</code> message is sent. It can be overridden in a derived port class to perform additional message checking and some user-defined actions. Returning <code>true</code> means that the message must be sent via this port; <code>false</code> - that the message should be discarded. By default the method returns <code>true</code> .
<code>void receive(Object msg)</code>	The method is called when the port receives a message. The <code>onReceive()</code> method is called to check if the port should accept the message. If needed, you can imitate the reception of a message from outside by calling the <code>receive()</code> method, e.g., from another active object. This way you can implement “direct” delivery of messages bypassing the connection lines.

<pre>boolean onReceive(Object msg)</pre>	<p>The method is called before the <code>msg</code> message is received at a port. It can be overridden in a derived port class to perform additional message checking and some user-defined actions. Returning <code>true</code> means that the message must be received by this port and processed in default way; <code>false</code> - that the message should be discarded. By default the method returns <code>true</code>.</p>
<pre>static void connect(Port a, Port b)</pre>	<p>The method connects two ports, located on the same level of containment hierarchy, namely a port of encapsulated object to a private port, or ports of two encapsulated objects. Ports to be connected are specified as method parameters.</p>
<pre>void connect(Port port)</pre>	<p>The method connects two ports, located on the same level of containment hierarchy. The method connects this port to another one specified as a parameter.</p>
<pre>static void disconnect(Port a, Port b)</pre>	<p>The method disconnects two ports, located on the same level of containment hierarchy.</p>
<pre>void disconnect(Port port)</pre>	<p>The method disconnects ports, located on the same level of containment hierarchy, namely this port and the port specified as a parameter.</p>
<pre>void map(Port port)</pre>	<p>The method connects ports of active objects, located on the different levels in the containment hierarchy, namely a public port to a private port or a public port to a port of an encapsulated object. The method connects this port object, located on the upper level of containment hierarchy to another one specified as a parameter.</p>
<pre>void unmap(Port port)</pre>	<p>The method disconnects this public port and an internal port of the active object specified as a parameter.</p>
<pre>void map(Statechart statechart)</pre>	<p>The method connects this port to the specified statechart.</p>

<code>void unmap(Statechart statechart)</code>	The method disconnects the port and the statechart.
<code>void breakLinks()</code>	The method disconnects this port from all connected ports and statecharts.
<code>void setParams()</code>	The method sets parameters of the port. The default implementation does nothing. You can override the method <code>setParams()</code> to implement a desired behavior in your derived port classes.
<code>void setBreakpoint(boolean enable)</code>	The method enables or disables breakpoint on this port by passing <code>true</code> or <code>false</code> as a parameter.
<code>boolean isBreakpoint()</code>	The method checks whether the breakpoint is set on the port.
<code>ActiveObject getOwner()</code>	The method returns the active object in which the port resides.
<code>boolean wasActive()</code>	The method returns <code>true</code> whether a message was passed through it on the previous model step and <code>false</code> otherwise.
<code>String getName()</code>	The method returns the port name.
<code>String getPortFullName()</code>	The method returns full name of the port.
<code>void setPortInspect(String value)</code>	The method sets the specified inspect string for this port. If <code>null</code> is specified then a default inspect value will be used.
<code>String getDefPortInspect()</code>	The method returns the default port inspect string.
<code>String toString()</code>	The method returns the actual inspect string of the port.
<code>void setup(String name, ActiveObject owner)</code>	The method is called after the port has been created to setup port system data.

Table 11. Port class methods

7.3.1.2 PortQueuing class

The class is derived from the `Port` class and adds the functionality for message queue management.

The methods of the `PortQueuing` class are listed in Table 12.

Method	Description
<code>void setQueueSize(int size)</code>	The method sets queue size.
<code>Object getBase()</code>	The method extracts the message from the queue and returns it. The queue size is decreased by one. The queue must contain at least one message or an exception will be thrown.
<code>Object peekBase()</code>	The method returns the message from the port queue without removing it, thus the queue is not modified by this method. The queue must contain at least one message or an exception will be thrown.
<code>int size()</code>	The method returns number of messages currently pending in the port queue.
<code>Object waitForMessage()</code>	The method is used only in AnyLogic threads. It can be called from a thread to wait for a message arrival to the port. If there are any messages stored in the queue, the first one will be returned. Otherwise the thread will be suspended until a message arrives. When a message arrives, the thread execution is resumed and the message is returned.

Object waitForMessage(double timeout)	The method is used only in AnyLogic threads. It can be called from a thread to wait for a message arrival to the port. If there are any messages stored in the queue, the first one will be returned. Otherwise the thread will be suspended until a message arrives. When a message arrives the thread execution is resumed and the message is returned. If the time interval specified as a parameter is expired while no messages arrived to the port, null is returned.
--	---

Table 12. PortQueuing class methods

7.4 Message passing use cases

7.4.1 Filtering messages by message contents

Sometimes you may need to filter incoming messages – i.e., accept only messages meeting your requirements and discard other ones. In AnyLogic you can implement frequently needed message contents checking.

Message contents checking at a port can be implemented in the *On receive action* section of the port properties. You can specify your own contents checking code and ignore or accept the arrived message by writing the `return false;` or the `return true;` statement correspondingly. If you have written any code in the *On receive action* section and the `return` statement is omitted, the message will be discarded.

Let's examine the message contents checking implementation by the client-server model example. The server processes only valid client requests. The requests with some of the required fields unfilled as well as the outdated requests are ignored. The message validity time period is defined by the server's `msgLifetime` parameter of `double` type.

Server receives messages of the `InfoMsg` class with the data structure shown below.

Type	Name	Default
------	------	---------

Double	time
--------	------

String	name
--------	------

String	address
--------	---------

The message contents checking operations are specified in the *On receive action* property of the port:

```

if (msg.time + msgLifetime < getTime() || msg.name==null ||
msg.address==null)

    return false;

else {

    // process the request

    return true;

}

```

The outdated requests are detected by comparing the message lifetime with the current model time. The messages with the invalid timestamp value as well as the messages with unfilled required fields are discarded.

7.4.2 Filtering messages by message type

Sometimes you may need to filter incoming messages by message type – i.e., accept only messages of the certain type and discard other ones. It is commonly used when an object broadcasts messages of different types to a group of recipients and it is necessary to receive messages of one type only in certain ports, while the messages of another type in another ones.

Message type checking is implemented by specifying the type of messages that are accepted by the port in the *Message type* property of the port. Those messages that cannot be cast to the specified message type are discarded. If *Message type* is left blank, the predefined Java class `Object` is assumed – i.e. the port accepts all messages.

Note that message type checking is performed only for messages received at a port, but not for messages sent through a port.

Let's examine the example where the message type checking is needed. Suppose you have to model a car wash. A car wash has two different sections, one for serving automobiles and one for trucks. You need to prevent trucks delivering to the automobile section and vice versa.

In our model the car wash is represented by the `CarWash` active object with two ports, `trucksection` and `carsection`, representing two car wash sections. The trucks and cars are represented by two different message classes: `Truck` and `Car`. We need to implement message type checking at ports to receive messages of the `Truck` type only at the `trucksection` port, while the messages of the `Car` type - at the `carsection` port. Thus, we need to specify these message classes in the *Message type* properties of the ports as shown in Figure 81.

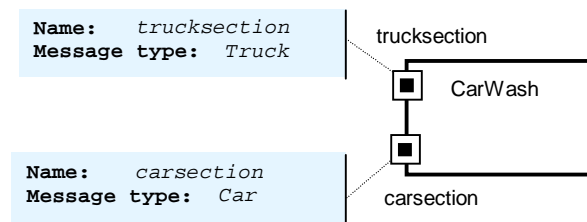


Figure 81. CarWash ports

The model works as shown in Figure 82. In the first case the `Car` message is accepted at the `carsection` port and discarded at the `trucksection`. And vice versa, in the second case the `Truck` message is received at the `trucksection` port and ignored at the `carsection` one.

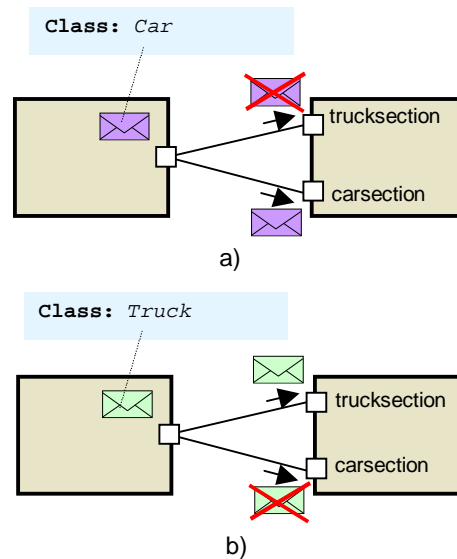


Figure 82. Message filtering by type

7.4.3 Sending a message with a delay

According to the time semantics of the message passing, once sent, the message gets to all destination ports immediately. However, it is frequently needed to delay message sending through a port for a specified period of time. In the case of entity flow, delay is typically used to model time spent on processing an entity at active object – e.g., for serving customers in service systems or for producing and processing parts or products in manufacturing models.

The delayed message passing can be implemented in different ways: you can model delay by using dynamic timers (see Chapter 8, “Timers”) or timed transitions in statecharts (see Chapter 9, “Statecharts”). However, using timers is more efficient.

You can implement delayed message sending in the following way: you create a dynamic timer that exists for a specified period of time, modeling message processing delay. When this time elapses, the timer expires and the message is sent via the port.

Therefore, define the `sendDelay()` method in the *Additional class code* property of the port. The method takes a message and a delay as parameters and passes them to the created instance of `SendTimer` timer class that you should define in your active object class.

```
void sendDelay( Object msg, double delay ) {
```

```

        new SendTimer( delay, msg );
    }

```

Type the following code in the *Additional class code* code section of the `SendTimer` class:

```

SendTimer( double delay, Object msg ) {
    super( delay );
    this.msg = msg;
}
Object msg;

```

This code defines the new timer constructor and stores a reference to a message that should be sent in the class member variable `msg`. The line `super(delay);` invokes base timer class constructor. The created dynamic timer schedules message sending on its expiration. This is defined by writing the following line of code in the *Expiry action* property of the timer:

```
port.send(msg);
```

Thus, you have implemented delayed message sending. Call the `sendDelay()` method of the port to send a message with a delay. However, you can simply call the `send()` method of the port to send a message immediately.

Since delayed message sending is frequently needed, you can create your own port class and place it in an external file for the reuse in other models as well. Then the class code should look like the following:

```

public class PortDelay extends Port {
    public void PortDelay(){
        super();
    }
    void sendDelay( Object msg, double delay ) {
        new SendTimer( delay, msg );
    }
    class SendTimer extends DynamicTimer {
        public void action() {
            port.send( msg );
        }
    }
}

```

```

    }

    SendTimer( double delay, Object msg ) {

        super( delay );

        this.msg = msg;

    }

    Object msg;

}

}

```

7.4.4 Sending a message with a delay dependent on the message field

In section 7.4.3, “Sending a message with a delay” you learned how to model message sending after a specified delay. It is a common situation when this delay represents the time spent on message processing at the active object. In the case of entity flow, delay commonly models entity processing time at active object and may depend on some characteristics of the processed entity. For instance, in manufacturing models, part processing time at a conveyor depends on the size of the part; in network models packet transfer time through a channel is proportional to the size of the sent packet. You can define a custom message field to carry the parameter affecting the message processing time and delay message sending proportional to its value.

You can model message sending delay dependent on the message field by implementing the delayed message sending mechanism, described in section 7.4.3, “Sending a message with a delay”. Actually, you should do a pair of modifications.

First, you need to create new message class `Message`, with the `attribute` field of type `double`, representing the characteristics of the passed entity, affecting the entity processing time.

Then you should define a new method in the *Additional class code* property of the port:

```

void sendDependentDelay(Message msg) {

    sendDelay(msg, msg.attribute);

}

```

Calling the `sendDependentDelay()` method with the message passed as a parameter, you can delay message sending via a port on the period of time equal to the message's `attribute` field value.

7.4.5 Getting information on connected objects

Sometimes you may need to get information about objects connected to a specific port. Thus, you can check your system topology. The list of connected objects can also be used as an object addresses list in message routing when you need to send messages only to the certain objects rather than to all of them (see section 7.4.7, "Sending a message to a specific recipient / a group of recipients").

Connected objects registration can be implemented in different ways. The following is the simplest: when created, an active object sends a message with identification information. At the destination port the identification information is extracted from the registration message and added to the list of connected objects.

The object identifiers must be unique to distinguish objects. Actually, you can use various data as an object identifier. You can define a class parameter and adjust its value for each active object. Actually, you can define a parameter of any type, supported by AnyLogic, e.g. numeric (`int`, `double` etc.), symbolic (`String`) etc. There is, however, one inconvenience: you will need to ensure the assigned identifiers' uniqueness. A common situation when the same identifier was in error assigned to several objects is shown in Figure 83.

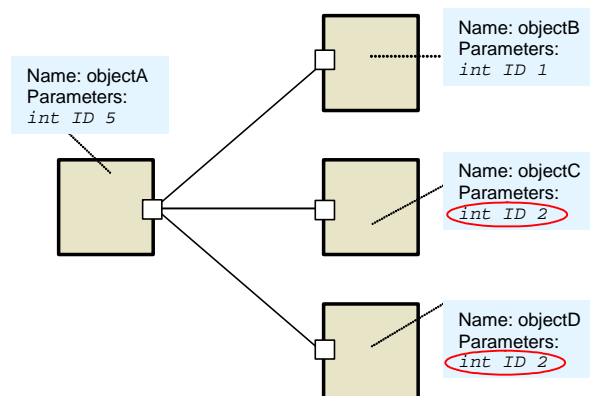


Figure 83.

So you need to avoid such situations and provide identifier uniqueness. Since active objects are standard Java objects, you can use one of the means described below as object identifiers:

- a unique integer value returned by the object method `hashCode()`,
- a unique object name in your model returned by the active object method `getName()`,
- a unique reference to the object returned by `this` command.

An object may have multiple ports. In this case you will need to make up a list of connected ports, not a list of connected objects to distinguish messages sent from the different ports of the same object. Since ports in AnyLogic are also Java objects, you can use any of the three described above approaches for defining unique port identifiers (use the port method `getPortFullName()` instead of the `getName()` method to get the port name).

Let's examine the example where the object registration is necessary. Suppose you have to model a distribution center shipping goods to the customers and you need to keep the customer list.

The model consists of a distributor object and a number of customer objects. Customers send requests to the distributor via input ports. Goods are sent through the distributor's output port and delivered to customers.

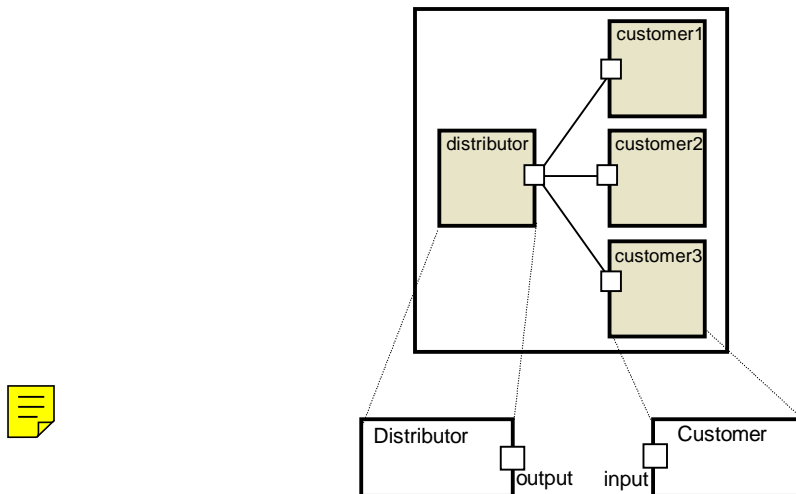


Figure 84. The goods distribution model

Let's implement customer registration at the distribution center.

First, you need to decide what kind of data to use as an object identifier. Let's choose a reference to the object.

Define the `RegMsg` message class for registration messages. The class has the `source` field of the `ActiveObject` type used to transfer the source object identifier.

Specify the `registerCustomer()` method in the *Additional class code* property of the customer's input port. This method sends the registration message, carrying the customer identifier (the reference to the active object is accessed by the method `getOwner()` of the port).

```
void registerCustomer() {
    RegMsg msg = new RegMsg ();
    msg.source = getOwner();
    send( msg );
}
```

Type the following line in the *Startup code* code section of the `Customer` class to call the `registerCustomer()` method upon the active object creation:

```
input.registerCustomer();
```

Define the `customers` member variable of the `Vector` type in the *Additional class code* property of the output port to store information on registered customers.

```
Vector customers = new Vector();
```

Type the following code in the *On receive action* property of the output port to check the type of arrived message. In the case it's a registration message, the message sender is added to the list of connected objects. Otherwise, the message is processed according to the logic of your model.

```
if( msg instanceof RegMsg ){
    RegMsg rm = (RegMsg)msg;
    customers.add( rm.source ); //add unknown customer
}
else {
    //it's not a registration message
```

```

}
return true;

```

Finally, your model should look like as shown in Figure 85:

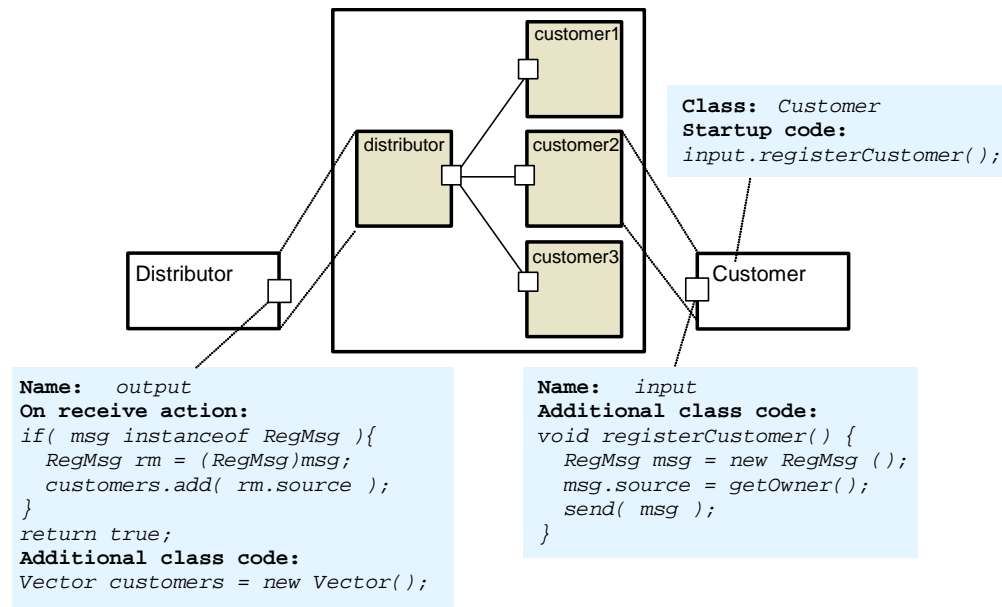


Figure 85. Customer registration at the distributor center

Thus, you have implemented customer registration at the distribution center. Namely, the references to all objects connected to output port are stored in its `customers` member variable. You can use this list to route messages to the certain addresses from the list, see section 7.4.7, “Sending a message to a specific recipient / a group of recipients”.

You can use the described mechanism to collect any information about objects you like by passing the required data in a custom field of a registration message.

7.4.6 Sending messages to all recipients

Sending messages to all the recipients is the default port behavior in AnyLogic. It is employed when a source object has multiple connected objects and needs to send some general information or a command to all of them.

All messages sent via a port are automatically forwarded to all the connected ports, as shown in Figure 86. Namely, the message sent through the portA port is received at recipA, recipB and recipC ports.

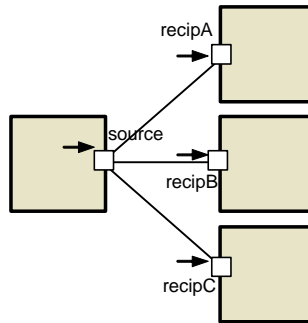


Figure 86. Sending a message to all recipients

It is important to understand that all the recipients get references to the same Java object, representing a message. Therefore, if any of them modifies the message, this affects all other recipients as well. Thus, to avoid sharing problems, you have to clone received messages explicitly, as described in section 7.2.2, “Cloning messages to avoid sharing violation”.

7.4.7 Sending a message to a specific recipient / a group of recipients

In AnyLogic, the sent message arrives to all ports connected to the sender port. However, you may need to send a message only to a specific recipient or to a specified group of recipients, rather than to all of them. You can send messages to specific object(s) by defining object addresses and marking outgoing messages with destination addresses. At the destination ports, messages are filtered by this address – i.e. only messages addressed to this certain recipient are accepted.

Obviously, the sender should have a list of recipient addresses to route messages only to the certain recipients. There are several ways of making up this list. Somewhat simpler is hard-coded defining of a connection list at the model design time. Figure 87 shows the example of

hard-coded object registration: both the ID port identifier values and the `connList` connection list are defined at the model design time.

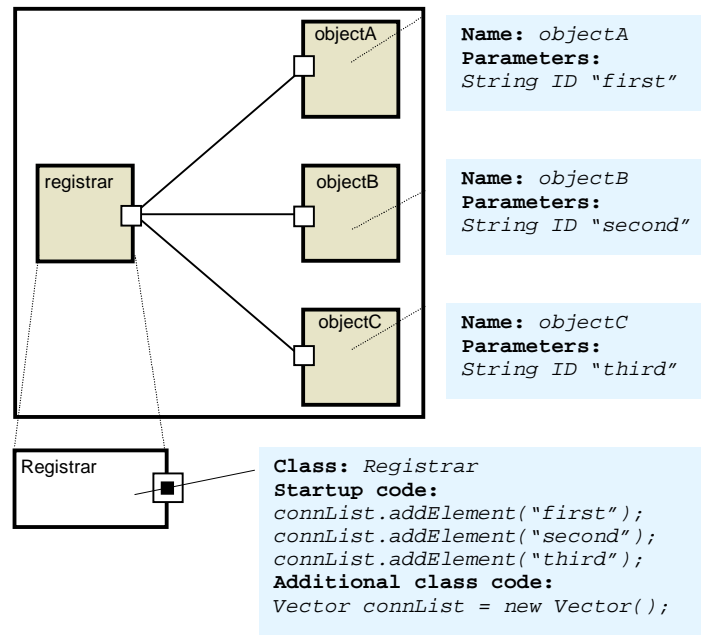


Figure 87. Making up connection list at the model design time

Commonly, this approach is inappropriate for the case of scalable models, since you will need to modify your connection list on every object added. In this case you can build a connection list programmatically at runtime. This can be done by sending identity messages with the source object addresses at the model startup. At the destination port the identification information is extracted from the message and added to the list of connected objects. This mechanism is described in section 7.4.5, “Getting information on connected objects”.

Let’s examine the example where sending messages to specific recipients is necessary. Suppose you have to model a post office. A post office delivers letters to the specified recipients according to their addresses. Periodicals are delivered to specified groups of subscribers. The bulletins with the general information on the new post office services are sent to all known addresses.

This model is shown in Figure 88. It consists of the `postOffice` object and a set of recipient objects. Messages are sent through `postOffice`’s output port and delivered to recipient’s input ports.

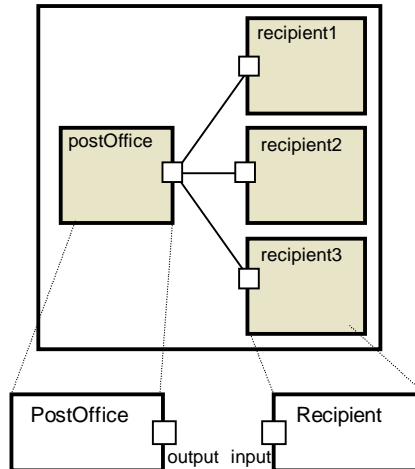


Figure 88. Post office model

First, implement the mechanism described in section 7.4.5, “Getting information on connected objects” Use the reference to the sender port as the port address.

Define the registration message class `RegMsg`, which has `source` field of the `Port` type to carry the recipient address to the post office.

Type	Name	Default
Port	source	

Define the `MailMsg` message class for post office mail. The `MailMsg` class has the `mail` data member of the custom `Mail` class representing the mail itself and the `address` field to carry the mail destination address.

Type	Name	Default
Port	address	
Mail	mail	

Declare the `addresses` member variable of the `Vector` type in the `output` port’s *Additional class code* to keep information on registered addresses at the post office.

```
Vector addresses = new Vector();
```

Specify the `registerAddress()` method in the input port's *Additional class code*. This method sends a registration message, carrying the recipient port address to all objects connected to this port.

```
void registerAddress() {
    RegMsg msg = new RegMsg();
    msg.source = this;
    send(msg);
}
```

Type the following line in the *Startup code* code section of the Recipient class to send a registration message to the post office upon the active object creation:

```
input.registerAddress();
```

Type the following code in the *On receive action* property of the output port to check the type of the arrived message. If it's a registration message, the message sender address is added to the address list. Otherwise, the message should be processed according to the logic of your model.

```
if( msg instanceof RegMsg ){
    RegMsg rm = (RegMsg)msg;
    addresses.add( rm.source ); //add unknown address
}
else {
    //it's not a registration message
}
```

Now the post office has an addresses list and you can send messages to the certain address by calling the method `sendTo()` of the `PostOffice` class, providing the destination port address and the message as the parameters.

```
void sendTo(MailMsg msg, Port destination){
    msg.address = destination;
    output.send(msg);
}
```

For instance, write the following code to deliver mail to the last registered address:

```
sendTo(msg, output.addresses.get(addresses.size()-1));
```

You need to implement message broadcasting in your system to send bulletins with information on new post office services to all known addresses. This can be implemented by sending the message with `null` destination value. Therefore, you should call:

```
sendTo(msg, null);
```

Type the following code in the input port *On receive action* to perform message filtering by its destination address:

```
if (msg.address == this || msg.address == null ) {  
    return true;    //accept message  
}  
else {  
    return false;  //do not accept message  
}
```

As shown above, the port checks the destination address of the arrived message. If the destination address is `null`, which in our implementation means broadcast, or if it is equal to this port address, the port accepts the message. Otherwise the message is discarded.

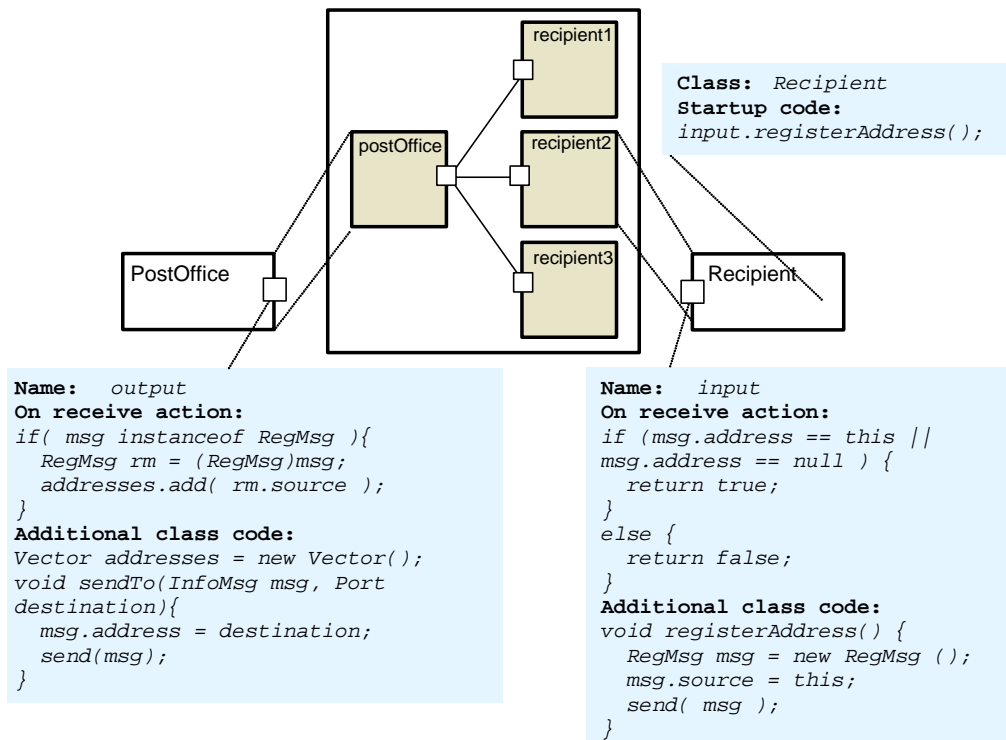


Figure 89. Sending messages to specified addresses

You have implemented message sending to the specific recipients in your system: sent messages are accepted only by the recipients with the specified destination addresses.

7.4.7.1 Sending a message to a specific group of recipients

Now you need to implement periodical delivery to the specified groups of subscribers. You need to keep subscriber addresses lists at the post office and deliver periodicals only to members of these lists. The post office announces new periodical and a recipient subscribes on it by sending a confirmation message on announce reception.

Define the `SubscrMsg` message class to represent periodical announces. The `periodName` member variable carries the periodical name. The `address` member variable is used to carry the subscriber destination address in the confirmation message.

Type	Name	Default
Port	address	

String	periodName
--------	------------

Define the `PeriodicalMsg` message class to represent periodicals. The `addresses` member variable is used to carry the subscriber addresses list. The `periodical` member variable represents the periodical itself.

Type	Name	Default
Mail	periodical	
Vector	addresses	

Type the following code in the *Additional class code* property of the output port.

```
String timesName = new String("Times");
String USATodayName = new String("USA Today");
Vector timesSubscribers = new Vector();
Vector USATodaySubscribers = new Vector();
void announcePeriodical( String periodicalName ){
    SubscrMsg msg = new SubscrMsg();
    msg.periodName = periodicalName;
    send(msg);
}
```

The `times` and `USAToday` member variables keep information about “Times” and “USA Today” subscribers. The `timesName` and `USATodayName` member variables define periodical names. The `announcePeriodical()` method announces a subscription on a periodical. The periodical name is specified as the method parameter.

Substitute the code in the *On receive action* properties of input ports with the following code:

```
if ((msg instanceof MailMsg)&&((msg.address==this)|| (msg.address==null) )){
    //process a mail addressed to this recipient or to all recipients
    return true;
}
else if ( msg instanceof SubscrMsg ) {
```

```

if (subscribe){
    msg.address = this;
    send(msg); // subscribes to the periodical if subscribe is true
}
return true;
}
else {
    PeriodicalMsg message = (PeriodicalMsg)msg;
    if message.addresses.contains(this){
        // process periodical sent to this subscriber
        return true;
    }
}
}

```

As shown above, if the recipient wants to subscribe to the announced periodical (it is defined by its `subscribe` member variable of boolean type), it returns the message to the post office in response to `announce`.

Substitute the code in the *On receive action* property of the output port with the following code:

```

if( msg instanceof RegMsg ){
    RegMsg rm = (RegMsg)msg;
    addresses.addElement( rm.source ); //add unknown address
}
else {
    SubscrMsg sm = (SubscrMsg)msg;
    if (sm.periodName.equals(timesName))
        timesSubscribers.addElement(sm.address);
    else if (sm.periodName.equals(USATodayName))
        USATodaySubscribers.addElement(sm.address);
}
}

```

Now the subscription messages are processed at the output port. The subscriber addresses are added to the corresponding address lists.

Specify the `sendToSubscribers()` method in the *Additional class code* property of the output port.

```
void sendToSubscribers( Vector subscribers, PeriodicalMsg periodical ){
    msg.addresses = subscribers;
    port.send(msg);
}
```

Call this method to send periodicals to the specified group of subscribers, providing the subscriber address list and the message as method parameters. For instance, to deliver new “USA Today” issue to the subscribers, you should write:

```
PeriodicalMsg issue = new PeriodicalMsg();
sendToSubscribers( USATodaySubscribers, issue );
```

7.4.7.2 Receiving a message only from a specific object or a group of objects

According to the message routing rules, ports accept every incoming message. However, it is frequently needed to receive a message only from a certain object or a group of objects, rather than from all of them.

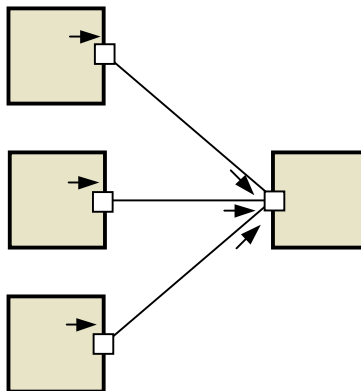


Figure 90. Default message reception

This task is bisymmetrical to the task described in section 7.4.7, “Sending a message to a specific recipient / a group of recipients”. Thus the implementation concept is the same: you need to define unique port identifiers to mark outgoing messages and filter messages at a recipient port by sender identifiers. Therefore, you need to define the message class with a custom field to carry the sender identifier and specify the list of authorized senders at the recipient object.

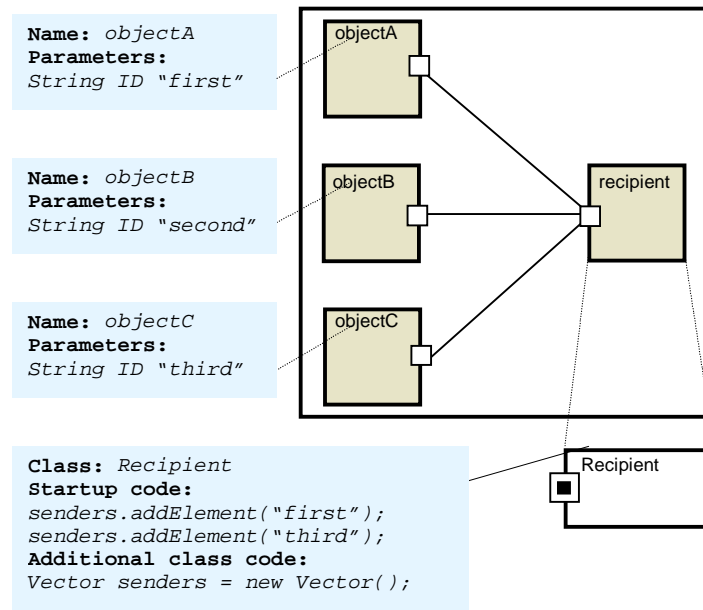


Figure 91. Making up authorized senders list

Again, as described in section 7.4.7, “Sending a message to a specific recipient / a group of recipients” you can build this list manually at the model design time as shown in Figure 91 or programmatically at runtime by implementing object registration mechanism described in section 7.4.5, “Getting information on connected objects” at the recipient object. Figure 92 shows one of possible implementations.

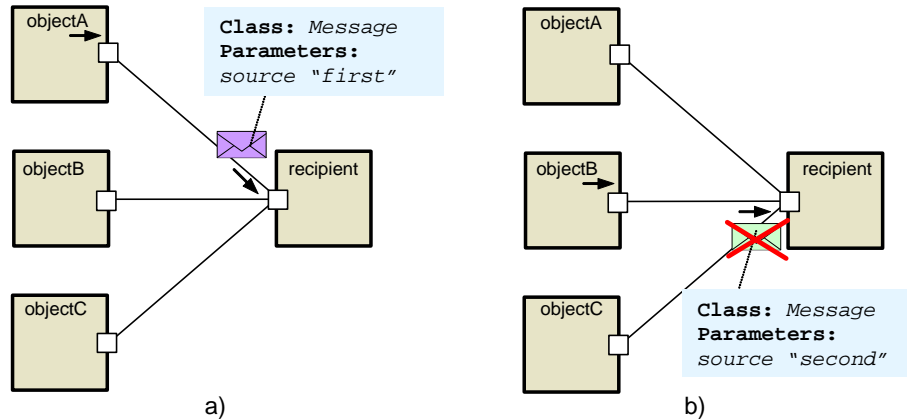


Figure 92. Message filtering

7.4.8 Verifying port connections at runtime

Sometimes you may need to verify port connections in your model. Port connection verifying is commonly used when your model has several port types and it's essential to allow message passing only between ports of certain types. When editing your model you can establish connection with a port of forbidden type that causes a logic error. Therefore, you need to detect an invalid connection at runtime by implementing your own checking mechanisms. You can define specific actions to be performed on invalid connection detection, the most obvious is showing an error message using runtime error ability.

Suppose you need to detect invalid port connections in the simple manufacturing model. In this model a raw material supplier supplies product processing machines with raw material. After processing at a processing machine, a resulting product is delivered for the storage to a warehouse.

The model structure is shown in Figure 93. Raw material supplier is represented by `supplier` object, machine - by `machine` object, warehouse - by `warehouse` object.

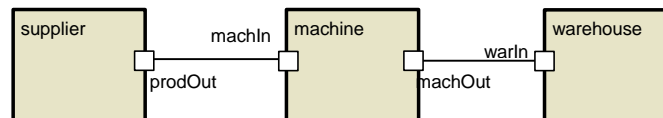


Figure 93. Simple manufacturing model

Constructing the model from a great number of objects, you may establish invalid connections. Suppose you have made a mistake when editing your model – e.g., you have connected two output ports in error. In this case you can hardly detect the error: no error message will be displayed while your model will work incorrectly.

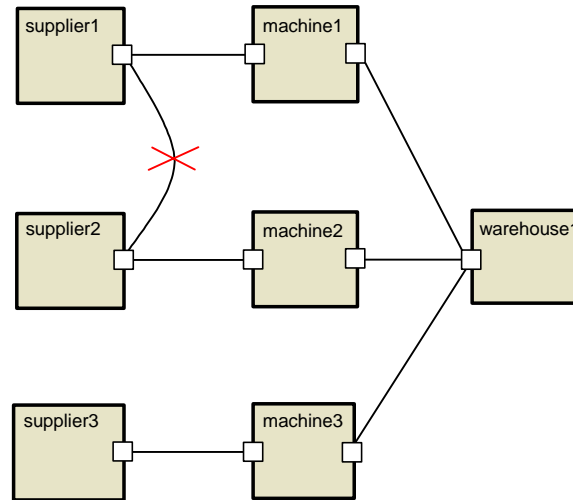


Figure 94. Invalid port connection in the manufacturing model

If you have well-defined input and output ports in your model, you can detect invalid connection of two output ports by message arrival to an output port. Type the following code in the *On receive action* property of the output port to terminate the model simulation and show the error message on message reception at the port:

```
Engine.error("Error: The output " + this.getFullName() + " port has invalid
connection. "); //terminates simulation
```

Generally, you should implement a more sophisticated mechanism to check proper material flow in this model. According to the model logic, an entity sent by a supplier should be sent to a machine first and then to a warehouse. Thus, only the `prodOut` ports can be connected with the `machIn` ports and the `machOut` ports can be connected only with the `warIn` ports.

You can define a dedicated message class for each port class. Then each port class will send only the messages of the specific type and you can verify port connections by checking incoming message types.

In our model each entity is represented by its own message class: raw material is represented by a message of the `Material` type and a product – by a message of the `Product` type. The

messages of the `Material` type should be received only at the `machIn` ports. Similarly, the messages of the `Product` type should be received only at the `warIn` port.

The checking procedure is specified in the *On receive action* property of the port. Type the following code to detect invalid connections of the `machIn` port:

```
if( ! (msg instanceof Material) ){
    Engine.error("Error: The " + this.getFullName() + " port has invalid
    connection. "); //terminates simulation
}
return true;
```

This code checks incoming message types. Only messages of `Material` type sent by `prodOut` ports are accepted. When a message of another type arrives to the port, the error message is shown.

Another approach consists in transmitting a reference to the sender port along with the message. Having got a reference to a port you can detect a connection with a port of forbidden type by checking the sender's port type.

Let us demonstrate you how to implement this mechanism. Suppose you have ports of `PortA` and `PortB` types in your model and `PortA` ports can be connected only with the `PortB` ones. You need to detect invalid connections of the ports of the same type.

Create `Message` message class with the `source` field of type `Port` to carry a reference to message sender port. Specify `Message` message class in the *Message type* property of the ports.

To send a message via the `port` port, you should write the following code in the active object code:

```
Message msg = new Message();
msg.source = port;
send( msg );
```

The sent message carries the reference to the sender port.

The checking procedure is specified in the *On receive action* of the port. For instance, write the following code for the ports of `PortA` type to allow connections only with the `PortB` ports:


```

if (!(msg.source instanceof PortB))

    Engine.error("Error: The " + this.getFullName() + " port is erroneously

    connected to the " + msg.source.getFullName() + " port." );

}

return true;

```

This code checks the sender's port type. The message from non `PortB` port reception indicates the invalid port connection and the error message is shown.

7.4.9 Defining message class constructors

You can initialize the created message data differently, depending on some external conditions. In AnyLogic you create a message instance by calling a message class constructor. The default constructor automatically generated by AnyLogic fills out all the fields of the created message instance in the same order you defined in the *Parameters* table. However, you may need to create the message with only a certain set of fields. For example, in the distributed database model, the data structure of the server response message depends on the type of the client query. You can define custom message class constructors to create messages with only those message fields that are necessary under the certain conditions.

For example, you need to model an elevator. A user controls the elevator by sending commands to the controller. A command may take a parameter or not, thus the data structure of the created message varies depending on the type of the command. Specifying your own constructors, you can create the command message with certain fields only.

Commands are represented by the messages of the `Command` class. The message class has the `cmd` member variable of type `String` representing the command instruction and the `arg` member variable representing the optional command argument.

Type	Name	Default
String	cmd	
Double	arg	

The “Move” command takes the parameter, specifying the movement step. In this case the default constructor is used to create the message.

The default constructor creates a new message with all the fields initialized:

```
Command ( String cmd, double arg ) {
    this.cmd = cmd;
    this.arg = arg;
}
```

However, the “Stop” command does not take any parameters. Thus you can define the custom constructor to create a command message without any parameters. The message class constructor is defined in the *Additional class code* section of the message class properties:

```
Command ( String cmd ) {
    this.cmd = cmd;
}
```

Calling different constructors you can create messages with different data structure. For example, the following lines of code create messages, representing “Move” and “Stop” commands correspondingly.

```
Command movecmd = new Command( "MOVE", 20.0 );
Command stopcmd = new Command( "STOP" );
```

In the second case, the custom constructor is called to create the message with the cmd field only.

Note that Java statements you specified in the *Constructor code* section of the message class properties will be inserted in the default constructor only. Thus this code will be executed only in the case of creating a message instance by the default constructor.

7.4.10 Modeling a LIFO queue

The default port queue in AnyLogic is the FIFO queue. However, you may need to model a LIFO queue – e.g., to model a LIFO buffer in queuing models. Messages are extracted from the LIFO queue in the Last-In First-Out order – i.e., the last placed in the queue message is extracted first as shown in Figure 95.

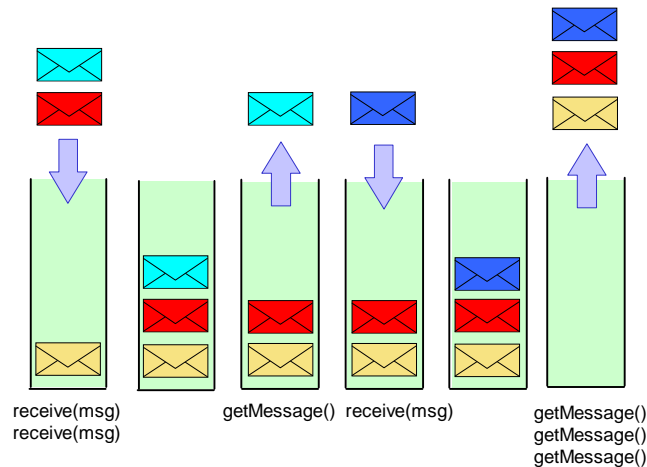


Figure 95. LIFO queue in a port

LIFO queue can be implemented by defining your own container for incoming messages in the port and providing a method to extract stored messages in LIFO order.

First, define the messages data variable of the `Vector` type in the *Additional class code* of the port to store incoming messages:

```
Vector messages = new Vector();
```

Type the following code in the *On receive action* property of the port to place the arrived message in our message container, representing a LIFO queue:

```
messages.add( 0, msg );
return false;
```

As shown above, the incoming message is placed at the head of our queue. Note that returning `false` we prohibit the default processing of the message to process it in a custom way. Thus received messages will neither be forwarded further nor stored in the default port queue.

You can extract the last arrived message from the LIFO queue by calling the `getMessage()` method, specified in the *Additional class code* property of the port:

```
Object getMessage(){
    return messages.remove(0);
}
```

Similarly, define the `peekMessage()` method to access the message without removing it from a queue:

```
Object peekMessage(){
    return messages.get(0);
}
```

Note that the extracted message is the `Object` class instance and you may need to cast it explicitly to the original message class.

For instance, we send messages of the `Message` type. Then we need to cast the extracted message instance to this class in the following way:

```
Message message = (Message)getMessage();
```

Generally, you will need to check the port queue size before accessing the messages from the queue:

```
if (messages.size() > 0)
    Message message = (Message)getMessage();
```

If needed, you can implement a limited queue. First, specify the capacity of the queue by defining `capacity` member variable in the *Additional class code* of the port:

```
int capacity = 44;
```

Queue would not accept an incoming message if the capacity is reached.

Substitute the code in the *On receive action* with the following code to perform the current queue size checking and discard messages if the queue is full.

```
if ( messages.size() < capacity )
    messages.add( 0, msg );
return false;
```

You can further change the queue behavior. For instance, you may want to store the latest messages in your queue. So when the message arrives to the port with a full queue you can remove the oldest message stored. Therefore, you should substitute the code in the *On receive action* with the following:

```
messages.add( 0, msg );
```

```

if ( messages.size() > capacity )
    messages.remove( messages.size() - 1 );
return false;

```

7.4.11 Modeling a priority queue

The default port queue in AnyLogic is FIFO queue. Messages are extracted from the FIFO queue in the First-In First-Out order – i.e., the first placed in the queue message is extracted first. You can model a priority queue in a port. It may be needed, for instance, in queuing models, to model a priority buffer. Priority queue can be implemented by defining your own message container in the port and placing incoming messages in this container regarding to the message priority.

First, define the `messages` member variable of the `Vector` type in the *Additional class code* of the port to store incoming messages:

```
Vector messages = new Vector();
```

Next, define the message class `PriorityMsg`, which has the `priority` field of type `int`, representing the message priority. The lower values represent higher priorities, 0 represents the highest priority.

Type the following code in the *On receive action* property of the port to place the arrived message in the queue, regarding to the message priority: the messages with the highest priority are placed in the head of our queue to be extracted first.

```

int i =0;

while(i<messages.size() && msg.priority >=
((PriorityMsg)messages.get(i)).priority)

    i++;

messages.add( i, msg );

return false; // do not store in a default queue

```

Note that returning `false` we prohibit the default processing of the message to process it in a custom way. Thus received messages will neither be forwarded further nor stored in the default port queue.

Extract the message with the highest priority from the priority queue by calling the `getMessage()` method, specified in the *Additional class code* property of the port:

```
PriorityMsg getMessage(){
    return (PriorityMsg)messages.remove(0);
}
```

Similarly, define the `peekMessage()` method to access the message without removing it from a queue:

```
PriorityMsg peekMessage(){
    return (PriorityMsg)messages.get(0);
}
```

Generally, you will need to check the port queue size before accessing messages from the queue:

```
if (messages.size() > 0)
    PriorityMessage message = getMessage();
```

7.4.12 Connecting ports at runtime

You can connect ports programmatically at runtime by calling methods `connect()` and `map()`. To disconnect ports, you call the methods `disconnect()` and `unmap()`. Thus you can easily create complex systems with sophisticated topologies. You can change port connections at runtime to model systems with dynamically changing connections, in particular, systems with mobile objects. The example of a system with dynamically changing connections is given in section 7.4.13, “Modeling a system with dynamically changing structure”.

Use the `connect()/disconnect()` methods to connect or disconnect ports, located on the same level of containment hierarchy and the `map()/unmap()` methods otherwise.

In other words, use the `connect()/disconnect()` methods to connect or disconnect:

- Ports of encapsulated objects
- A private port with a port of an encapsulated object

Use the `map()` / `unmap()` methods to connect or disconnect:

- A public port with a port of an encapsulated object
- A public port with a private port
- A port with a statechart

Using the `connect()`/`disconnect()` methods

Figure 96 and Figure 97 show the situations when the `connect()`/`disconnect()` methods are used.

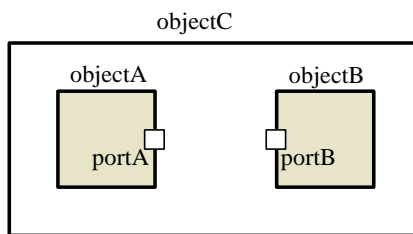


Figure 96. Ports of encapsulated objects

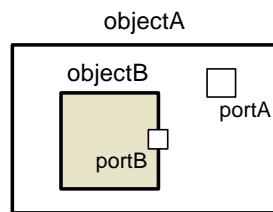


Figure 97. A private port and a port of an encapsulated object

In both cases ports are located on the same level of the containment hierarchy.

Call the `connect()` method to connect `portA` with `portB`:

```
Use Port.connect( portA, portB ); statement, or  

portA.connect( portB );
```

Call the `disconnect()` method to disconnect ports in a similar manner:

```
Use Port.disconnect( portA, portB ); statement, or
portA.disconnect( portB );
```

Note that it does not really matter in what order the ports are specified. You can write the following line as well:

```
portB.disconnect( portA );
```

You can write this code anywhere you like in the parent active object class (objectC in the first case and objectA in the second one).

Using the map()/unmap() methods

The method map() is used for connecting ports located on the different levels of the containment hierarchy, namely a public port with a private port or a public port with a port of an encapsulated object. Ports are disconnected by calling the unmap() method.

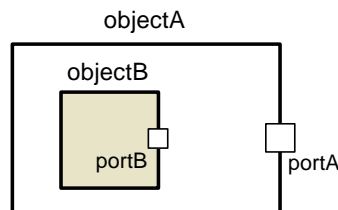


Figure 98. A public port and a port of an encapsulated object

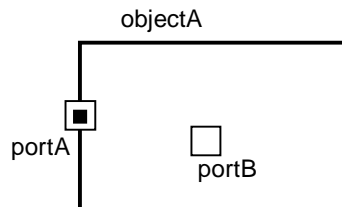


Figure 99. A public port and a private port

Figure 98 and Figure 99 show the situations when the map()/unmap() methods are used.

To connect these ports, call the `map()` method of the public port (`portA`) with the internal port (`portB`), specified as a parameter anywhere you like in the parent active object class (`objectA`):

```
portA.map( portB );
```

To disconnect these ports, call the `unmap()` method:

```
portA.unmap( portB );
```

Note that it is significant that the method `map()/unmap()` of the public and not of the internal port is called. The following line of code is invalid:

```
portB.map( portA );
```

You can connect or disconnect a port with a statechart at runtime. The method syntax is the same both for public and private ports:

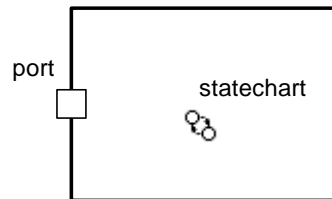


Figure 100. A public port and a statechart



Figure 101. A private port and a statechart

Connect a port with a statechart by writing

```
port.map( statechart );
```

Disconnect a port with a statechart by writing

```
port.unmap( statechart );
```

7.4.13 Modeling a system with dynamically changing structure

AnyLogic enables you to model not only systems with static structure, but also systems where object interconnections change dynamically. Thus you can easily create complex models with flexible structures. AnyLogic is the only visual tool that supports creation of truly dynamic models – the ones with dynamically evolving structure and component interconnection.

You can model systems with dynamically changing structure by changing port connections at runtime. To connect ports at runtime use the port methods `connect()` and `map()`. To disconnect ports, use the methods `disconnect()` and `unmap()`. See section 7.4.12, “Connecting ports at runtime” for more details on using these methods.

Suppose you are modeling an object moving in 1D space along the array of sensor stations. The object emits signals, which are accepted by the nearest station.

The model is shown in Figure 102. The mobile object is represented by wanderer object. Sensor stations are represented by the replicated object sensors.

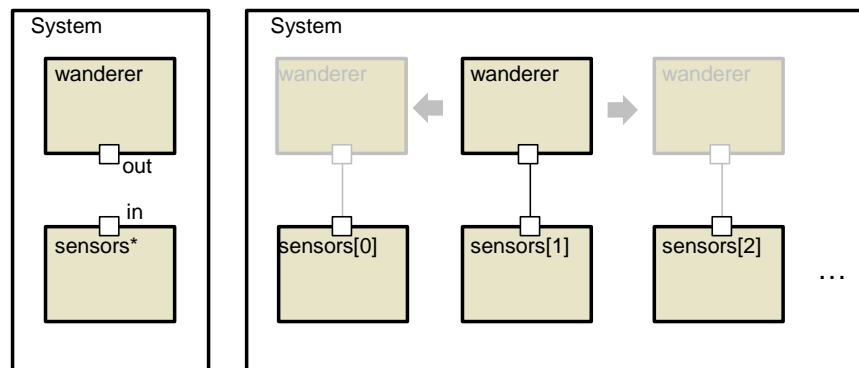


Figure 102. A wandering object

The active object `wanderer` sends messages through its output port `out`. These messages are accepted by the sensor that is currently connected to the `wanderer`. At random time moments, the class `System` moves the `wanderer`.

You can implement this as described below. Type the following code in the *Additional class code* section of the class `System` properties:

```
int x = 0;

public void onCreate() {

    wanderer.out.connect( sensors.item( x ).in );

}

void move() {

    wanderer.out.disconnect(sensors.item(x).in); // wanderer is disconnected

    x ++;

    x = x % sensors.size(); // wanderer location changed

    wanderer.out.connect(sensors.item(x).in); //wanderer is connected

}
```

The `x` member variable defines the wanderer location.

The initial location of the wanderer is defined in the method `onCreate()` that is called at the model startup. Initially wanderer is located near the first sensor station. So it is connected via its out port with the in port of the first element of the `sensors` replicated object. The individual element of a replicated object `sensors` is accessed by the method `sensors.item(index)`. See Chapter 2, “Replicated objects” for more details on replicated objects.

The wanderer displacement is implemented by the method `move()` of the class `System`. Namely, the out port of the `wanderer` object is disconnected from the in port of the current element of `sensors` object and connected to the in port of the next element of `sensors` object.

7.4.14 Implementing instantaneous data feedback

In AnyLogic when an object needs to send some data to another object, it sends a message carrying this data object. You can instantaneously modify the original data stored at the sender object on this message reception at a recipient port. Thus you can implement instantaneous data feedback between active objects.

The data feedback technique is the following. Since the data is represented by a Java object, and the recipient receives the reference to this object, when the recipient modifies this data object, it affects the original data object stored at the sender as well. Thus you can instantaneously modify the sender's data sent within a message without sending any response message, carrying modified data back to the sender.

For instance, you send the message with the information in the `number` member variable.

```
DataMsg dm = new DataMsg();  
dm.number = 38;  
traceln("Original number " + dm.number );  
port.send(msg);  
traceln("Modified number " + dm.number );
```

You can instantaneously change the value of the sent data object by changing the value of the passed object in the recipient port's *On receive action* property:

```
DataMsg data = (DataMsg)msg;  
data.number=35;  
return true;
```

The `dm.number` value stored at the sender object will be changed immediately (it is evidenced with the original and modified values, printed in the AnyLogic Log window). You do not need to send a supplementary message, carrying the new value to the message sender.

7.4.15 Implementing instantaneous message exchange

You can implement instantaneous message exchange between active objects by sending a response message on a message reception. It is important that at the time the initial message sending is finished all the responses from the recipients are already received. Thus, you can perform some helper operations, e.g. pass messages with some information about the current state of recipient or implement object polling.

Instantaneous message exchange is implemented by specifying response message sending back to the message sender in the port's *On receive action* property. You can perform several message exchanging iterations between objects. The message exchange is performed by a

single message sending act, since code, specified in the *On receive action* property of the recipient port is executed amid the sender method `send()` execution.

Let's examine the message exchange in the model shown in Figure 103.

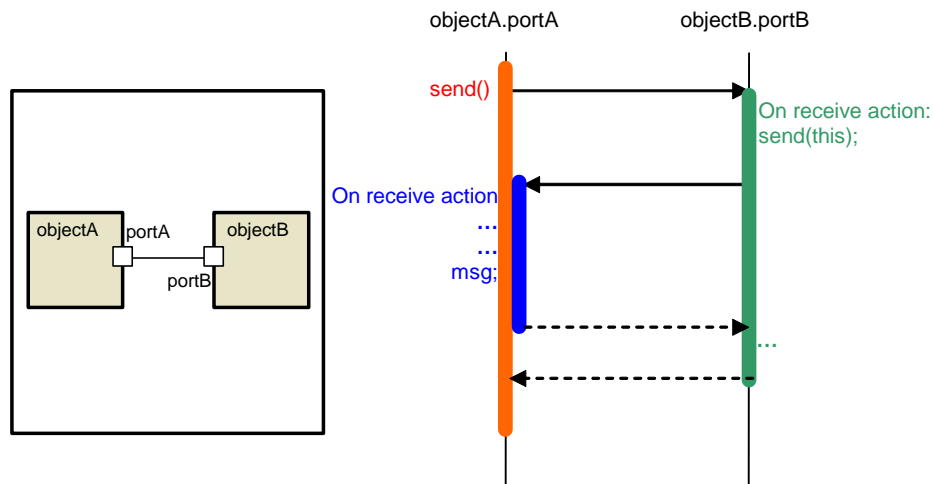


Figure 103. Immediate reply. Method call sequence

Initially `objectA` sends a message to `objectB`.

```
portA.send(msg);
```

The message is received at the `portB` port and the code specified in the `portB` *On receive action* property is executed. In particular, the `portB` sends back a message carrying the reference to this port.

```
send(this);
```

The response message is received at the `portA` and processed in a custom way specified in the `portA` *On receive action*. When this code finishes, the `portB` *On receive action* code execution is proceeded. In turn, after it is finished, the `send()` method of the `portA` finishes its work.

The method call sequence is illustrated in Figure 103.

Let's implement immediate object polling in a product delivering model using the described mechanism. A factory ships products to a number of warehouses. Products are sent only to the certain warehouses that have sufficient place to store a product. Therefore, before sending the product, requests are sent to all the warehouses to determine which warehouse

is capable of storing the product. A warehouse checks its spare place and sends back a confirmation only if it is not full. Finally, the product is sent to one of the spare warehouses.

The model structure is shown in Figure 104.

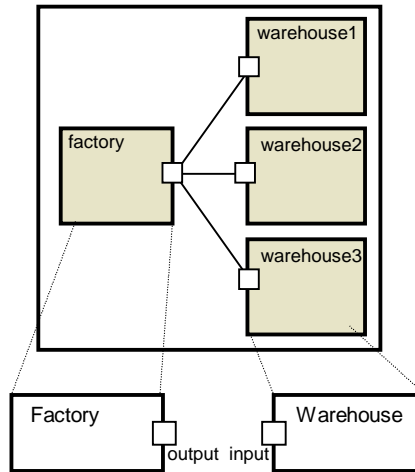


Figure 104.

The request and confirmation messages are represented by the `ReqMsg` message class. The message class contains the `recip` field of type `Port` used for transferring the warehouse address within a confirmation message.

Type	Name	Default
Port	recip	

The products are represented by the `Product` messages. This class has the `product` data member of `Product` type representing the product itself and the `dest` member variable of type `Port` used to carry the destination warehouse address.

Type	Name	Default
Port	dest	
Product	product	

The model works in the following way. First, the spare warehouse is searched. Therefore the request message is sent through the output port of the factory object:

```
output.send(new ReqMsg());
```

The request message is sent to all the recipients. When the message arrives at the warehouse the code specified in *On receive action* property of the input port is executed. This code checks the current warehouse spare place, defined by the capacity member variable. If the warehouse has sufficient place to store the product, it sends a confirmation message, stamped with the address of this warehouse back to the factory.

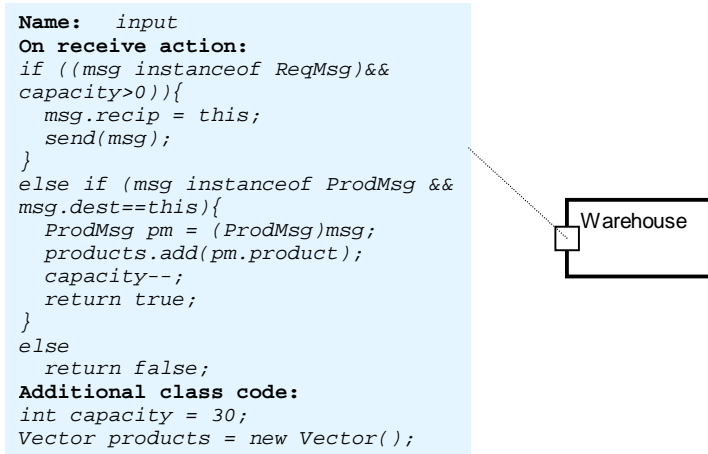


Figure 105. Warehouse object

When a confirmation is received at the factory, the warehouse address is added to the warehouses list. The request is sent to the next warehouse and so forth.

After the requests have been sent to all the warehouses and the `onReceive()` of the last polled warehouse is finished, the method `send()` execution is finished.

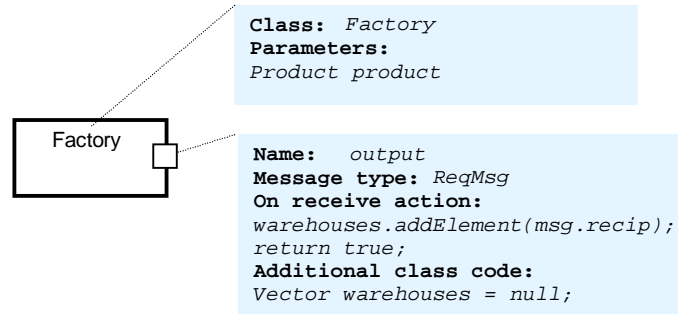


Figure 106. Factory object

Finally, the product is delivered to one of the warehouses that has confirmed the delivery request. Actually, the message is sent to all the warehouses, but is filtered at the recipient ports by the destination warehouse address, specified in the message `dest` field. When the product is delivered to the warehouse, its spare place value is decreased and the product is stored in the `products` Vector.

```

output.send(new ReqMsg());

//the warehouses list is already built

ProdMsg pm = new ProdMsg();

pm.product = product;

pm.dest = output.warehouses.get(0);

output.send(pm);

output.warehouses.removeAllElements();
  
```

Finally, the warehouses list is cleared.

Figure 107 illustrates how this model works. The request is sent to all the warehouses. In the concerned case, the first and second warehouses have sufficient place to store the product, so they send confirmation to the factory. The third warehouse is full and cannot store product. Finally, the product is sent to the first warehouse.

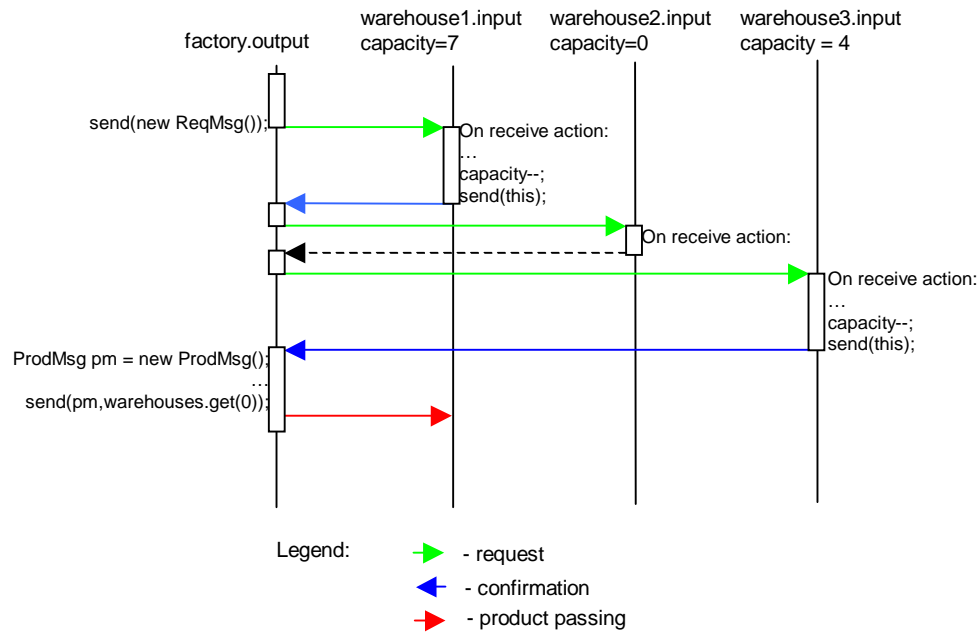


Figure 107. Method call sequence

7.4.16 Message passing in Enterprise Library

This section contains the detailed information about the Enterprise Library port classes. It describes the library objects interaction protocol and may be helpful in creating custom objects in addition to those in the Enterprise Library.

Active objects of the Enterprise Library follow certain rules when they pass entities via ports one to another. Generally, port can be input or output – i.e., it transfers entities only in one direction (passing resource units and transporters in the network is not considered). An input port may only be connected to an output port.

7.4.16.1 Entity passing protocol

When the entity is passed, the objects follow the specific protocol:

1. When an object intends to output an entity, it sends a notification to all connected inputs.

2. If the object wants to receive an entity, it sends an entity request to all the ports that have entities. Actually, if an input port is connected to multiple output ports, it can accept an entity from any of them.
3. The entity is passed via the output port on the first request reception.

Therefore the entity can never exit or enter an object without its prior agreement.

This protocol is implemented on top of standard AnyLogic ports by defining two port classes: `EntityInPort` for input ports and `EntityOutPort` for outputs. There are also two protocols for resource units exchange between the `Resource` object and `SeizeQ` and `Release` objects and for transporters exchange between `Node` and `Segment` objects, but they are left out of consideration because of their specificity. See AnyLogic Enterprise Library Reference Guide for a detailed description of these objects.

Let's examine how the entity passing protocol works in the model shown in Figure 108. Entities are passed via the output ports `outA`, `outB`, `outC1` and `outC2` to the `inK` and `inL` input ports of `objectK` and `objectL` correspondingly.

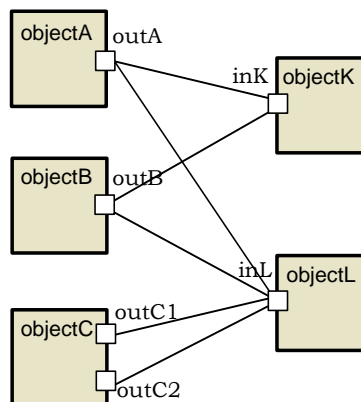


Figure 108.

Figure 109 shows the method call sequence. At the initial time entities are already pending at the `outA` and `outC1` ports.

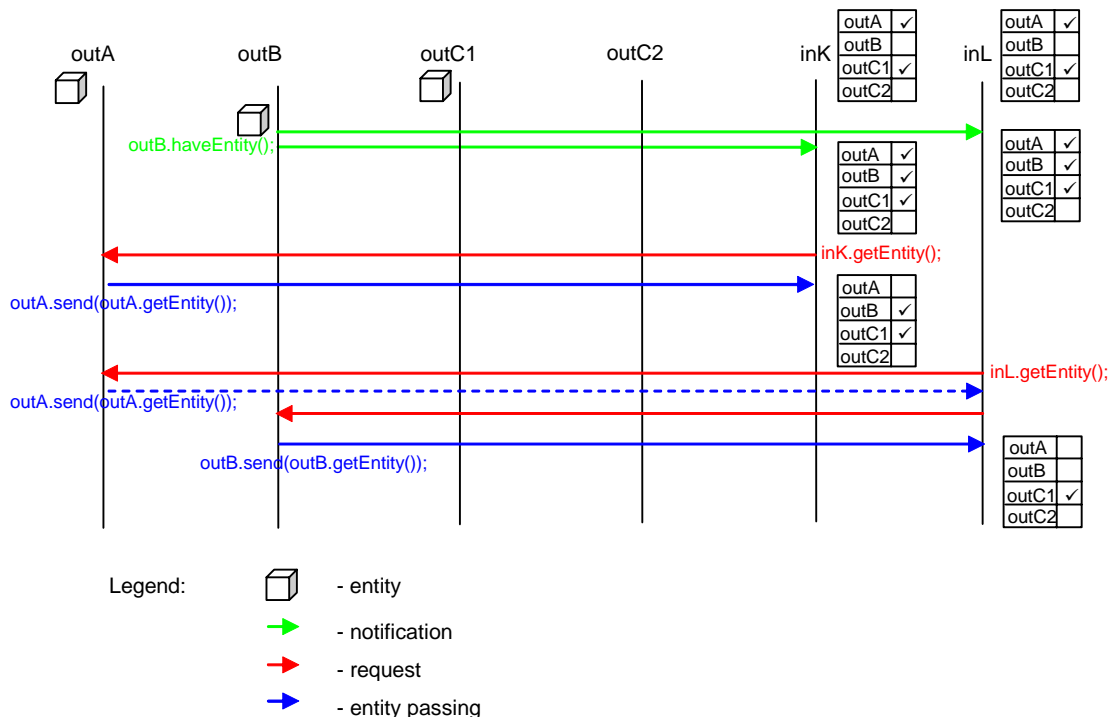


Figure 109. Entity passing protocol

The objectB passes entity at the outB port for output. The outB port sends the notification to all connected inputs (namely, inK and inL ports) by calling the `haveEntity()` method.

The notification is received at all the connected inputs. Each input port stores a table of the connected outputs. Ports ready to output entities are marked in this table with special flags. When a notification from the outB port is received at an input port, the corresponding flag in its table is set up.

If more than one entity is currently pending at the object, the `haveMoreEntities()` method is called. It schedules the successive notification sending at the same model time instant.

After some time objectK requests an entity by calling the `getEntity()` method of its inK port. All the connected outputs that have entities to output are requested in a round-robin manner. It is implemented by sending request messages to the marked outputs from the port table one by one in a loop.

First the request is received at the `outA` port. The entity for output is returned by the `getEntity()` method of the `outA` port. Since the `outA` port has an entity to output, it is passed to the requestor object. When the entity arrives at the `inK` port, the entity request procedure is finished and the entity requestor method `getEntity()` returns the received entity to the object.

Then the `inL` port requests an entity. First, the request is sent to the first marked port in the `inL` outputs table, namely to the `outA` port. But the request arrives by the time there is no entity at the `outA` port (the object has already output an entity to `objectK` on its request), so `null` is passed to the entity requestor. Since no entity was received from the `outA` port, the request is sent to the next marked port in the `inL` outputs table, namely to the `outB` port. The `outB` port has an entity to output, so it is passed to the `inL` port.

In case several objects were ready and sent their requests, the order in which they arrive is arbitrary, so the entity will be passed to a randomly chosen object.

In the case all the requested outputs already had no entities to output, the request initiator method `getEntity()` returns `null`.

The arbitrary connections allowed in AnyLogic Enterprise Library have an important consequence. It is important to understand how entities are passed in case there are several alternative possibilities. As long as an object may be accepting an entity from multiple sources, (which, in turn may be connected to multiple recipients), it can never know for sure which entity will come or even that an entity will come at all until it actually arrives. Symmetrically, an object can never know that another object will accept an entity until it actually requests it.

The important property of the entity exchange protocol is that an object may sometimes be unable to output an entity because of inability of the other objects to accept it. Therefore, you should organize your entity flow diagram in such a way that entities are always able to exit whenever they are not allowed to stay by adding a buffering object, or increasing a capacity of the existing object. There are only several objects that would allow an entity to stay and wait until it can be passed out: `Queue`, `Conveyor` and `Lane`. All other objects will report an error if an entity spends a non-zero time waiting at the output.

7.4.16.2 Enterprise Library port classes

This section gives the detailed description of AnyLogic Enterprise Library port classes. It may be helpful for creating your own port classes with customized behavior. AnyLogic has two predefined port classes: `Port` for a port without a queue, and `PortQueueing` for a port with a queue. If you want to customize the default behavior of ports you need to define your own port class, derived from one of these base classes.

EntityInPort class

The input ports in the Enterprise Library are represented by the instances of the `EntityInPort` port class.

The methods of the `EntityInPort` class are listed in Table 13.

Method	Description
<code>void haveEntity()</code>	The method is called on notification arrival. You can override this method in the port instance of your object to specify custom actions to be performed – e.g., request an entity.
<code>Entity getEntity()</code>	The method requests an entity from connected outputs. The method returns the received entity. If no entities were received, <code>null</code> is returned.
<code>boolean hasEntity()</code>	The method checks if any connected output has an entity. If so, <code>true</code> is returned; otherwise <code>false</code> is returned.
<code>void processAuxiliary(EntityMsg fm)</code>	AnyLogic Enterprise Library enables you to send auxiliary messages between ports in both directions. You can process the received auxiliary message in a custom way by redefining this method in the port instance.
<code>ActiveObject getConnectedObject()</code>	The method returns first connected object, if any are known.

Table 13. EntityInPort class methods

EntityOutPort class

The output ports in Enterprise Library are represented by the instances of the `EntityOutPort` port class.

The methods of the `EntityOutPort` class are listed in Table 14.

Method	Description
<code>void haveEntity()</code>	The method notifies connected input ports that this object has an entity to send via this port.
<code>void haveMoreEntities()</code>	The method schedules sending a notification to the connected input ports that this object has more entities to send via this port.
<code>abstract Entity getEntity()</code>	The method extracts the entity for output. The method is abstract and is overridden in the port instances of the Enterprise Library objects, since the entity obtaining logic may depend on the object work logic.
<code>int getCount()</code>	The method returns the number of entities exited through the port.
<code>int size()</code>	The method returns the number of pending entities.
<code>void resetStats()</code>	The method resets the collected statistics on exited entities.
<code>boolean isEmpty()</code>	The method checks if there are any entities pending in the port (the method returns <code>false</code> if there are any pending entities and <code>true</code> otherwise).
<code>void processAuxiliary(EntityMsg fm)</code>	AnyLogic Enterprise Library enables you to send auxiliary messages between ports in both directions. You can process the received auxiliary message in a custom way by redefining this method in the port instance.

Table 14. `EntityOutPort` class methods

EntityOutPortQueue class

The `EntityOutPortQueue` port class, derived from `EntityOutPort`, is used to model output ports with queues. This port is commonly used in entity processing objects. If more than 1000 entities are pending, the port signals an error.

The methods of the `EntityOutPortQueue` class are listed in Table 15.

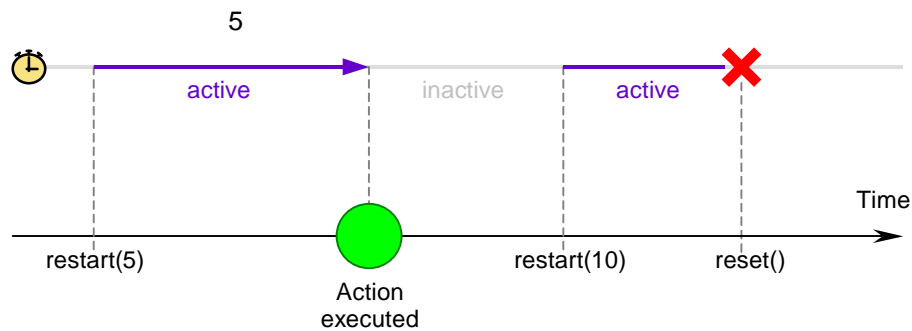
Method	Description
<code>void take()</code>	The method places the entity passed as the method parameter in the port queue.
<code>Entity getEntity()</code>	The method extracts the first entity from the port queue.
<code>boolean isEmpty()</code>	The method checks if there are any entities pending in the queue (the method returns <code>false</code> if there are any pending entities and <code>true</code> otherwise).
<code>void setCanWait(boolean cw)</code>	The method allows or forbids the entity waiting at the output port queue by passing <code>true</code> or <code>false</code> correspondingly. (However, multiple entities can be passed to an output as long as they all are passed further in zero time).
<code>int size()</code>	The method returns the number of pending entities.

Table 15. `EntityOutPortQueue` class methods

8. Timers

A timer is the simplest way to schedule some action in the model. Thus, timers are used to model delays and timeouts. Sometimes you do the same using timed transitions in statecharts, but timers might be more efficient. There are cases when desired behavior can be modeled only using timers. For example, a communication channel which is able to transmit an arbitrary number of messages concurrently can be modeled with the help of dynamic timers that are created for each message.

Static/Chart timers



Dynamic timers

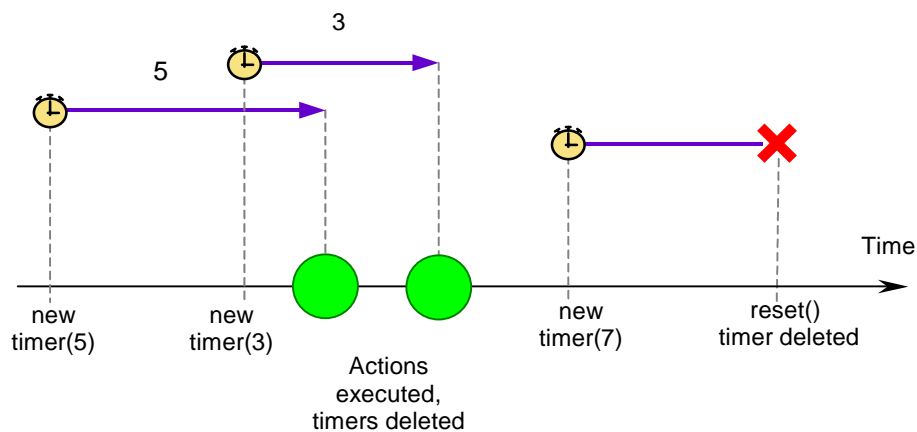


Figure 110. Static and dynamic timers

There are two types of timers: dynamic timers and static timers. The latter may be declared graphically, in which case a timer is called chart timer. The difference between the dynamic timer and the static timer is that the dynamic timer deletes itself upon expiry, whereas static timer survives and can be restarted. The timeout of a dynamic timer is passed to its constructor, whereas the timeout of a static timer is specified by calling the method `restart()` of the timer. A chart timer has even more features: you can specify that it expires either once or cyclically, or works in the manual mode like an ordinary static timer.


A timer has an action associated with it – code that is executed when the timer expires.

8.1 Dynamic timers

A dynamic timer is created using the operator `new` and the timeout is passed to the object's constructor. Time counting begins at the moment of timer construction. When the timeout expires, AnyLogic calls the method `action()` of the timer and then deletes the timer. If the method `reset()` of the timer is called before expiry, the timer is deleted and its method `action()` is never called.

The easiest way to define a dynamic timer class is to use the Project window of AnyLogic. However, you can define a dynamic timer class anywhere in the code – e.g., in the *Additional class code* or in an external file. Then you would have to subclass from `DynamicTimer` and override the method `action()`.

► To define a new dynamic timer class

1. Click the *New Dynamic Timer Class*  toolbar button, or Choose *Insert | New Dynamic Timer Class...* from the main menu. The *New Dynamic Timer Class* dialog box is displayed. Specify the name of the new dynamic timer class, choose the active object class, which will contain the timer class, and click *OK*.
2. Alternatively, in the Project window, right-click the active object class, which will contain the timer class, and choose *New Dynamic Timer Class...* from the popup menu. The *New Dynamic Timer Class* dialog box is displayed. Specify the name of the new dynamic timer class and click *OK*.

A dynamic timer class has the following properties, specified on the *General* and *Code* pages of the Properties window correspondingly:

General properties

Name – name of the dynamic timer class.

Parameters – [optional] set of formal parameters of the class. Every parameter should be declared in form: *Type Name Default*, where *Type* is the type of the parameter, *Name* is the name of the parameter, *Default* is the optional default value of the parameter. When instantiating the class, actual parameters may be specified or default ones may be left. The parameters can be accessed as member variables of the timer object.

Exclude from build – if set, the dynamic timer class is excluded from the model.

Code properties

Constructor code – [optional] sequence of Java statements to be executed on dynamic timer construction.

Expiry action – [optional] sequence of Java statements to be executed on timer expiry.


Additional class code – [optional] Java code to be inserted into the class definition. Constants, variables, and methods can be defined here.

8.2 Static and chart timers

Unlike a dynamic timer, a static timer can expire any number of times. To start a static timer, you call its method `restart()`, specifying the timeout value as a parameter. When the timer expires, it calls the method `action()`. By calling the method `reset()`, you deactivate the timer until the next call to `restart()`.

The easiest way to define a static timer is to place a chart timer on the structure diagram of an active object class. However, you can define a static timer class anywhere else in the model – e.g., in the *Additional class code* or in an external files. Then you would have to subclass from `StaticTimer` class and override the method `action()`. A chart timer adds more features to a static timer: you can specify that it expires either once or cyclically, or works in the manual mode like an ordinary static timer.

► To declare a chart timer

1. Click the *Chart Timer*  toolbar button, or Choose *Draw | Structure | Chart Timer* from the main menu.
2. Click the place on the diagram where you want to place the chart timer. A chart timer appears, displayed as an icon, see Figure 111.

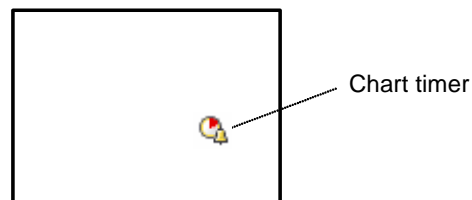


Figure 111. Chart timer

Chart timer defines a static timer within an active object. Chart timer is displayed as an icon and can be placed anywhere on the structure diagram – inside or outside this object.

Chart timer has the following properties:

Properties

Name – name of the chart timer.

No expiry/Expire once/Cyclic – determines whether the timer is in the manual mode, or expires once, or expires cyclically.

Expire at startup – applies to a cyclic timer only. If set, the timer first expires immediately at model startup. Otherwise it first expires after a timeout.

Timeout – expression that is evaluated to obtain the timeout. This can be a real number, call to a method, random expression, etc.

Expiry action – [optional] sequence of Java statements to be executed on expiry.

Exclude from build – if set, the chart timer is excluded from the model.

Show name – if set, the name of the chart timer is shown on the structure diagram.

9. Statecharts

During its lifetime an active object performs operations in response to external or internal events and conditions. Existence of a state within an active object means that the order in which operations are invoked is important. For some objects, this event- and time-ordering of operations is so pervasive that you can best characterize the behavior of such objects in terms of a state transition diagram – a statechart. A statechart is used to show the state space of a given algorithm, the events that cause a transition from one state to another, and the actions that result from state change.

By using statecharts you can visually capture a wide variety of discrete behaviors, much more rich than just idle/busy, open/closed, or up/down status offered by most block-based tools.


AnyLogic statecharts are UML compliant. They preserve graphical appearance, attributes, and execution semantics defined in UML.

AnyLogic supports hybrid statecharts – the most natural and powerful way to integrate discrete logic and continuous time behavior. In hybrid statecharts you can associate a set of differential and algebraic equations with a state on a statechart diagram. When a state transition is taken as a result of, e.g., some discrete event, the system of equations changes – this way, discrete logic affects the continuous time behavior. If you specify a condition over continuously changing variables as a trigger of a transition, you get the opposite effect: continuous time behavior impacts the discrete part of the system.

9.1 Creating a statechart

A statechart is declared on a structure diagram and implemented using a statechart diagram.

► To create a statechart

1. Click the *Statechart*  toolbar button, or Choose *Draw | Structure | Statechart* from the main menu.
2. Click the place on the structure diagram where you want to place the statechart. A statechart appears, displayed as an icon, see Figure 112.

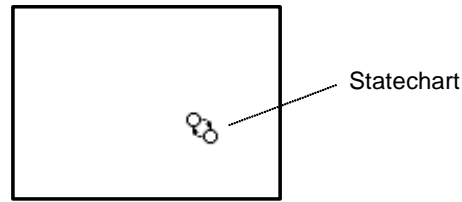


Figure 112. Statechart

Once the statechart is created, you can specify the statechart name in the text line editor opened on the right of the statechart in the structure diagram.

A statechart has the following set of properties:

Properties

Name – name of the statechart.

Deferred events – [optional] a set of signal event descriptors separated by commas, e.g.:
`new SignalEvent1(), new SignalEvent2("param")`. Deferred events for this statechart are events that match the given descriptors.

After initialize action – [optional] the code to be called when the statechart is about to run after it has been initialized.

Before step action – [optional] the code to be called each time before the statechart switches from one state to another.

After step action – [optional] the code to be called each time after the statechart switches from one state to another.

Node action – [optional] the code to be called each time before the statechart could execute an action associated with entering a state or a transition (including pseudo states, branch transitions, etc.), even if such action is not defined. Please note that the function may be called several times during a single step.

Exclude from build – if set, the statechart is excluded from the model.

Show name – if set, the name of the statechart is shown on the structure diagram.

9.2 Statechart diagram

The behavior of a statechart is defined on a statechart diagram. AnyLogic supports the following statechart constructs: state (see section 9.2.1.1, “State”), transition (see section 9.2.1.2, “Transition”), initial state pointer (see section 9.2.1.3, “Initial state pointer”), final state (see section 9.2.1.5, “Final state”), branch (see section 9.2.1.6, “Branch”), shallow and deep history states (see section 9.2.1.7, “Shallow history and deep history states”), and text box (see section 9.2.1.8, “Text box”), see Figure 113. In this section all these constructs are described in detail.

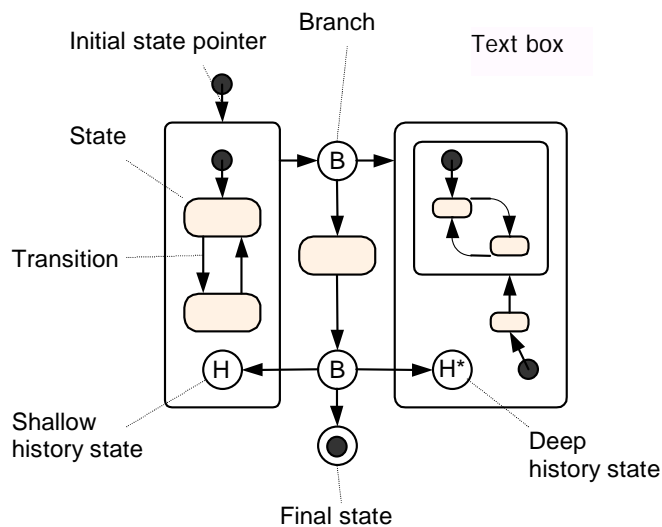


Figure 113. Statechart diagram

A statechart diagram is edited in the statechart editor using the statechart toolbar, see Figure 114.

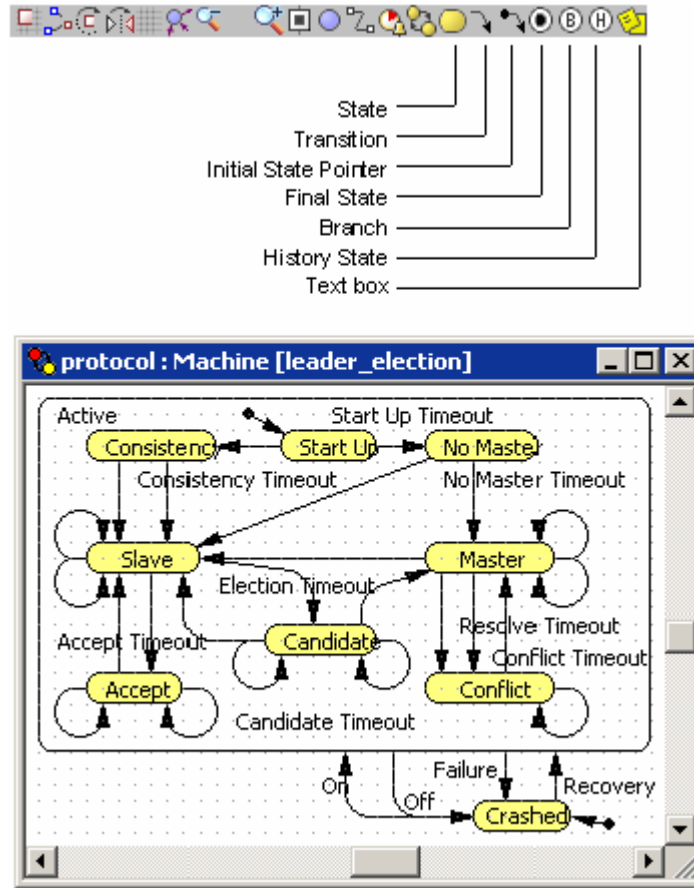


Figure 114. Statechart editor and toolbar

► **To open the statechart diagram of a statechart**

1. Double-click the statechart in the Project window, or
Double-click the statechart on a structure diagram, or
Right-click the statechart on a structure diagram and choose *Open Statechart* from the popup menu.

► **To open the structure diagram of the statechart owner object**

1. Right-click the empty area of the statechart diagram and choose *Up to Parent* from the popup menu

Statechart editor shares a large set of generic editing operations described in section 1.5.2, “Diagram editors. Generic operations”.

9.2.1.1 State

A state represents a location of control with a particular set of reactions to conditions and/or events. A state can be either simple or, if it contains other states, composite. Control always resides in one of simple states, but the current set of reactions is a union of those of the current simple state and of all composite states containing it – i.e., a transition exiting any of these states may be taken.

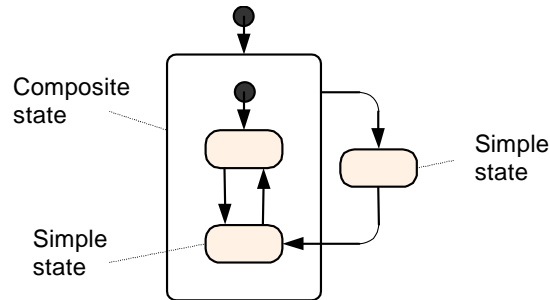



Figure 115. States

► To draw a state

1. Click the *State*  toolbar button, or Choose *Draw | Statechart | State* from the main menu.
2. Click the place in the diagram where you want to put the state. Then drag to choose the size of the state.

A state has the following properties:

Properties

Name – name of the state.

Deferred events – [optional] set of signal event descriptors separated by commas, e.g.:
`new SignalEvent1(), new SignalEvent2("param")`. Deferred events for this state are events that match the given descriptors.

Equations – [optional] set of differential equations, algebraic equations, and formulas given in form described in Chapter 5, “Equations”. These equations are active while the statechart stays in the state.

Entry action – [optional] sequence of Java statements to be executed when the statechart enters the state.

Exit action – [optional] sequence of Java statements to be executed when the statechart exits the state.

Exclude from build – if set, the state is excluded from the model.

Show name – if set, the name of the state is shown on the statechart diagram.

9.2.1.2 Transition

A transition (see Figure 116) denotes a switch from one state to another. A transition indicates that if the specified trigger event occurs and the specified guard condition is true, the statechart switches from one state to another and performs the specified action. When this occurs, we say that the transition is taken.

The starting point of a transition lies on the border of the transition's source state. The end point of a transition lies on the border of the transition's destination state. A transition may freely cross simple state and composite state borders. If the source of a transition lies either on or inside a state, and the destination of that transition lies outside of the state, then that state is considered exited by the transition. If such a transition is taken, the exit action of the exited state is executed. If the source of a transition lies outside a state, and the destination of that transition lies either on or inside the state, then that state is considered entered by the transition. If such a transition is taken, the entry action of the entered state is executed. In case a part of a transition lies inside a state, but both source and destination are outside the state, this state is considered neither entered nor exited.

There is a special type of transition called internal transition. An internal transition lies inside a state, and both start and end points of the transition lie on the border of this state (see Figure 116). Since an internal transition does not exit the enclosing state, neither exit nor entry actions are executed when the transition is taken. Moreover, the current simple state within the state is not exited too. Therefore, the internal transition is very useful for implementing simple background jobs, which should not interrupt the main activity of the composite state.

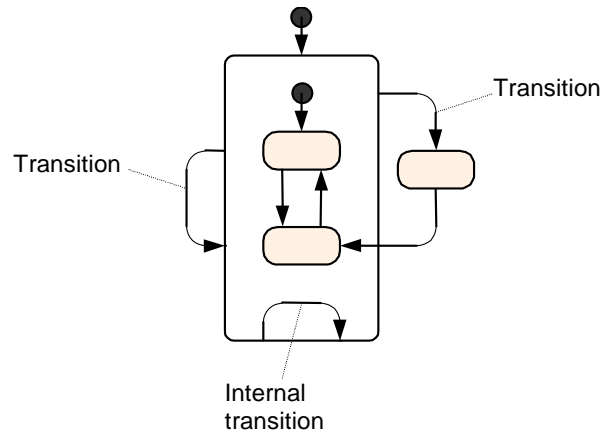



Figure 116. Transitions

► To draw a transition

1. Click the *Transition*  toolbar button, or Choose *Draw | Statechart | Transition* from the main menu.
2. Click the starting point of the transition (the border of a state or pseudo state).
3. Click the points where the transition should turn.
4. Click the ending point of the transition (the border of a state or pseudo state).

Properties

Name – name of the transition.

Fire – the trigger type.

Immediately – the transition is triggered immediately.

After timeout – the transition is triggered after the specified timeout elapses.

If signal event occurs – the transition is triggered on the specified signal event occurrence.

If change event occurs – the transition is triggered on the specified change event occurrence.

If event occurs – the transition is triggered on the specified custom event occurrence.

All these trigger types are described in section 9.4, “Triggering a transition”.

Timeout – [for transitions triggered *After timeout only*] specifies a timeout that triggers the transition.

Signal event – [for transitions triggered *If signal event occurs only*] specifies signal event that triggers the transition.

Change event – [for transitions triggered *If change event occurs only*] specifies change event that triggers the transition.

Trigger – [for transitions triggered *If event occurs only*] specifies custom event that triggers the transition.

Guard – [optional] boolean expression that allows (if `true`) or prohibits (if `false`) the transition. If not specified, `true` is assumed.

Action – [optional] sequence of Java statements executed when the transition is taken.


Exclude from build – if set, the transition is excluded from the model.

Show name – if set, the name of the transition is shown on the statechart diagram.

► To move a point of a transition


1. Drag the point.

► To add a salient point to a transition

1. Select the transition.
2. Click the *Edit Points*  toolbar button, or Choose *Draw | Edit Points* from the main menu, or Right-click the transition and choose *Edit Points* from the popup menu. The points of the transition should turn yellow.
3. Drag a segment of the transition to create a salient point, or Right-click the segment and choose *Add Point* from the popup menu.

► To remove a salient point from a transition

1. Select the transition.

2. Click the *Edit Points*  toolbar button, or Choose *Draw | Edit Points* from the main menu, or Right-click the transition and choose *Edit Points* from the popup menu.
3. Right-click the point and choose *Delete Point* from the popup menu, or Drag the point to an adjacent point of the transition. The dragged point disappears.

9.2.1.3 Initial state pointer

Initial state pointer (see Figure 117) points to the initial state within a particular level of state hierarchy.

If the control is passed to a composite state, a simple state is found inside it by following the initial state pointers down the state hierarchy, and this state becomes current. There should be exactly one initial state on each level – i.e., on the upper level and in each composite state.

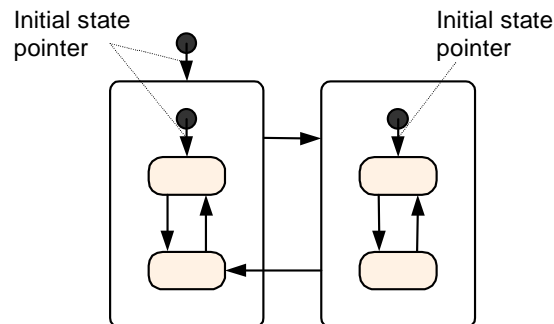



Figure 117. Initial state pointer

► To draw an initial state pointer

1. Click the *Initial State Pointer*  toolbar button, or Choose *Draw | Statechart | Initial State Pointer* from the main menu.
2. Click the starting point of the initial state pointer.
3. Click the points where the initial state pointer should turn.

4. Click the ending point of the initial state pointer (the border of a state or pseudo state).

The generic transition editing operations (see section 9.2.1.2, “Transition”) can be applied to initial state pointers.

Properties

Name – name of the initial state pointer.

Action – [optional] sequence of Java statements executed when the initial state pointer forwards the control to an initial state.

Exclude from build – if set, the initial state pointer is excluded from the model.

Show name – if set, the name of the initial state pointer is shown on the statechart diagram.

9.2.1.4 Pseudo states

A pseudo state is a special type of a node on a statechart diagram. Control never stays in a pseudo state; it always passes through. Therefore, triggers cannot be specified for transitions exiting pseudo states. When control passes a pseudo state, pseudo state's action is executed.

There are four types of pseudo states:

- Final state
- Branch
- Shallow history state
- Deep history state

They all have the following set of properties:

Properties

Name – name of the pseudo state.

Exclude from build – if set, the pseudo state is excluded from the model.


Show name – if set, the name of the pseudo state is shown on the statechart diagram.

Action – [optional] sequence of Java statements executed when the control passes the pseudo state.

9.2.1.5 Final state

A final state (see Figure 118) is a termination point of a statechart. When control enters a final state, its action is executed, and the statechart terminates. Transitions may not exit a final state.

► To draw a final state

1. Click the *Final State*  toolbar button, or Choose *Draw | Statechart | Final State* from the main menu.
2. Click the place on the diagram where you want to put the final state.

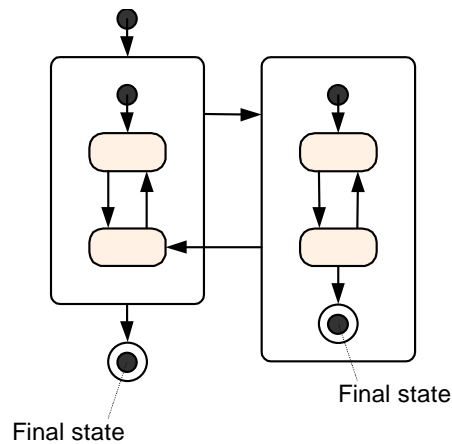



Figure 118. Final states

9.2.1.6 Branch

A branch (see Figure 119) represents a transition branching and/or connection point. Using branches you can create a transition that has more than one destination state, as well as several transitions that merge together to perform a common action.

When control passes a branch, its action is executed, and then the guards of transitions exiting the branch are evaluated. The first enabled transition – i.e., the transition whose guard evaluates to true – is taken.

► To draw a branch state

1. Click the *Branch*  toolbar button, or Choose *Draw | Statechart | Branch* from the main menu.
2. Click the place on the diagram where you want to put the branch state.

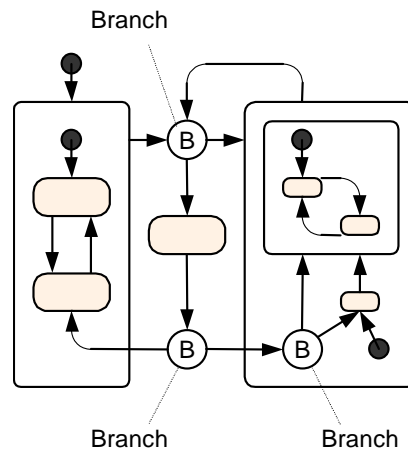


Figure 119. Branches

A branch may have at most one special outgoing transition marked default branch exit. This transition is taken in case all other outgoing transitions are closed.

Transitions exiting branch states have the following properties, slightly different from other transitions properties.

Properties

Name – the name of the transition.

Fire – the trigger type.

If guard is open – the transition is triggered if the specified guard is open.

If all other guards are closed – the transition is the default branch exit.

Guard – [for transitions triggered *If guard is open* only] boolean expression that allows (if `true`) or prohibits (if `false`) the transition. If not specified, `true` is assumed.

Action – [optional] sequence of Java statements executed when the transition is taken.

Exclude from build – if set, the transition is excluded from the model.


Show name – if set, the name of the transition is shown on the statechart diagram.

If all outgoing transitions are closed and there is no default exit from a branch, a runtime error is issued.

9.2.1.7 Shallow history and deep history states

A composite state may contain shallow history and deep history states. A shallow history state is a reference to the most recently visited state on the same hierarchy level within the composite state. Deep history state is a reference to the most recently visited simple state within the composite state. When the control comes to a shallow/deep history state, its action is executed, and the control is immediately passed to the “real” state referred by it.

► To draw a history state

1. Click the *History state*  toolbar button, or
Choose *Draw | Statechart | History State* from the main menu.
2. Click the place on the diagram where you want to put the history state.
3. In the Properties window, choose whether the history state is *Deep* or *Shallow*.

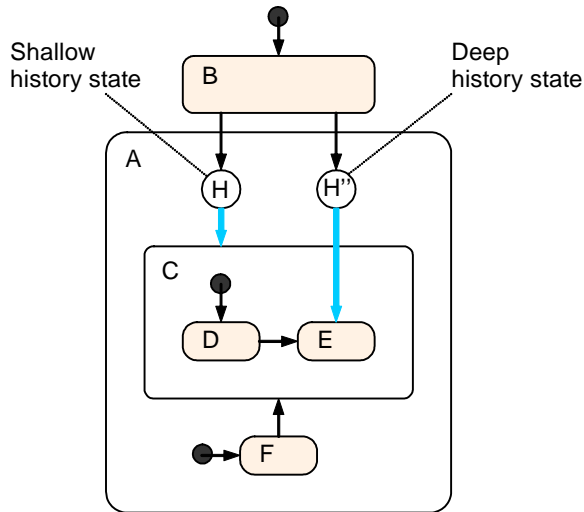


Figure 120. Shallow history and deep history states

Figure 120 illustrates the difference between shallow and deep history states. Suppose E is the most recently visited simple state inside the composite state A. If the control reaches the deep history state H*, it passes to E, whereas shallow history state H passes the control to C – the most recently visited state on the same hierarchy level. Then the standard procedure of finding the initial state within C is invoked, and the statechart ends up in D.

In case there is no visited state at all within the scope of a history state (no history exists yet), the control goes to the corresponding initial state, unless there is a transition exiting the history state and pointing to the so-called default history state (see Figure 121). There may be at most one such transition (with *If there is no history* trigger type) for a history state.

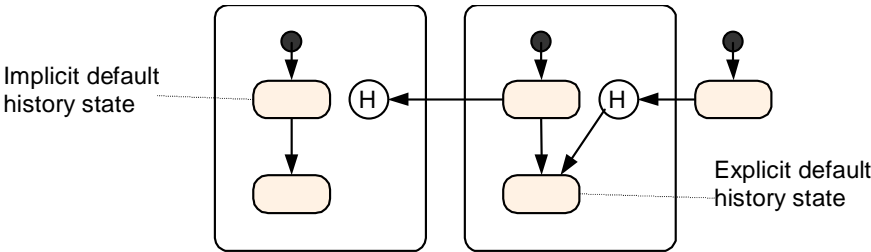



Figure 121. Default history states

9.2.1.8 Text box

A text box is used to put a comment on a statechart diagram. It does not affect the model behavior.

► To draw a text box

1. Click the *Text Box*  toolbar button, or Choose *Draw | Text Box* from the main menu.
2. Click the place on the diagram where you want to put the text box. Then drag to choose the size of the shape.

► To modify the content of a text box

1. Double-click the text box.
2. Edit the content of the text box.
3. Click the empty area of the diagram or press Esc to store the modified text.

You can also modify the text of the text box using its Properties window.

Properties

Text – content of the text box.

9.3 Execution order

It is important to know exactly what the order is of the execution of statechart elements' actions. For this reason we present the following algorithm.

When a transition is taken, transition and state actions are executed in the following order:

1. State exit actions starting with the old simple state up to the outermost exited composite state.
2. Transition action.

3. State entry actions starting with the outermost entered composite state down to the new simple or pseudo state.
4. If the control enters a pseudo state, its action code is executed, and then the control goes out of the pseudo state immediately, and this algorithm applies again from the beginning.

Actions associated with statechart elements (states and transitions) are executed atomically and in zero model time. Therefore they cannot contain synchronization and delay operations, or call methods directly or indirectly containing them.

Example

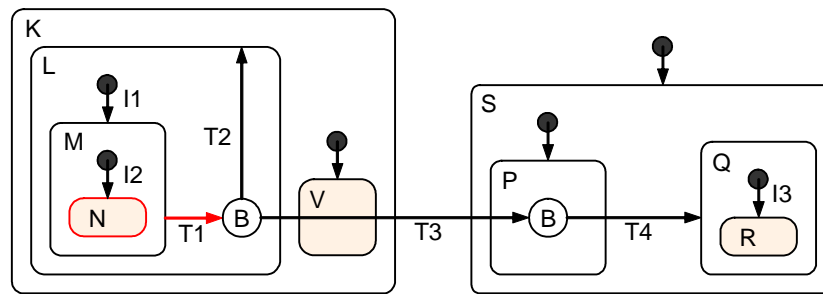


Figure 122. Execution order illustration

Consider the example shown in Figure 122. Suppose N is the current simple state and transition $T1$ has been selected to be taken. Then the actions are executed in the following order:

1. N state exit action
2. M state exit action
3. $T1$ transition action
4. Branch action

Then the transition $T2$ or $T3$ is selected depending on guards of the transitions. In case the selected transition is $T2$, the following actions are executed:

5. $T2$ transition action

6. I1 initial state pointer action (exit and entry actions of the state L are not executed since that state is not exited)
7. M state entry action
8. I2 initial state pointer action
9. N state entry action

In case the selected transition is T3, the following actions are executed:

10. L state exit action
11. K state exit action (actions of the state V are not executed)
12. T3 transition action
13. S state entry action
14. P state entry action
15. Branch action
16. P state exit action
17. T4 transition action (the guard of this transition must be true as this is the only exit from the branch)
18. Q state entry action
19. I3 initial state pointer action
20. R state entry action

9.4 Triggering a transition

When a statechart enters a simple state, the triggers of all outgoing transitions (including the transitions outgoing all composite states containing the simple state) are collected and the statecharts begins to wait for any of them to occur. When a trigger event occurs, the guard

of the corresponding transition is evaluated. If the guard is `true`, then the transition may be taken (we say “may be” because there may be alternative simultaneous events at AnyLogic simulation engine, which may reset the trigger). This algorithm of guard evaluation is called “guards-after-triggers”.

If several triggers are signaled at the same time, and the corresponding guards are true, the transition to be taken can be chosen randomly or deterministically, see section 14.2.1, “Event processing at the simulation engine”.

Transition can be triggered as a result of various types of events occurred, namely:

- Immediately;
- After the specified timeout elapses;
- When the change event occurs;
- When the signal event occurs;

You specify the trigger type in the *Fire* property of a transition.

This section gives the detailed description of all transition trigger types.

9.4.1 Immediate triggering

Transition can be triggered immediately on entering the transition's source state.

► To define immediately triggered transition

1. Click the transition on the diagram.
2. In the Properties window, choose *Immediately* from the *Fire* drop-down list.
3. Specify transition guard in the *Guard* section of the edit box.
4. Specify transition action in the *Action* section of the edit box.

9.4.2 Triggering after a timeout

The trigger is interpreted as time if it evaluates to `double` or `Distr`. In case of `Distr`, the time value is evaluated as a sample of the distribution (see section 10.2, “Probability distributions”). The transition becomes enabled after the specified amount of time elapses, since the statechart comes to the source state of the transition. Such transition may be used to model delays and, combined with alternative transitions, timeouts.

► To define transition triggered after a timeout

1. Click the transition on the diagram.
2. In the Properties window, choose *After timeout* from the *Fire* drop-down list.
3. Specify *Timeout*, *Guard* and *Action* for the transition.

Example

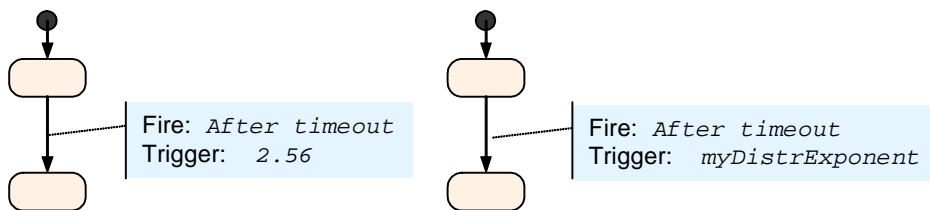


Figure 123. Transitions triggered by time

9.4.3 Change event trigger

A trigger is considered as change event if it evaluates to `boolean`. A transition with such a trigger is enabled when the expression is `true`. If by the time the statechart comes to the source state of such transition the expression is `true` already, the transition becomes enabled immediately. Otherwise, it becomes enabled as soon as the expression becomes `true` actions – e.g., as a result of equation solving, as a change event may contain variables changing continuously according to a set of differential and algebraic equations. When the expression becomes `true`, AnyLogic determines the switch point – the moment when the expression becomes `true` – with the accuracy set by the user.

► To define transition triggered by change event

1. Click the transition on the diagram.
2. In the Properties window, choose *If change event occurs* from the *Fire* drop-down list.
3. Specify *Change event*, *Guard* and *Action* for the transition.

Example

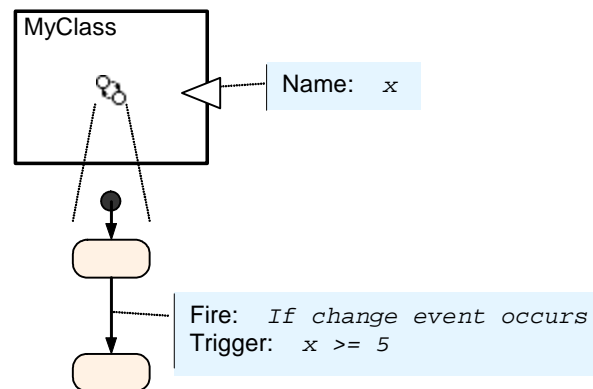


Figure 124. Transition triggered by change event

When specifying a change event, you should keep in mind the so-called sensitivity problem. Let the transition wait for the Boolean expression $x \geq 5$, and let x changes continuously in time as shown in Figure 125. As the numeric equation solver works by steps, it may happen that x will exceed the value 5 and get back in between the two steps. In this case the change event will not be detected. You should be aware of such situations when modeling systems where such error might be critical. If you encounter such a problem, you should make numerical method accuracies smaller (see section 5.6, “Numerical methods”).

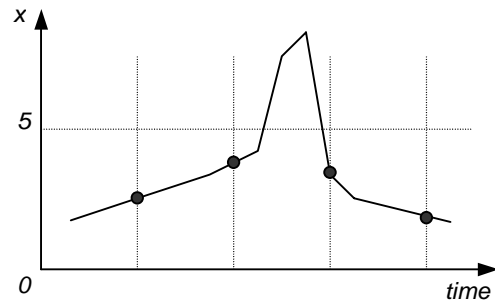


Figure 125. Sensitivity problem

Example

You can trigger statechart transition if the specified port has any messages in its queue (please do not confuse it with the statechart queue), see Figure 126. If by the time the statechart comes to the source state of such transition the port queue already has messages, the transition becomes enabled immediately. Otherwise it becomes enabled as soon as a message is placed in the queue.

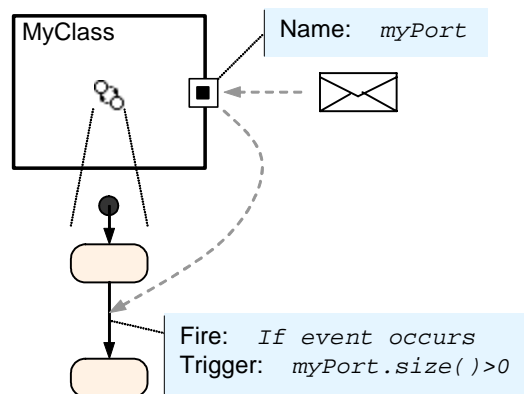


Figure 126. Transition triggered by port

Please note that triggering of such a transition does not delete the message from the port queue. You can call the method `get()` of the port to consume messages from the message queue, or the method `peek()` (see section 7.1.2, “Port queue” to know how to manage port queue). You can access the queue e.g. in the *Action* of the transition.

9.4.4 Signal event

Statechart transition can be triggered by a signal event. A signal event is an instance of an arbitrary Java class. When someone calls the method `fireEvent()` of a statechart, a signal event is added to the statechart event queue (please do not confuse it with a port queue), and then it can trigger a transition.

You can connect a port and a statechart. Then messages coming into the port will be routed to the statechart and will generate signal events. See section 7.1.7, “Receiving messages” for details.

The event that has just triggered the transition is available for further analysis via the method `getEvent()` of a statechart. You can access it, e.g., in the transition guard and action.

Simplified forms

There are two simplified forms of using signal events:

- If you are happy with using strings as signal events, you can simply type, e.g., “MYSTRING” in the *Signal event* property of a transition. Such a transition is triggered when the method `fireEvent(“MYSTRING”)` of a statechart is called.
- If you need just to check the type of your signal event to trigger a transition, you can type `MyType` in the *Signal event* property. Such a transition is triggered by a call to `fireEvent()` with any object of type `MyType` as an argument.

► To define transition triggered by signal event

1. Click the transition on the diagram.
2. In the Properties window, choose *If signal event occurs* from the *Fire* drop-down list.
3. Specify *Signal event*, *Guard* and *Action* for the transition.

Example

```
public class SomeEvent {  
}
```

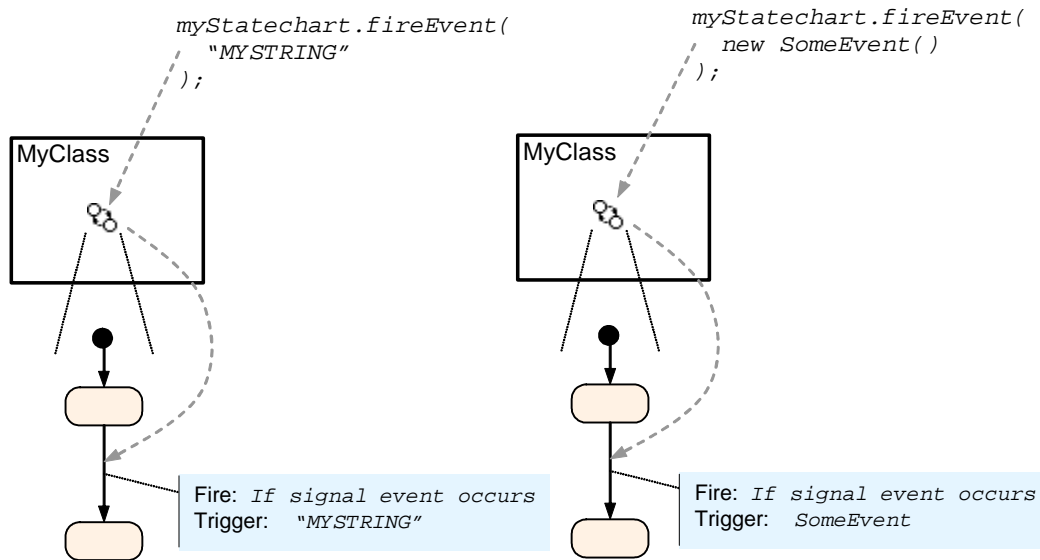


Figure 127. Transition triggered by a signal event (simplified forms)

General form

In general, to let a transition be triggered by a signal event, you should specify a reference to an event descriptor in the trigger. Event descriptor is an object, which is compared with a signal event to decide if a transition should be triggered – a filter for signal events. When a signal event is fired at a statechart, AnyLogic calls the method `equals()` of an event descriptor, giving a reference to the event as a parameter. If it returns `true`, then the event matches the descriptor and the transition should be taken. `false` means no match. The method `equals()` may use just event type information, or may look at event parameters.

You can use an instance of the same signal event class – a prototype – as an event descriptor. To do that, you should define the method `equals()` in the signal event class, which may e.g. compare member variables. Sometimes, however, the usage of a prototype is not convenient.

As a signal event can be an object of any class, you do not always need to define a special class for a signal event. For example, you may (re)use a message class.

Below we give examples of usage of signal events. In these examples an event has a parameter. The parameter is used to determine the match. In the first example we use a prototype as the event descriptor. In the second example we use a special class.

Example

```

public class CharEvent {

    public CharEvent( char param ) {
        this.param = param;
    }

    public boolean equals( Object obj ) {
        if( obj instanceof CharEvent ) {
            CharEvent ev = (CharEvent)obj;
            return ev.param == this.param;
        }
        else {
            return false;
        }
    }

    public char param;
}

```

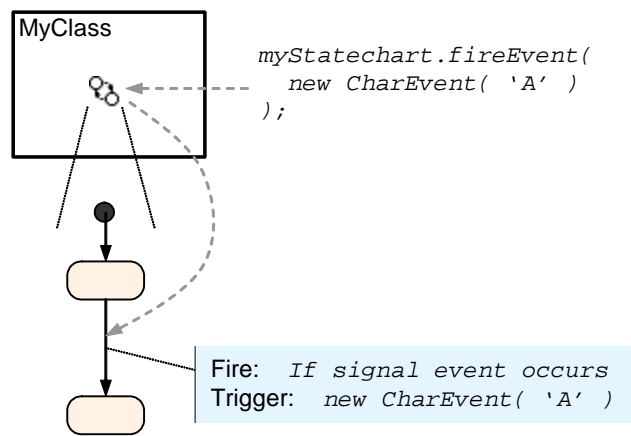


Figure 128. Transition triggered by a signal event (I)

Example

```

public class CharEvent {

    public CharEvent( char param ) {
        this.param = param;
    }

    public char param;
}

```

...

```

class CharEventDescr {

    public CharEventDescr( char lower, char upper ) {
        this.lower = lower;
        this.upper = upper;
    }

    public boolean equals( Object obj ) {
        if( obj instanceof CharEvent ) {
            CharEvent ev = (CharEvent)obj;
            return ev.param >= this.lower && ev.param <= this.upper;
        }
        else {
            return false;
        }
    }
}

char lower;
char upper;
}

```

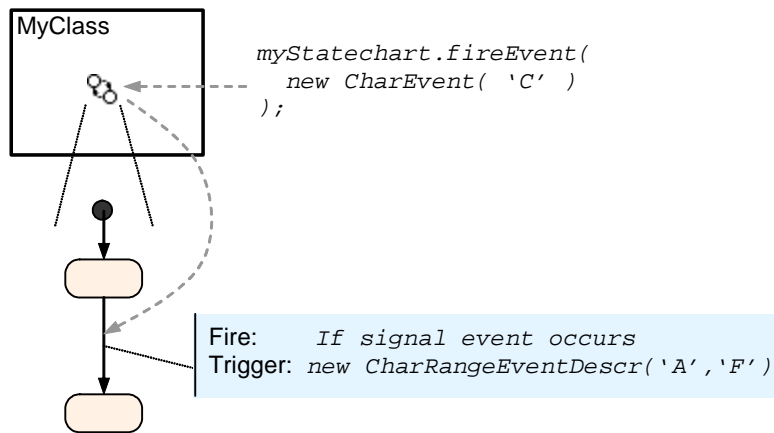


Figure 129. Transition triggered by signal event (II)

9.4.4.1 Statechart queue processing

A statechart handles signal events in the so-called event queue. The event queue is necessary because signal events may occur at those moments of time in which the statechart cannot

react to events (e.g. when a transition is executed). The event queue is processed by a statechart according to the following algorithm:

The event processing starts every time something occurs to the statechart, e.g. `fireEvent()` is called or the statechart makes a step.

- If there is one or several transitions outgoing the current simple state or any of its container states, whose trigger matches the first event in the queue, such transitions become enabled, i.e. one of it is taken depending on guards.
- If there are no such transitions, but the event matches any of the deferred event descriptors of the current simple state or any of its container states, or of the whole statechart, the event is kept in the queue, and the next event is processed.
- If no match is found in the deferred event lists either, the event is deleted from the queue, and the next event is processed.

A statechart can make several consequent steps processing several signal events from the queue (these steps take zero model time). When the statechart finishes processing and starts waiting, the event queue is either empty or contains only currently deferred events.

9.5 Observing statechart at runtime

9.5.1 Animated statechart diagram

A statechart running in the model is visualized in the animated statechart diagram window, see Figure 130. Animated statechart diagram looks similar to the statechart diagram editor, but editing is not allowed and color animation, breakpoints, and other viewer features are enabled.

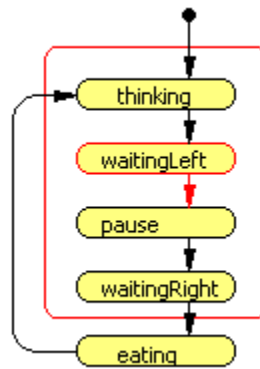


Figure 130. Animated statechart diagram

You can open the animated statechart diagram either from the Model Explorer or from the animated structure diagram.

► **To open the animated statechart diagram of a statechart**

1. Double-click the statechart, or
Right-click the statechart and choose *Statechart* from the popup menu.

To help you to locate the current activity within the model, AnyLogic highlights active objects in the animated statechart diagram, see section 11.2.3.1, “Color highlighting of model items”.

9.5.2 Debugging a statechart

Using the animated statechart diagram window you can set breakpoints on states and transitions of statecharts.

► **To set/clear breakpoint on a state or transition**

1. Right-click the state or transition and choose *Breakpoint* from the popup menu.

► **To set/clear breakpoint on a statechart**

1. Right-click the statechart and choose *Breakpoint* from the popup menu.

You can display the information about the current state of a statechart in statechart's inspect window. More precisely, the inspect window displays the inspect string associated with the object. You can set inspect string for a statechart manually.

► **To open the inspect window of a statechart**

1. Right-click the statechart and choose *Inspect* from the popup menu.

You can define custom inspect strings for a statechart. This is done using the following API (for more information, please consult AnyLogic Class Reference):

Related method of Statechart

```
void setThreadInspect( java.lang.String value ) – sets the inspect string  
for the statechart.
```

10. Stochastic modeling

A model can be stochastic as well as deterministic. There are many different ways to incorporate nondeterminism into a model. For example, you can assign a randomly generated time value to a transition, timer, or delay operation. Or a random value or its derivative can be used to determine a message destination address, evaluate a guard expression, or otherwise impact the model behavior.

There is also a case when the model can have stochastic behavior, even if you do not specify it explicitly using randomly generated values: this is random serialization of simultaneous events, see section 14.2.1.3, “Event step”. If several events are available at the same time, AnyLogic can make nondeterministic choice with equal probability for each event.

Otherwise, the model behavior is deterministic and 100% reproducible irrespective of the seed of the random number generator.

10.1 Random number generator

Stochastic models require a random seed value for the pseudorandom number generator. In this case model runs cannot be reproduced since the model random number generator is initialized with different values for each model run. Specifying the fixed seed value, you initialize the model random number generator with the same value for each model run, thus the model runs are reproducible.

All AnyLogic distribution classes are implemented on the basis of the standard Java random number generator `java.util.Random`. However, if you wish to use your own one, you should call the static method `setRandomGenerator(java.util.Random gen)` of the class `Distr`.

The seed of the standard random number generator can be defined in the experiment properties, or, if you are running the model from command line, using the command line option `--seed` (see section 18.1, “Running a model from the command line”).

► To set random/fixed random number generator seed value

1. In the Project window, click the experiment you are setting random number seed value for.
2. On the *Additional* page of the Properties window, choose the *Random seed (unique experiments)/Fixed seed (reproducible experiments)* option.
3. For the fixed seed, enter the seed value in the *Seed value* edit box.

If the model does not receive any external input (either data or user actions), the behavior of the model in two simulations with the same initial seeds is identical. The random number generator is initialized once when the model is created and is not reinitialized between model replications.

10.2 Probability distributions

You may need to model non-deterministic processes, e.g. weather changing, product demand changing, or a random choice. Stochastic processes can be modeled using probability distributions.

AnyLogic comes with a large set of probability distributions. The corresponding classes are defined in the package `com.xj.random`. The generic class `Distr` has just one abstract method `get()`, which should return a generated sample. You can create empirical distributions from your own data - then you should define your own distribution class, inheriting from `Distr`.

The probability distribution classes inherit from `Distr`. They have names like `DistrExponential`, `DistrChi`, `DistrNormal`, etc. They support two forms of obtaining random values:

- You can instantiate a distribution class with particular parameters, e.g. `Distr myDistr = new DistrExponential(0.3);` and then call its method `myDistr.get()`
- You can call the static method `sample()`, defined in each class to obtain random values without instantiating the distribution class. For the above distribution this looks like `DistrExponential.sample(0.3)`.

This is the API you use (please consult AnyLogic Class Reference for more details):

Related method of Distr

`abstract double get()` – returns a random value depending on the actual probability distribution implemented by a derived class.

Related method of Distr<xxx>

`static <type> sample(<params>)` – returns a random value of either `int` or `double` type distributed according to the distribution. Return type and parameters depend on the distribution.

Supported distributions are summarized in Table 16.

Type	Class	Parameters and formulas
Bernoulli	DistrBernoulli	double p $\text{prob}(0) = 1-p, \text{prob}(1) = p$
Beta	DistrBeta	double a, double b $p(x) = (\text{Gamma}(a + b)/(\text{Gamma}(a)*\text{Gamma}(b))) x^{(a-1)} (1-x)^{(b-1)}$
Binomial	DistrBinomial	double p, int n $\text{prob}(k) = n!/(k!(n-k)!) * p^k (1-p)^{(n-k)}$ for $k = 0, 1, \dots, n$
Cauchy	DistrCauchy	double mu $p(x) = (1/(\pi \mu)) (1 + (x/\mu)^2)^{-1}$
Chi	DistrChi	int num

Chi2	DistrChi2	double nu $p(x) = (1/\text{Gamma}(\text{nu}/2)) (x/2)^{(\text{nu}/2 - 1)} \exp(-x/2)$ for x = 0 ... +infty
Constant	DistrConst	double value
Erlang	DistrErlang	double a, int n $p(x) = x^{(n-1)} \exp(-x/a) / ((n-1)!a^n)$ for x = 0 ... +infty
Exponential	DistrExponential	- double mu $p(x) = \exp(-x/\text{mu})/\text{mu}$ for x = 0 ... +infty
Exponential Power	DistrExpPow	double mu, double a $p(x) = (1/(2 \text{ mu Gamma}(1+1/a))) * \exp(- x/\text{mu} ^a)$ for -infty < x < infty.
F	DistrF	double nu1, double nu2 $p(x) = (\text{nu1}^{(\text{nu1}/2)} \text{nu2}^{(\text{nu2}/2)} \text{Gamma}((\text{nu1} + \text{nu2})/2) / \text{Gamma}(\text{nu1}/2) \text{Gamma}(\text{nu2}/2)) * x^{(\text{nu1}/2 - 1)} (\text{nu2} + \text{nu1} * x)^{-(\text{nu1}/2 - \text{nu2}/2)}$
Gamma	DistrGamma	double a, double b $p(x) = \{1 / \backslash\text{Gamma}(a) b^a\} x^{\{a-1\}} e^{\{-x/b\}}$ for x>0.

Gaussian	DistrGaussian	- double sigma double u = x / fabs (sigma); double p = (1 / (sqrt (2 * M_PI) * fabs (sigma))) * exp (-u * u / 2); return p;
GaussianTail	DistrGaussianTail	double a, double sigma
Geometric	DistrGeometric	double p prob(k) = p (1 - p)^(k-1) for n = 1, 2, 3, ...
Gumbel1	DistrGumbel1	double a, double b $p(x) dx = a b \exp(-(b \exp(-ax) + ax)) dx$
Gumbel2	DistrGumbel2	double a, double b $p(x) dx = b a x^{-(a+1)} \exp(-b x^{-a}) dx$
Hypergeometric	DistrHypergeometric	int n1, int n2, int t prob(k) = choose(n1,t) choose(n2, t-k) / choose(n1+n2,t) where choose(a,b) = a!/(b!(a-b)!)
Laplace	DistrLaplace	- double mu $p(x) = (1/(2 \mu)) * \exp(- x/\mu)$ for $-\infty < x < \infty$
Levy	DistrLevy	double mu, double alpha $p(x) = (1/(2 \pi)) \int dt \exp(it(\mu-x) - ct ^{\alpha})$ with $0 < \alpha \leq 2$

Logarithmic	DistrLogarithmic	double p $\text{prob}(n) = p^n / (n \log(1/(1-p)))$ for $n = 1, 2, 3, \dots$
Logistic	DistrLogistic	double mu $p(x) = (1/\mu) \exp(-x/\mu) / (1 + \exp(-x/\mu))^2$ for $-\infty < x < \infty$
Lognormal	DistrLognormal	double zeta, double sigma $p(x) = 1/(x * \sqrt{2 \pi \sigma^2}) \exp(-(\ln(x) - zeta)^2/2 \sigma^2)$ for $x > 0$.
Negative Binomial	DistrNegativeBinomial	double p, double n $\text{prob}(k) = \text{Gamma}(n + k)/(\text{Gamma}(n) \text{Gamma}(k + 1)) p^n (1-p)^k$ for $k = 0, 1, \dots$
Normal	DistrNormal	- double sigma double sigma, double mean
Pareto	DistrPareto	double a double a, double b $p(x) = (a/b) / (x/b)^{a+1}$ for $x \geq b$
Pascal	DistrPascal	double p, int n $\text{prob}(k) = (n - 1 + k)! / (n!(k - 1)!) * p^n (1-p)^k$ for $k = 0, 1, \dots, n$

Poisson	DistrPoisson	double mu $p(n) = (\mu^n / n!) \exp(-\mu)$ for $n = 0, 1, 2, \dots$
Rayleigh	DistrRayleigh	- double sigma $p(x) = (x / \sigma^2) \exp(-x^2 / (2 \sigma^2))$ for $x = 0 \dots +\infty$
RayleighTail	DistrRayleighTail	double a double a, double sigma
Student	DistrStudent	int num
T	DistrT	double nu $p(x) = (\Gamma((\nu + 1)/2) / (\sqrt{\pi \nu} \Gamma(\nu/2))) * (1 + (x^2)/\nu)^{-((\nu + 1)/2)}$
Triangular	DistrTriangular	double a, double b double a, double b, double c $p(x) = 2*(x-a)/((b-a)*(c-a))$ if $a \leq x < c$; $p(x) = 2*(b-x)/((b-a)*(b-c))$: $c \leq x \leq b$; $p(x) = 0$: otherwise,
Uniform	DistrUniform	- double max double min, double max $p(x) = 1/(b-a)$ if $a \leq x < b$ $p(x) = 0$, otherwise
Weibull	DistrWeibull	double mu, double a $p(x) = (a/\mu) (x/\mu)^{a-1} \exp(-(x/\mu)^a)$

Table 16.

To simplify work with probability distributions, AnyLogic `ActiveObject` class has static methods (inherited from `Func` class), wrapping the described above methods of probability distribution classes. You can simply call the static method of the `ActiveObject` class to obtain random values without instantiating the distribution class, e.g. `ActiveObject.exponential(0.6)`. For more information please consult AnyLogic Class Reference.

11. Running and observing a model

If you have already built up your model, you may run the model simulation. There are several ways to run the model generated by AnyLogic. You can run the model:

- Directly from AnyLogic.
- Using command line on any Java-enabled machine, see section 18.1, “Running a model from the command line”.
- As an applet, see section 18.2, “Running a model as an applet”.

This chapter describes how to run a model from AnyLogic. This is the simplest way of running the model. The same instance of AnyLogic that was used for editing and building the model launches and serves as a model viewer, controller, and debugger. If you wish, you can specify arbitrary options for Java virtual machine.

AnyLogic gives you the maximum control over the model execution. AnyLogic is an ideal environment for iterating the design and debugging your model (the detailed information on debugging AnyLogic models is given in Chapter 14, “Debugging a model”). AnyLogic shows you the running model in terms of design notation. AnyLogic visual model viewer/debugger features include:

- Step and run modes with a variety of options
- Easy navigation with Model Explorer: any model element is accessible
- On-the-fly animation of structure and statecharts
- Graphical breakpoints, log and inspect windows
- Dataset visualization (scatter, histogram, Gantt) and export to other applications
- Event viewer
- Variable and parameter modification at runtime.

Please note that you cannot use AnyLogic to view a model running as an applet.

11.1 Running the model with AnyLogic


View and debug your model using the *Model* and *View* items in the main menu and the *Model toolbar*, Figure 131. Not all of these buttons are present on the toolbar by default. To add/remove buttons, use the *Customize* dialog box (see section 21.1, “Customizing toolbars and menus”).




Figure 131. Model toolbar

11.1.1 Creating and destroying the model

► **To bring the project up to date and create the model (but not start it)**

1. Click the *Step*  toolbar button, or
Choose *Model|Step* from the main menu, or
Press F10.

► **To bring the project up to date and run the model**

1. Click the *Run*  toolbar button, or
Choose *Model|Run* from the main menu, or
Press F5.

You can tell AnyLogic to save your project every time before creating the model.

► **To set project autosave before creating the model**


1. Choose *Tools|Options...* from the main menu.
The *Options* dialog box is displayed.
2. On the *Miscellaneous* page, select the *Autosave project on model run* check box.
3. Click *OK*.

You can set up a timeout for creating a model. When you are creating the model and the specified timeout elapses, AnyLogic stops model creation and displays the message box, with the “*Error occurred during the model creation*” error message.

► **To set up a model creation timeout**

1. Choose *Tools|Options...* from the main menu.
The *Options* dialog box is displayed.
2. Click the *Viewer* tab of the dialog box.
3. In the *Model creation timeout (sec)* edit box, specify the timeout for model creation (in seconds).

► **To destroy the created model**

1. Click the *Stop*  toolbar button, or
Choose *Model|Stop* from the main menu, or
Press Shift+F5.

11.1.2 Controlling the model execution


AnyLogic can execute the model in a variety of modes. The modes differ in how frequently the viewer windows are updated (e.g., on every simulation step, each 5 seconds, on every visible change, etc.), and when the model stops to give control back to the user.

The simulation performance depends on the execution mode and on the windows opened in the viewer. The fewer windows are opened, the faster the simulation runs.

There is a notion of simulation step. Each step corresponds to an event in AnyLogic simulation engine, see section 14.2.1, “Event processing at the simulation engine”. In complex models there may be a large density of events, and normally you do not need to trace the model execution to that level of detail. Moreover, depending on which windows are opened in AnyLogic viewer, some steps may appear invisible for the user, because they correspond to events of the objects currently not displayed. Therefore, for user convenience, “regular” *Run* and *Step* commands make steps with respect to visible changes only. However,

in case you wish to see all primitive steps, you may use *Detailed Play* and *Primitive Step* commands.

► To run the model

1. Click the *Run*  toolbar button, or
Choose *Model|Run* from the main menu, or
Press F5.

When a model starts running, current model time, step and replication values are displayed in the status bar, see Figure 132.

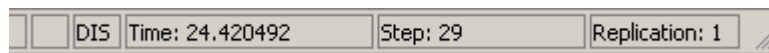


Figure 132. Status bar

You can change the period of status bar update during model run.

► To set up the status bar update period

1. Choose *Tools|Options...* from the main menu.
The *Options* dialog box is displayed.
2. Click the *Viewer* tab of the dialog box.
3. In the *Model status refresh interval (ms)* edit box, specify the update period in milliseconds.

By default, the *Run* command executes the model as fast as possible, refreshing windows rarely with the period of time specified in AnyLogic options. However, you can refresh windows when needed.

► To refresh windows (excluding animation window) when running the model

1. Choose *Model|Refresh Windows* from the main menu.


You can configure the viewer update period during model run using the *Viewer* page of the *Options* dialog box.

► **To set up the viewer update period**


1. Choose *Tools|Options...* from the main menu.
The *Options* dialog box is displayed.
2. Click the *Viewer* tab of the dialog box.
3. In the *Forced screen refresh interval during play (ms)* edit box, specify the viewer update period (in milliseconds).

You can turn the Views Auto Refresh mode on, in which case the model runs in such a way that each change in open windows is reflected.


► **To turn the Views Auto Refresh mode on/off**

1. Click the *Views Auto Refresh*  toolbar button, or
Choose *Model|Auto Refresh Views* from the main menu.
If Views Auto Refresh mode is set, the button is shown pressed.

► **To make a step (run until a change in any window, then stop)**

1. Click the *Step*  toolbar button, or
Choose *Model|Step* from the main menu, or
Press F10.

► **To pause the model**

1. Click the *Pause*  toolbar button, or
Choose *Model|Pause* from the main menu, or
Press Ctrl+F10.

► **To play the model with respect to the current window
(as run, screen updated on any change in the current window)**

1. Choose *Model|Play in Window* from the main menu.

► **To make one step with respect to the current window
(run until a change in the current windows, then stop)**

1. Choose *Model|Step in Window* from the main menu.

► **To make one primitive step
(move to the next event of the simulation engine, then stop)**

1. Choose *Model|Primitive Step* from the main menu, or Press F11.

► **To play the model displaying maximum details
(non-stop execution, screen updated on each event of the simulation engine)**

1. Choose *Model|Detailed Play* from the main menu, or Press Ctrl+F11.

► **To run the model until a particular time or step condition holds**

1. Choose *Model|Run to* from the main menu.
The *Run To Condition* dialog box is displayed, as shown in Figure 133.

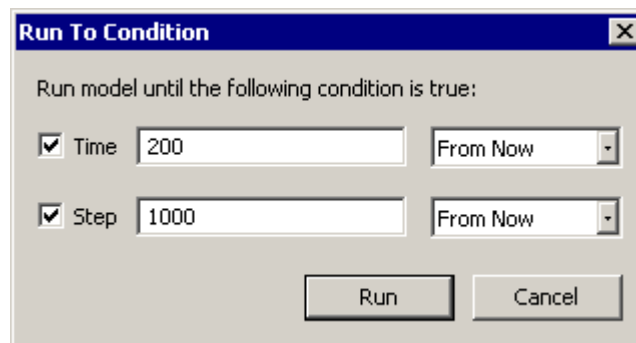



Figure 133. Run To Condition dialog box

2. Check the type of stop condition (*Time*, *Step*, or both).
3. Type the time in the *Time* edit box. In the corresponding drop-down list specify whether this time is absolute or relative to the current model time.

4. Type the step in the *Step* edit box. In the corresponding drop-down list, specify whether this step number is absolute or relative to the current model step.
5. Click the *Run* button.

► **To restart the model**

1. Click the *Restart Model*  toolbar button, or Choose *Model|Restart* from the main menu, or Press Ctrl+Shift+F5.


The model restart means that all objects in the model are destroyed and created again. The model time is set to 0. No steps are executed. For more information on initialization order see section 22.2.1, “Model initialization order”.

The commands *Model|Run*, *Model|Step*, etc. become disabled when there is no activity in the model. This indicates that your model has finished.

11.1.3 Setting up model simulation speed

AnyLogic model can be run either in real time or virtual time mode. In real time mode, the mapping of AnyLogic model time to the real time is made. It is frequently needed when you have developed some animation and want it to appear as in real life. In virtual time mode the model runs at its maximum speed and no mapping is made between model time units and a second of astronomical time.



► **To set virtual/real time mode**

1. Click the *Enable virtual time mode*  toolbar button.
If virtual time mode is set, the button is shown pressed.

In the real time mode, you can increase or decrease model speed by changing the model simulation speed scale (see section 13.1, “Simulation speed” to know how to define model simulation speed). The model speed scale value is displayed in the *Model speed* toolbar combo box. The default 1x scale means that the model is simulated with the model simulation speed defined in the properties of the current AnyLogic experiment, 2x means that model is run

twice faster than the specified model speed, etc. For instance, if model speed is 6 model time units per second, 2x means that 12 model time units correspond to 1 second. You can change the model simulation speed as you like.

► **To change the model simulation speed scale**

1. To decrease model speed, click the *Decrease model speed*  toolbar button, or Choose from the main menu.
2. To increase model speed, click the *Increase model speed*  toolbar button, or Choose from the main menu.

11.2 Viewing and controlling the model

You view and control the running model using the following windows (see Figure 134): the Model Explorer window, animated structure diagram window, animated statechart diagram window, inspect, log, chart windows.

The Model Explorer window displays the model elements, organized in a tree, and provides easy navigation and fast access to any of them. The animated structure diagram window visualizes the structure diagram of an active object and provides access to its elements. The variables and parameters changes are automatically displayed and active elements of model are graphically highlighted.

Using these windows you can:

- Open log windows, displaying textual output of a model or of an active object.
- Open inspect windows, displaying the information about the current status of model elements.
- Modify parameters and variables of active objects.
- Visualize datasets by various types of charts (scatter, histogram, Gantt).
- Set up breakpoints on elements of the model.

The animated statechart diagram window provides on-the-fly animation of a statechart and provides access to statechart elements.

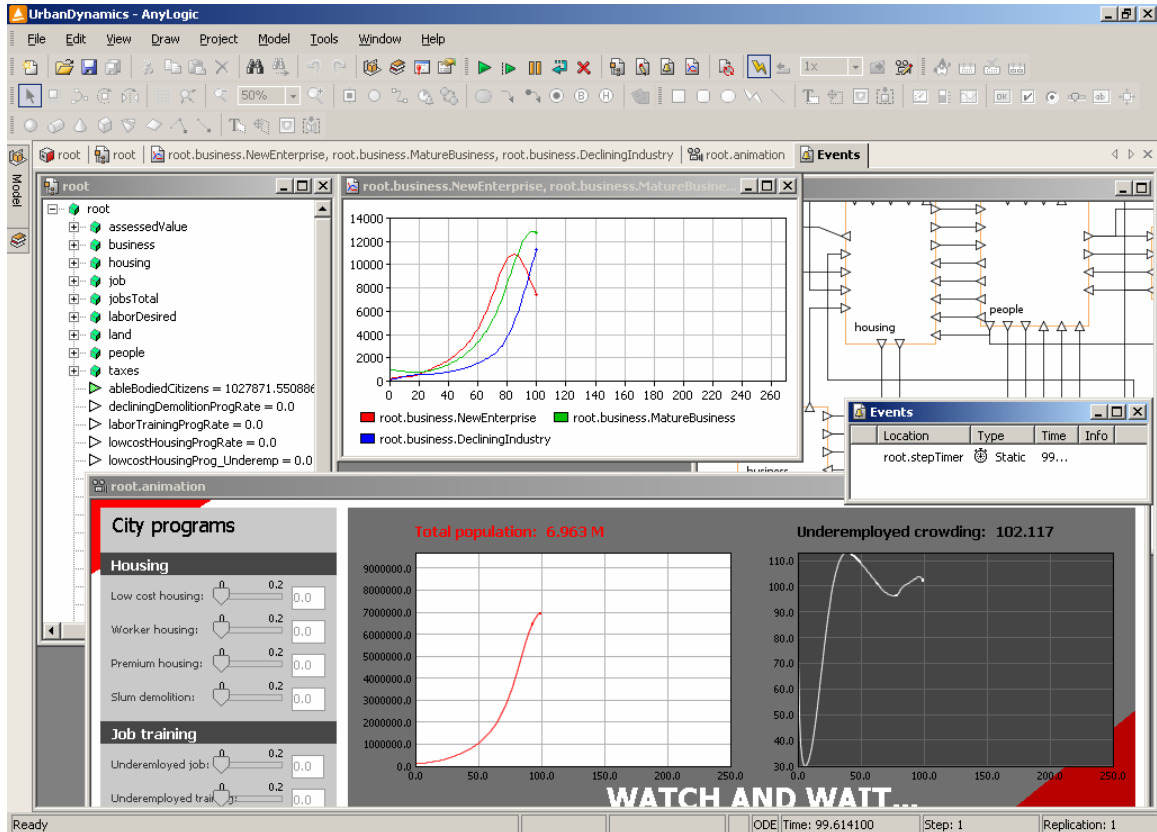


Figure 134. AnyLogic viewer windows

Use AnyLogic *Window* menu options to arrange open windows (see section 1.3.3, “Arranging windows”). If needed, set custom color scheme for the AnyLogic viewer using the *Colors* page of the *Options* dialog box, see section 21.2, “Customizing colors”.

11.2.1 Model Explorer

The Model Explorer window, see Figure 135, displays the objects currently existing in the model and provides easy navigation and fast access to any of them. Since an AnyLogic model is hierarchical, the objects are organized in a tree, with the root object at the root. The structure of an AnyLogic model may change dynamically, and the explorer tree changes to reflect this.

Note that while items in the Project window are classes, items in the Model Explorer are instances.

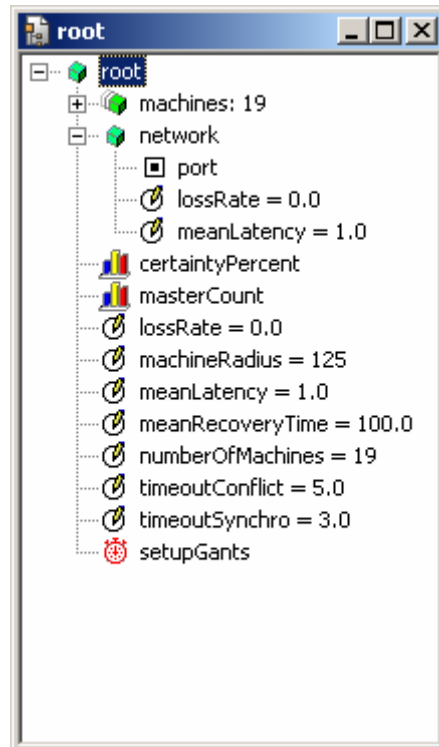


Figure 135. Model Explorer

► To open the Model Explorer of the root object

1. Click the *Model Root Object*  toolbar button, or Choose *View | Model Root Object* from the main menu.

If you select an item in the Model Explorer, the Properties window displays the inspect window of the item. The inspect window shows the item's inspect string obtained from the model. In addition, if the selected item is an active object, the Properties window displays the log window.

You can open more explorers having different objects at roots of their trees. The new Model Explorer displays the subtree of the object the explorer is opened on. This is a way of shifting the Model Explorer base along the object hierarchy. There is also a backward

operation: the Model Explorer can go up through the object hierarchy until the root object is met.

► **To explore an active object in a new Model Explorer**

1. Right-click the active object in an existing Model Explorer and choose *Explore* from the popup menu.

► **To shift the Model Explorer up one level**

1. Right-click the Model Explorer and choose *Up* from the popup menu.

The Model Explorer can show a large set of items: active objects, ports, variables, parameters, statecharts, datasets, threads, chart timers, chart events. Sometimes it is desirable to hide some of them. AnyLogic allows the user to choose what to show and what to hide.

► **To show/hide items**

1. Right-click the Model Explorer and choose the corresponding item from the *Hide Items* item of the popup menu, e.g. choose *Hide Items | Parameters* to show/hide parameters.

► **To show/hide behavior items (statecharts, threads, events, and timers)**

1. Right-click the Model Explorer and choose *Hide Items | Behavior Items* from the popup menu.

► **To show/hide structure items (objects, ports, variables, parameters, and datasets)**

1. Right-click the Model Explorer and choose *Hide Items | Structure Items* from the popup menu.

► **To show/hide all items except objects**

1. Right-click the Model Explorer and choose *Hide Items | All Except Objects* from the popup menu.

Every item in the Model Explorer has a set of actions associated with it, accessible through the item's popup menu. For example, an encapsulated object can be explored in a new Model Explorer, its structure diagram, log, and inspect windows can be opened, and a breakpoint can be set on it.

11.2.1.1 Modifying variables and parameters

You can modify parameters and variables of active objects from the Model Explorer.

► To modify a variable/parameter

1. Double-click the variable/parameter, or
Right-click the variable/parameter and choose *Modify* from the popup menu.
The *Modify* dialog box is displayed.
2. Type a new value in the *Enter new value* edit box.
3. Click *OK*.

11.2.1.2 Color highlighting of model items

Objects displayed in the Model Explorer (active objects, ports, threads, timers, etc.) may be involved in steps executed by the simulation engine. For example, an event step may be associated with a timer expiry. Such shapes are highlighted in the Model Explorer to help you to locate the current activity within the model.

The default highlight colors and their meanings are given in the table below. You can change the default color scheme using the *Colors* page of the *Options* dialog box.

Item	Color (default)	Status
Timer	Red	Chosen. Will expire at this step.
Thread	Red	Chosen. Will advance at this step.
Statechart	Red	Chosen. One of the statechart's transitions is chosen.

Table 17.

Please note that an active object is highlighted only if an activity takes place exactly at its own statecharts, timers, or threads. The activity of an encapsulated object does not affect highlighting of its parent objects.

11.2.2 Animated structure diagram

The animated structure diagram window, see Figure 136, visualizes the structure diagram of an active object. It looks like the structure diagram editor, but editing is not allowed and color animation, breakpoints, and other viewer features are enabled.

While the structure diagram editor window corresponds to an active object class, the animated structure diagram window corresponds to an instance of an active object class.

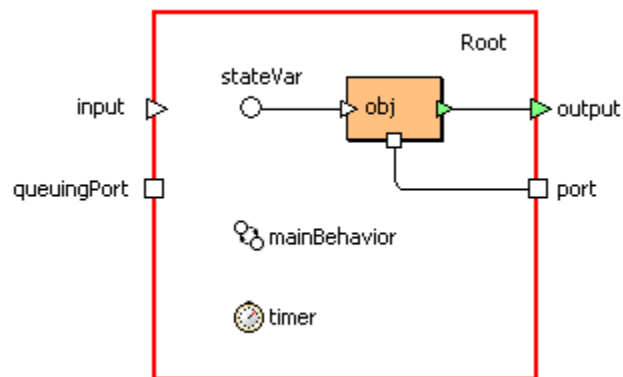


Figure 136. Animated structure diagram window

The animated structure diagram displays chart timers only, whereas the Model Explorer displays all currently existing timers: chart timers as well as static and dynamic timers.

You can open the animated structure diagram from the Model Explorer, or from another animated structure diagram.

► **To open the animated structure diagram of an active object from the Model Explorer**

1. Double-click the active object in the Model Explorer, or
Right-click the active object in the Model Explorer and choose *Structure* from the popup menu.

► **To open the animated structure diagram of an encapsulated object from the animated structure diagram of a parent object**

1. Double-click the encapsulated object, or
Right-click the encapsulated object and choose *Structure* from the popup menu.

► **To open the animated structure diagram of a parent object from the animated structure diagram of an encapsulated object**

1. Right-click the empty area of the animated structure diagram and choose *Up to Parent* from the popup menu.

You can get the image of the animated structure diagram (including color highlighting) on the Clipboard:

► **To copy the image of the animated structure diagram on the Clipboard**

1. Click on the diagram, and choose *Draw | Copy Image* from the main menu, or
Right-click the empty area of the diagram and choose *Copy Image* from the popup menu.

Every shape on the animated structure diagram has a set of actions associated with it, accessible through the shape's popup menu. For example, an encapsulated object can be explored in the Model Explorer, its structure diagram, log, and inspect windows can be opened, and a breakpoint can be set on it.

► **To explore an active object in the Model Explorer**

1. Right-click the active object and choose *Explore* from the popup menu.

11.2.2.1 Modifying variables and parameters

You can modify parameters and variables of active objects from the animated structure diagram.

► **To modify a variable/parameter**

1. Double-click the variable/parameter, or
Right-click the variable/parameter and choose *Modify* from the popup menu.
The *Modify* dialog box is displayed.
2. Type a new value in the *Enter new value* edit box.
3. Click *OK*.

11.2.2.2 Color highlighting of model items

Objects, involved in steps executed by the simulation engine are highlighted in animated diagrams. The default highlight colors and their meanings are given in the table below.

Item	Color (default)	Status
Active object	Red	Chosen. One of the statecharts or timers of this active object is chosen.
Chart timer	Red	Chosen. Will expire at this step.
Statechart icon	Red	Chosen. One of the statechart's transitions is chosen.

Table 18.

Please note that an active object is highlighted only if an activity takes place exactly at its own statecharts, timers, or threads. The activity of an encapsulated object does not affect highlighting of its parent objects.

11.2.2.3 Animating active objects

If you have developed some icon for an active object (see section 1.5.3, “Active object icon”), each time an instance of this object appears as an encapsulated object on an animated structure diagram, this icon is displayed. Since you can link properties of the shapes your icon is constructed from to active object data, you can animate active objects on the animated structure diagram.

11.2.3 Animated statechart diagram

The animated statechart diagram window, see Figure 137, visualizes a statechart running in the model. It looks similar to the statechart diagram editor, but editing is not allowed and color animation, breakpoints, and other viewer features are enabled.

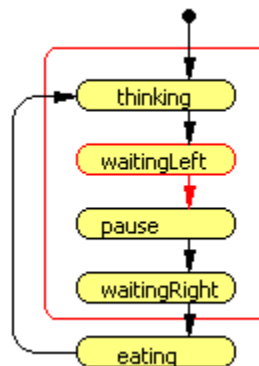


Figure 137. Animated statechart diagram

You can open the animated statechart diagram from the Model Explorer or from the animated structure diagram.

► To open the animated statechart diagram of a statechart

1. Double-click the statechart, or
Right-click the statechart and choose *Statechart* from the popup menu.

► **To open the animated structure diagram of a parent object from the animated statechart diagram**

1. Right-click the empty area of the animated statechart diagram and choose *Up to Parent* from the popup menu.

You can get the image of the animated statechart diagram (including color highlighting) on the Clipboard:

► **To copy the image of the animated statechart diagram on the Clipboard**

1. Click on the diagram, and choose *Draw | Copy Image* from the main menu, or Right-click the empty area of the diagram and choose *Copy Image* from the popup menu.

Using the animated statechart diagram window you can set breakpoints on states and transitions of statecharts.

11.2.3.1 Color highlighting of model items

Objects involved in steps executed by the simulation engine are highlighted in the animated statechart diagram. The default highlight colors and their meanings are given in the table below.

Item	Color (default)	Status
State	Red	Active. The control is at this state. If this is a composite state, the exact location of control is in one of the inner simple states.
Transition	Red	Chosen. Will be taken at this step.
Transition	Blue	Enabled. Could be taken at this step, but some other event has been chosen. This can be changed using the events window, see section 14.2, “Viewing and modifying AnyLogic events”.

Table 19.

11.2.4 Inspect window

You can display the information about the current state of a model element in the element's inspect window. More precisely, the inspect window displays the inspect string associated with the object. Some objects (datasets, variables) have hard coded inspect, see Figure 138. Inspect strings for active objects, statecharts and ports can be set manually.

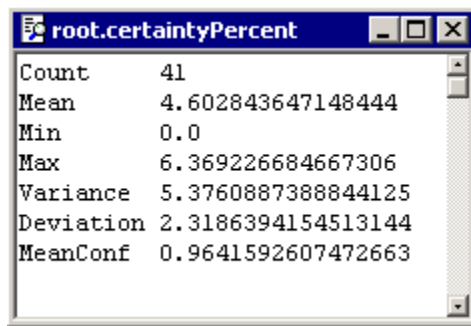


Figure 138. Inspect window of a dataset

The inspect window appears in the Properties window when you select a model item. Also, a standalone inspect window can be opened from the popup menu of an item.

► To open the inspect window of a model item

1. Right-click the item in the Model Explorer or on the animated structure diagram and choose *Inspect* from the popup menu.

11.2.4.1 Defining custom inspect

You can define custom inspect strings for an active object, a port and a statechart. This is done using the following API (for more information, please consult AnyLogic Class Reference):

Related method of ActiveObject

`void setInspect(java.lang.String value)` – sets the inspect string for the active object.

Related method of Port

`void setPortInspect(java.lang.String s)` – sets the inspect string for the port.

Related method of Statechart

`void setThreadInspect(java.lang.String value)` – sets the inspect string for the statechart.

11.2.5 Log window

The log window, see Figure 139, displays textual output of a model (global log) or of an individual active object. The log is displayed as read-only text which can be copied onto the Clipboard. The log window of an active object appears in the Properties window when you select an active object. Also a standalone log window can be opened from the popup menu of an active object.

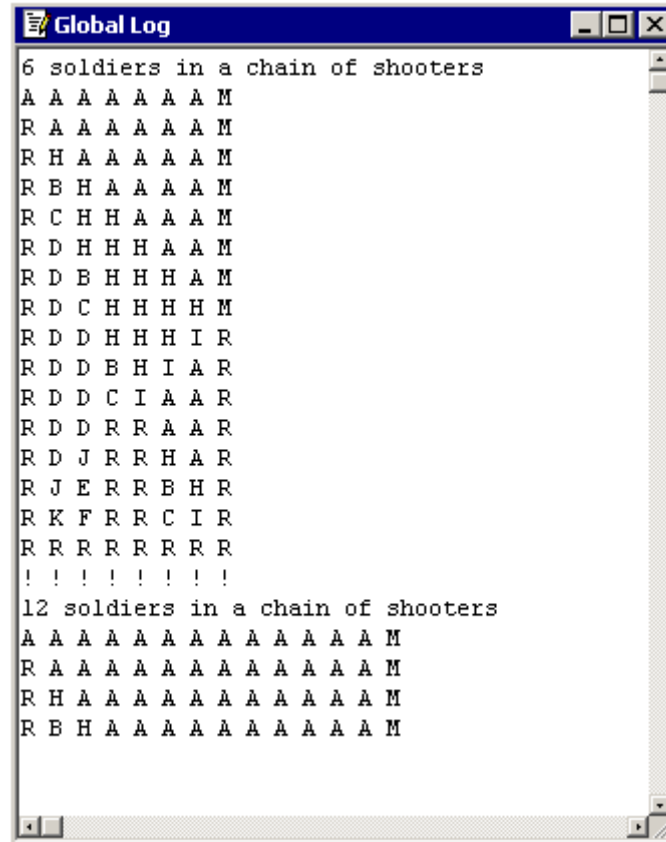


Figure 139. Log window

The global log is convenient for output of information across several model replications, because it is not reset in between replications. It can also be used as a debugging tool, for example, to find out what is the order in which the model executes actions of different objects.

To write to the global log, you use the object `Engine.log` or the methods `trace()` and `traceln()` of the class `ActiveObject`. To write to the active object's log, you use the member variable `log` of an active object. See section 14.4, “Logging a model” to know how to work with logs.

► To open the global log

1. Click the *Global Log*  toolbar button, or Choose *View|Global Log* from the main menu.

11.2.6 Chart window

Chart windows are used to visualize and export data collected in a model. You can visualize data collected explicitly in datasets you have created as well as implicitly since AnyLogic automatically collects datasets for variables. See Chapter 17, “Collecting data and performing statistical analysis” for more information about collecting and analyzing data.

► To open a blank chart window

1. Click the *New Chart*  toolbar button, or Choose *View|New Chart* from the main menu.

► To open a chart window with a particular dataset displayed

1. Double-click the dataset in the Model Explorer, or Right-click the dataset in the Model Explorer and choose *Chart* from the popup menu.

► To open a chart window with a particular variable displayed

1. Right-click the variable in the Model Explorer and choose *Chart* from the popup menu.

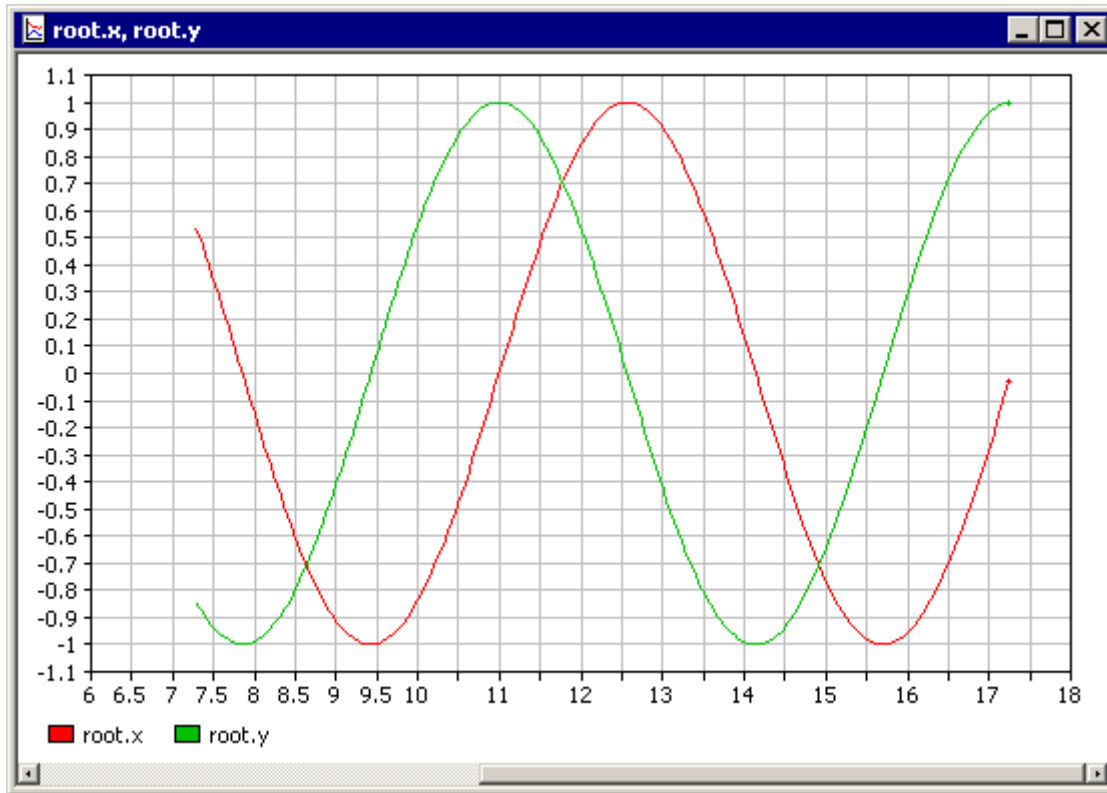


Figure 140. Chart window

Chart window looks as shown in Figure 140.


Datasets and variables can be added to a chart window using drag and drop or using the *Chart Setup* dialog box.

► **To add a dataset or a variable to a chart window**

1. Drag the dataset/variable from the Model Explorer onto the chart window.

► **To set up the chart**

1. Right-click the chart window and choose *Chart Setup...* from the popup menu. The *Chart Setup* dialog box is displayed (see Figure 141).
2. To add a variable to the chart, double-click the corresponding item in the *Variables, parameters and datasets* list.

3. To remove a variable from the chart, double-click the corresponding item in the *Axis Y* list.
4. By default, *Time* is chosen in the *Axis X* list, that is the chart is timed. If you need to plot one variable against another variable, dataset, or parameter, make the chart phased. Set up the variable/dataset/parameter to be displayed on the x-axis by clicking the corresponding item in the *Variables, parameters and datasets* list, and then clicking the  button to the left of the *Axis X* list. To make plot timed again, remove the variable/dataset/parameter item from the *Axis Y* list by double-clicking.
5. Click *OK*.

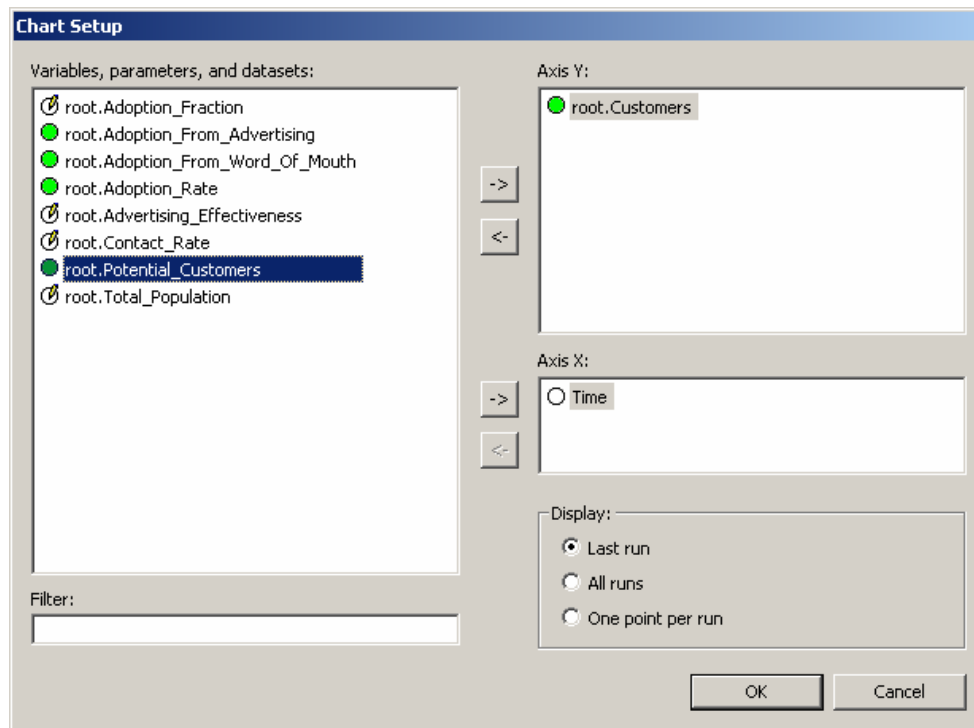


Figure 141. Chart Setup dialog box

11.3 Debugging the model

AnyLogic provides various tools for convenient model debugging.

AnyLogic supports on-the-fly checking of model syntax.

AnyLogic allows you to view what is happening at the simulation engine at the lowest level details and make some changes to the event processing.

AnyLogic allows you to debug your model by throwing runtime errors and tracing model execution by setting breakpoints on model elements and writing custom information to AnyLogic logs.

AnyLogic detects logical errors of model execution and errors in your Java code. If such an error occurs, AnyLogic stops the model and notifies you with error message.

AnyLogic enables you to debug Java code using a third-party debugger by running the model within the debugger using command line execution feature or by attaching the debugger to the currently running model.

The detailed information on debugging a model is given in Chapter 14, “Debugging a model”.

11.4 Setting up model execution parameters

An AnyLogic model is executed with a set of simulation settings (simulation speed, number of model runs). You can adjust model execution settings by customizing AnyLogic experiments. See Chapter 13, “Simulation settings” to know how to set required model execution settings.

11.5 Optimizing a model

If you need to run a simulation and observe system behavior under certain conditions, as well as improve system performance, for example, by making decisions about system parameters and/or structure, you can use the optimization capability of AnyLogic. Simulating a model with different parameters, AnyLogic automatically finds the optimal values of model parameters, with respect to certain constraints. See Chapter 16, “Optimization” for details.

12. Animation

AnyLogic offers a unique animation technology that enables you to construct very complex 2D and 3D animations rapidly using the object structure of your model. It features:

- *Modular development.* You develop animations in a modular way, separately for each object. AnyLogic takes care of assembling the picture, performs placement and transformation of its elements.
- *Scalability.* Animation scales with your model as you vary its size.
- *Reuse.* Animations are associated with active objects. They can be incorporated into any higher-level animation scene associated with a container object. If you put an object into a library, animation is stored there as well.
- *Integrated animation editor.* AnyLogic animation editor is integrated with its model development environment, sharing the Project and Properties windows.
- *Rich API.* For sophisticated cases that cannot be specified using the graphical animation editor, AnyLogic offers a rich API capable of solving virtually any animation need.
- *100% Java.* AnyLogic animations are 100% Java.
- *Web.* Animations can be accessed over the Internet and displayed in Web browser as applets.
- *Interactivity.* 2D animations are interactive, offering the user a full range of controls (buttons, text inputs, checkboxes, sliders, knobs, etc.).

The animation window of the root object of a model automatically appears when the model starts. The speed of animation and, correspondingly, the speed at which the model runs, can be set in AnyLogic by specifying the mapping of model time units to seconds in real time mode.

12.1 Animation concepts

The basic idea behind AnyLogic animation technology is that animations (drawings constructed of elementary shapes) are associated with model components – active objects –

and are composed according to the model hierarchy. In this section we describe animation of a standalone active object (section 12.1.1, “Reflecting the state of an object in animation”), then animation reflecting a hierarchical model (section 12.1.2, “Animating hierarchical models”). Interactive animations are considered in section 12.1.3, “Interactive control of animation”.

12.1.1 Reflecting the state of an object in animation

Each active object can have an associated animation. Animation is a drawing composed of various shapes: circles, rectangles, lines, etc., and also indicators and controls. Each shape has a number of properties defining its visual appearance: position, height, width, color, and so on. These properties are typically organized as shown in Figure 142:

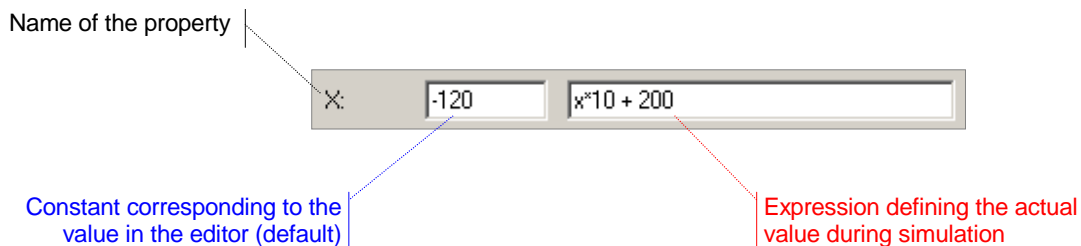


Figure 142. Property of an animation shape

The static value on the left shows the value of the property as defined while drawing in the editor. It is also treated as a default value. The expression on the right defines the actual value during simulation. This is the place where you can link the appearance of a shape to any data of the active object. The data may change and it will be reflected in the picture. In case the expression is empty, the property retains the default static value throughout the whole simulation.

The actual values are only evaluated when the shape is visible.

An example of associating graphical properties of animation shapes with active object data is shown in Figure 143. Here the coordinates of the circle are dynamically defined by the variables `x` and `y` of the active object, and the rotation angle of the rectangle is defined by the object member variable `alpha`.

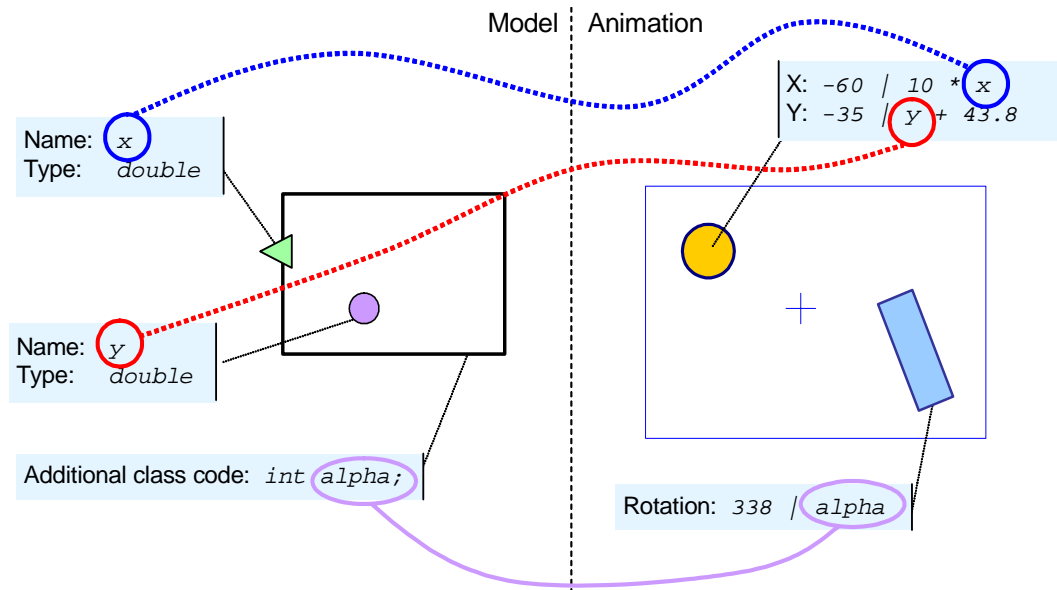


Figure 143. Associating graphical properties with model data

The blue cross marks the center (0, 0) point and indicates the axis direction. The meaning of the blue frame is explained in details in section 12.2.1.1, “Animation origin, axis and frame”.

To define the scale of mapping of the model values to the graphical coordinates, you should use the properties *Scale*, *X Offset*, and *Y Offset* of the animation diagram.

You can add code to the properties *Setup code* and *Update code* of the animation diagram to define more complex relationships between the animation and the model than fixed setting of the animation properties to some expression in the model. Animation object names are used to address the individual objects from code.

Example

This example shows how a simple mechanical model – a pendulum – can be animated.

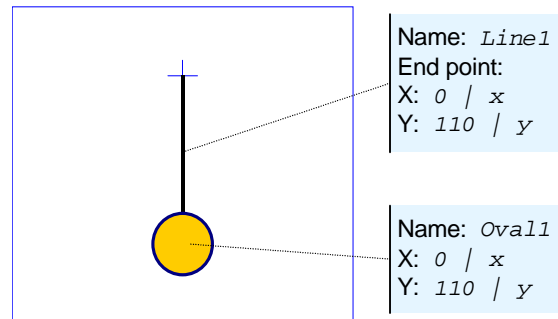


Figure 144. Pendulum animation

12.1.2 Animating hierarchical models

When you create an encapsulated object, its animation automatically appears on the animation of a container object. This is called the encapsulated animation shape and is drawn as a rectangle showing the content of the animation of the encapsulated object. You can move, scale, and rotate an encapsulated animation shape in the animation editor, or you can assign expressions to necessary properties to allow a model to move, scale, and rotate an encapsulated animation shape at runtime.

Motion of a shape in an encapsulated animation is a composition of its motion defined in an encapsulated object and the motion of an encapsulated animation shape defined in a container object.

The animation of a container object can, in its turn, be encapsulated somewhere else, and so on to any desired level. This way, you can construct very complex animations in a modular way, developing individual pictures independently one from another.

Position and size of an encapsulated animation on a container animation can be changed dynamically. This can be defined in two ways:

- You can assign expressions to the properties of the animation developed for the class of the encapsulated object, or
- You can assign expressions to properties of the encapsulated animation shape on the animation diagram of the container.

If an encapsulated object itself knows its position with respect to a parent, then you use the first technique. In this case, you have to leave properties of the encapsulated animation shape blank, because properties of an encapsulated animation shape (if they are defined) override properties of an encapsulated animation. If a container knows position of an encapsulated object, then you use the second technique.

In case an encapsulated object is created and destroyed dynamically, its animation appears and disappears synchronously with the object.

In some cases, you have to draw an encapsulated animation shape manually. For example, if you create an encapsulated object, and the encapsulated object class does not have animation defined for it, then the encapsulated animation shape is not created. If you define animation for the encapsulated object class after that, you have to manually create an encapsulated animation shape. To create an encapsulated animation shape, you click the corresponding toolbar button, place a shape on the animation diagram of a container object, and specify the name of an encapsulated object this shape refers to. Once this is done, the encapsulated animation shape shows the content of the animation of the encapsulated object. Another example: if you animate replicated objects, you may also need to draw one or more encapsulated animation shapes manually.

12.1.2.1 Animating replicated objects

If an encapsulated object is replicated, then there are two options to place its animations on the animation of a container:

- To display all elements of the replicated object, you draw an encapsulated animation shape with the encapsulated object name set to the whole “vector” of objects, e.g., `cars`, `server`. In that case, positions of different encapsulated animations are usually specified in the properties of the animation of the encapsulated object class (first technique, see above).
- To display only the selected elements of the replicated object, you draw as many encapsulated animation shapes as necessary and specify the encapsulated object name for each shape in the form of `<encapsulated object name>-<number>`, e.g., `cars-5`, `server-0`, etc. Position of such encapsulated animations is usually specified directly in the properties of the encapsulated animation shapes (second technique, see above).

12.1.2.2 Animating structures not matching the model hierarchy

Although in the majority of cases the animation structure naturally reproduces the model structure, sometimes they do not match. This means that sometimes you compose an animation of shapes corresponding to active objects lying in different levels of model hierarchy. There are two ways to do so:

- If those active objects have their own animations defined, and you wish to display them at some animation several objects up the model hierarchy, then you define animations for all intermediate active objects, which just contain necessary encapsulated animations and no other shapes.
- In case you just wish to define animation for one (e.g. root) active object, you can draw all shapes on its animation diagram, and then associate them with the data of encapsulated active objects. Obviously, you should make sure those data are accessible.

12.1.3 Interactive control of animation

In AnyLogic 2D animation can be made interactive by adding various types of controls to the animation diagram: buttons, edit boxes, check boxes, radio buttons, sliders, etc.

12.2 Animation diagram

Each active object class may have an animation diagram associated with it. An animation diagram is a collection of shapes. An animation diagram links shape properties to active object data and encapsulated objects, and also defines where the animation of this active object appears in the animation of a container, if the latter exists.

12.2.1 Animation editor

An animation diagram is edited in the animation editor using the animation toolbar, see Figure 145.

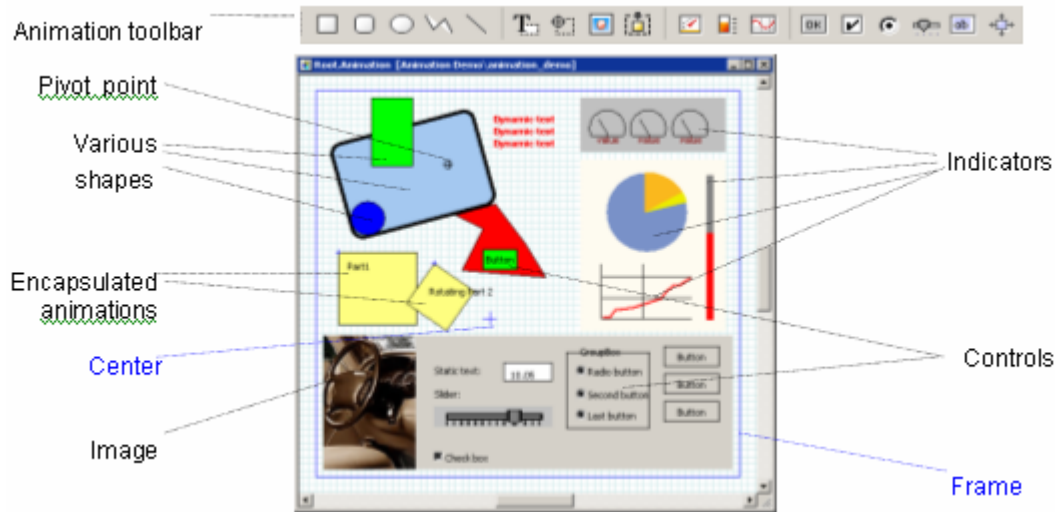



Figure 145. Animation editor and toolbar

► To add an animation to an active object class

1. Click the *New Animation*  toolbar button, or Choose *Insert | New Animation...* from the main menu. The *New Animation* dialog box is displayed. Choose the active object class, which will contain the animation from the *Choose active object* drop-down list.
2. Alternatively, in the Project window, right-click the active object class, which will contain the animation, and choose *New Animation...* from the popup menu. The *New Animation* dialog box is displayed.
3. Enter the name of the new animation in the *Name of the new animation* edit box.
4. If needed, select the *Add encapsulated animations* check box to add animations of the encapsulated objects to this animation. Moreover, you can add links between encapsulated animations by setting the *Add links between encapsulated animations* check box.
5. Click *OK*.

► To open the existing animation diagram of an active object class

1. Double-click the animation in the Project window, or
Right-click the animation in the Project window and choose *Open Animation* from the popup menu.

Animation editor shares a set of generic editing operations described in section 1.5.2, “Diagram editors. Generic operations”.

An animation diagram is always associated with an active object class and has the following properties:

Properties

Name – animation name.

X – [optional] dynamic expression of the x-coordinate of the animation on the container animation (pixels).

Y – [optional] dynamic expression of the y-coordinate of the animation on the container animation (pixels).

Rotation – [optional] dynamic expression of the rotation of the animation on the container animation (radians).

Scale – [optional] if defined, the positions and sizes of all shapes on this diagram are multiplied by this factor.

X Offset – [optional] if defined, the x-coordinates of all shapes on this diagram are increased by this value.

Y Offset – [optional] if defined, the y-coordinates of all shapes on this diagram are increased by this value.

Exclude from build – if set, the animation is excluded from the model.

Prevent frame selection – if set, the animation frame becomes non-selectable. This may be useful if your animation contains, e.g., a rectangle covering the entire animation and you want to select such a rectangle. The point is that in this case, the rectangle and the animation frame coincide. So, if this property is not set, you cannot know for sure what you select: the rectangle or the animation frame.

Flip Y-axis – changes the Y-axis direction.

Properties specifying the location and rotation of the animation on the container animation (*X*, *Y*, *Rotation*) are suppressed by the dynamic expressions of the corresponding properties of the encapsulated animation shape in the container animation, if the latter is defined.

12.2.1.1 Animation origin, axis and frame

The blue cross is the origin point (0, 0) of the animation diagram. Origin point also indicates the axis direction. AnyLogic enables you to change the Y-axis direction since some users got used to working with a frame with Y-axis directed down, while others, with the up-directed Y-axis frame.

► To set up/down Y-axis direction

1. Click the animation item in the Project window.
2. In the Properties window, select/clear the *Flip Y-axis* check box.

The blue rectangle, which you cannot delete, is the animation frame.

The animation frame has two meanings. In case this is a root animation, it denotes the window area. Otherwise, in case this animation is encapsulated in another animation, it is used for scaling the encapsulated animation: the scale is evaluated as the ratio of sizes of the animation frame and the encapsulated animation shape in the container animation. Note that no clipping occurs when animations are encapsulated.

The animation frame has properties generic for all animation shapes (see section 12.2.1.2, “Generic properties of animation shapes”).

You can specify a background image for the animation.

► To set a background image for an animation

1. Click the animation frame in the animation window.
2. Click the *Frame* tab of the Properties window.

3. Click the *Browse* button.
The *Open* dialog box is displayed.
4. Browse for the image file you want to use.
Double-click the file or click the *Open* button to select the file.
5. Select *Stretch* | *Tile* | *Center* option to choose the manner the background image is displayed.

12.2.1.2 Generic properties of animation shapes

All shapes of an animation diagram have the following common properties that are displayed on the *General* page of the Properties window:

Properties

Name – name of the shape, which may be used to access it from code.

X – static value | [optional] dynamic expression of the x-coordinate (pixels).

Y – static value | [optional] dynamic expression of the y-coordinate (pixels).

Rotation – static value (degrees) | [optional] dynamic expression of the rotation angle (radians).

Width – static value | [optional] dynamic expression of the width (pixels).

Height – static value | [optional] dynamic expression of the height (pixels).

Fill color – static value | [optional] dynamic expression of the fill color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.

Line color – static value | [optional] dynamic expression of the line color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.

Line width – static value | [optional] dynamic expression of the line width (pixels).

Visible – [optional] dynamic boolean expression determining if the shape is visible.

Replication – replication factor of the shape.

Lock aspect ratio – if set, the aspect ratio is locked for this shape.

Show name – if set, the name of the shape is shown on the animation diagram editor (but not in the animation).

Exclude from build – if set, the shape is excluded from the animation.

If a generic property is not applicable to a particular shape, it is disabled.


Names of animation shapes are used only for code generation and, correspondingly, to access shapes from code. Names do not appear in the animation window.

12.2.2 Animation shapes

In this section the detailed description of the shapes that can be drawn on animation diagram is given.

12.2.2.1 Rectangle


► To draw a rectangle

1. Click the *Rectangle*  toolbar button, or
Choose *Draw | Animation | Rectangle* from the main menu.
2. Click or drag the rectangle on the diagram.

The rectangle has no specific properties.

12.2.2.2 Rounded rectangle

► To draw a rounded rectangle


1. Click the *Rounded Rectangle*  toolbar button, or
Choose *Draw | Animation | Rounded Rectangle* from the main menu.
2. Click or drag the rounded rectangle on the diagram.

Properties

Radius – static value | [optional] dynamic expression of the corner radius of the rectangle (pixels).

12.2.2.3 Line

► To draw a line

1. Click the *Line*  toolbar button, or
Choose *Draw* | *Animation* | *Line* from the main menu.
2. Drag the line on the diagram.

Properties

The following properties are set individually for line's *Begin point* and *End point*.

X – static value | [optional] dynamic expression of the x-coordinate of the point (pixels).


Y – static value | [optional] dynamic expression of the y-coordinate of the point (pixels).

Style – point style. If *Arrow* is set, the arrow is drawn.

Size – the arrow size.

12.2.2.4 Polyline

► To draw a polyline

1. Click the *Polyline*  toolbar button, or
Choose *Draw* | *Animation* | *Polyline* from the main menu.
2. Click at each polyline point on the diagram.
3. Double-click to finish.

Properties

points – [optional] dynamic expression of number of points of the polyline.

X[index] – [optional] dynamic expression of the x-coordinate of the polyline's point (pixels).

Y[index] – [optional] dynamic expression of the y-coordinate of the polyline's point (pixels).

Begin | End point style – the style of the begin | end point of the polyline. If *Arrow* is set, the arrow is drawn.

Begin | End point size – the size of the begin | end point's arrow.


Closed polyline – if checked, the closing segment is created.

Each point of the polyline can be controlled during the model execution. You can specify dynamic expression, defining the number of points. The coordinates of the polyline's points can also be defined by dynamic expressions. Use the predefined symbol “index” in *X*, *Y* expressions to refer to the current point index. The index value is zero based, i.e., the first point has index of 0.

► To move a point of a polyline


1. Drag the point.

► To add a salient point to a polyline

1. Select the polyline.
2. Click the *Edit Points*  toolbar button, or Choose *Draw | Edit Points* from the main menu, or Right-click the polyline and choose *Edit Points* from the popup menu. The points of the polyline should turn yellow.
3. Drag a segment of the polyline to create a salient point, or Right-click the segment and choose *Add Point* from the popup menu.


► To remove a salient point from a polyline

1. Select the polyline.

2. Click the *Edit Points*  toolbar button, or
Choose *Draw | Edit Points* from the main menu, or
Right-click the polyline and choose *Edit Points* from the popup menu.
3. Right-click the point and choose *Delete Point* from the popup menu, or
Drag the point to an adjacent point of the polyline.
The dragged point disappears.

12.2.2.5 Oval

► To draw an oval

1. Click the *Oval*  toolbar button, or
Choose *Draw | Animation | Oval* from the main menu.
2. Click or drag the oval on the diagram.


Properties

Radius 1 – static value | [optional] dynamic expression of the first (horizontal) oval radius (pixels).

Radius 2 – static value | [optional] dynamic expression of the second (vertical) oval radius (pixels).

12.2.2.6 Image

► To draw an image

1. Click the *Image*  toolbar button, or
Choose *Draw | Animation | Image* from the main menu.
2. Click or drag a rectangle area on the diagram.

Properties

Image index expression – [optional] dynamic expression defining the index of the image in the list to be displayed (integer, zero-based). If left blank, 0 is assumed.

Images – list of file names containing images. Use *Add* and *Remove* buttons to edit the list. The *Image index expression* property defines which image is currently displayed.

Original size – if checked, the original image size is preserved.

If you intend to move your project file, first embed your images. Otherwise, you will need to update paths to all image files used. Embedded images are stored in the AnyLogic project file. If needed, they can be exported to a graphical file anew.

► **To embed an image**


1. Select the image filename in the *Images* list.
2. Click the *Embed* button.

► **To export an embedded image**

1. Select the image name in the *Images* list.
2. Click the *Export* button.
The *Save As...* dialog box is displayed.
3. Specify the name of the image file.
4. Browse for the folder where you want to store the file.
5. Click the *Save* button.

12.2.2.7 Text

► **To draw a text**

1. Click the *Text*  toolbar button, or
Choose *Draw | Text* from the main menu.
2. Click or drag a rectangle area on the diagram.

► To modify the content of a text box

1. Double-click the text.
2. Edit the content of the text.
3. Click the empty area of the diagram to store the modified text, or Press Esc to finish editing.

To create a multiline text, use properties of the text shape.

Following Java convention, the origin point of the text box is the bottom left corner of the first line.

Properties

Text – [optional] content of the text box.

Color – static value | [optional] dynamic expression of the text color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.


Font – the text font.

Choose – the button opens the *Font* dialog box for changing the font properties.

12.2.2.8 Pivot

Pivot is used to group animation shapes, rotate the group, and shift the coordinate system. By specifying dynamic properties of a pivot (*X*, *Y*, *Rotation*, etc.), you can move a group of shapes and rotate it around the pivot. A pivot itself is not visible.

► To draw a pivot

1. Click the *Pivot*  toolbar button, or Choose *Draw* | *Animation* | *Pivot* from the main menu.
2. Click the place on the diagram where you want to put the pivot.

► To add/remove shapes from the pivot group

1. Right-click the pivot and choose *Add/Remove Shapes* from the popup menu.
2. Click on a shape to add/remove it to/from the pivot group.
3. Click the empty area to finish.

When a shape is added to a pivot group, the pivot becomes the origin of its dynamic coordinates, instead of the animation origin point (0,0). This can be used to shift the coordinate system for a part of an animation.

Properties

Custom shape template – if set, the group of shapes added to pivot is considered as a custom shape. You can create as many such custom shapes as you like at runtime.

Setup code – [optional] Java statements to be inserted at the end of the method `setup()` of the pivot class. The code is called during the set up phase of the pivot group. The set up phase is executed only once when the animation creates.

Update code – [optional] Java statements to be inserted at the end of the method `update()` of the pivot class. The code is called each time the pivot group is about to be redrawn.

Additional class code – [optional] Java code to be inserted into the pivot class declaration.


If a pivot point is used as a custom shape template, the shape is not created automatically, but you can create as many custom shapes as you need at runtime. For each custom shape, AnyLogic generates a class derived from the pivot class `Group`. The class has the shape name – e.g., for the `Pivot1` shape, the `Pivot1` class is generated.

Note that the name of the pivot point used as a custom shape template should be capitalized.

You can modify properties of a custom shape and its shapes at runtime using the methods of the `Group` class (for more information, please consult AnyLogic Class Reference).

12.2.2.9 Encapsulated animation

► To draw an encapsulated animation

1. Click the *Encapsulated Animation*  toolbar button, or Choose *Draw | Animation | Encapsulated Animation* from the main menu.
2. Click or drag a rectangle area on the diagram.

Properties

Object – name of the encapsulated object this shape refers to. In case the object is replicated, you may specify a particular element, e.g., `cars-5`.

Original size – if checked, the original size of the animation suppresses the size of the shape.


Lock aspect ratio – applicable to encapsulated animations with non-original size, keeps the ratio between height and width constant during resize operation.

12.2.3 Indicators

Indicators visualize dynamically changing numerical values in a model.

12.2.3.1 Arc indicator

► To draw an arc indicator

1. Click the *Arc Indicator*  toolbar button, or Choose *Draw | Animation | Arc Indicator* from the main menu.
2. Click or drag the indicator area on the diagram.

Properties

Value to indicate – dynamic expression whose value is indicated.

Min value – [optional] minimum value of the indicator.

Max value – [optional] maximum value of the indicator.

Value color – static value | [optional] dynamic expression of the value text color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.

Scale color – static value | [optional] dynamic expression of the scale text color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.


Show value – if checked, the current value is textually displayed at the bottom of the indicator.

Show scale – if checked, the scale is displayed.

The color setup of an arc indicator is the following. The inner area is filled with *Fill color*. The scale and the pointer are of *Line color*. The scale text is of *Scale color*, and the value text is of *Value color*.

12.2.3.2 Bar indicator

► To draw a bar indicator

1. Click the *Bar Indicator*  toolbar button, or Choose *Draw | Animation | Bar Indicator* from the main menu.
2. Click or drag the indicator area on the diagram.

Properties

Value to indicate – dynamic expression whose value is indicated.

Vertical | Horizontal – orientation of the bar indicator.

Min value – [optional] minimum value of the indicator.

Max value – [optional] maximum value of the indicator.

Value color – static value | [optional] dynamic expression of the value color (java.awt.Color). If the checkbox is not checked, the static color is

transparent. If the expression evaluates to null, the dynamic color is transparent.

Scale color – static value | [optional] dynamic expression of the scale text color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.

Show value – if checked, the current value is textually displayed at the bottom of the indicator.


Show scale – if checked, the scale is displayed.

The color setup of a bar indicator is the following. The background bar area is filled with *Fill color*. The scale and the frame are of *Line color*. The scale text is of *Scale color*, and the value text and the value bar are of *Value color*.

12.2.3.3 Chart indicator

The chart indicator displays a dataset or a variable in one of the following forms: scatter, Gantt, pie chart, or bar chart.

► To draw a chart indicator

1. Click the *Chart Indicator*  toolbar button, or Choose *Draw | Animation | Chart Indicator* from the main menu.
2. Click or drag the indicator area on the diagram.

Properties

Value to indicate – dynamic expression whose value is indicated.

Type – the type of the chart: *Scatter*, *Gantt*, *Pie Chart*, or *Bar Chart*.

Window size – displayed time window or number of samples (for scatters and Gantt charts).

Min value – [optional] minimum value of the indicator.

Max value – [optional] maximum value of the indicator.

Min color – minimum color (for Gantt charts)

Max color – maximal color (for Gantt charts)

Value color – static value | [optional] dynamic expression of the scatter line color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.

Scale color – static value | [optional] dynamic expression of the scale text color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.

Show value – if checked, the current value is textually displayed at the bottom of the indicator.

Show scale – if checked, the scale is displayed.


The color setup of a chart indicator is the following. The background area is filled with *Fill color*. The scale and the frame are of *Line color*. The scale text is of *Scale color*, and the scatter line is of *Value color*.

12.2.4 Controls

AnyLogic offers a set of controls (buttons, checkboxes, edit boxes, etc.) for creating interactive animations.

12.2.4.1 Button

► To draw a button

1. Click the *Button*  toolbar button, or Choose *Draw | Animation | Button* from the main menu.
2. Click or drag the button rectangle on the diagram.

Properties

Label – [optional] button text displayed on the screen.


Variable name – [optional] name of the variable in a model. The variable must be of boolean type. It is `true` if the button is being pressed by the user, and `false` otherwise.

Enable expression – [optional] boolean expression determining whether the button is enabled or disabled.

Event handling code – [optional] code to be executed when the button is clicked.

12.2.4.2 Check box

► To draw a check box

1. Click the *Check Box*  toolbar button, or Choose *Draw | Animation | Check Box* from the main menu.
2. Click or drag the check box area on the diagram.

Properties

Label – [optional] check box text displayed on the screen.


Variable name – [optional] name of the variable in a model. The variable must be of boolean type. It is `true` if the check box is checked and `false` otherwise.

Enable expression – [optional] boolean expression determining whether the check box is enabled or disabled.

Event handling code – [optional] code to be executed when the check box is clicked.

12.2.4.3 Radio buttons

► To draw a group of radio buttons

1. Click the *Radio Buttons*  toolbar button, or Choose *Draw | Animation | Radio Buttons* from the main menu.
2. Click or drag the shape area on the diagram.

Properties

Label – [optional] text of the group box containing the radio buttons.

Variable name – [optional] name of the variable in a model. The variable must be of type `double`. When you choose a radio button, the variable is set to the value associated with that radio button.

Enable expression – [optional] boolean expression determining whether the group box is enabled or disabled.


Orientation – either *Vertical* or *Horizontal*. Determines how the radio buttons are arranged.

Button-Value – list of radio buttons and `double` values associated with them.

Event handling code – [optional] code to be executed when any of the radio buttons is clicked.

12.2.4.4 Slider

► To draw a slider

1. Click the *Slider*  toolbar button, or Choose *Draw | Animation | Slider* from the main menu.
2. Click or drag the slider area on the diagram.

Properties

Label – [optional] slider label displayed on the screen.

Variable name – [optional] name of the variable in a model. The variable must be of type `double`. It reflects the position of the slider set by the user. The value is a linear interpolation between *Min value* and *Max value*.

Enable expression – [optional] boolean expression determining whether the slider is enabled or disabled.

Orientation – orientation of the slider, either *Vertical* or *Horizontal*.


Min value – [optional] minimum value of the slider.

Max value – [optional] maximum value of the slider.

Event handling code – [optional] code to be executed when the user changes the slider position.

12.2.4.5 Edit box

► To draw an edit box

1. Click the *Edit Box*  toolbar button, or Choose *Draw | Animation | Edit Box* from the main menu.
2. Click or drag the edit box on the diagram.

Properties

Label – [optional] edit box label displayed on the screen.

Variable name – [optional] name of the variable in a model. The variable must be of type `java.lang.String`. It reflects the content of the edit box entered by the user.


Enable expression – [optional] boolean expression determining whether the edit box is enabled or disabled.

Event handling code – [optional] code to be executed when the user changes text in the edit box.

12.2.4.6 Handle

Handle is a moveable rectangle. The user may move a handle in an animation window, and the variables specified in the properties *X* and *Y* of the handle change correspondingly. You cannot specify expressions in the properties *X* and *Y*. You must specify single variables.

► To draw a handle

1. Click the *Handle*  toolbar button, or Choose *Draw | Animation | Handle* from the main menu.
2. Click the place on the diagram where you want to put the handle.

Properties

Event handling code – [optional] code to be executed when the user moves the handle.

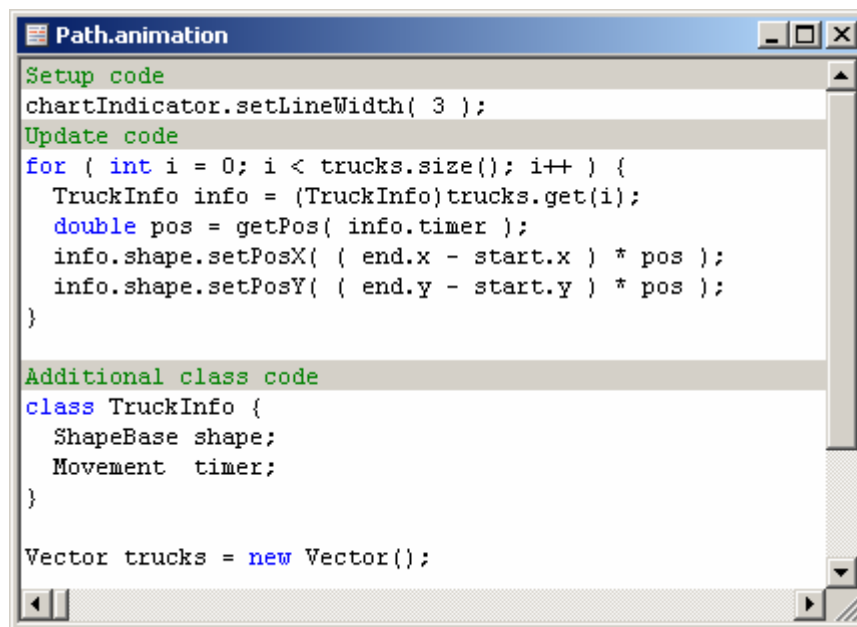
12.2.5 Writing code for an animation

You can write code for the animation object in the Code window of the animation.

► To open the Code window of an animation

1. In the Project window, right-click the *Code* item in the animation subtree of workspace tree and choose *Open Code* from the popup menu, or Double-click the *Code* item in the animation subtree.

The Code window of the animation is displayed (see Figure 146)



```

Path.animation
Setup code
chartIndicator.setLineWidth( 3 );
Update code
for ( int i = 0; i < trucks.size(); i++ ) {
    TruckInfo info = (TruckInfo)trucks.get(i);
    double pos = getPos( info.timer );
    info.shape.setPosX( ( end.x - start.x ) * pos );
    info.shape.setPosY( ( end.y - start.y ) * pos );
}
Additional class code
class TruckInfo {
    ShapeBase shape;
    Movement timer;
}
Vector trucks = new Vector();
  
```

Figure 146. Code window of an animation

The Code window has the following sections, where you can specify your own Java code to be executed on different occurrences:

Setup code – the sequence of Java statements to be executed on the animation setup.

This code is inserted at the end of the method `update()` of the animation.

Update code – the sequence of Java statements to be executed on each animation update performed. This code is inserted at the end of the method `setup()` of the animation.

Additional class code – arbitrary constants, variables and methods can be defined here.

This code is inserted into the animation class declaration.

12.3 3D animation diagram

Each active object class may have a 3D animation diagram associated with it. A 3D animation diagram is a collection of 3D shapes. A 3D animation diagram links shape properties to the active object data and encapsulated objects, and also defines where the 3D animation of this active object appears in the 3D animation of a container, if the latter exists.

12.3.1 3D animation editor

A 3D animation diagram is edited in the 3D animation editor using the 3D animation toolbar, see Figure 147.

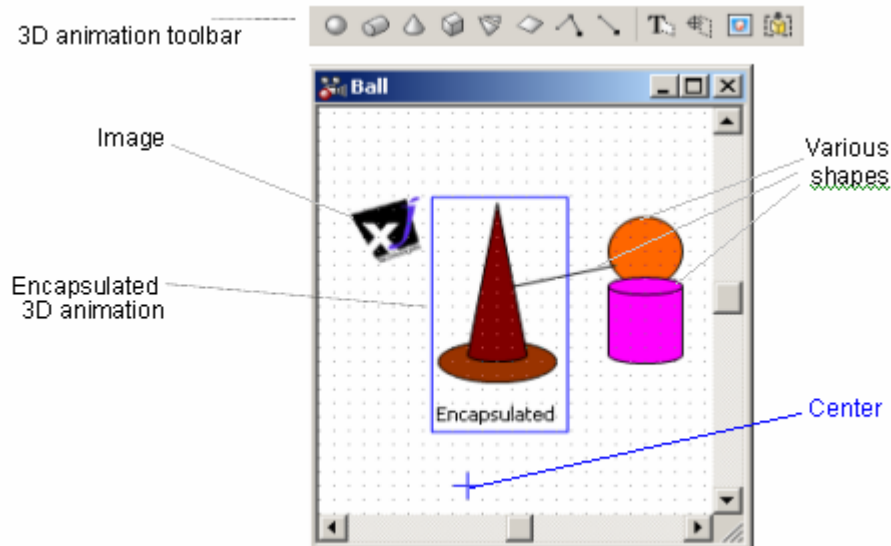



Figure 147. 3D animation editor and toolbar

► To add a 3D animation to an active object class

1. Click the *New 3D Animation*  toolbar button, or Choose *Insert | New 3D Animation...* from the main menu. The *New 3D Animation* dialog box is displayed. Choose the active object class, which will contain the 3D animation, from the *Choose active object* drop-down list.
2. Alternatively, in the Project window, right-click the active object class, which will contain the 3D animation, and choose *New 3D Animation...* from the popup menu. The *New 3D Animation* dialog box is displayed.
3. Enter the name of the new 3D animation in the *Name of new 3D animation* edit box.
4. If needed, select the *Add encapsulated 3D animations* check box to add 3D animations of the encapsulated objects to this 3D animation. Moreover, you can add links between encapsulated 3D animations by setting the *Add links between encapsulated 3D animations* check box.
5. Click *OK*.

► **To open the existing 3D animation diagram of an active object class**

1. In the Project window, double-click the 3D animation item, or
Right-click the 3D animation item and choose *Open Animation* from the popup menu.

3D animation editor shares a set of generic editing operations described in section 1.5.2, “Diagram editors. Generic operations”.

A 3D animation diagram is always associated with an active object class and has the following properties:

General properties

Name – 3D animation name.

X – [optional] dynamic expression of the x-coordinate of the 3D animation on the container 3D animation.

Y – [optional] dynamic expression of the y-coordinate of the 3D animation on the container 3D animation.

Z – [optional] dynamic expression of the z-coordinate of the 3D animation on the container 3D animation.

X rotation – [optional] dynamic expression of the counter clockwise rotation of the 3D animation on the container 3D animation about the x-axis (radians).

Y rotation – [optional] dynamic expression of the counter clockwise rotation of the 3D animation on the container 3D animation about the y-axis (radians).

Z rotation – [optional] dynamic expression of the counter clockwise rotation of the 3D animation on the container 3D animation about the z-axis (radians).

Scale – [optional] if defined, the positions and sizes of all shapes on this diagram are multiplied by this factor.

X Offset – [optional] if defined, the x-coordinates of all shapes on this diagram are increased by this value.

Y Offset – [optional] if defined, the y-coordinates of all shapes on this diagram are increased by this value.

Z Offset – [optional] if defined, the z-coordinates of all shapes on this diagram are increased by this value.

Background – static value | [optional] dynamic expression of the background color.

Exclude from build – if set, the animation is excluded from the model.

Properties specifying the location and rotation of the 3D animation on the container 3D animation (*X*, *Y*, *Z*, *X rotation*, *Y rotation*, *Z rotation*) are suppressed by the dynamic expressions of the corresponding properties of the encapsulated 3D animation shape in the container 3D animation, if the latter is defined.

On the *Miscellaneous* page of 3D animation's properties window you specify animation window size, and light sources properties. AnyLogic 3D animation is lightened with one ambient and two directional lights. Ambient light is constant low level light. Because ambient lighting is uniform, it produces uniform shade. Directional light sources make animation more interesting. Directional light is an oriented light with an origin at infinity. A directional light has parallel light rays that travel in one direction along the specified vector. The portion of a scene where visual objects are illuminated by a particular light source is called that light object's region of influence. The influencing bounds of a light determine, which objects to light. When a light source's influencing bounds intersect the bounds of a visual object, the light is used in shading the entire object.

Miscellaneous properties

Window size – the width and the height of 3D animation window.

Ambient – the color of the ambient light.

Direct 1 – the color of the first directional light.

X, *Y*, *Z* – static values | [optional] dynamic expressions of the x, y, z-coordinates of the vector, in which the first directional light shines.

Direct 2 – the color of the second directional light.

X, *Y*, *Z* – static values | [optional] dynamic expressions of the x, y, z-coordinates of the vector, in which the second directional light shines.

Bounds – light sources influencing bounds.

The blue cross is the origin point (0, 0) of 3D animation diagram.

12.3.1.1 Generic properties of 3D animation shapes

All shapes of a 3D animation diagram share a set of common properties described below. General properties are accessible from the *General* page of the animation shape's properties window.

General properties

Name – name of the shape, which may be used to access it from code.

X – static value | [optional] dynamic expression of the x-coordinate of the shape.

Y – static value | [optional] dynamic expression of the y-coordinate of the shape.

Z – static value | [optional] dynamic expression of the z-coordinate of the shape.

X rotation – static value (degrees) | [optional] dynamic expression of the counter clockwise rotation of the shape about the x-axis (radians).

Y rotation – static value (degrees) | [optional] dynamic expression of the counter clockwise rotation of the shape about the y-axis (radians).

Z rotation – static value (degrees) | [optional] dynamic expression of the counter clockwise rotation of the shape about the z-axis (radians).

Width – static value | [optional] dynamic expression of the width of the shape.

Height – static value | [optional] dynamic expression of the height of the shape.

Depth – static value | [optional] dynamic expression of the depth of the shape.

Color – static value | [optional] dynamic expression of the color (`java.awt.Color`). If the checkbox is not checked, the static color is transparent. If the expression evaluates to `null`, the dynamic color is transparent.

Visible – [optional] dynamic boolean expression determining if the shape is visible.

Replication – replication factor of the shape.

Lock aspect ratio – if set, the aspect ratio is locked for this shape.

Show name – if set, the name of the shape is shown on the animation diagram editor (but not in the animation).

Exclude from build – if set, the shape is excluded from the animation.

Appearance properties are accessible from the *Appearance* page of the animation shape's properties window

Appearance properties

Enable lighting – if set, the shape is lightened and thus visible.

Shininess – static value | [optional] dynamic expression of the shininess of the lightened shape, specifies how shiny a material surface is. This value (in the range 1.0 to 128.0) is used in calculating the specular reflection of a light from a visual object. The higher the value, the more concentrated the specular reflection is.

Transparency – static value | [optional] dynamic expression of the shape's opacity (where 0.0 denotes fully opaque and 1.0 denotes fully transparent shape).

Polygon mode – *Fill* | *Point* | *Line* the style of how the polygons making up the figure are rendered. If *Fill* is set, polygons are filled, if *Point* – they are rendered as the points only, if *Line* – with lines only.

Render face – *Front* | *Back* | *Both* type of rendering the faces of 3D animation shapes. For many visual objects, only one face of the polygons need to be rendered. To reduce the computational power required to render the polygonal surfaces, the renderer can cull the unneeded faces.

If *Front* is set, front facing polygons are rendered only.

If *Back* is set, back facing polygons are rendered only.

If *Both* is set, all polygons are rendered, no matter which direction they are facing.

If a generic property is not applicable to a particular shape, it is disabled.


Names of animation shapes are used only for code generation and, correspondingly, to access shapes from code. Names do not appear in the animation window.

12.3.2 3D animation shapes

In this section, the detailed description of the 3D animation shapes is given.

12.3.2.1 Ellipsoid

► To draw an ellipsoid

1. Click the *Ellipsoid*  toolbar button, or
Choose *Draw* | *3D Animation* | *Ellipsoid* from the main menu.
2. Click or drag the ellipsoid on the diagram.

Properties

Radius X – static value | [optional] dynamic expression of the ellipsoid radius along its X-axis.


Radius Y – static value | [optional] dynamic expression of the ellipsoid radius along its Y-axis.

Radius Z – static value | [optional] dynamic expression of the ellipsoid radius along its Z-axis.

Slices – the number of the circle elements the ellipsoid surface is formed from.

12.3.2.2 Cylinder

► To draw a cylinder

1. Click the *Cylinder*  toolbar button, or
Choose *Draw* | *3D Animation* | *Cylinder* from the main menu.
2. Click or drag the cylinder on the diagram.

Properties


Radius – static value | [optional] dynamic expression of the radius of the cylinder.

Length – static value | [optional] dynamic expression of the cylinder length.

Slices – the number of the circle elements the cylinder surface is formed from.

12.3.2.3 Cone

► To draw a cone

1. Click the *Cone*  toolbar button, or
Choose *Draw | 3D Animation | Cone* from the main menu.
2. Click or drag the cone on the diagram.

Properties


Radius – static value | [optional] dynamic expression of the radius of the cone.

Length – static value | [optional] dynamic expression of the cone length.

Slices – the number of the circle elements the cone surface is formed from.

12.3.2.4 Parallelepiped


► To draw a parallelepiped

1. Click the *Parallelepiped*  toolbar button, or
Choose *Draw | 3D Animation | Parallelepiped* from the main menu.
2. Click or drag the parallelepiped on the diagram.

Parallelepiped has no specific properties.

12.3.2.5 Rectangle


► To draw a rectangle

1. Click the *Rectangle*  toolbar button, or
Choose *Draw | 3D Animation | Rectangle* from the main menu.
2. Click or drag the rectangle on the diagram.

Rectangle has no specific properties.

12.3.2.6 Line

► To draw a line

1. Click the *Line*  toolbar button, or
Choose *Draw | 3D Animation | Line* from the main menu.
2. Drag the line on the diagram.

Properties

The following properties are set individually for the polyline's *Begin point* and *End point*.

Begin | End point X – static value | [optional] dynamic expression of the x-coordinate of the line's begin | end point (pixels).


Begin | End point Y – static value | [optional] dynamic expression of the y-coordinate of the line's begin | end point (pixels).

Begin | End point Z – static value | [optional] dynamic expression of the z-coordinate of the line's begin | end point (pixels).

Line width – static value | [optional] dynamic expression of the line width.

12.3.2.7 Polyline

► To draw a polyline

1. Click the *Polyline*  toolbar button, or
Choose *Draw | 3D Animation | Polyline* from the main menu.
2. Click at each polyline point on the diagram.
3. Double-click to finish.

The generic 2D polyline editing operations (see section 12.2.2.4, “Polyline”) can be applied to 3D polyline.

Properties

points – [read only] static value | [optional] dynamic expression of number of points of the polyline.

X[index] – [optional] dynamic expression of the x-coordinate of the polyline’s point.

Y[index] – [optional] dynamic expression of the y-coordinate of the polyline’s point.

Z[index] – [optional] dynamic expression of the z-coordinate of the polyline’s point.

Line width – static value | [optional] dynamic expression of the polyline width.

Closed polyline – if checked, the closing segment is created.


Points – coordinates of the polyline points, specified in the form #, X, Y, Z, where # is the point’s index, X, Y, Z – x-, y- and z-coordinates of the point correspondingly.

Each point of the polyline can be controlled during the model execution. You can specify dynamic expression, defining the number of points. The coordinates of the polyline’s points can also be defined by dynamic expressions. Use the predefined symbol “index” in X, Y, Z expressions to refer to the current point index. The index value is zero based – i.e. the first point has index of 0.

12.3.2.8 Mesh

You can use mesh to define custom geometry shapes. The mesh is defined as follows: you define the vertices of the mesh and then choose the mesh type – the way the array of vertices is drawn (individual groups of vertices can form lines, triangles, quadrilaterals, etc.).

► To draw a mesh

1. Click the *Mesh*  toolbar button, or
Choose *Draw* | *3D Animation* | *Mesh* from the main menu.
2. Click at each mesh vertex on the diagram.
3. Double-click to finish.

The generic polyline editing operations (see section 12.2.2.4, “Polyline”) can be applied to mesh.

Properties

Point array – the array of vertices is drawn as individual points.

Line array – the array of vertices is drawn as individual line segments. Each pair of vertices defines a line to be drawn. The number of vertices of a line array mesh must be divisible by 2.

Triangle array – the array of vertices is drawn as individual triangles. Each group of three vertices defines a triangle to be drawn. The number of vertices of a triangle array mesh must be divisible by 3.

Quad array – the array of vertices is drawn as individual quadrilaterals. Each group of four vertices defines a quadrilateral to be drawn. The number of vertices of a quad array mesh must be divisible by 4.

Line strip array – the array of vertices is drawn as a set of connected line strips. An array of per-strip vertex counts specifies where the separate strips appear in the vertex array. For every strip in the set, each vertex, beginning with the second vertex in the array, defines a line segment to be drawn from the previous vertex to the current vertex.

Triangle strip array – the array of vertices is drawn as a set of connected triangle strips. An array of per-strip vertex counts specifies where the separate strips appear in the vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array, defines a triangle to be drawn using the current vertex and the two previous vertices.

Triangle fan array – the array of vertices is drawn as a set of connected triangle fans. An array of per-strip vertex counts specifies where the separate strips (fans) appear in the vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array, defines a triangle to be drawn using the current vertex, the previous vertex and the first vertex. This can be thought of as a collection of convex polygons.

points – [read only] static value | [optional] dynamic expression of number of vertices of the mesh.

X[index] – [optional] dynamic expression of the x-coordinate of the mesh's vertex.

Y[index] – [optional] dynamic expression of the y-coordinate of the mesh's vertex.


Z[index] – [optional] dynamic expression of the z-coordinate of the mesh's vertex.

Points – coordinates of the mesh vertices, specified in the form #, X, Y, Z, where # is the vertex's index, X, Y, Z – x-, y- and z-coordinates of the vertex correspondingly.

Each vertex of the mesh can be controlled during the model execution. You can specify dynamic expression, defining the number of vertices. The coordinates of the mesh's vertices can also be defined by dynamic expressions. Use the predefined symbol “index” in X, Y, Z expressions to refer to the current vertex index. The index value is zero based, i.e., the first vertex has index of 0.

12.3.2.9 Text

► To draw a text

1. Click the *Text*  toolbar button, or Choose *Draw | Text* from the main menu.
2. Click or drag a rectangle area on the diagram.

► To modify the content of a text box

1. Double-click the text.
2. Edit the content of the text.
3. Click the empty area of the diagram to store the modified text, or Press Esc to finish editing.

To create a multiline text use properties of the text shape.

Following Java convention, the origin point of the text box is the bottom left corner of the first line.

Properties

Text – static value | [optional] dynamic expression of the content of the text box.

Color – static value | [optional] dynamic expression of the text color (java.awt.Color). If the checkbox is not checked, the static color is transparent. If the expression evaluates to null, the dynamic color is transparent.


Font – the font of the text.

Choose – the button opens the *Font* dialog box for changing the font properties.

12.3.2.10 Pivot

Pivot is used to group animation shapes, rotate the group, and shift the coordinate system. By specifying dynamic properties of a pivot (*X*, *Y*, *Rotation*, etc.), you can move a group of shapes and rotate it around the pivot. A pivot itself is not visible in an animation.

► To draw a pivot

1. Click the *Pivot*  toolbar button, or Choose *Draw | 3D Animation | Pivot* from the main menu.
2. Click the place on the diagram where you want to put the pivot.

► To add/remove shapes from the pivot group

1. Right-click the pivot and choose *Add/Remove Shapes* from the popup menu.
2. Click on a shape to add/remove it to/from the pivot group.
3. Click the empty area to finish.

When a shape is added to a pivot group, the pivot becomes the origin of its dynamic coordinates, instead of the animation origin point (0, 0). This can be used to shift the coordinate system for a part of an animation.

Properties

Custom shape template – if set, the group of shapes added to pivot is considered as a custom shape. You can create as many such custom shapes as you like at runtime.

Setup code – [optional] Java statements to be inserted at the end of the method `setup()` of the pivot class. The code is called during the set up phase of the pivot group. The set up phase is executed only once when the animation creates.

Update code – [optional] Java statements to be inserted at the end of the method `update()` of the pivot class. The code is called each time the pivot group is about to be redrawn.

Additional class code – [optional] Java code to be inserted into the pivot class declaration.


If a pivot point is used as a custom shape template, the shape is not created automatically, but you can create as many custom shapes as you need at runtime. For each custom shape, AnyLogic generates a class derived from the pivot class `Group3D`. The class has the shape name – e.g., for the `Pivot1` shape, the `Pivot1` class is generated.

Note that the name of the pivot point used as a custom shape template should be capitalized.

You can modify properties of a custom shape and its shapes at runtime using the methods of the `Group3D` class (for more information, please consult AnyLogic Class Reference).

12.3.2.11 Image

► To draw an image

1. Click the *Image*  toolbar button, or Choose *Draw* | *Animation* | *Image* from the main menu.
2. Click or drag a rectangle area on the diagram.

Properties

Image index expression – [optional] dynamic expression defining the index of the image in the list to be displayed (integer, zero-based). If left blank, 0 is assumed.

Images – list of file names containing images. Use *Add* and *Remove* buttons to edit the list. The *Image index expression* property defines which image is currently displayed.

Original size – if checked, the original image size is preserved.

If you intend to move your project file, first embed your images. Otherwise, you will need to update paths to all image files used. Embedded images are stored in the AnyLogic project file. If needed, they can be exported to a graphical file anew.

► **To embed an image**


1. Select the image filename in the *Images* list.
2. Click the *Embed* button.

► **To export an embedded image**

1. Select the image name in the *Images* list.
2. Click the *Export* button.
The *Save As...* dialog box is displayed.
3. Specify the name of the image file.
4. Browse for the folder where you want to store the file.
5. Click the *Save* button.

12.3.2.12 Encapsulated 3D animation

► **To draw an encapsulated 3D animation**

1. Click the *Encapsulated 3D Animation*  toolbar button, or
Choose *Draw | 3D Animation | Encapsulated Animation* from the main menu.
2. Click or drag a rectangle area on the diagram.

Properties

Object – name of the encapsulated object this shape refers to. In case the object is replicated, you may specify a particular element, e.g., *cars-5*.

Scale – static value | [optional] dynamic expression of the scale factor applied to the original size of the animation.

12.3.3 3D animation rendering principles

To form the rendered image, the animation content is projected onto an image plate. Figure 148 shows the relationship between the image plate, the camera position, and the animation world. The camera position is behind the image plate. The visual objects in front of the image plate are rendered to the image plate. Rendering can be thought of as projecting the visual objects to the image plate. This idea is illustrated with the four projectors in the image (dashed lines).

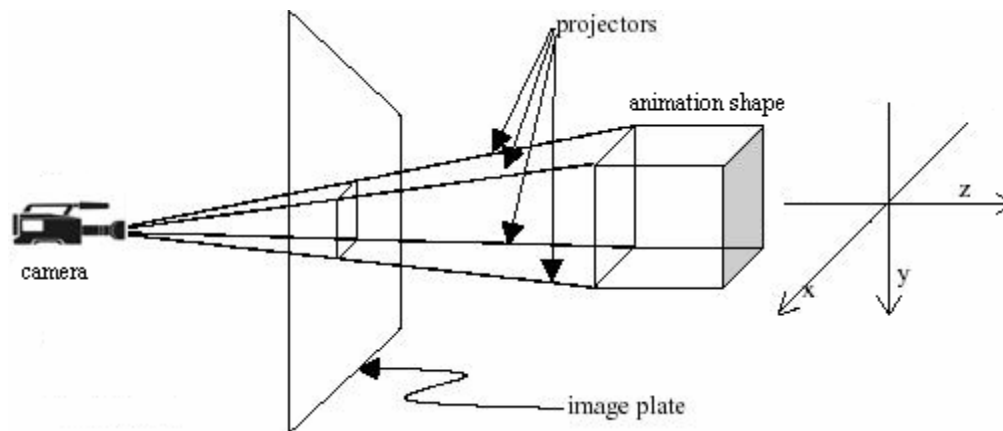


Figure 148. 3D animation rendering principles

The coordinate system is right-handed.

12.3.4 Managing a camera

The AnyLogic 3D animation world is combined from the group of animation shapes. Only the specific part of the world is visible – it is defined by the camera's parameters: field of view, position and orientation.

Camera can be positioned automatically by AnyLogic. In this case, camera is positioned to make the animation appear as in the animation editor. The orientation of axis is the same as

in the animation editor (Figure 148 shows the orientation with respect to the viewer (the x-axis is positive to the right, y-axis is positive down, and z-axis is negative toward the viewer). To fit all animation shapes into the image plate and to make them appear with the same sizes as in animation editor, the camera is moved along the z-axis back to negative values.

However, you may need to position the camera differently; therefore, AnyLogic enables you to specify the camera's parameters on your own.

► **To calculate camera parameters automatically/manually**

1. Click 3D animation in the Project window.
2. Click the *Camera* tab of the Properties window and go to the *Position* section.
3. Select/clear the *Automatically calculate camera parameters* check box.

Note that when you set the camera's parameters on your own, camera orientation differs from the orientation set for automatically managed camera. In this case the default axis orientation is the following: the x-axis is positive to the right, y-axis is positive up, z-axis is positive toward the viewer. The default camera location is the image plate center ((0,0,0) point). Since the camera must be positioned behind the image plate to see animation shapes rendered onto the image plate, you need to move the camera to positive z values.

You can define the clip distances and the camera's position and orientation on the *Camera* page of the 3D animation's properties page.

12.3.4.1 Defining clip distances

The back clip distance specifies the distance from the camera in the direction of gaze to where objects begin disappearing. Objects farther away from the camera than the back clip distance are not drawn. The default value is 10.0 meters.

The front clip distance specifies the distance away from the eyepoint in the direction of gaze where objects stop disappearing. Objects closer to the eye than the front clip distance are not drawn. The default value is 0.1 meters.

There are several considerations that need to be taken into account when choosing values for the front and back clip distances:

- The front clip distance must be greater than 0.0 in physical eye coordinates.
- The front clipping plane must be in front of the back clipping plane; that is, the front clip distance must be less than the back clip distance in physical eye coordinates.
- Try not to assign large values to the front and back clip distances, since they are in physical eye coordinates, not in pixels.
- The ratio of the back distance divided by the front distance, in physical eye coordinates, affects Z-buffer precision. Values of 100 to less than 1000 will produce better results.

Violating any of the above rules will result in undefined behavior. In many cases, no picture will be drawn.

► To define clip distances

1. In the Project window, click 3D animation item in the workspace tree.
2. Click the *Camera* tab of the Properties window and go to the *Position* section.
3. Specify static values or dynamic expressions of front and back clip distances in the *Front* and *Back* edit boxes correspondingly.

12.3.4.2 Defining camera position

► To define camera position

1. In the Project window, click 3D animation item in the workspace tree.
2. Click the *Camera* tab of the Properties window and go to the *Position* section.
3. Specify static values or dynamic expressions of x-, y- and z-coordinate of the camera in the *X*, *Y* and *Z* edit boxes correspondingly.

12.3.4.3 Defining camera orientation

If needed, you can define the orientation of the camera by specifying either camera rotation angles, or the point the camera is looking at.

► **To rotate the camera**

1. In the Project window, click 3D animation item in the workspace tree.
2. Click the *Camera* tab of the Properties window and go to the *Orientation* section.
3. Choose the *Rotation* option.
4. Specify static values (in degrees) or dynamic expressions (in radians) of the counter clockwise rotations of the camera about the x-axis, y-axis and z-axis in the *X*, *Y* and *Z* edit boxes correspondingly.

► **To define a point the camera is looking at**

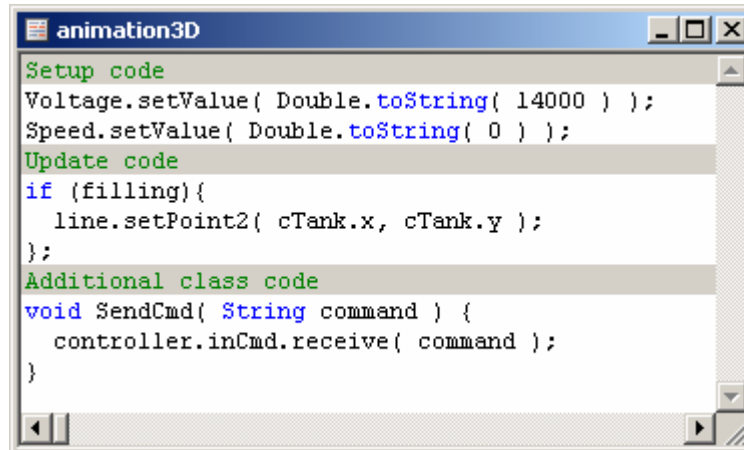
1. In the Project window, click 3D animation item in the workspace tree.
2. Click the *Camera* tab of the Properties window and go to the *Orientation* section.
3. Choose the *Look at* option.
4. Specify static values or dynamic expressions of the x-, y- and z-coordinates of the point the camera is looking at in the *X*, *Y* and *Z* edit boxes correspondingly.
5. Specify the camera's up direction vector in the *Up X*, *Up Y*, *Up Z* edit boxes. The default up direction vector is (0,1,0). Changing it, e.g., to (0,-1,0) flips the rendered image upside-down.

12.3.5 Writing code for 3D animation

You can write code for 3D animation object in the Code window of 3D animation.

► **To open the Code window of 3D animation**

1. In the Project window, right-click the *Code* item in the animation subtree of the workspace tree and choose *Open Code* from the popup menu, or Double-click the *Code* item in the animation subtree.
3D animation's Code window is displayed (see Figure 149)



```

animation3D
Setup code
Voltage.setValue( Double.toString( 14000 ) );
Speed.setValue( Double.toString( 0 ) );
Update code
if (filling){
    line.setPoint2( cTank.x, cTank.y );
};
Additional class code
void SendCmd( String command ) {
    controller.inCmd.receive( command );
}

```

Figure 149. Code window of 3D animation

This window has the following sections, where you can specify your own Java code to be executed on different occurrences:

Setup code – the sequence of Java statements to be executed on the animation setup.

This code is inserted at the end of the method `update()` of the animation.

Update code – the sequence of Java statements to be executed on each animation update performed. This code is inserted at the end of the method `setup()` of the animation.

Additional class code – arbitrary constants, variables and methods can be defined here.

This code is inserted into the animation class declaration.

12.4 Running animation

An AnyLogic animation is run synchronously with the model simulation. Animation is displayed in the standalone animation window (see Figure 150).



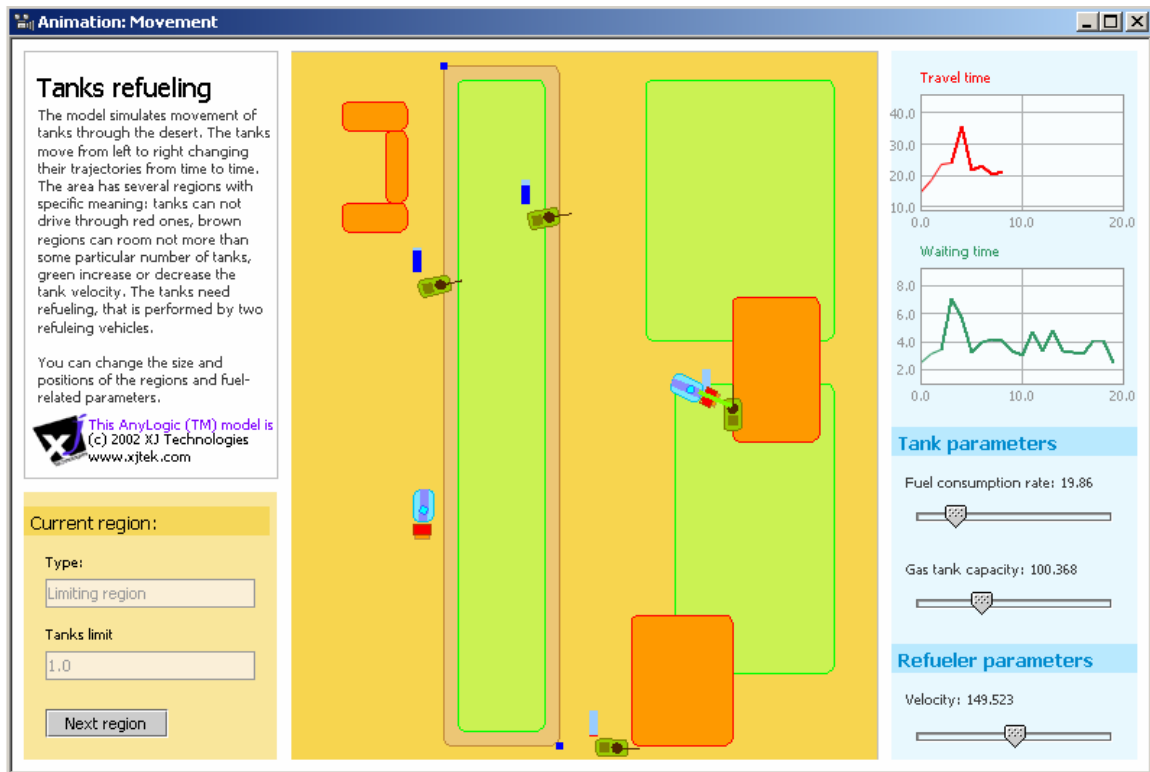



Figure 150. Animation window

The animation window is automatically displayed when the model starts running. Sometimes you may need to close the window to reduce data display overhead. You can open it later when needed.

► To open an animation window

1. Click the *Animation*  toolbar button, or Choose *View|Animation* from the main menu.

12.5 Running 3D animation

AnyLogic 3D animation is run synchronously with the model simulation. 3D animation is displayed in the 3D animation window.

The 3D animation window is automatically displayed when the model starts running. Sometimes you may need to close the window to reduce data display overhead. You can open it later when needed.

► **To open 3D animation window**

1. Choose *View | 3D Animation* from the main menu.

12.5.1 Moving and rotating the animation

You can rotate the animation scene and change the camera position: move the camera in the XY-plane to shift the animation scene or move it toward/off the scene to zoom the scene in/out.

► **To rotate the scene**

1. Click in the 3D animation window and, while holding the left mouse button down, move the mouse in the required rotation direction.

► **To shift the animation scene**

1. Right-click in the 3D animation window and, while holding the right mouse button down, move the mouse in the direction you want to move your camera.

► **To zoom the scene in/out**

1. Click in the 3D animation window and, while holding Alt and the left mouse button down, move the mouse up/down.

12.6 Configuring an animation run

Configure and control AnyLogic animation using the animation settings toolbar (see Figure 151). Namely, you can set up the animation update rate and toggle the anti-aliasing option on/off. These settings apply both to the AnyLogic animation and 3D animation windows.




Figure 151. Animation settings toolbar

12.6.1 Setting up animation update rate

AnyLogic enables you to set up the animation update rate. The greater update rate you specify, the smoother animation will appear. However, animation rendering takes longer, and frequent animation update will slow the model simulation. So, choose between smooth animation and fast simulation and set up the animation update rate according to your needs.

You can explicitly specify fixed update rate in frames per second. Alternatively, you can specify adaptive update rate. Adaptive update rate will be recalculated during the model simulation to fix up the specified ratio between simulation speed and animation smoothness.

► To set up the animation update rate

1. Click the *Animation Settings*  toolbar button, or Choose *Model|Animation Settings...* from the main menu. The *Animation Settings* dialog box is displayed, as shown in Figure 152.
2. To define the fixed update rate, choose the *Fixed* option and explicitly specify the update rate (in frames per second) with a slider.
3. Otherwise, to define adaptive update rate, choose the *Adaptive* option and specify the update rate with a slider. Choose between smooth animation (*Smooth*) and fast simulation (*Fast*).
4. Click *Apply* to apply changes. The animation will appear with the specified update rate.
5. If needed, repeat steps 2-4 to specify another update rate and click *OK* when finished.

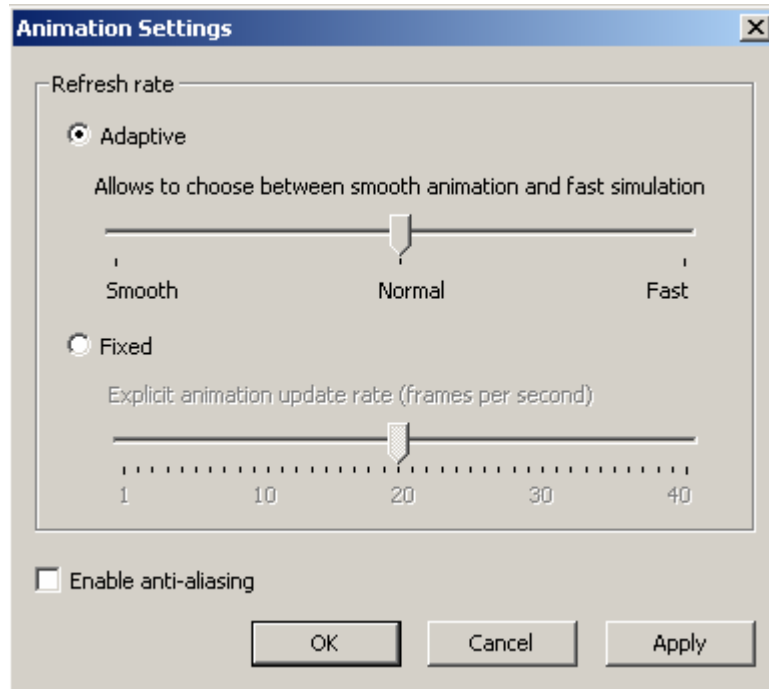



Figure 152. Animation Settings dialog box

12.6.2 Setting up animation anti-aliasing

AnyLogic animation supports anti-aliasing – one of the most important techniques in making graphics more smooth and pleasing to the eye. Anti-aliasing is a process of smoothing the drawing of points or lines that would otherwise appear jagged. However, note that more time is spent on rendering the animation with the anti-aliasing set.

► To enable/disable anti-aliasing

1. Click the *Animation Settings*  toolbar button, or Choose *Model|Animation Settings...* from the main menu. The *Animation Settings* dialog box is displayed, see Figure 152.
2. Select/clear the *Enable anti-aliasing* checkbox.
3. Click *OK*.

13. Simulation settings

AnyLogic enables you to control model simulation.

You can set up model simulation speed, specifying the mapping between the model time units and seconds. This is frequently needed when you need your animation to appear as in real life. See section 13.1, “Simulation speed” for details.

You can define the number of model replications – single model runs the model simulation contains of. See section 13.2, “Model replications” for details.


You may also stop the simulation at some event (at specified model time, or simulation stop condition). See section 13.3, “Simulation stop conditions” for details.

All these settings are defined individually for each AnyLogic experiment. Thus you can simply control your model simulation by creating several experiments with different simulation settings and simulating your model with different current experiments.

13.1 Simulation speed

An AnyLogic model can be run either in real time or virtual time mode. In real time mode, the mapping of AnyLogic model time to the real time is made. It is frequently needed when you have developed some animation and want it to appear as in real life. In virtual time mode, the model runs at its maximum speed and no mapping is made between model time units and a second of astronomical time.

► To set virtual/real time mode

1. Click the *Enable virtual time mode*  toolbar button.
If virtual time mode is set, the button is shown pressed.

► To specify model simulation speed

1. In the Project window, click the experiment, for which you want to specify model simulation speed.

- Specify the model simulation speed in the *Simulation speed* section of the *Additional* page of the Properties window (see Figure 153).

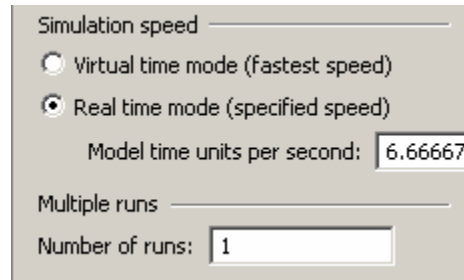


Figure 153. Simulation speed and Multiple runs experiment's properties

In the *Simulation speed* section you can set up the following settings:

Virtual time mode (fastest speed) – if set, the model is run in virtual time mode, the model runs at its maximum speed and no mappings are made between model time unit and seconds of astronomical time.

Real time mode (specified speed) – if set, the model is run in real time mode, i.e., one second takes *Model time units per second* model units.

Model time units per second – for real time mode, specifies how many model time units one second takes.

Note that in the case you specify excessive speed your model cannot keep, a real-time violation occur. You can handle it by overriding the dedicated method in the *Additional class code* of the root active object class (please consult AnyLogic Class Reference for more details):

Related method of ActiveObject

`boolean onRTViolation (double delay)` – the method is called for the root object when a real-time violation has occurred; that is, the difference between the expected and the actual model time exceeds `delay` milliseconds.

13.2 Model replications

Model simulation contains of one or several single model runs – model replications. You can define how many replications to execute. Using several replications in one simulation, you can, for example, vary model parameters to plot a chart of output versus model parameter. Repetitive runs of a model are particularly useful for periodic models and for stochastic models where many runs are required to assess the effect of random factors.

► To define a number of model replications

1. In the Project window, click the experiment to set number of replications for.
2. Type the number of model replications in the *Number of runs* edit box in the *Multiple runs* section of the *General* page of the Properties window (see Figure 153).

13.3 Simulation stop conditions

Sometimes you need simulation stopped at some specific event.

You can set up your model to be stopped:

- At the specified model time.
- When the mean confidence of the specified dataset is less than the threshold.
- When the specified variable steps over the threshold.
- When the specified boolean condition becomes `true`. Expression can include checks of dataset mean confidence, variable values, etc.

If no stop condition is defined, model works until you stop it manually.

Simulation stop conditions for an experiment are defined in the *Model stop condition* section of the *Additional* page of experiment's properties window (Figure 154).

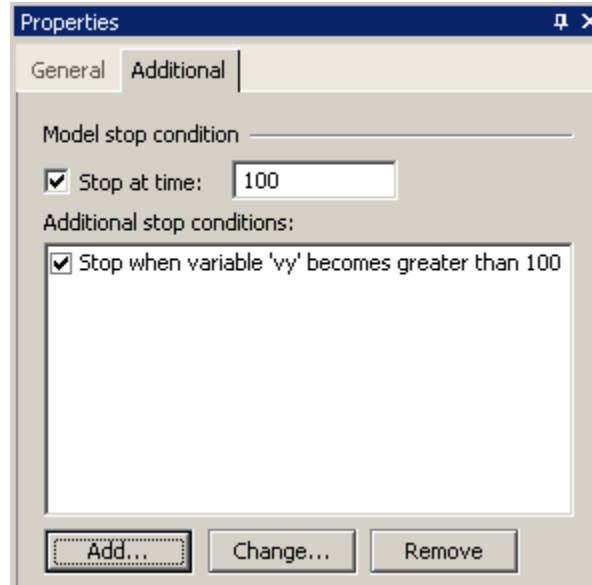


Figure 154. Additional property page of an experiment. Model stop condition section

13.3.1 Defining a model time stop condition

► To define time the model should stop at

1. In the Project window, click the experiment to specify the simulation stop condition for.
2. Select the *Stop at time* check box on the *Additional* page of the Properties window.
3. In the edit box on the right, specify the model stop time in model time units.

13.3.2 Defining additional stop conditions

Other stop conditions (based on checks of a variable value, a dataset mean confidence or a boolean condition) are named *additional stop conditions*.

Additional stop conditions of an experiment are listed in the *Additional stop conditions* list on the experiment's properties window. You define them in the *Additional stop condition* dialog box.

► **To open the Additional stop condition dialog box**

1. In the Project window, click the simulation experiment to define additional stop conditions for.
2. Click the *Add* button on the *Additional* page of the Properties window. The *Additional stop condition* dialog box is displayed, see Figure 155.

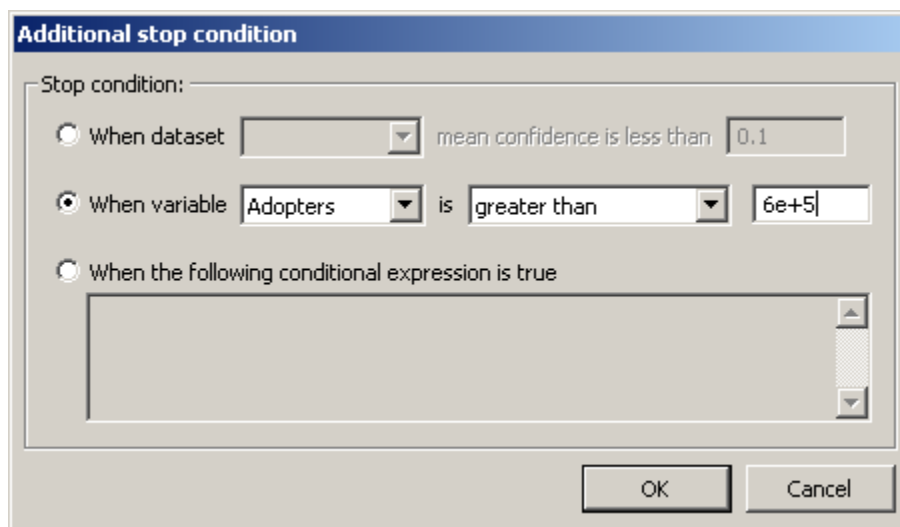


Figure 155. Additional stop condition dialog box

► **To define a statistics confidence level as a stop condition**

1. Open the *Additional stop condition* dialog box.
2. Choose the *When dataset...* option.
3. Specify the dataset in the combo box on the right.
4. Type the confidence level value in the edit box on the right.

► To define a variable level as a stop condition

1. Open the *Additional stop condition* dialog box.
2. Choose the *When variable...* option.
3. Choose a variable from the drop-down list on the right.
4. Choose the *less than* | *less than or equal to* | *greater than* | *greater than or equal to* comparison operation from the drop-down list on the right.
5. Type the threshold value in the edit box on the right.

► To define a boolean expression as a stop condition

1. Open the *Additional stop condition* dialog box.
2. Choose the *When the following conditional expression is true* option.
3. Specify the boolean expression in the edit box below the option.

You can simply disable the additional stop condition. The disabled stop condition is not applied but remains in the project.

► To enable/disable an additional stop condition

1. Select/clear the check box to the left of the stop condition description in the *Additional stop conditions* list.

You can modify stop conditions and delete them.

► To modify an additional stop condition

1. Select the condition in the *Additional stop conditions* list.
2. Click *Change* button.
3. Change the condition in the *Additional stop condition* dialog box.

► **To remove an additional stop condition**

1. Select the condition in the *Additional stop conditions* list.
2. Click the *Remove* button.

13.4 Controlling model replications

AnyLogic enables you to control model replications either using AnyLogic experiment's properties, or programmatically.

13.4.1 Writing code to be executed between model replications

AnyLogic enables you to specify arbitrary actions to be performed between model replications. You can write any Java code to be executed before and after each model replication on the *Code* page of the project's properties window.

► **To write code to be executed between model replications**

1. In the Project window, click the project item (the top-most item in the workspace tree).
2. On the *Code* page of the Properties window, type code to be executed before each model replication in the *Before replication* section.
3. Type code to be executed after each model replication in the *After replication* section.

13.4.2 API to control replications

You can control simulations and replications programmatically by overriding the method `executionControl()` of the root object of the model. The mechanism of controlling replications is based on the following feature of the root object:

The tree of active objects is constructed before each replication and deleted afterwards. The root object is responsible for creation and destruction of the tree. The root object itself survives between subsequent replications.

This means that the member variables of the root object remain untouched. You can use this property of the root object, for example, to collect datasets across multiple replications or perform parameter iteration.

Please do not confuse the terms simulation and replication. Replication is one execution of a model. Simulation is an execution of one or more replications. The result of a simulation is the value of the observable to be optimized by AnyLogic optimization subsystem. Several simulations take place only if you invoke optimization. Otherwise, only one simulation takes place. By overriding the method `executionControl()`, you control how much replications are executed in one simulation. The root object does not survive between simulations.

By default, a simulation performs one replication and stops. To tell AnyLogic to execute multiple replications, you override the method `executionControl()` of the root object. In this method, you call `Engine.execute()` to perform a replication. You can program any algorithm (e.g. nested loops, an optimization strategy) around replications. To override the method `executionControl()`, you use the *Additional class code* code section of the root object class.

The default implementation of `executionControl()` calls `Engine.execute()` once.

Example

In the following example, one replication calculates the result `result` depending on the parameter `param`. The optimization algorithm defined by the method `executionControl()` finds the parameter value (with the given accuracy `eps`), with which the `result` equals 0.

```
public void executionControl() {
    double a = 0;
    double b = 10;
    double eps = 0.01;

    param = a;
    Engine.execute();
}
```



```
double fa = result;

param = b;
Engine.execute();
double fb = result;

assert(
    fa * fb < 0,
    "The function sign must be different on the interval ends"
);

while ( b - a > eps ) {
    param = ( a + b ) / 2;
    Engine.execute();
    if ( result == 0 )
        break;
    if ( result * fa > 0 ) {
        fa = result;
        a = param;
    }
    else {
        fb = result;
        b = param;
    }
}
}
```

14. Debugging a model

AnyLogic supports model debugging. This chapter provides information on AnyLogic debugging tools.

- AnyLogic supports on-the-fly checking of types, parameters, and diagram syntax. The errors found during code generation and compilation are displayed in AnyLogic Output window and graphically highlighted in error location windows.
- Using AnyLogic Events window, you can view the event queue of AnyLogic simulation engine to view what is happening at the simulation engine at the lowest level details and to make some changes to the event processing.
- You can debug your model by setting a breakpoint on a model element to stop the model execution when this element becomes active, examine the model state, and perform some actions in response.
- AnyLogic supports runtime error ability. You can throw runtime error and terminate model execution as a reaction to different undesirable occurrences.
- You can trace model execution by writing custom information to AnyLogic log windows on different occurrences.
- AnyLogic detects errors in Java code written by the user and logical errors of model execution (simulation errors). If such an error occurs, AnyLogic stops the model and notifies you with error message.
- You can debug Java code using a third-party debugger by running the model within the debugger using command line execution feature or by attaching the debugger to the currently running model.


14.1 Checking model syntax

AnyLogic supports on-the-fly checking of types, parameters, and diagram syntax. The errors found during code generation and compilation are displayed in AnyLogic Output window (see Figure 156). For each error, the Output window displays description and location.

Description	Location
<identifier> expected	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 406:2
not a statement	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 450:1
',' expected	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 450:27
<identifier> expected	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 501:2
<identifier> expected	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Root.java, 1444:2
',' expected	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 1534:3
cannot resolve symbol: class a	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 406:1
cannot resolve symbol: class a	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Network.java, 121:1
cannot resolve symbol: class a	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Root.java, 1444:1
cannot resolve symbol: class c	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 501:1
cannot resolve symbol: method initGhantts ()	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 47:1
cannot resolve symbol: method updateStatistic...	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 491:10
cannot resolve symbol: variable masterCount	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Machine.java, 841:1
cannot resolve symbol: variable masterCount	P:\AnyLogic\Setup\Target\Examples\Computers & Networks\leader_election\Root.java, 78:17

Figure 156. Output window

► To show/hide the Output window

1. Click the *Output*  toolbar button, or
Choose *View | Output* from the main menu, or
Press Alt+2.

You can open an error. Depending on the error, opening it may result in displaying different windows. If, for example, it is a graphical error, the corresponding diagram is opened with invalid shapes highlighted.


► To open an error

1. Double-click the error in the Output window.

It is not always possible to give an exact error location in AnyLogic windows. For example, if you are trying to use an identifier Java cannot resolve, it could be an undeclared variable, or a parameter, or anything else. In such cases, AnyLogic displays a .java file and positions the cursor at the error location. This file is opened read-only and it is up to you to track down the real error location in AnyLogic.

► To copy error messages on the Clipboard

1. Select the error messages you want to copy.

2. Click the *Copy*  toolbar button, or
Choose *Edit|Copy* from the main menu, or
Right-click the error message and choose *Copy* from the popup menu, or
Press Ctrl+Ins.

14.2 Viewing and modifying AnyLogic events

AnyLogic simulates the model as a sequence of time steps and event steps. The information presented in section 14.2.1, “Event processing at the simulation engine”, might be useful for better understanding of how the AnyLogic engine simulates discrete events and continuous behavior. Section 14.2.2, ”Events window”, describes how to view the event queue of AnyLogic simulation engine to view what is happening at the simulation engine at the lowest level details and to make some changes to the event processing.

14.2.1 Event processing at the simulation engine

AnyLogic simulates the model as a sequence of time steps and event steps. During a time step:

- The model clock is advanced.
- The “discrete” state of the model (the statechart, port, event, thread, etc. states) remains unchanged.
- Active equations, if any, are being solved numerically and the variables are changed correspondingly.
- Awaited change events are tested for occurrence.

During an event step:

- No model time elapses.
- The actions of states, transitions, timers, ports, etc. corresponding to this event are executed.
- The state of the model may change.

- Some scheduled events may be deleted, and the new events may be scheduled in the AnyLogic Engine event queue.

14.2.1.1 Engine events

AnyLogic engine events are events that occur at runtime. Please do not confuse them with static/dynamic events that are a part of AnyLogic modeling language. There are several types of engine events:

- **current** – events that can be executed at the time now
- **chosen** – one of the enabled events that is chosen to be executed next
- **enabled** – other current events (those that potentially could be executed next)
- **scheduled** – events scheduled at some particular known time in the future
- **pending** – events that may occur in the future, but the time is not known

Engine events reside in the engine event queue. Any event present in the engine event queue may be associated with:

- An active timer
- A transition triggered on a timeout expiry
- A thread executing a `delay()` statement

In addition, current events may be associated with something that has just happened as a result of other event execution:

- A transition being triggered by a port, immediately, or by a static, signal or change event
- A thread successfully exiting its `waitEvent()` or `waitForMessage()` statement

14.2.1.2 Time step

If there are no current events, AnyLogic makes a time step to the nearest event (or events) in the queue, i.e., advances its clock. During a time step a change event may occur. The discrete part of AnyLogic engine does not know when a change event associated with a transition occurs: it depends on the equation set being solved numerically by a continuous part of the

engine. Once this happens, the clock is advanced to the time reported by the continuous-time equation solver, and the event step is executed.

14.2.1.3 Event step

Several events may be scheduled to occur at the same moment of time. If there are several current events, AnyLogic chooses one and executes it. This is repeated until there are no current events. Thus, several event steps may be made in succession, whereas a time step is always followed by an event step. Simultaneous events may depend on each other or be truly concurrent. The serialization of concurrent events is called interleaving a model.

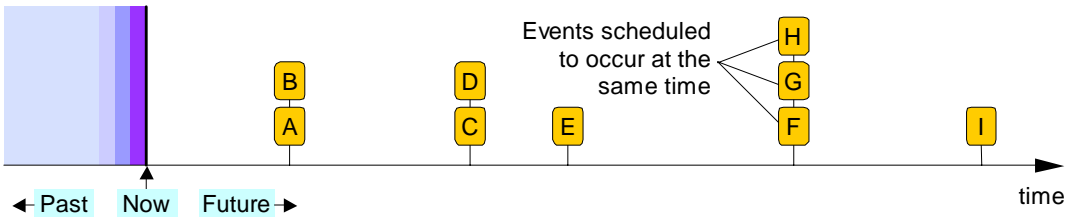
Depending on your task, you can tell AnyLogic simulation engine to do random or deterministic serialization of events. Random instead of deterministic serialization ensures that a bigger part of the system state space is covered by a simulation, so it is more likely that an undesirable behavior will be detected. Note that random event serialization slightly slows your model simulation.

► To set up deterministic/random event serialization

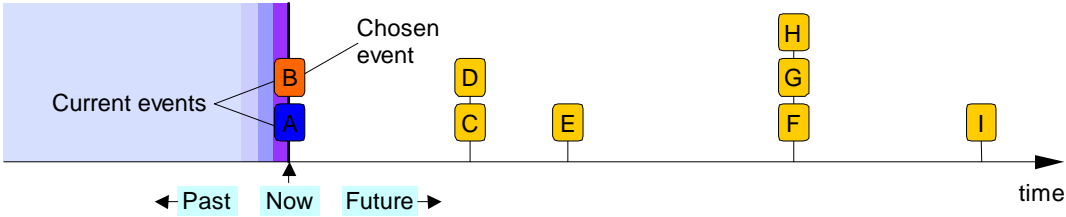
1. In the Project window, click the project item (the top-most item in the workspace tree).
2. On the *General* page of the Properties window, choose *Deterministically/Randomly* from the *Event scheduling algorithm* drop-down list.

The execution of a timer event is actually the execution of the timer's action code. The execution of a transition event is the execution of a set of actions associated with the transition. As a result of the event execution, the discrete state of the model may change: statecharts may change their states, other equations may be activated, other transitions may begin waiting, and other timers may be activated. Thus, some events may be deleted from the event queue and other events may be added to it.

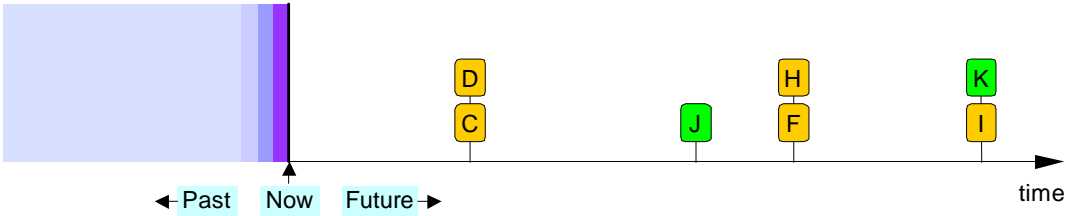
The example of AnyLogic event queue processing is shown in Figure 157.



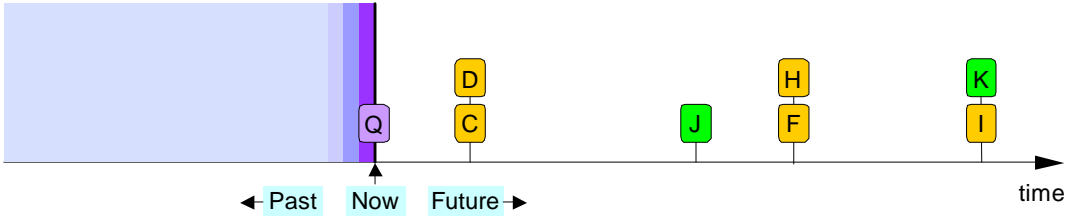
Time step: the clock is advanced to A and B - the head of the event queue. Active algebraic-differential equations are being solved. The event queue remains unchanged.



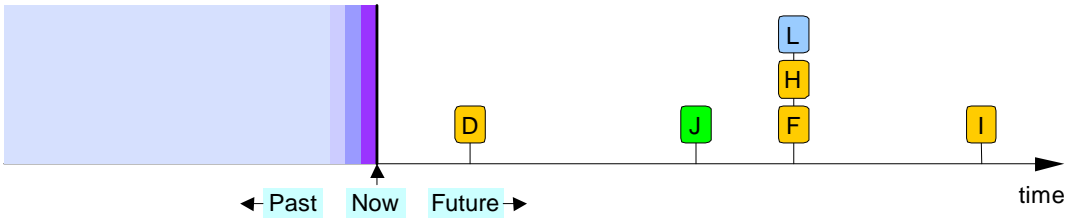
Event step: B is chosen and occurs. No time elapses. Model state changes. A, E and G are deleted from the event queue. J and K are scheduled.



Time step: the clock is advanced to C and D. Active algebraic-differential equations are being solved. Suddenly change event Q is detected.



Event step: Q is chosen and occurs. No time elapses. Model state changes. C and K are deleted from the event queue. L is scheduled.

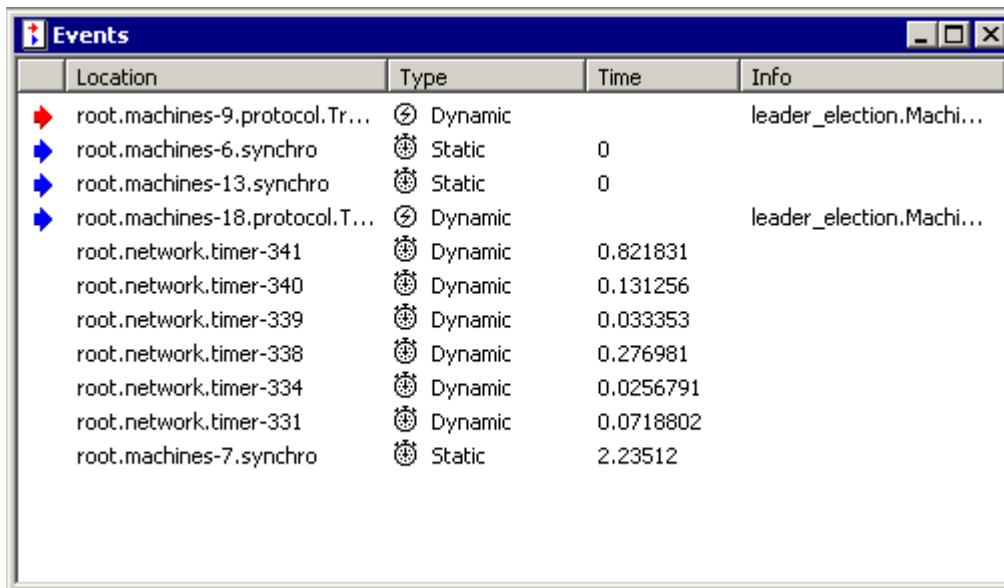


Time step: the clock is advanced to D ...

Figure 157. AnyLogic event queue (pending events not shown)

14.2.2 Events window

The events window, see Figure 158, displays the event queue of AnyLogic simulation engine either for the whole model or for a particular active object. You can use the events window for debugging purposes to view what is happening at the simulation engine at the lowest level details and to make some changes to the event processing. The user often works with the events window using the *Primitive Step* and *Detailed Play* commands, see section 11.1.2, “Controlling the model execution”.





Location	Type	Time	Info
root.machines-9.protocol.Tr...	Dynamic		leader_election.Machi...
root.machines-6.synchro	Static	0	
root.machines-13.synchro	Static	0	
root.machines-18.protocol.T...	Dynamic		leader_election.Machi...
root.network.timer-341	Dynamic	0.821831	
root.network.timer-340	Dynamic	0.131256	
root.network.timer-339	Dynamic	0.033353	
root.network.timer-338	Dynamic	0.276981	
root.network.timer-334	Dynamic	0.0256791	
root.network.timer-331	Dynamic	0.0718802	
root.machines-7.synchro	Static	2.23512	

Figure 158. Events window

► To open the global events window

1. Choose *View|Model Events* from the main menu.

Each event in the events window is displayed in the following form:

Flag –  for the chosen event,  for enabled events, no flag for other events.

Location – model path to the object that is associated with this event.

Type – “Dynamic” for dynamic events and dynamic timers, “Static” for static timers, “Change” for change events, and “Timeout” for timed transitions.

Time – relative or absolute occurrence time for scheduled events, no value for other events.

Info – inspect string for the event. E.g., if this is a message acting as a dynamic event, this field displays the string returned by the method `toString()` of the message.

At the event steps of the simulation, there exist current events – those that can be executed at the time now. Among the current events, there is one chosen to be executed next and others are considered as enabled (potentially could be executed next). You can control manually the choice among the current events:

► To change the chosen event

1. Right-click the enabled event and choose *Set chosen* from the popup menu.

Among the non-current events (those that cannot be executed at the time now) there may be events scheduled at some particular time known in the future, and others that do not know their times (for example, a transition waiting for a change event or a port). The latter ones are called pending, and are not shown in the Events window by default, although you can view them if you wish:

► To show/hide pending events

1. Right-click the events window and choose *Hide Pending Events* from the popup menu.

The occurrence times of the scheduled events may be displayed either as relative to “now” or as absolute values.

► To display the occurrence times in relative/absolute values

1. Right-click the events window and choose *Show Absolute Time* from the popup menu.


The occurrence time of a scheduled event may be modified.

► To change the occurrence time of a scheduled event

1. Right-click the scheduled event and choose *Modify* from the popup menu.
The *Modify Event Scheduling* dialog box opens.

2. Specify the new relative time of the event occurrence in the *Time* edit box.
3. Click *OK*.

14.3 Breakpoints

You can debug your model by setting a breakpoint on almost any element of the model – on active object, statechart, state, transition, chart timer, dynamic timer. When an element with a breakpoint on it becomes active during the model execution, the model stops, and the icon  is displayed in the status line. Thus you can trace model execution, detect some undesired activities or events in your model and perform some actions in response to the occurrences.

A breakpoint can be set from the Model Explorer or from an animated diagram. Breakpoints are displayed in animated diagrams dashed red. They are not displayed in the Model Explorer.

► To set/clear breakpoint on a model element

1. Right-click the element and choose *Breakpoint* from the popup menu.

You can manage (remove, enable, and disable) breakpoints using the *Edit Breakpoints* dialog box, see Figure 159.

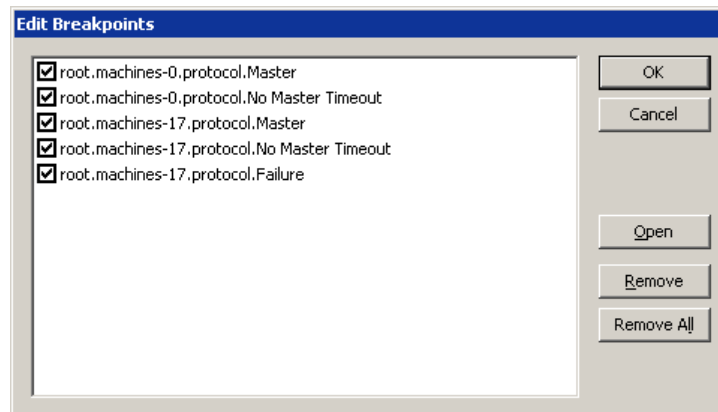


Figure 159. Edit Breakpoints dialog box

► **To open the Edit Breakpoints dialog box**

1. Choose *Model|Edit Breakpoints* from the main menu, or Click Alt+F9.
The *Edit Breakpoints* dialog box is displayed.

► **To enable/disable a breakpoint**

1. Select/clear the checkbox to the left of the breakpoint.

► **To remove a breakpoint**

1. Select the breakpoint.
2. Click *Remove* button.

► **To remove all breakpoints**

1. Click *Remove All* button.

► **To open the location of a breakpoint**

1. Select the breakpoint.
2. Click *Open* button.

14.4 Logging a model

14.4.1 Log window

You can output textual information for a model (global log) or for an individual active object during the model execution in AnyLogic log windows, see Figure 160. You can use it in debugging purposes to trace model execution by writing specific text to the log on different occurrences. The log is displayed as a read-only text, which can be copied onto the Clipboard. The log window of an active object appears in the Properties window when you

select an active object. Also a standalone log window can be opened from the popup menu of an active object.

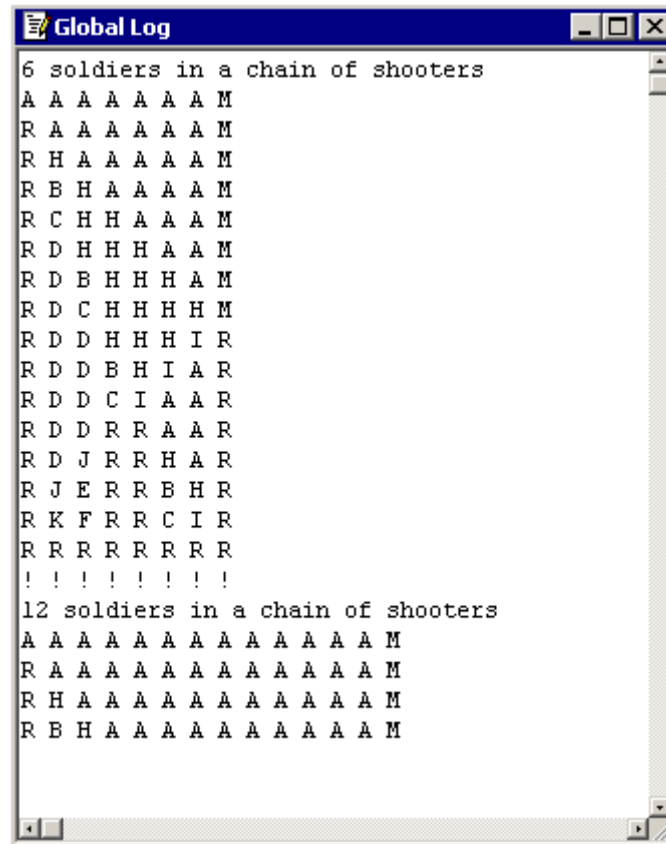


Figure 160. Log window

► **To open the global log**

1. Click the *Global Log*  toolbar button, or Choose *View | Global Log* from the main menu.

► **To open the log window of an active object**

1. Right-click the active object in the Model Explorer or on the animated structure diagram and choose *Log* from the popup menu.

14.4.2 Writing to logs

Each active object has a log. This is the variable `log` of type `PrintWriter` defined in the class `ActiveObject`. You write to `log` in the same way as you write to, e.g., `System.out`. Usually, you use the methods `print()` and `println()`.

In addition, a model has the so-called global log. This is the variable `log` of the same type `PrintWriter` defined as a static member variable of the class `Engine`. You can access the global log as `Engine.log`, or simply call the methods `trace()` and `traceln()` of the class `ActiveObject`. The method `trace()` calls `Engine.log.print()`, and the method `traceln()` calls `Engine.log.println()`.

The global log is convenient for output of information across several model replications, because it is not reset in between replications. It can also be used as a debugging tool, for example, to find out what is the order in which the model executes actions of different objects.

You work with logs using the following API (for more information, please consult AnyLogic Class Reference):

Related member variables and methods of `ActiveObject`

`java.io.PrintWriter log` – log of the active object.

`static void trace(java.lang.String value)` – prints a string to the global log.

`static void traceln(java.lang.String value)` – prints a string with a line delimiter to the global log.

Related member variables of `Engine`

`java.io.PrintWriter log` – the global model log.

14.5 Runtime errors

Various errors may occur during the model execution. Runtime errors can be of two types:

- Java exceptions

- Simulation errors.

You can throw runtime error and terminate model execution as a reaction to different undesirable occurrences.

14.5.1 Java exceptions

Java code written by the user may contain unintentional errors like division by zero, accessing null pointer and so on. Such errors are detected by Java runtime environment. If such an error occurs, Java throws an exception.

AnyLogic catches all exceptions. If an exception occurs, AnyLogic stops the model, notifies the user with a message box and dumps the exception to the global log. You can examine the log to find out where there is a bug.

14.5.2 Throwing runtime errors

You can debug your model at runtime by throwing runtime errors as a reaction to undesirable occurrences using the static methods `error()` and `assert()` of the class `Engine`. The `error()` method throws runtime exception to the simulation engine. The `assert()` method checks if the model behaves normally by checking the specified Boolean condition to be `true` and throws a runtime exception if it is `false`. Consult AnyLogic Class Reference for more details.

Related methods of Engine

`static void error(String message)` – the method throws a runtime exception to the simulation engine. AnyLogic immediately stops model execution, shows the window, displaying the message and dumps the exception to the AnyLogic global log.

`static void assert(boolean assertion, String message)` – the method checks the assertion condition, and in case it's `false`, calls the `error()` method with specified message to raise a runtime error.

14.5.3 Simulation errors

Simulation errors are logical errors of model execution. For example, if a statechart is unable to exit a branch because all exiting transitions are closed, it is a simulation error. Simulation errors are detected by AnyLogic rather than by Java runtime environment. If a simulation error occurs, AnyLogic stops the model and notifies the user with an error message.

14.6 Debugging Java code

AnyLogic does not have a built-in Java code debugger. However, you can debug Java code using a third-party debugger. The user is free to run a model within the debugger using command line execution feature (see section 18.1, “Running a model from the command line”) or to attach the debugger to the currently running model. Using the debugger control, you can trace the code, set breakpoints, output debug information, and so on.

The source code of AnyLogic Engine is not included in the distribution set. You are able to debug your own Java code, and not the engine code.

15. Creating a model with dynamically changing structure

AnyLogic is the only visual tool that supports creation of truly dynamic models – the ones with dynamically evolving structure and component interconnection.

AnyLogic supports:

- Dynamic creation and removing of encapsulated objects. It is described in subsection 15.1, “Manual creation and destruction of encapsulated objects”. Moreover, dynamic creation and removing elements of a vector of replicated objects is enabled. See section 2.2, “Accessing and modifying a replicated object at runtime” for details.
- Dynamic changing of interface elements connections. It is described in subsection 15.2, “Dynamically changing connections”.

15.1 Manual creation and destruction of encapsulated objects

There are several situations when you may need to take care of the creation of encapsulated objects and tell AnyLogic not to do it automatically:

- The object has a limited lifetime and should be created and destroyed dynamically as the model evolves.
- The object has several constructors and it is not known in advance which one should be called.
- The actual class of the object is not known in advance.

To create an encapsulated object manually, it is sufficient to write the code equivalent to the one generated by AnyLogic. The algorithm of writing the code is the following:

1. Prepare the place where you will store the reference to the encapsulated object. This can be a member variable of the corresponding type, an array, a vector, etc. In case

the encapsulated object is present on the structure diagram, but its property *Auto create* is not set, the member variable for the reference is generated by AnyLogic, but the encapsulated object is not created. Thus you can create the encapsulated object manually and store the reference in the member variable generated by AnyLogic.

2. When you wish to create an encapsulated object, construct an instance of the corresponding class using operator `new` and store the reference in the prepared place. Then call the method `setup_myObject1()` generated by AnyLogic or call the equivalent code. Do not forget to call `register()`. This method registers the object in AnyLogic simulation engine.
3. If you are creating an encapsulated object during the model initialization, pass `false` in the parameter `autoRun` of the method `setup_myObject1()`. If you are doing this during the model execution, pass `true` – this makes the encapsulated object automatically start all its activities. For your convenience AnyLogic generates the overloaded method `setup_myObject1()` that sets the parameter `autoRun` to `true`.
4. When you wish to destroy the encapsulated object, call the method `dispose_myObject1()` or call the equivalent code. Do not forget to call `unregister()` – this deletes the reference to the object from AnyLogic simulation engine. Make sure nobody in the model refers to the object and the object itself does not refer to anybody. Note that the object will be deleted by the garbage collector with a certain delay.

In case an encapsulated object is created and destroyed dynamically, its animation appears and disappears synchronously with the object.

15.1.1 Writing code executed on object creation and destruction

AnyLogic enables you to write code to be executed on various stages of active object creation and destruction. Thus, you can specify arbitrary actions to be performed on object creation and destruction.

15.1.1.1 Writing code executed on object creation

You can define code to be executed on various stages of active object creation. You can:

- Define one or several constructors of active objects
- Override the method `onCreate()`
- Define the code section *Startup code*.

The order of execution of this code is the following:

1. First, the object constructor is called. It is fed with parameters specified in the *Constructor parameters* code section of the encapsulated object. You can define your own constructor and place the code there to be executed during the construction. Custom constructors can be defined in the *Additional class code* code section of the active object class.
2. The method `create()` of the active object is called. You cannot insert code into the method `create()`. AnyLogic places the code there that creates elements defined on the structure diagram of this active object class.
3. The method `onCreate()` of the active object is called. The user can override the method `onCreate()` to perform some actions, if necessary. Usually, this includes manual connection of ports and variables and manual creation of encapsulated objects. By the time `onCreate()` is called, all elements of the structure diagram are already created, except statecharts. The method `onCreate()` can be defined in the *Additional class code* code section of the active object class.
4. Finally, the method `startup()` is called. At the beginning of this method AnyLogic starts statecharts defined on the structure diagram. At the end of this method, AnyLogic inserts the code specified by the *Startup code* code section of the active object class. The execution of the startup code is the final stage of object creation.

15.1.1.2 Writing code executed on object destruction

You can define code to be executed on various stages of active object destruction. You can:

- Override the method `cleanup()`
- Override the method `onDestroy()`

The order of execution of this code is the following:

1. The destruction starts with the call to the method `cleanup()`. By the time `cleanup()` is called, all elements of the structure diagram still exist. You can override the method `cleanup()` to do some custom cleanup. The method `cleanup()` can be defined in the *Additional class code* code section of the active object class.
2. The method `destroy()` is called. The user cannot insert the custom code into the method `destroy()`. AnyLogic places there the code destroying all elements of the structure diagram.
3. Finally, the method `onDestroy()` is called. The user can override the method `onDestroy()` to do some additional necessary work. Usually in `onDestroy()` you manually destroy encapsulated objects which were created manually. By the time this method is called, all elements of the structure diagram are already destroyed. You can define the method `onDestroy()` in the *Additional class code* code section of the active object class.

15.2 Dynamically changing connections

You can model systems with dynamically evolving component interconnection by changing interface elements connections at runtime. See section 4.3.2.2, “Connecting variables at runtime” and section 7.4.12, “Connecting ports at runtime” for more details.

16. Optimization

If you need to run a simulation and observe system behavior under certain conditions, as well as improve system performance, for example, by making decisions about system parameters and/or structure, you can use the optimization capability of AnyLogic. Optimization is the process of finding the optimal combination of conditions resulting in the best possible solution. Optimization can help you find, for example, the optimal performance of a server or the best method for processing bills.

AnyLogic optimization is built on top of the OptQuest Optimization Engine¹, one of the most flexible and user-friendly optimization tools on the market. The OptQuest Engine automatically finds the best parameters of a model, with respect to certain constraints. AnyLogic provides a convenient graphical user interface to set up and control the optimization.

The optimization process consists of repetitive simulations of a model with different parameters. Using sophisticated algorithms, the OptQuest Engine varies controllable parameters from simulation to simulation to find the optimal parameters for solving a problem.

► To optimize your model

1. Specify the function to be minimized or maximized, parameters to be varied, and constraints to be met. See section 16.1, “Setting up an optimization”.
2. Optionally, adjust the OptQuest Engine settings for your problem. See section 16.3, “Optimization settings”.
3. Run the optimization. See section 16.4, “Running the optimization”.

¹ OptQuest is a registered trademark of OptTek Systems, Inc. For advanced information about the OptQuest Engine, please visit OptTek’s web site www.opttek.com.


16.1 Setting up an optimization

You set up an optimization with these five steps:

1. Create an optimization experiment
2. Define the objective
3. Define optimization parameters
4. Define constraints (optional)
5. Specify the simulation stop condition
6. Specify the optimization stop condition

16.1.1 Creating an optimization experiment

► **To create an optimization experiment**

1. Click the *New Experiment*  toolbar button, or Choose *Insert | New Experiment...* from the main menu, or In the Project window, right-click the *Experiments* item and choose *New Experiment...* from the popup menu. The *Create a new experiment* dialog box is displayed.
2. Select the *Optimization experiment* option.
3. Type the experiment name in the *Name* edit box.
4. Choose the root object of the experiment from the *Root object* drop-down list.
5. Click *OK*.

You need to set the created optimization experiment as a current experiment.

► **To set an experiment to be a current experiment**

1. In the Project window, right-click the optimization experiment and choose *Set as Current* from the popup menu.

The optimization experiment has the following properties, set up in the experiment's properties window (Figure 161).

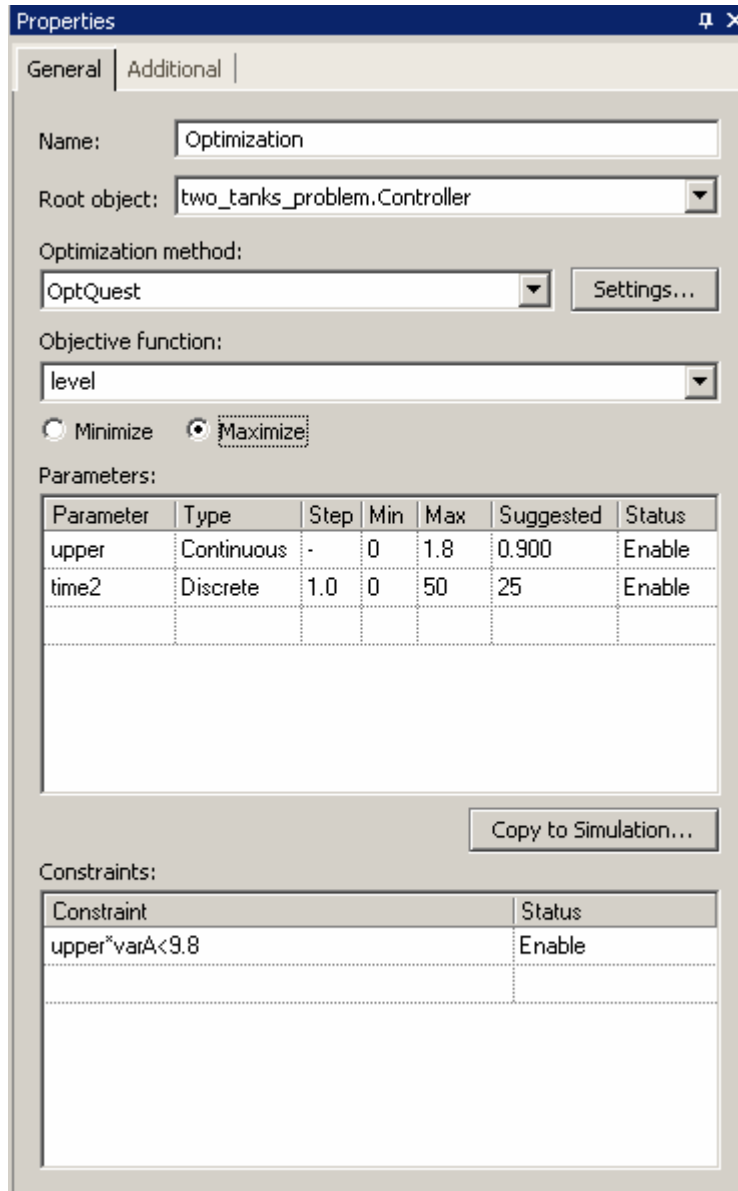


Figure 161. General page of optimization experiment's Properties window

Properties

Name – the name of the experiment.

Root object – the root object of the experiment.

Optimization method – the optimization method used. *OptQuest* method is set by default.

Settings – the button opens the *OptQuest settings* dialog box. See section 16.3, “Optimization settings” for the details on changing optimization settings.

Objective function – the objective function. See section 16.1.2, “Defining the objective” to know how to define an objective.

Minimize | *Maximize* – the optimization criterion. Determines whether the objective function should be minimized or maximized.

Parameters – the set of optimization parameters. See section 16.1.3, “Defining optimization parameters” to know how to define parameters.

Constraints – [optimal] the set of constraints. Constraints are checked at the end of each simulation, and if they are not met, the parameters used are rejected. Otherwise the parameters are accepted. See section 16.2, “Constraints” for details.

16.1.2 Defining the objective

The goal of the optimization process is to find the parameter values that result in a maximum or minimum of a function called the *objective function*. Objective function is a mathematical expression describing a relationship of the optimization parameters or the result of an operation (such as simulation) that uses the optimization parameters as inputs. The optimization *objective* is the objective function plus optimization criterion. The latter determines whether the goal of the optimization is to minimize or maximize the value of the objective function.

► To define the objective

1. In the Project window, click the optimization experiment.
2. In the Properties window, specify the objective function in the *Objective* edit box. Choose one of the variables of the root active object from the drop-down list, or enter a custom expression.

3. Define the optimization criterion.

Choose the *Minimize/Maximize* option to minimize/maximize your objective function.

You can enter any Java expression as an objective function, including an arithmetic expression or method call. Since expression is considered to be in context of the root active object class, it can access variables and parameters of the root active object. If your function is rather sophisticated, you can define a method, calculating the function, in the *Additional class code* code section of the root active object class, and place the method call in the *Objective* edit box. If you use a method, it is called only once after each simulation rather than numerous times during a simulation. If an objective function calculation is time-consuming, placing it in a method is preferred.

The OptQuest Engine obtains a sample of the objective function at the end of each simulation. The engine analyzes a sample, modifies optimization parameters according to its optimization algorithm, and starts a new simulation.

Therefore, optimization is an iterative process where:

- The OptQuest Engine calculates possible solutions for the parameters
- The objective function and constraints are evaluated using the suggested solutions
- The results are analyzed by the OptQuest Engine, and a new set of possible solutions is calculated

16.1.3 Defining optimization parameters

An optimization parameter (or a decision variable, in the terms of optimization) is a model parameter to be optimized. For example, the number of nurses to employ during the morning shift in an emergency room may be an optimization parameter in a model of a hospital. The OptQuest Engine searches through possible values of optimization parameters to find optimal parameters. It is possible to have more than one optimization parameter.

Only a parameter of the root active object class can be an optimization parameter. If you need to optimize parameters of encapsulated objects, you should use parameter propagation.

An AnyLogic variable cannot be an optimization parameter. This is because of the difference between variables and parameters. A variable represents a model state, and can change

during simulation. A parameter is normally a constant in a single simulation, and is something that influences the behavior of the model.

In order to perform optimization, you must have at least one parameter in the root active object class.

16.1.3.1 Optimization parameter types

During the optimization process, the parameter's value is changed in accordance to its type within an interval, specified by lower and upper bounds. There are the following types of optimization parameters:

- Continuous parameter
- Discrete parameter
- Design parameter

Continuous parameter can take any value from the interval. The parameter precision determines the minimal value continuous parameters can change.

Discrete parameter is represented by a finite set of decisions with essential direction: the parameter influences the objective like a numeric parameter, but can take values from the specified set only. It begins at a lower bound and increments by a step size up to an upper bound. For example, a discrete parameter with 1.0 step represents a set of integer values.

Sometimes the range and step are exactly defined by the problem; but generally you will have to choose them. If you specify the step for the parameter, only the discrete points will be involved in the optimization, so it will be impossible to determine optimal parameter value more precisely than defined by the step. So, if you are not sure what the step should be, choose the *Continuous* rather than the *Discrete* parameter type.

Design parameter is represented by a finite set of decisions, where there is no clear sense of direction. Value of design parameter represents an alternative but not a quantity. It begins at a lower bound and increments by a step size up to an upper bound. Values order is inconsequential. Using design parameters you can model choosing the best alternative from the catalog, where the choices are not in a specific order. For example, a design

parameter, which can take values 0 or 1 (min=0, max=1, step=1) may represent a choice between: a model has some element or has not.

Optimization parameters are defined in the *Parameters* table on the optimization experiment's properties page. Each parameter is defined in individual row.

► **To define an optimization parameter**

1. Click the optimization experiment in the Project window.
2. Go to the last row in the *Parameters* table.
3. Click the *Parameter* field and choose a parameter from the drop-down list.
4. Click the *Type* field and choose the *Continuous* | *Discrete* | *Design* parameter type from the drop-down list.
5. Specify the range for the parameter. Enter the parameter's lower bound in the *Min* field and the parameter's upper bound in the *Max* field.
6. For *Discrete* and *Design* parameters, specify the parameter step. Double-click the *Step* field and enter the step value.
7. Suggest the initial value for the parameter in the *Suggested* field. Initially, the value is set to the parameter's default value, but you can enter any other value.
8. Finally, click the *Status* field and choose *Enable/Disable* from the drop-down list to add the parameter to the optimization process or to eliminate the parameter from it.

The dimension of the search area depends on the number of optimization parameters. Each new parameter expands the search area, thus slowing down the optimization. If you have N optimization parameters, their ranges form the N-dimensional square search area. Obviously, that area must be wide enough to contain the optimal point. However, the wider the range is, the more time is needed to find the optimum in the search area. On the other hand, suggested parameter values located near the optimal value can shorten the time it takes to find the optimal solution.

16.2 Constraints

AnyLogic supports *constraints* - additional restrictions imposed on the solution found by the optimization engine.

16.2.1 Defining a constraint

In AnyLogic, a constraint is a range specified for a model variable. This means that if all variables meet their constraints at the end of the simulation, the corresponding set of parameter values is considered feasible, and the result of the simulation is accepted by the optimization engine. Otherwise, parameter values are considered infeasible, and the result is rejected.

A constraint is a well-formed arithmetic expression describing a relationship between the optimization parameters. It always defines a limitation by specifying a lower bound and/or an upper bound.

```
VarA + VarB + 2*VarC = 10
```

```
VarC - VarA*VarB >= 300
```

It can also be a restriction on a response that requires its value to fall within a specified range. Constraint may contain any variables of the active object, the model time symbol t , any arithmetic operations and method calls, such as, e.g., `sin()`, `cos()`, `sqrt()`, etc., or calls to your own methods.

```
0 <= 2*VarB - dataset1.max() <= 500
```

```
sqrt(VarC)>=49
```

► To define a constraint

1. Click the optimization experiment in the Project window.
2. Go to the last row in the *Constraints* table.
3. Double-click the *Constraint* field. Specify the constraint expression.
The constraint is defined as a string.

You can disable the constraint. The disabled constraints are eliminated from optimization process and do not affect its results.

► **To disable/enable a constraint**

1. Click the optimization experiment in the Project window.
2. Go to the row with the constraint in the *Constraints* table.
3. Click the *Status* field and choose *Disable/Enable* from the drop-down list.

Optimization performance usually suffers when you include constraints. Therefore, you should try to avoid using constraints whenever possible.

Note that a linear constraint defined upon optimization parameters only is managed significantly faster than more complex ones. Such a constraint defines a range for an optimization parameter. Each time the optimization engine generates a new set of values for the optimization parameters, it creates feasible solutions, satisfying this constraint; thus the space of searching is reduced, and the optimization is performed faster.

16.2.2 Feasible and infeasible solutions

A feasible solution is one that satisfies all constraints.

The optimization engine makes finding a feasible solution its highest priority. Once it has found a feasible solution, it concentrates on finding better solutions.

The fact that a particular solution may be infeasible does not imply that the problem itself is infeasible. However, infeasible problems do exist. Here is an example:

```
varA + varB <= 4
varA + varB >= 5
```

Clearly, there is no combination that will satisfy both of these constraints.

If a model is constraint-feasible, the optimization engine will always find a feasible solution and search for the optimal solution (i.e., the best solution that satisfies all constraints).

If the optimization engine cannot find any feasible solutions, you should check the constraints for feasibility and fix the inconsistencies of the relationships modeled by the constraints.

16.3 Optimization settings

To enable optimization, you must ensure that each simulation ends. By default, a simulation never ends; therefore the optimization engine gets no samples of the objective function. To ensure that each simulation ends, you must define a simulation stop condition (see section 13.3, “Simulation stop conditions” for details). Note that a simulation may involve several replications. A sample of the objective function is the result of a simulation rather than a replication.

In general, a simulation stop condition should be specified in such a way that the value of the objective function is significant when simulation stops. The model should be stable, transient processes should be finished, and statistics should be representative.

Optimization can stop under two circumstances: the maximum number of simulations is exceeded or the value of the objective function stops improving. The latter is also known as *automatic stop*. You can use either of these conditions to stop an optimization. If more than one condition is specified, optimization stops when the first condition is satisfied.

Set up these settings of an optimization process using the *OptQuest settings* dialog.

► To open the OptQuest settings dialog

1. Click the optimization experiment in the Project window.
2. In the Properties window, click the *Settings* button.
The *OptQuest settings* dialog box is displayed, see Figure 162.

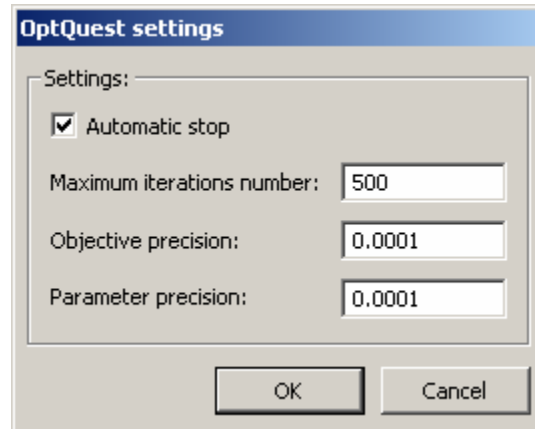


Figure 162. OptQuest settings dialog box

The following options are set in the *OptQuest settings* dialog box:

Automatic stop – if set, the optimization stops when the value of the objective function stops improving. An objective function is said to stop improving if its values differ by less than p during n iterations, where p is the objective function precision, and n is calculated as 5% of the *Maximum iterations number*, but n cannot be less than 50. You can control this by specifying the objective function precision and number of iterations.

Maximum iterations number – the maximum number of simulations. The optimization stops when this number is exceeded.

Objective precision – the objective precision. If the difference between two objective function values is less than the objective precision, the solutions are considered equal.

Parameter precision – the optimization parameter precision. If the difference between two parameter values is less than the parameter precision, the values are considered equal. The parameter precision specifies the minimum amount a continuous parameter can change.

The number of simulations influences the optimization strategy. If the number of simulations is small, the OptQuest Engine uses an aggressive search strategy to exploit the parameter space. If the number is rather large, the OptQuest Engine uses a more conservative strategy to thoroughly explore the search space.


16.4 Running the optimization

After you set up the optimization, you are ready to run it. This involves the following steps:

- Start the optimization
- Observe the status
- Determine the best solution

If desired, you can pause the optimization while it is running. This section describes how you can control running the optimization.

► To start the optimization

1. Set the optimization experiment as a current experiment.
2. Click the *Run*  toolbar button, or Choose *Model|Run* from the main menu.
This starts the optimization process and opens the Optimization window as shown in Figure 163.



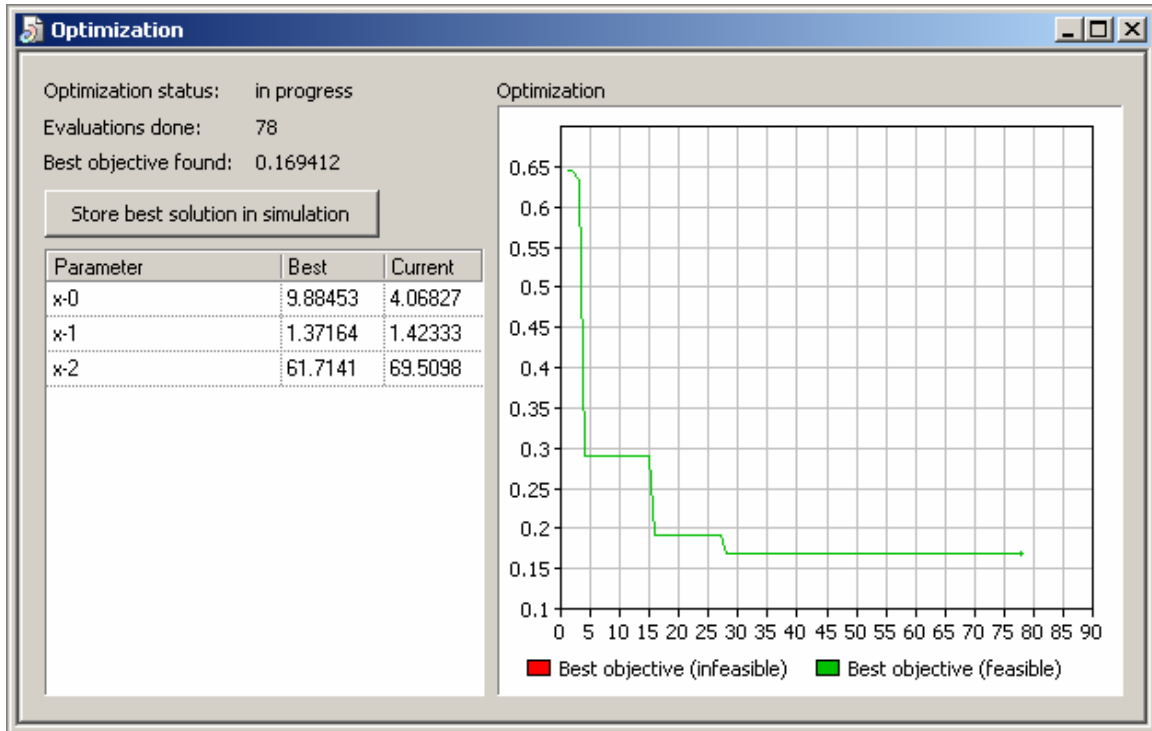


Figure 163. Optimization window

The Optimization window is automatically displayed when the model starts running. Sometimes you may need to close the window to reduce data display overhead. You can open it later when needed.

► To open the Optimization window

1. Click the *Optimization*  toolbar button, or
Choose *View | Optimization* from the main menu.

The Optimization window displays all necessary information regarding optimization, such as the current status of the optimization and a chart that visually illustrates the optimization process. The following information is displayed in the upper left-hand corner of the window:

Optimization status – reflects the current state of the optimization: *not started*, *in progress*, or *finished*.

Evaluations done – number of simulations already completed.

Best objective found – the best value of the objective function.

Store best solution in simulation – the button stores the best found solution in a simulation experiment.

The chart in the Optimization window illustrates the progress of the optimization. The X-axis represents simulations, and the Y-axis represents the best value of the objective function found for each simulation.

The table above the chart displays information about optimization parameters. The table includes the following information:

Parameter – name of the optimization parameter.

Best – parameter value corresponding to the best solution found.

Current – value used in the current simulation.

The values in the column *Best* form the best solution found up to the current time. Once optimization has finished, this solution is considered to be optimal. The best value of the objective function shown as *Best objective found* corresponds to this optimal solution.

► To stop the optimization

1. Click the *Stop*  toolbar button, or
Choose *Model|Stop* from the main menu.

You can suspend optimization and examine the model using various windows and commands, as described in Chapter 11, “Running and observing a model”. This becomes useful if you want to examine precisely how the model performs under the current set of optimization parameters. You can even modify some parameters; keep in mind that those modifications affect the current simulation only.

► To suspend optimization

1. Click the *Pause*  toolbar button, or
Choose *Model|Pause* from the main menu.

When suspended, you can resume optimization or debug it by running the model step-by-step.

► To resume optimization

1. Click the *Run*  toolbar button, or
Choose *Model|Run* from the main menu.



► To run the model step-by-step

1. Click the *Step*  toolbar button, or
Choose *Model|Step* from the main menu.

You can use other commands such as *Primitive Step* and *Step In Window*, as well.

You can also restart the optimization.

► To restart optimization

1. Click the *Restart Model*  toolbar button, or
Choose *Model|Restart* from the main menu.
2. Click the *Run*  toolbar button, or
Choose *Model|Run* from the main menu.

16.5 How to increase optimization performance

If you find optimization performance unsatisfactory, consider the following recommendations:

► Reduce data display overhead

- Reset the *Model|Auto Refresh Views* option.
- Close, or at least minimize all windows except the *Optimization* window.
- Minimize AnyLogic.

Be careful when using the Automatic stop option. In the case of an optimization jam, (i.e., when the objective function is changing too slowly), it is possible that optimization will stop long before the real optimal solution is found. If you encounter this problem, decrease the objective function precision, suggest other parameter values, or do not use Automatic stop.

► Find an optimal solution faster

- Adjust the OptQuest Engine settings for your problem. Fine-tuning these settings can increase optimization performance (for more information, please consult AnyLogic Class Reference):
- Suggest initial values for optimization parameters as close to the optimal value as possible
- Reduce the search space by specifying ranges for optimization parameters
- Exclude parameters that do not influence the objective function
- Avoid using constraints. First, optimize your model without constraints; then check the optimal solution for constraints. If some constraints are not satisfied, start optimizing with constraints.
- Solve the optimization problem interactively and iteratively:
 1. Initially, use a rough approximation of the problem: wide ranges, big steps, low precisions.
 2. Run the optimization until the best-found value starts changing slowly.
 3. Set up the optimization more precisely. Reduce ranges and steps of optimization parameters. Start with optimization parameter values obtained on the previous step.
 4. Run the optimization until the best-found value starts changing slowly. If you are satisfied with the results, stop the optimization. Otherwise, go back to step 3.

► Speed up simulation

- Make sure the model runs in virtual time rather than in real time. See section 13.1, “Simulation speed” to know how to set virtual time mode.
- Keep the simulation stop simple, see section 13.3, “Simulation stop conditions”.

- Use formulas instead of equations wherever possible.
- Increase model performance in any other way possible: Optimize user-defined Java code. Eliminate unnecessary objects and communications.

In general, the optimization process may be very time-consuming, especially if there are multiple parameters and constraints. If nothing from the list above helps improve performance, try using a more powerful workstation or schedule more time for optimization.

16.6 Tips and notes

Here are some tips you may find helpful:

- Debug your model before starting the optimization. If your model cannot run properly under some values of optimization parameters, you can restrict the search space using ranges. Otherwise, be aware of possible incorrect model operation.
- If the optimization engine finds all solutions to be infeasible (drawn red on the *Optimization* chart, see Figure 163), this indicates that there is no solution in the parameter space satisfying all the constraints. Possible reasons for this are:
 1. The constraints are inconsistent. Check for conflicting constraints, such as $x > 25$, $x < 24$. If variables appearing in the constraints are calculated using some formulas inside the model, the inconsistency can be in those formulas.
 2. The ranges of the optimization parameters conflict with the constraints.

17. Collecting data and performing statistical analysis

AnyLogic supports collecting and visualizing data during model execution.

The data is collected in AnyLogic datasets. It can be collected explicitly by adding data by user or implicitly by collecting data on variables during the model execution. The collected data is visualized in chart windows and AnyLogic animation.

AnyLogic also supports the statistics block for analyzing data: calculating statistical information on datasets, such as mean value, minimum, maximum, etc. Collected statistics can be exported to external applications.

17.1 Collecting data in datasets

AnyLogic supports datasets – objects that help you to collect, display, and analyze data during model execution.

A dataset has three main elements:

- Array of 2D data points of a finite size,
- Array of intervals where sample hits are counted (for histograms and Gantt charts),
- A block of statistical data for all data ever added to the dataset.

Each data point of a dataset has two values: x and y. The y-value is usually a sample of some observable of a model. Depending on the dataset, the x-value can represent one of the following (see Figure 164):

- Sample number – for a non-timed dataset that records y-value samples.
- Logical time – for a timed dataset that records how the y-value changes over time.
- Another observable – for a phase dataset that records pairs of arbitrary samples showing how some y-value changes depending on some x-value.

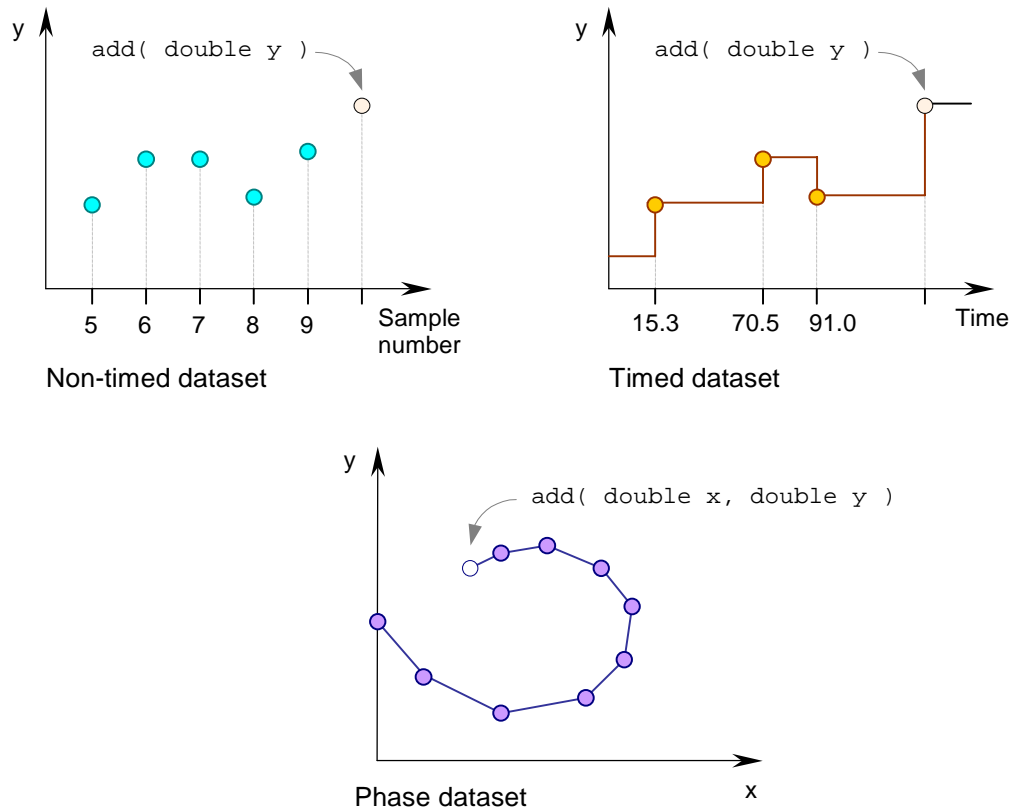


Figure 164. Non-timed, timed, and phase datasets

You use method `add(...)` to add a data point to a dataset. The parameters of this method differ depending on the dataset type. The number of points stored in a dataset is limited, and the “oldest” point is deleted when the limit is reached.


If intervals are defined for a dataset, the dataset collects the number of interval hits. Intervals are used when the dataset is displayed as a histogram or a Gantt chart. The number of hits per interval is accumulated characteristics of the dataset, where all data points ever added, including the lost ones, count.

The block of statistical data also stores the accumulated statistical characteristics of the dataset, where all data points ever added count.

17.1.1 Defining a dataset

Datasets are defined within active object classes. You usually define datasets graphically in the Project window.

► To add a dataset to an active object class

1. Click the *New Dataset*  toolbar button, or Choose *Insert | New Dataset...* from the main menu. The *New Dataset* dialog box is displayed. Specify the name of the new dataset, choose the active object class, which will contain the dataset, and click *OK*.
2. Alternatively, in the Project window, right-click the active object class and choose *New Dataset...* from the popup menu. The *New Dataset* dialog box is displayed. Specify the name of the new dataset and click *OK*.

A dataset has the following properties:

Properties

Name – name of the dataset.

Tail size – [optional] maximum number of points the dataset can store.

Phase – if set, the dataset is of phase type.

Timed – if set, the dataset is of timed type. Does not apply to phase datasets.

Smooth – if set, the dataset is of smooth (averaged) type. Does not apply to phase datasets.

Window size – averaging parameter. Applies only to smooth datasets.

Regular intervals – [optional] set of regular intervals for the dataset. Every interval is specified in the form: *Lower bound Upper bound Number of intervals*, where *Lower bound* is the lower limit of the intervals, *Upper bound* is the upper limit of the intervals, *Number of intervals* is the number of intervals.

Custom intervals – [optional] set of custom intervals for the dataset. Every interval is specified in the form: *Lower bound Upper bound Color*, where *Lower bound* is the

lower limit of the interval, *Upper bound* is the upper limit of the interval, *Color* is the Java color of the interval.

Exclude from build – if set, the dataset is excluded from the model.

17.1.2 Dataset types

17.1.2.1 Timed dataset

A timed dataset is used when an observable value is associated with a time moment when it is sampled. A queue length, a vehicle coordinate, a fluid level are the examples of such values.

To add a point to a timed dataset, you use the following API (please consult AnyLogic Class Reference for more details):

Related method of `TimedDataSet`

`void add(double y)` – adds a pair <current time, y> to the dataset.

17.1.2.2 Non-timed dataset

A non-timed dataset is used when an observable value is associated with a discrete event and not with a time moment when it is sampled. A packet end-to-end delay or success/failure Boolean data are examples of such values.

To add a point to a non-timed dataset, you use the following API (please consult AnyLogic Class Reference for more details):

Related method of `DataSet`

`void add(double y)` – adds a pair <sample number, y> to the dataset.

17.1.2.3 Smooth timed and non-timed datasets

If you wish to observe values averaged over some period of time (or over some number of samples) preceding the actual value occurrence, you should use a smooth dataset. When you add a point to a smooth dataset, its history (either continuous or discrete) is weighted taking the new point into account, and the averaged value is actually added. The weighting function is exponential, so that the older is the point the less is its weight, see Figure 165.

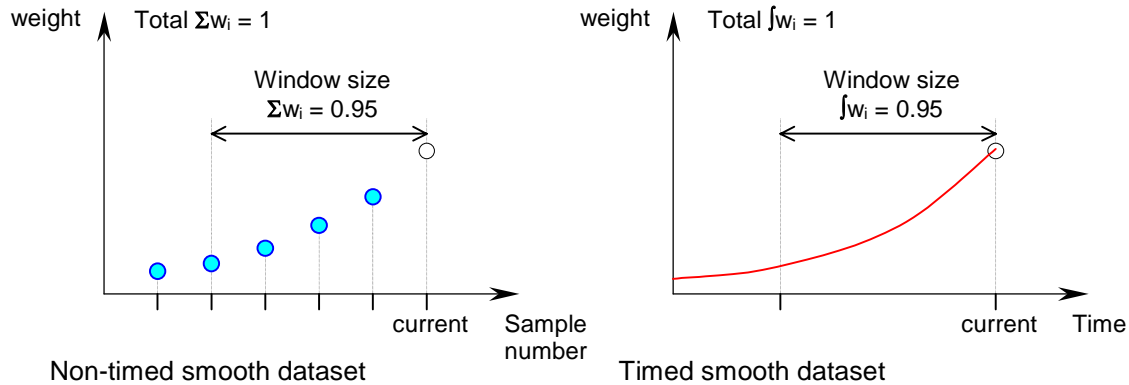


Figure 165. Weight function of smooth datasets

Smooth datasets have a property *Window size*, defining the size of history being averaged. In case of timed dataset it is the “backward” time period covering 95% of weight. In case of non-timed dataset it is the number of samples with the total weight of 95%.

The corresponding Java classes are `SlidingWindowDataSetT` for timed and `SlidingWindowDataSet` for non-timed datasets. Consult AnyLogic Class Reference for more details.

17.1.2.4 Phase dataset

A phase dataset is used when you wish to record the dependency of one value on another.

For example, you could use a simulation parameter as x-value, and some observable as y-value. Then several model replications with different parameters show how the observable depends on the parameter. A single model replication with the fixed parameter gives just one data point.

To add a point to a phase dataset, you use the following API (please consult AnyLogic Class Reference for more details):

Related method of PhaseDataSet

```
void add( double x, double y ) – adds a pair <x, y> to the dataset.
```

17.1.3 Intervals associated with datasets

A dataset may have intervals. Intervals are used when the dataset is displayed as a histogram or a Gantt chart. The histogram displays the number of sample hits for each interval. Gantt chart displays the multicolor stripe along the X-axis where the color is either the color of the interval hit by the current y-sample. Intervals can be zero-length (contain one point), or have lower and upper bounds.

If intervals are defined, a dataset collects interval-based information. A non-timed dataset accumulates the number of hits of y-coordinate into each interval. A timed dataset accumulates the total time during which the y-coordinate was within each interval range.

To define intervals, you either edit the *Regular intervals* or the *Custom intervals* property for datasets displayed as histograms or Gantt charts correspondingly, or use the following API (for more information please consult AnyLogic Class Reference):

Related methods of DataSetBase

```
void addInterval( double point, java.awt.Color color ) – adds a point  
(zero-length) interval to the dataset.
```

```
void addInterval( double from, double to, java.awt.Color color ) –  
adds an interval to the dataset.
```

17.1.4 Statistics block

The statistics block calculates statistical information on a dataset, such as mean value, minimum, maximum, etc. The statistics block works differently for different types of datasets. For example, timed dataset samples are time-persistent; that is, y-value persists in time until the next change. The mean of such dataset is a time-weighted value. Non-timed

dataset samples are not time-persistent. They occur as isolated, discrete points in time, so the mean is simply the sum of individual samples divided by number of samples.

Information calculated by the statistics block can be seen in a dataset inspect window in AnyLogic. Also, the statistics block is accessible via the following API (for more information please consult AnyLogic Class Reference):

Related methods of DataSet

`int count()` – returns the number of samples added to the dataset.

`double min()` – returns minimum sample added to the dataset.

`double max()` – returns the maximum sample added to the dataset.

`double mean()` – returns the mean value for the dataset.

`double deviation()` – returns the standard deviation of the dataset.

`double meanConfidence()` – returns mean confidence interval for the dataset.

`void reset()` – resets the statistical data.

`double variance()` – returns variance of the samples added to the statistics.

`Statistics getStatistics()` – returns the statistics block of the dataset.

17.2 Collecting datasets for variables

AnyLogic supports collecting data on variable during the model simulation in a dataset. Samples of variables can be collected automatically from the beginning of the simulation.

► To collect a dataset for a variable automatically

1. Select variable on the structure diagram.
2. In the Properties window, select the *Auto collect dataset* check box.

If auto collecting is set, whenever you add a variable to a chart window, all the collected data on a variable from the beginning of simulation is displayed. If auto collecting data is not set, data collecting starts only when and if the variable is added to a chart window. See section 11.2.6, “Chart window” for more information on chart windows.

17.3 Visualizing collected data

AnyLogic supports visualizing the data collected during model execution. Datasets can be displayed with variety of options in chart windows or in an AnyLogic animation.

17.3.1 Visualizing collected data in Model Viewer

Data collected in a model can be displayed using AnyLogic chart windows. Data may be collected explicitly in datasets created by the user as well as implicitly – for example, AnyLogic automatically collects datasets for variables (see section 17.2, “Collecting datasets for variables” for details).

► To open a blank chart window

1. Click the *New Chart*  toolbar button, or
Choose *View|New Chart* from the main menu.

► To open a chart window with a particular dataset displayed

1. Double-click the dataset in the Model Explorer, or
Right-click the dataset in the Model Explorer and choose *Chart* from the popup menu.

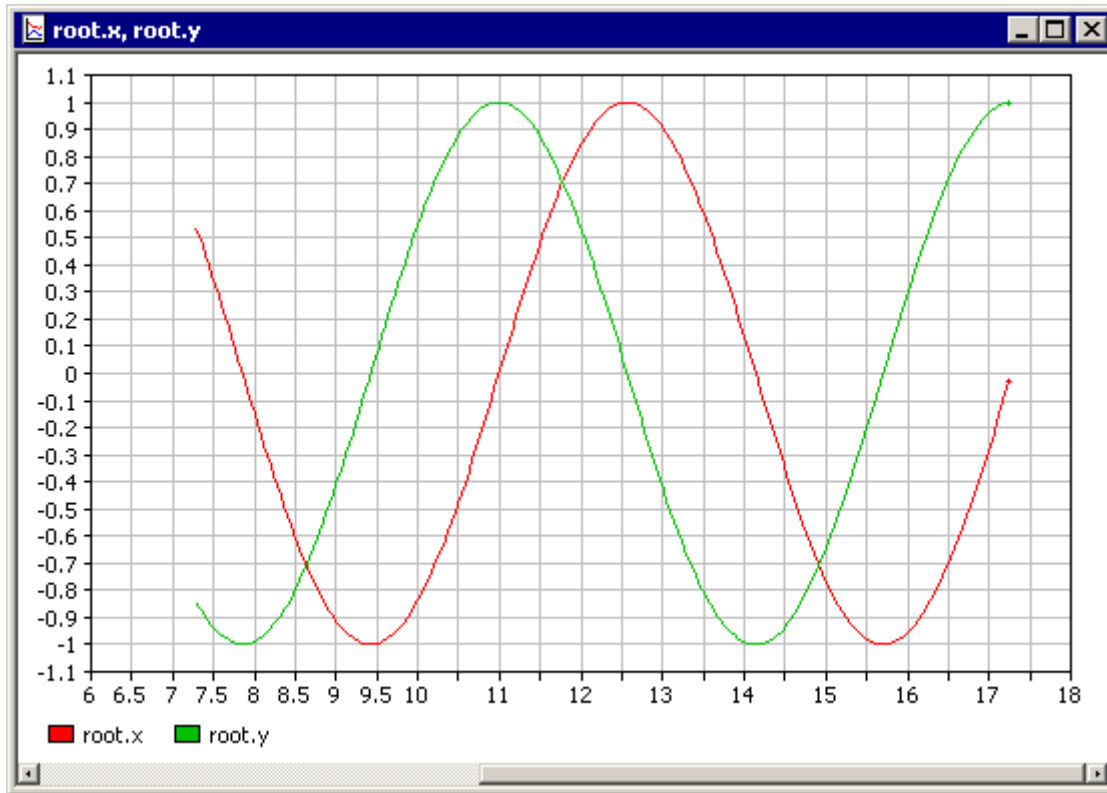


Figure 166. Chart window


Chart window looks as shown in Figure 166. Datasets can be added to a chart window using drag and drop or using the *Chart Setup* dialog box.

► **To add a dataset to a chart window**

1. Drag the dataset from the Model Explorer onto the chart window.

► **To set up the chart**

1. Right-click the chart window and choose *Chart Setup...* from the popup menu. The *Chart Setup* dialog box is displayed (see Figure 167).
2. To add a dataset to the chart, double-click the corresponding item in the *Variables, parameters and datasets* list.
3. To remove a dataset from the chart, double-click the corresponding item in the *Axis Y* list.

4. By default, *Time* is chosen in the *Axis X* list, that is the chart is timed. If you need to plot one dataset against another dataset, variable, or parameter, make the chart phased. Set up the variable/dataset/parameter to be displayed on the x-axis by clicking the corresponding item in the *Variables, parameters and datasets* list, and then clicking the  button to the left of the *Axis X* list. To make plot timed again, remove the variable/dataset/parameter item from the *Axis Y* list by double-clicking.
5. Click *OK*.

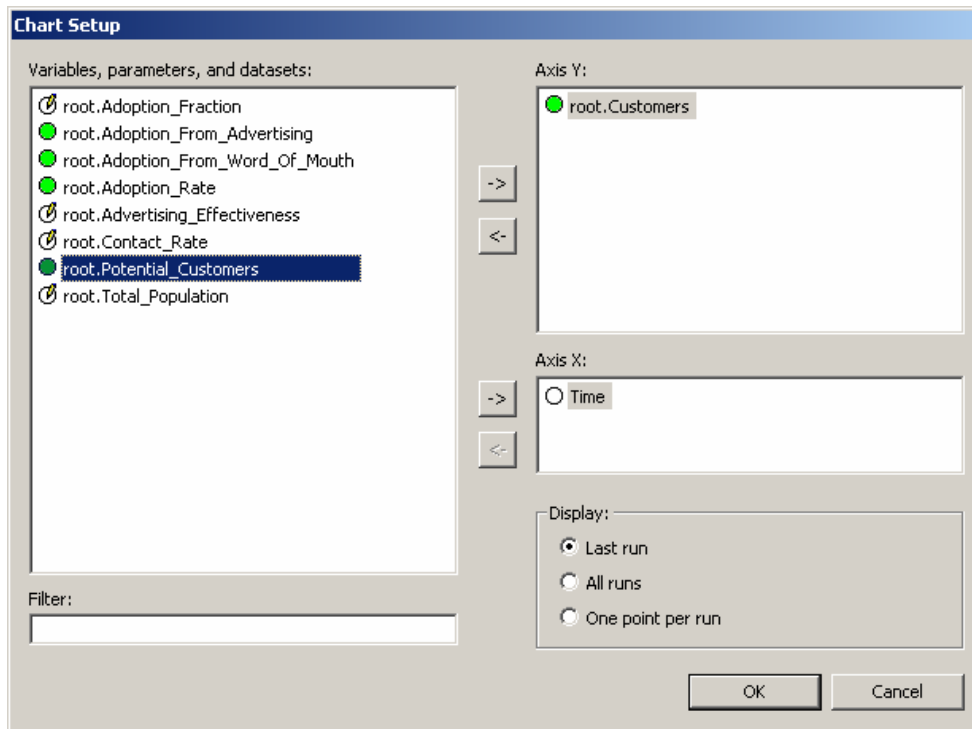


Figure 167. Chart Setup dialog box

17.3.1.1 Specifying chart window options

A chart window has large set of options. To specify options of a chart window, you use the *Chart Options* dialog box.

► **To specify options of a chart window**

1. Double-click the chart window, or
Right-click the chart window and choose *Chart Options* from the popup menu.
The *Chart Options* dialog box is displayed, as shown in Figure 168.
2. Modify options using the *General*, *Axes*, and *Datasets* pages (description of pages is given below)
3. Click *OK* to close the dialog box and apply changes.

General page

The *General* page of the *Chart Options* dialog box is shown in Figure 168. It is used to specify the chart type and some presentation options.

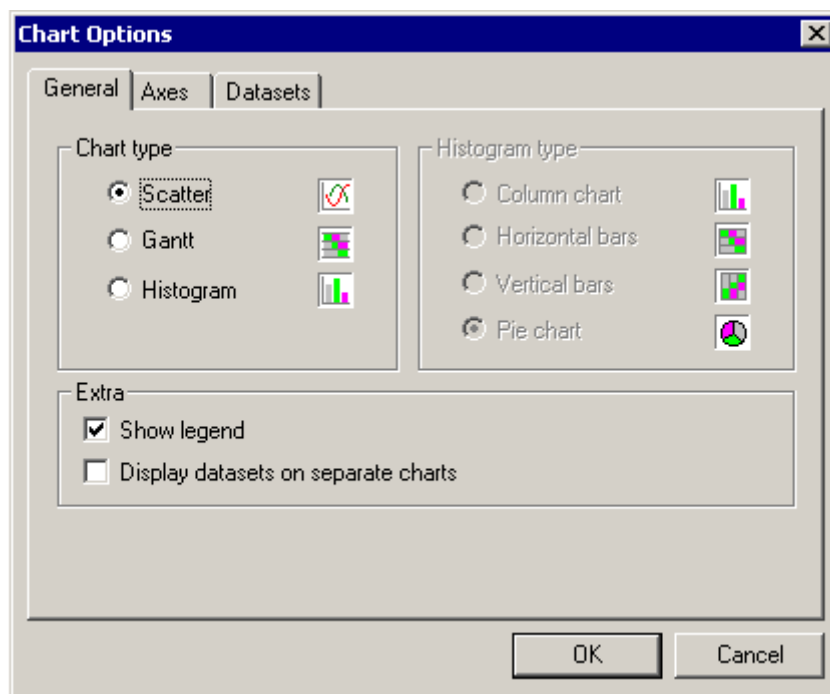


Figure 168. Chart Options dialog box. General page

Chart types are *Scatter*, *Gantt*, and *Histogram*.

Scatter is a conventional 2D chart with x- and y-axes.

Gantt chart is a stripe along the X-axis where y-value is represented by the stripe color. The color is taken from the interval hit by y-value. If a dataset has no intervals, the default color range is from white to the default line color.

Histogram is a set of bars showing the number of hits of each interval by y-value (non-timed histogram), or the percentage of time during which the value was staying within each interval (timed histogram), relative to other intervals. Histograms do not display individual data points. If the dataset has no intervals, no meaningful histogram can be displayed for it. A histogram can be of four types: column chart, horizontal bars, vertical bars, or pie chart.

► **To specify chart type**

1. Choose *Scatter*, *Gantt*, or *Histogram* option.

► **To specify histogram type (if chart type is set to histogram)**

1. Choose *Column chart*, *Horizontal bars*, *Vertical bars*, or *Pie chart* option.

► **To show/hide legend**

1. Select/clear the *Show legend* check box.

► **To display datasets on separate charts/share one chart**

1. Select/clear the *Display datasets on separate charts* check box.

Axes page

The *Axes* page of the *Chart Options* dialog box looks as shown in Figure 169. It is used to specify various properties of chart axes.

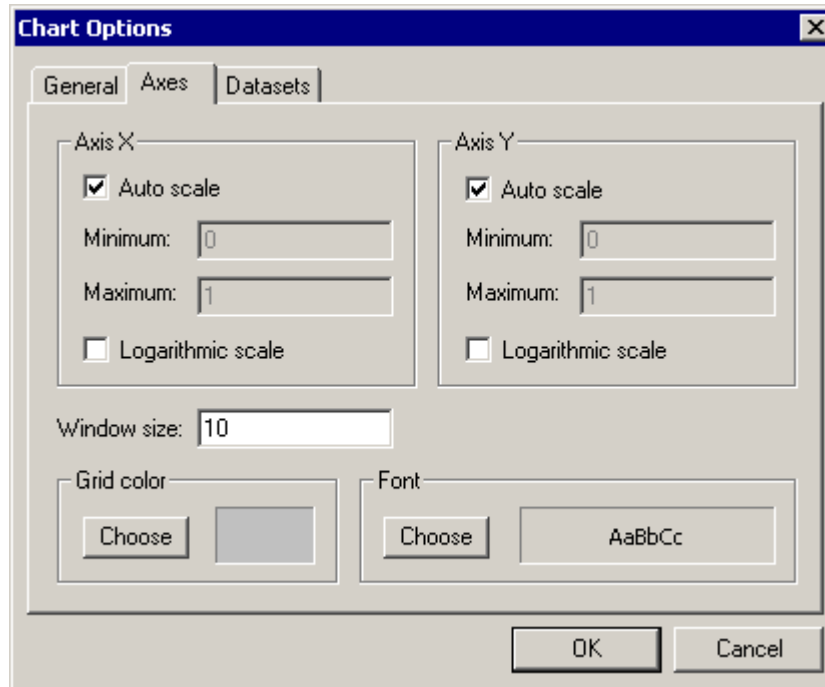


Figure 169. Chart Options dialog box. Axes page

Controls on this page have intuitive meaning, and only the *Window size* edit box needs to be discussed. This edit box determines the interval of time (for timed datasets) or the number of points (for non-timed datasets) displayed in the window.

Datasets page

The *Datasets* page of the *Chart Options* dialog box looks as shown in Figure 170. The *Datasets* page is used to specify individual options for each dataset. Also it is used to turn a chart window into a phase diagram.

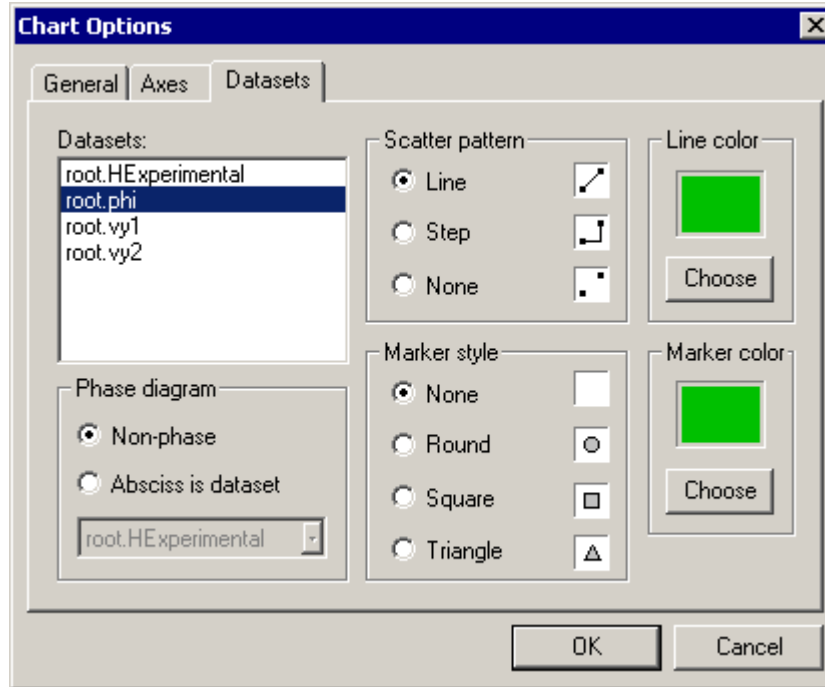


Figure 170. Chart Options dialog box. Datasets page

► **To select a dataset you wish to customize**

1. Click a dataset or a variable in the *Datasets* list box.

The following settings apply to a selected dataset.

► **To specify scatter pattern**

1. Select appropriate pattern in the *Scatter pattern* group.

► **To specify marker style**

1. Select appropriate style in the *Marker style* group.

► **To specify the color of a line**

1. Click *Choose* button in the *Line color* group box.
The *Color* dialog box is displayed.

2. Select color using the *Color* dialog box.

► **To specify the color of a marker**

1. Click *Choose* button in the *Marker color* group box.
The *Color* dialog box is displayed.
2. Select color using the *Color* dialog box.

Each dataset is a set of value pairs $\langle x,y \rangle$. When displaying several datasets in one window, you can either map x-values of all datasets to the X-axis, or you can specify that a y-value of some dataset should be mapped to X-axis, and y-values of other datasets should be mapped to the Y-axis. The latter case is called a phase diagram, and it is a sort of a scatter.

► **To let all datasets map their x-values to the X-axis**

1. Select the *Non-phase* option in the *Phase diagram* group box.


► **To turn the chart window into a phase diagram**

1. Select the *Absciss is dataset* option in the *Phase diagram* group box.
2. Choose the dataset whose y-values should be mapped to the X-axis.

17.3.2 Visualizing collected data in AnyLogic animation

Dataset plots can also be displayed in an AnyLogic animation using a special control – a chart indicator. A chart indicator displays a dataset in one of the following forms: scatter, Gantt, pie chart, or bar chart. See section 12.2.3.3, “Chart indicator” for the detailed description of chart indicator.

► **To create a chart indicator**

1. Click the *Chart Indicator*  toolbar button, or
Choose *Draw | Animation | Chart Indicator* from the main menu.
2. Click or drag the indicator area on the animation diagram.

3. In the Properties window, specify the dataset in the *Value to indicate* combo box.

17.4 Exporting statistical data to other applications

Data presented on a chart window can be copied on the Clipboard in the textual form (tab-separated) or exported to other applications, e.g., MS Excel. You can also get a graphical image of a window on the Clipboard.

► To copy datasets on the Clipboard in textual form

1. Right-click the chart window and choose *Copy Data* from the popup menu.

Because “x-values” of datasets displayed in the same chart window may not be identical, in the textual form each dataset forms an individual table with two columns. The left column contains “x-values” of that particular dataset (time in case of timed dataset, sample numbers in case of non-timed datasets, some user-specified values in case of phase dataset), and the right column contains the “y-values” of that dataset. The tables are placed one below the other and x- and y-values are tab-separated.

► To copy datasets on the Clipboard and paste them to MS Excel

1. Right-click the chart window and choose *Copy to Excel* from the popup menu.

► To copy the image of a chart window on the Clipboard

1. Right-click the chart window and choose *Copy Image* from the popup menu.

18. Standalone model running

AnyLogic gives you the unique ability to run simulation models standalone:

- From the command line on any Java-enabled platform,
- As applets in Web browsers.

18.1 Running a model from the command line

You can run AnyLogic model from command line on any Java-enabled machine. In this case, the model displays animation windows if you have developed them. However, you are not able to use AnyLogic windows to view and control the model.

First, you should specify that model files, specifically `.class` files, should be generated to the dedicated folder.

► To generate `.class` files to the dedicated folder

1. In the Project window, click the project (the top most item in the workspace tree).
2. In the Properties window, specify the folder path in the *Folder for generated files* edit box.

Once you specify the folder to store generated model files, build the model, set the folder specified as the current folder and enter the following shell command:

```
<java> -classpath <xjanylogic5engine.jar>;<mylib1.jar>;...;<mylibn.jar>; <root>
<options>
```

Where:

<java> – Java interpreter also known as Java virtual machine (for example, `java.exe`). If the operating system does not know a path to this application, you must specify the full path manually.

<xjanylogic5engine.jar> – file `xjanylogic5engine.jar` distributed as a part of AnyLogic.

- <mylib1.jar>;...;<mylibn.jar> – list of Java libraries used by the model. The list may include your own libraries, third-party libraries, and libraries coming with AnyLogic.
- <root> – full name of the class of the root object of the model. The class name must include the name of the package where the class can be found.
- <options> – combination of options listed below:
- debug – tells the model to output debug information.
 - norun – tells the model to stop after initialization. If omitted, the model starts running right after initialization.
 - seed <num> – value used to seed the pseudorandom-number generator. If 0, the model chooses the seed randomly using the system timer.
 - <name>=<value> – sets the parameter <name> of the root object to the value <value>.


18.2 Running a model as an applet

AnyLogic gives you the unique ability to make your models available online over the Internet. As long as an AnyLogic model is a 100% Java application, it can be presented as an applet and viewed in HTML browsers remotely. The simulation engine (including discrete part and numerical methods), the compiled model, and the animation will be transferred to the client machine and run there – with full degree of interactivity.

With respect to download time, the size of the engine is less than 500KB, which is small enough for one-time download at reasonable connection speeds.

Please note that multiple model applets can share the same engine file, so it is actually downloaded to the client machine only once – while the first model is accessed. For every next model, only the actual model code (normally of much smaller size) is downloaded. Check out how this works at <http://www.xjtek.com>.

► To generate an applet from the model

1. Make sure you have created animation for the root object of the model. This animation picture is displayed in the applet window.
2. Click the *Create Applet*  toolbar button, or Choose *Model | Create Applet* from the main menu.
The browser window opens with the applet displayed, as shown in Figure 171.

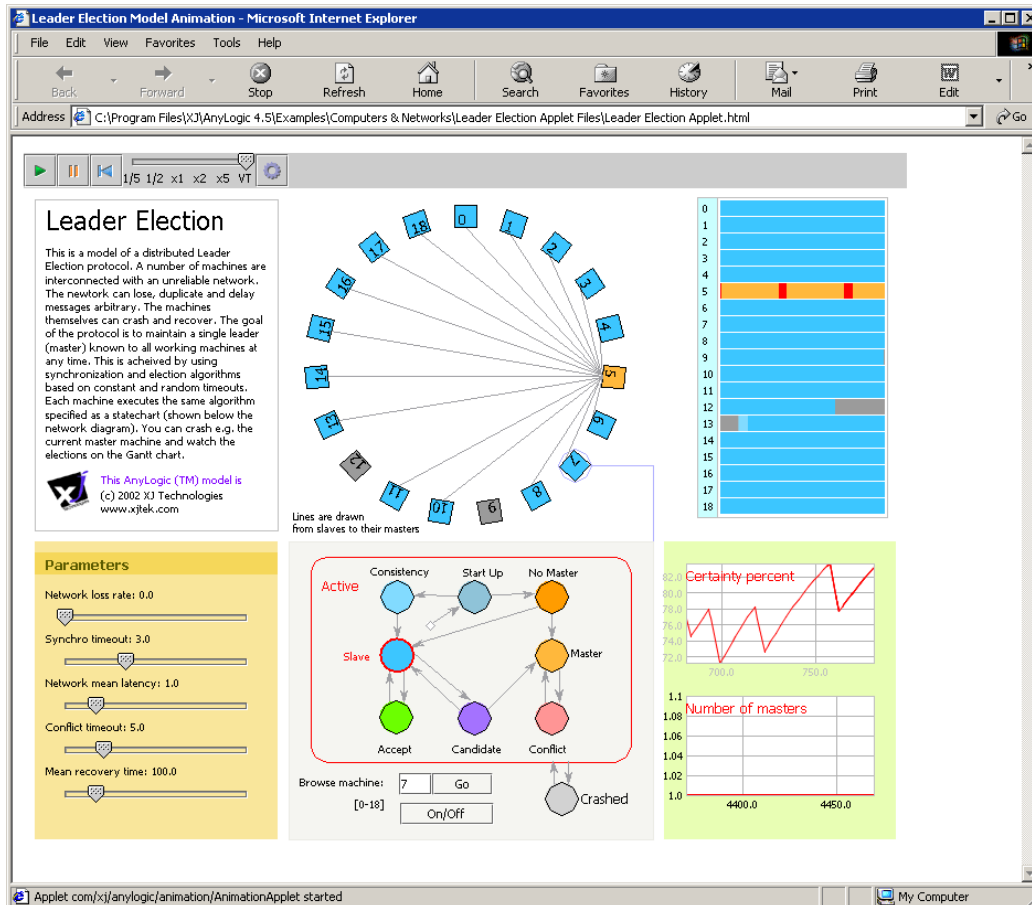


Figure 171. AnyLogic applet viewed in a Web browser

During this operation AnyLogic creates the folder “<project name> Applet Files” at the same location where the project resides. AnyLogic puts three files in this folder:

- xjanylogic5engine.jar - a copy of AnyLogic simulation engine.
- <project name>.jar - Java archive with the model code.

- `<project name> Applet.html` - a sample HTML file displaying the applet.

This is a self-sufficient set ready to be published on your Web site. To publish the applet, you would need to copy the first two files to the corresponding locations on your Web server (remember that you would need only one copy of the simulation engine for all model applets, because it can be shared), and to write your own HTML file using `<project name> Applet.html` as a template.

Each applet window contains a control bar (see Figure 171). It can be used to stop, run and restart the simulation, adjust the speed of model execution, and set up some appearance parameters.

18.2.1 Configuring your Web browser to view Java applets

Some Web browsers have built-in capability of displaying Java applets; some require Java plug-in to be installed. The installation is done only once, before you view the first applet. Normally, the browser automatically detects whether the plug-in needs to be installed and offers to download and install the plug-in for your particular platform and browser version. If this does not happen, and you are unable to view Java applets, please consult your browser/OS manufacturer.

18.3 Controlling the model simulation

When AnyLogic simulation model is run either from a command line or as an applet, only animation and 3D animation windows are displayed if you have developed them. In this case, you can control the model simulation and configure animation using the toolbar, displayed in the top of AnyLogic animation and 3D animation windows (see Figure 172).

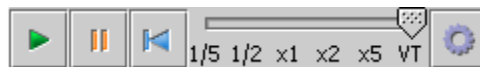


Figure 172. Animation window toolbar

Namely, you can run, pause and reset the model, and set up model simulation speed, animation update rate and toggle the on/off anti-aliasing option. All these settings apply both to AnyLogic animation and 3D animation windows.

18.3.1 Controlling the model simulation

From the animation control bar, you can start, pause or reset the model run.

► To run the model

1. Click the *Run*  toolbar button.

► To pause the model

1. Click the *Pause*  toolbar button.

► To reset the model

1. Click the *Reset*  toolbar button.

18.3.2 Setting up model speed

You may need to map the AnyLogic model time to the real time. It is frequently needed when you have developed some animation and want it to appear as in real life. Mapping of AnyLogic model time and the real time is described in section 13.1, “Simulation speed”.

You can set up either real time or virtual time mode. In real time mode, the mapping of model time units to seconds is made – i.e., you define that one model time unit takes the specified number of seconds. This mode can be used to make animation appear as in real life.

In the real time mode, you can increase or decrease model speed by changing the model simulation speed scale. You can change the model speed scale using the *Model Speed* toolbar slider. The default x1 scale means that the model is simulated with the model simulation

speed defined in the properties of the current AnyLogic experiment; x2 means that model is run twice faster than the specified model speed, etc. For instance, if the model speed is 6 model time units per second, x2 means that 12 model time units correspond to 1 second. You can change the model simulation speed as you like.

Also, you can run model in virtual time mode – the model runs at its maximum speed and no mapping is made between model time unit and seconds of astronomical time.

► **To set virtual time mode**

1. Move the *Model Speed* slider to the *VT* value.

18.3.3 Configuring animation


You can configure animation rendering options; namely, specify animation update rate and enable or disable anti-aliasing.

18.3.3.1 Setting up animation update rate

AnyLogic enables you to set up the animation update rate. The greater update rate you specify, the smoother animation will appear. However, animation rendering takes some period of time and frequent animation update will slow the model simulation. So, choose between smooth animation and fast simulation and set up the animation update rate according to your needs.

You can explicitly specify fixed update rate in frames per second. Alternatively, you can specify adaptive update rate. Adaptive update rate will be recalculated during the model simulation to fix up the specified ratio between simulation speed and animation smoothness.

► **To set up the animation update rate**

1. Click the *Settings*  toolbar button.
The *Settings* dialog box is displayed, as shown in Figure 173.
2. To define the fixed update rate, choose the *Fixed* option and explicitly specify the update rate in frames per second with a slider.

3. Otherwise, to define adaptive update rate, choose the *Adaptive* option and specify the update rate with a slider. Choose between smooth animation (*Smooth*) and fast simulation (*Fast*).
4. Click the *Apply* button to apply changes.
The animation will appear with the specified update rate.
5. If needed, repeat steps 2-4 to specify another update rate and click *OK* when finished.

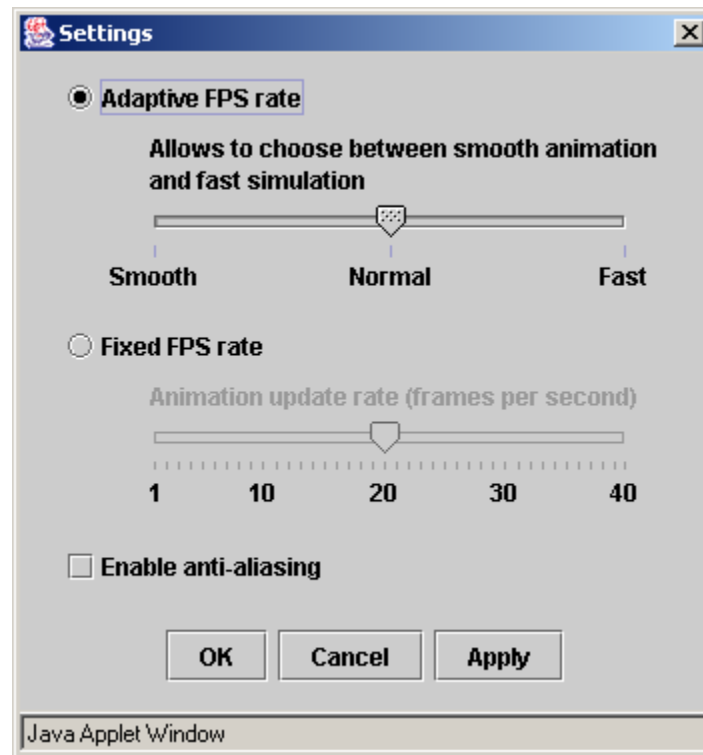



Figure 173. Settings dialog box

18.3.3.2 Animation anti-aliasing

AnyLogic animation supports anti-aliasing – one of the most important techniques in making graphics more smooth and pleasing to the eye. With anti-aliasing set, graphics look smoother due to the performed approximation of the colors of rendered pixels. However, note that more time is spent on rendering the animation with the anti-aliasing set.

► **To enable/disable anti-aliasing**

1. Click the *Animation Settings*  toolbar button.
The *Animation Settings* dialog box is displayed, see Figure 173.
2. Select/clear the *Enable anti-aliasing* checkbox.
3. Click *OK*.

19. Libraries and external files

19.1 External files

AnyLogic allows the user to add external Java files to a project. This capability is used to add code of any kind to a model. AnyLogic does not modify external files during the code generation. Just before the model is built, all external files are copied into build directories with respect to their packages names. When AnyLogic builds the model, external files are compiled together with the code generated by AnyLogic.

► To add an external file to a package

1. Choose *Insert | Add External File...* from the main menu.
The *Add External File* dialog box is displayed.
Choose the package, which will contain the external file, and click *OK*.
2. Alternatively, in the Project window, right-click the package and choose *Add External File...* from the popup menu.
3. The *Open* dialog box is displayed.
4. Browse for the existing Java file, or type the name of the new file you want to add to the package.
5. Click the *Open* button.

External files reside in AnyLogic packages, as well as active object classes. If the external file is added to a package, it must be logically placed inside Java package having the same name. For example, if the external file is added to the AnyLogic package `mypackage`, it must contain the line `package mypackage;`. If a new file is created when you add an external file to a package, this line is added automatically.

Properties

Name – [read only] the name of the external file.

Path name – [read-only] the full path the external file is loaded from.

Persist as – [read-only] shows how the file is saved in the project. This field can contain a relative or an absolute path. If you move the project to another folder, the information of this field will be used to obtain the full path and load the file.

Exclude from build – if set, the file is excluded from the model, i.e., it is not compiled.

19.2 Libraries

Libraries are collections of active object classes, animations, timer classes, message classes, and Java modules developed for some particular application area or modeling task. Libraries have several benefits:

- Provide for better reuse of classes across multiple models. A class can be developed and stored once and referenced from several projects.
- Libraries enable you to organize teamwork in AnyLogic projects: a part of the model developed by a team member may be put into a library, and others use consistent versions of the library in their work.
- By developing the right library, you can convert AnyLogic into a high-level modeling tool with point-and-click interface for a specific domain.

19.2.1 Creating a library

A library is actually a project, which is compiled and packed into a Java archive. Any project can be made a library if you specify its property *Target file* and build it.

► To create a library

1. Open a project which you would like to make a library, or create a new project.
2. Click the project in the Project window.
3. Type the name of the Java archive (e.g. `MyLibrary.jar`) in the *Target file* property of the project.

You do not need to specify the root object class for a library. However, if you specify it, you can run the library having that object as a root. This might be useful for testing while developing a library.

If you modify a library being used in a project, do not forget to rebuild the library (you need a separate instance of AnyLogic to open it). Note that when a project using a library is being compiled, the library itself is not compiled.

We recommend you to take special note of the future reuse of the class you wish to place in a library. Add comprehensive descriptions to the library classes. Keep them simple, flexible, and independent one from another as much as possible.

AnyLogic standard distribution includes several libraries, covering specific domains. They are: Enterprise Library, Dynamic Systems Library, Material Flow Library. AnyLogic libraries are located in the `Lib` directory. Have a look at these libraries to get an idea of how to develop your own ones.

19.2.2 Working with libraries

AnyLogic shows libraries in the Libraries window (a page in the Workspace window, see Figure 174). Each AnyLogic library is represented with an individual stencil containing library object classes. Each class is represented with the icon designed for it. The Libraries window does not allow you to modify anything within libraries.

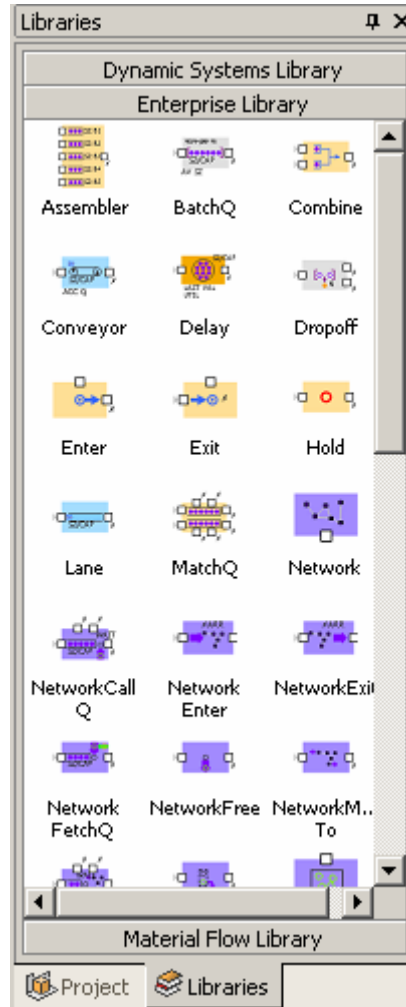



Figure 174. Libraries window

► To show the Libraries window

1. Click the *Libraries*  toolbar button, or
Choose *View|Libraries* from the main menu, or
Press Alt+1.

If you have created a library and wish to use it in other projects, you must tell AnyLogic where the library is located. Only libraries located in folders, specified in AnyLogic options, are displayed in the Libraries window. To use your own or third-party library, you should list

the folder containing the library in AnyLogic options or copy the library into the folder C:\Program Files\AnyLogic 5.0\Lib, where the standard libraries are located.

► **To add a library folder to AnyLogic options**

1. Choose *Tools | Options...* from the main menu.
The *Options* dialog box is displayed.
2. On the *Miscellaneous* page, type the name of the folder in the *Library folders* edit box.
Use a semicolon to separate your folder from other folders.
3. Click *OK*.

If you create a model in a specific domain, which is covered by a library, you usually reuse active object classes defined in a library just by dragging them on a structure diagram and connecting them.

► **To create an instance of an active object class defined in a library**

1. Drag an active object class from the Libraries window onto a structure diagram.
The instance of the active object class is created.

Moreover, you can pass a message defined in a library between your custom model objects. Or you can use a message defined in a library as a base class or as a parameter of your custom message (all message classes defined in AnyLogic libraries are listed in the *Base class* and the *Type* message class' properties correspondingly).

20. Database Support

Sometimes you need your model to access databases. For example, you may want to construct and connect active objects based on information stored in a database. AnyLogic allows you to link a database to your model.

You need this when:

- You have values of active object parameters in a database already;
- There are too much data to encode them in a model;
- A database is shared between your model and other applications.

To access a database, you have to code it manually using Java database connectivity technology – JDBC. JDBC documentation is included into Java SDK documentation available at:

<http://java.sun.com/docs>

There is also a basic JDBC tutorial at:

<http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html>

In addition, AnyLogic Engine provides `com.xj.anylogic.DataSource` class that wraps JDBC and simplifies access to databases. See AnyLogic Class Reference for description of this class.

20.1 Introduction

What is a database? A database is a structured storage of information. What does this mean? Consider that most ubiquitous of databases – the phone book. The phone book contains several items of information – name, address, and phone number – about each phone subscriber in a particular area. Each subscriber's information takes the same form.

In the database terms, the phone book is a *table* that contains a record for each subscriber. Each subscriber record contains three fields: name, address, and phone number. Other

examples of databases are: customer lists, library catalogs, parts inventories. The list of possible databases is infinite. Using a database management system such as Microsoft Access you can design your own database.

Information in a database table is held in *records*. A record contains information about a single entity – for example, a person, a product. Records are represented as *rows* in tables. For every record, the data is held in *fields*. The fields contain individual pieces of data about the record – for instance, name, telephone number. A table usually has one or more fields uniquely identifying a record in the table. Such a field is called the *key field*. The key field provides the means to distinguish one record from all the others in a table. It allows one to identify, locate, and refer to one particular record in the table.

A database can contain a single table of information, such as the phone book, or several tables.

A database can contain *queries*. A query is a filter used to obtain specific information from a database. The result of a query is a *view* – virtual table that contains records matching the query. Once a view is defined, it can be used as an ordinary table.

If you need views (virtual tables) other than those available in a database itself, you create them directly in AnyLogic as described in section 20.4, “Custom queries”.

20.2 Creating a data source

A data source is an element of an AnyLogic project that represents an actual database and is responsible for communication with the database. By creating a data source, you declare that your model will be able to access a corresponding database. That is, you cannot access a database unless you have created a data source representing that database.

► To create a data source

1. Choose *Insert|New Data Source...* from the main menu.
The *New Data Source* dialog box is displayed.
Specify the name of the new data source, choose the package, which will contain the data source, and click *OK*.

2. Alternatively, in the Project window, right-click the package, which will contain the data source, and choose *New Data Source...* from the popup menu. The *New Data Source* dialog box is displayed. Specify the name of the new data source and click *OK*.

The name is used to identify and access the data source. An AnyLogic data source name is not related to either a database filename or ODBC data source name.

After a data source is created, it is necessary to associate it with a database. You can associate it either with a database file (see subsection 20.2.1, “Associating with a database file”) or with an ODBC data source (see subsection 20.2.2, “Associating with an ODBC data source”).

20.2.1 Associating with a database file

You use association with a database file to access, e.g., an MS Access database or MS Excel spreadsheet. This approach is simple and does not require you to have anything else than one of the mentioned tools. Its drawback is that it requires storing an absolute file path in an AnyLogic project. Therefore, if you change the location of a database, you have to change a corresponding path manually in AnyLogic project.

► To associate a data source with a database file

1. In the Project window, click the data source.
2. In the Properties window, select the *File* option.
3. Click the *Browse* button. The *Open* dialog box is displayed.
4. Browse for the MS Access or MS Excel file you want to use. Double-click the file or click the *Open* button to select the file.
5. Enter *Login* and *Password* if they are required by the database.

20.2.2 Associating with an ODBC data source

The other way to connect a data source to a database is to associate it with an ODBC data source. This technique uses the so-called ODBC drivers and makes your model independent of the database type and location.

To associate an AnyLogic data source with an ODBC data source, you must have the latter defined in the system. You define ODBC data sources using the ODBC Data Source Administrator tool. You need ODBC installed on your computer; otherwise, this tool is not available.

Each ODBC data source has unique name – a DSN. Note that the name of an AnyLogic data source is not related to DSN.

► To create a new DSN

1. Start ODBC Data Source Administrator located, e.g., under the following windows menu path: *Start | Settings | Control Panel | Administrative Tools | Data Sources (ODBC)*.
2. Activate the *User DSN* page if you want to create a user DSN. Activate the *System DSN* page if you want to create a system DSN. A system DSN is visible to all applications started by any user, whereas a user DSN may be accessed only by applications started by the user that created that DSN.
3. Click the *Add* button.
4. Choose the driver.
5. Click the *Finish* button.
This opens the dialog box that varies depending on the driver selected. However, some settings are common and they are described below. You can also define options specific for the driver.
6. Type data source name in *Data Source Name* edit box. Do not enter an existing DSN name; otherwise that data source will be overwritten.
7. Type *Description*, if necessary.
8. Specify path to the database if it is stored in a file, or choose the database name if it is stored on a server.

9. Click *OK*.

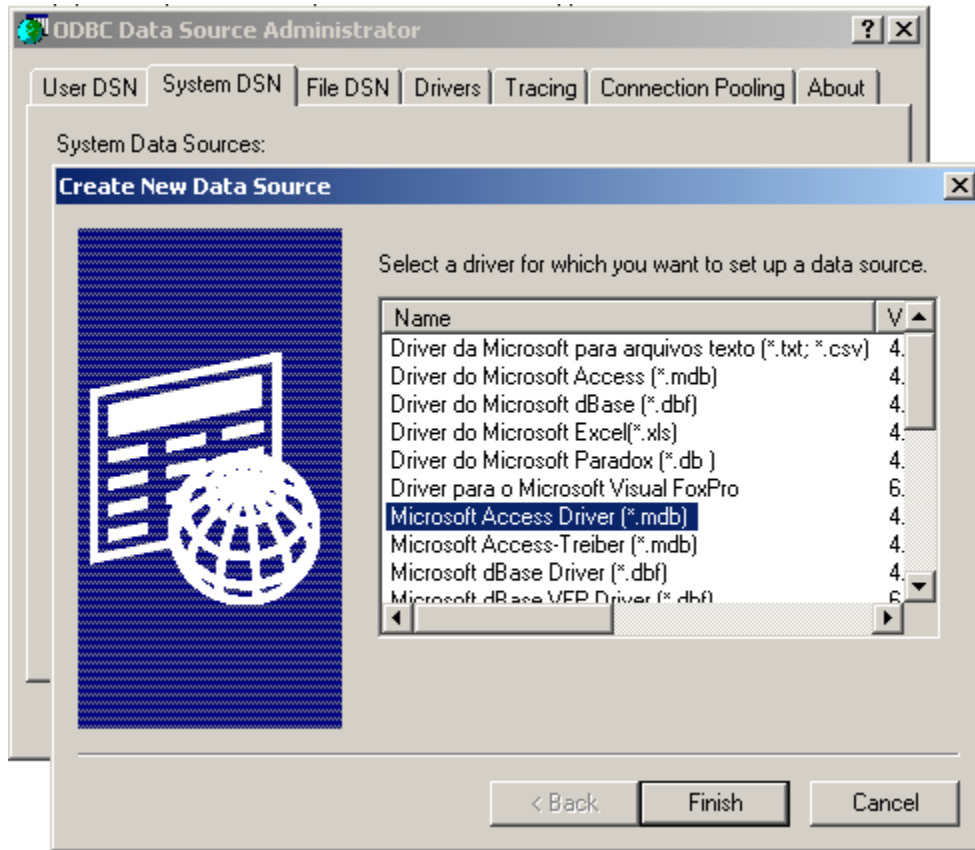


Figure 175. Selecting driver

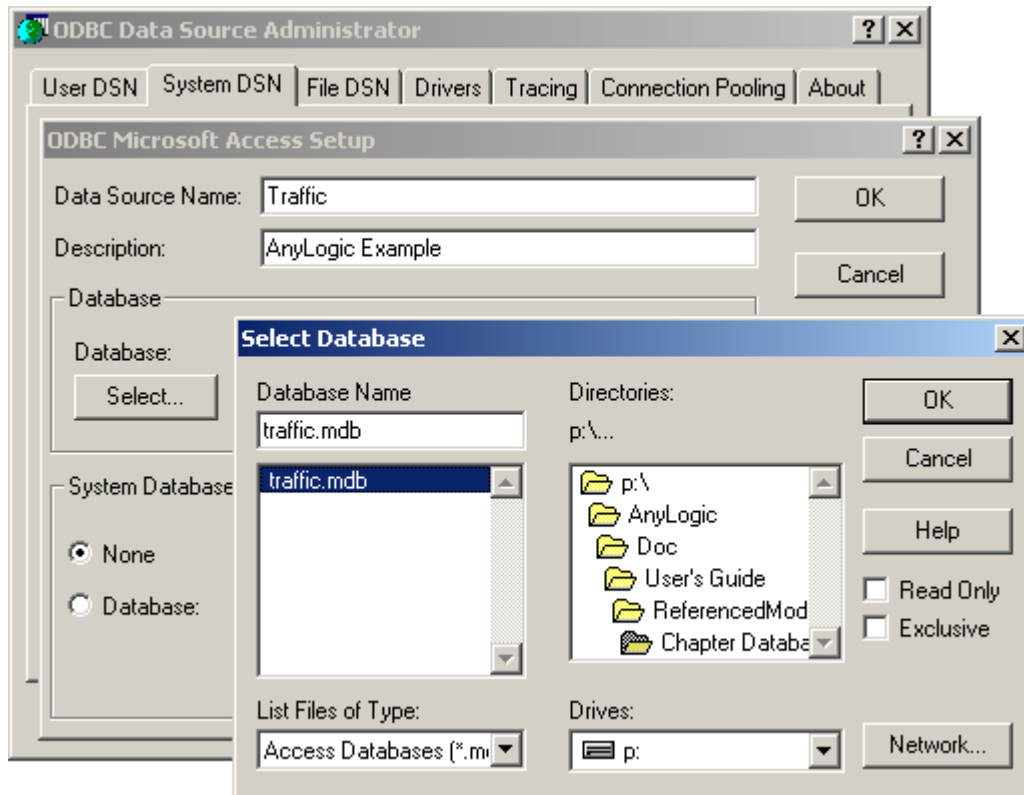


Figure 176. Selecting database file

To specify database file, click the *Select* button and browse for your database file. After closing the dialog boxes you see the newly created DSN in the list:

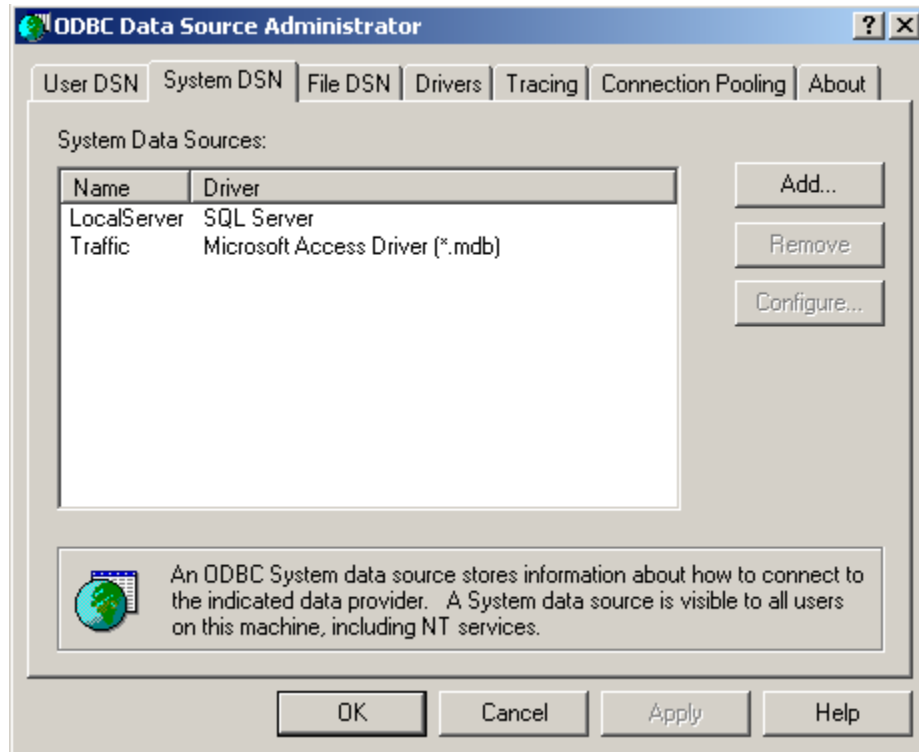


Figure 177. Newly created DSN

Once the ODBC data source is created, you can associate an AnyLogic data source with it.

► **To associate an AnyLogic data source with an ODBC data source**

1. In the Project window, click the data source.
2. In the Properties window, select the *ODBC Data Source* option and choose an ODBC data source from the drop-down list.
3. Enter *Login* and *Password* if they are required by the database.

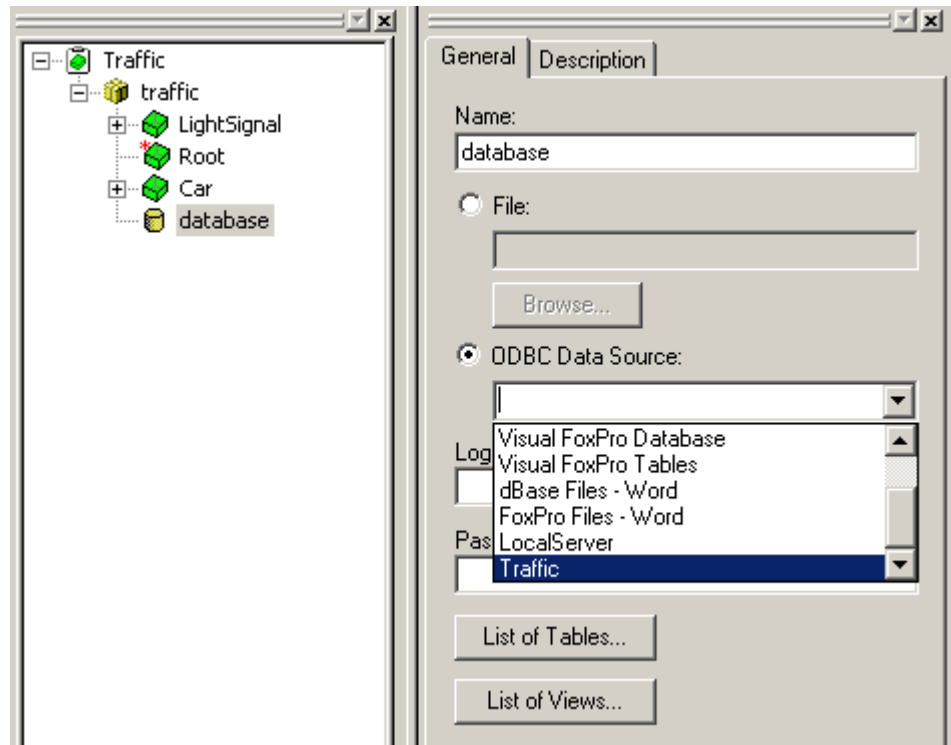


Figure 178. Associating an AnyLogic data source with an ODBC data source

20.2.3 Viewing data source content

You can examine tables and views available in a data source by clicking *List of tables* or *List of views* button in the data source's properties. Once you've pressed the button, you can see a dialog box with two list boxes. The upper one displays available tables/views. When you select a table/view in it, the lower list box shows table's/view's fields.

Using these dialog boxes, you check connection to the database. Tables, views, and their fields should match the ones defined in the database-authoring tool.

20.3 Working with data sources

For each defined data source, AnyLogic generates a new data source class derived from AnyLogic data source class `com.xj.anylogic.DataSource`. This class wraps JDBC and simplifies access to databases. To access a database, you have to code it manually using methods of this class (for more information please consult AnyLogic Class Reference).

Also to simplify access to data sources, each data source class has static methods automatically generated by AnyLogic. They are listed in Table 20.

Method	Description
<code>static boolean connect()</code>	The method connects a data source, specified by constructor parameters.
<code>static boolean disconnect()</code>	The method disconnects the data source.
<code>static java.sql.Connection getConnection()</code>	The method returns connection.
<code>static Integer getFieldType(String sTableName, String sFieldName)</code>	The method returns SQL-type of specified field.
<code>static Object getMatrix(String sSqlQuery, String sType)</code>	The method executes specified SQL query and returns the produced values as 2D array of values of specified type.
<code>static ResultSet getResultSet(String sSqlQuery)</code>	The method executes specified SQL query and returns the produced result as <code>java.sql.ResultSet</code> object.
<code>static ResultSet getQueryResultSet(String sQueryText, String sListOfFields, String sKeyField, String sKeyFieldValue)</code>	The method returns result set produced by querying data from specified fields and rows of specified query.

<pre>static ResultSet getTableResultSet(String sTableName, String sListOfFields, String sKeyField, String sKeyFieldValueString sSqlQuery)</pre>	The method returns result set produced by querying data from specified fields and rows of specified table.
<pre>static String getValue(String sSqlQuery)</pre>	The method executes specified SQL query and returns the produced value.
<pre>static boolean modify(String sSqlQuery)</pre>	The method executes specified SQL query.
<pre>static java.sql.Statement getStatement()</pre>	The method returns statement.
<pre>static java.util.Hashtable getRow(String sSqlQuery)</pre>	The method executes specified SQL query and returns the produced values.

Table 20. Methods of data source class generated by AnyLogic

20.4 Custom queries

If you need a view different from views available in a database, and you do not want to add that view to the database, you can create a query directly in AnyLogic.

► To create a new query

1. Choose *Insert | New Query...* from the main menu.
The *New Query* dialog box is displayed.
Specify the name of the new query, choose the data source and click *OK*.
2. Alternatively, in the Project window, right-click the data source and choose *New Query ...* from the popup menu.
The *New Query* dialog box is displayed.
Specify the name of the new query and click *OK*.

3. In the Properties window, enter the query code in the *SQL Query* edit box. This must be a `SELECT` statement written in SQL. Be sure that there is no semicolon at the end of the statement; otherwise it will be considered empty.
4. Click *Query Fields* button to open the list of query fields and see if the query is correct.

20.5 Tips and notes

If AnyLogic reports that it cannot connect to a database or you cannot find tables, views, fields, or data you expect to see, this indicates that connection to a database is faulty. You can do the following to fix this:

- Check that your data source is set up properly; that is, the path to a file or DSN name is correct.
- Check that the database file is not removed or moved to another folder.
- Check that an AnyLogic data source is not removed or renamed.

Note that usually a database-enabled model cannot run as an applet due to Internet security settings.

21. Customizing AnyLogic UI

AnyLogic allows you to customize the UI to make your work more convenient. Namely, you can customize AnyLogic menu, toolbars (create your own toolbars, control the layout of toolbars and toolbar buttons) and change the colors of AnyLogic structure diagrams elements.

21.1 Customizing toolbars and menus

AnyLogic toolbars and menus are customizable. You can create your own toolbars, delete toolbars, make them invisible, etc. The layout of buttons on toolbars can also be changed. You can customize the AnyLogic menu appearance also. To customize toolbars and menu appearance, you use the *Customize* dialog box.

► To open the Customize dialog box

1. Choose *Tools | Customize* from the main menu.
2. Click the *Toolbars* tab if you are going to create, delete, show, or hide toolbars. See section 21.1.1, “Toolbars page” for details.
3. Click the *Commands* tab if you are going to add or remove toolbar buttons. See section 21.1.2, “Commands page” for details.
4. Click the *Keyboard* tab if you are going to configure the shortcut keys for the commands. See section 21.1.3, “Keyboard page” for details.
5. Click the *Menu* tab if you are going to customize AnyLogic menu appearance. See section 21.1.4, “Menu page” for details.
6. Click the *Options* tab if you are going to customize the toolbar buttons appearance. See section 21.1.5, “Options page” for details.

21.1.1 Toolbars page

The *Toolbars* page (see Figure 179) enables you to create your own toolbars, delete toolbars and control their visibility.

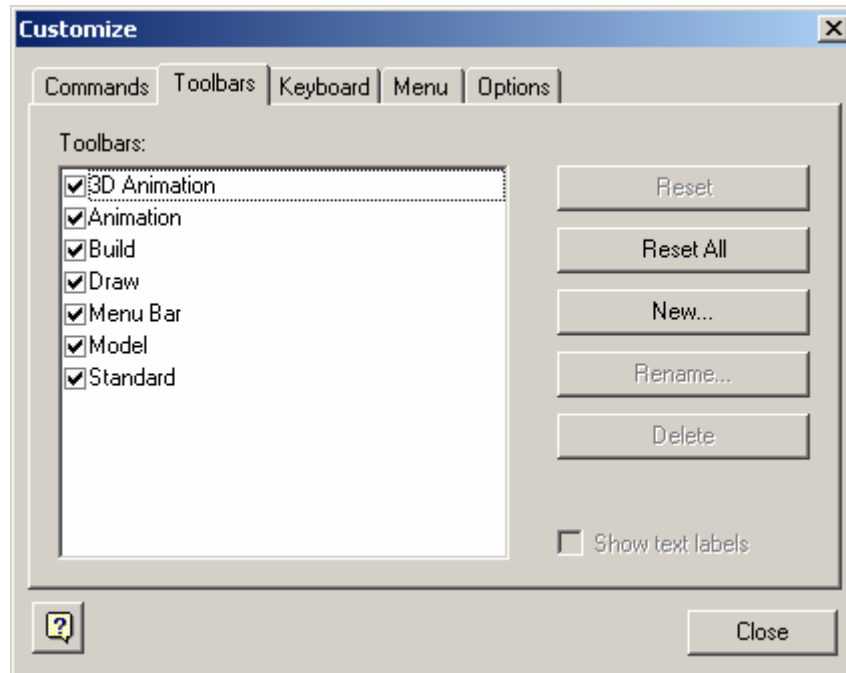


Figure 179. Customize dialog box. Toolbars page

► To show/hide a toolbar

1. Select the toolbar in the *Toolbars* list.
2. Select/clear the checkbox to the left of the toolbar name.

► To show/hide text labels under toolbar buttons

1. Select the toolbar in the *Toolbars* list.
2. Select/clear the *Show text labels* check box.

► To create a custom toolbar

1. Click *New* button.
The *Toolbar Name* dialog box is displayed.
2. Type the name of the new toolbar in the *Toolbar Name* edit box.
3. Click *OK*.

► To rename a custom toolbar

1. Select the toolbar in the *Toolbars* list.
2. Click the *Rename* button.

► To delete a custom toolbar

1. Select the toolbar in the *Toolbars* list.
2. Click the *Delete* button.

► To reset a toolbar to the default state

1. Select the toolbar in the *Toolbars* list.
2. Click the *Reset* button.

► To reset all toolbars to default states

1. Select the toolbar in the *Toolbars* list.
2. Click the *Reset All* button.

21.1.2 Commands page

The *Commands* page (see Figure 180) enables you to add, delete, and change the layout of toolbar buttons.

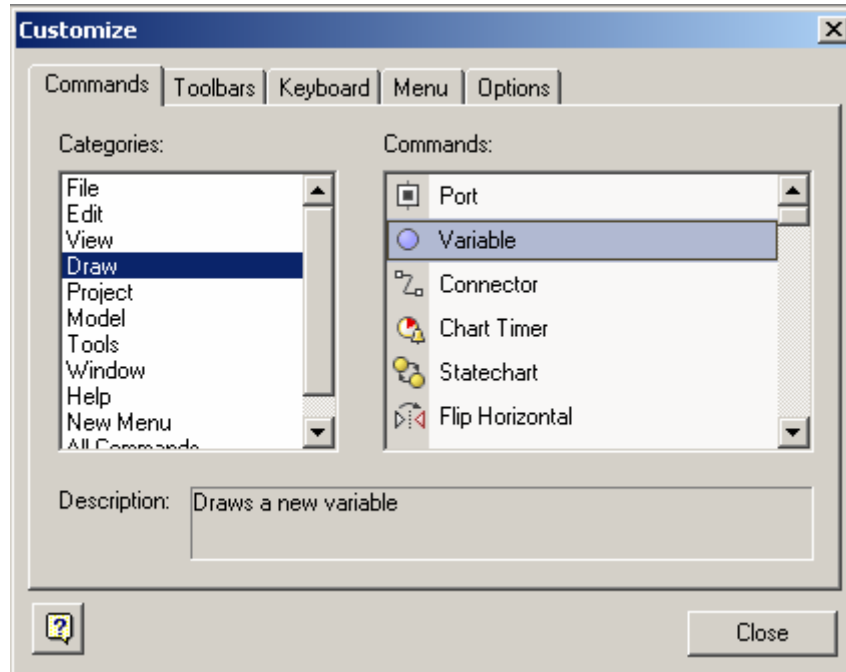


Figure 180. Customize dialog box. Commands page

► **To add a button to a toolbar**

1. Select the category of buttons in the *Categories* list.
2. Select the command in the *Commands* list.
The *Description* box displays the description of the selected command.
3. Drag the command button from the *Commands* box onto the toolbar.

► **To remove a button from a toolbar**

1. Drag the button from the toolbar onto the *Customize* dialog box.

► **To add or remove a separator to the left/right of a toolbar button**

1. Drag the button in the toolbar a little right/left.

21.1.3 Keyboard page

The *Keyboard* page (see Figure 181) enables you to change the shortcut keys for the commands.

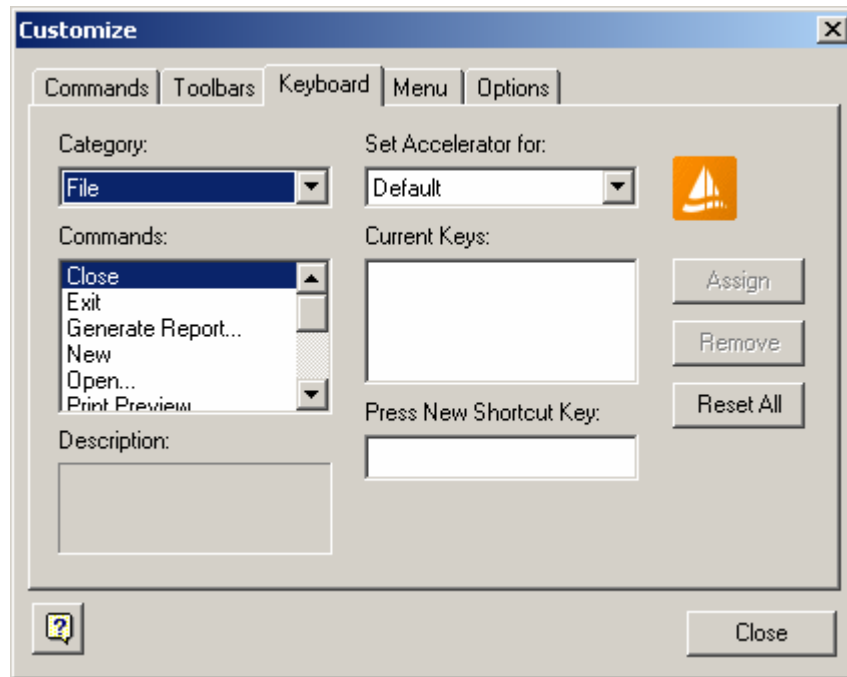


Figure 181. Customize dialog box. Keyboard page

► To set new shortcut key for a command

1. Select the category from the *Category* drop-down list.
2. Select the command from the *Command* drop-down list.
The *Description* box displays the description of the selected command.
The *Current Keys* list displays the current shortcut keys for the selected command.
3. Click inside the *Press New Shortcut Key* box and press the combination of keys that you want to make shortcut keys.
4. Click the *Assign* button.
5. Click the *Close* button.

You cannot assign the combination of keys to the command if it is already assigned to other command. The line below the *Press New Shortcut Key* box displays, whether this key combination is assigned to any other command or not.

► **To remove command shortcut key**

1. Select the category from the *Category* drop-down list.
2. Select the command from the *Command* drop-down list.
The *Current Keys* list displays the current shortcut keys for the selected command.
3. In the *Current Keys* list, select the shortcut key you want to remove.
4. Click the *Remove* button.
5. Click the *Close* button.

► **To reset all shortcut keys to default**

1. Click the *Reset All* button.

21.1.4 Menu page

The *Menu* page (see Figure 182) enables you to customize the AnyLogic menu appearance; namely, to choose how to display menus and how to animate opening menus.

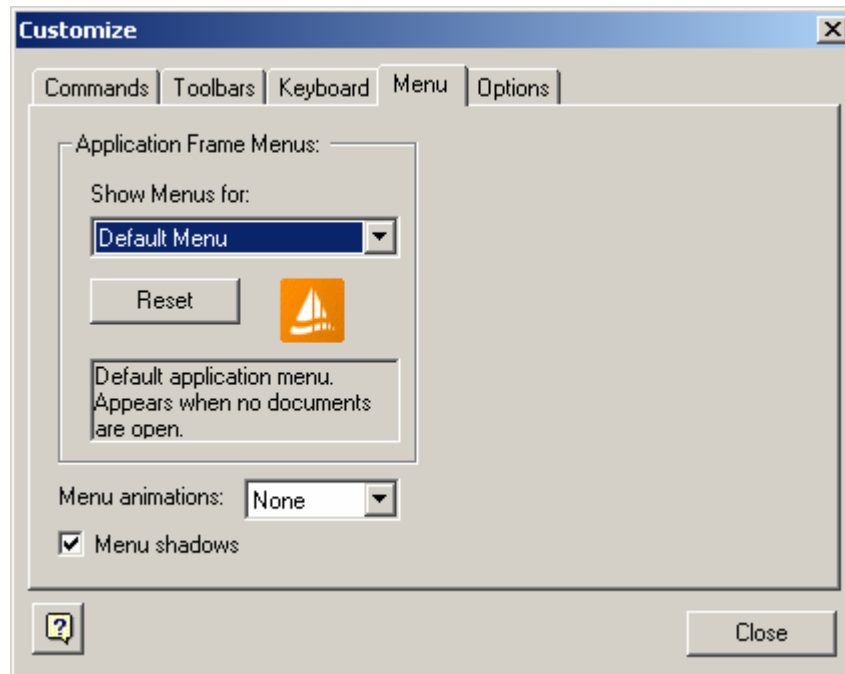


Figure 182. Customize dialog box. Menu page

► To choose the animation effect for opening menus

1. Choose the animation effect from the *Menu animations* drop-down list. You may choose from *None*, *Unfold*, *Slide*, *Fade*, where *None* means no effect applied, *Unfold* – that menus are animated unfolded, *Slide* – that menus are animated sliding down, *Fade* – that menus are animated fading in.

► To display menus with/without shadows

1. Select/clear the *Menu shadows* check box.

21.1.5 Options page

The *Options* page (see Figure 183) enables you to customize the toolbar buttons appearance, namely to choose whether to display tooltips or not (with shortcut keys or not) and whether to display buttons with standard or large icons.

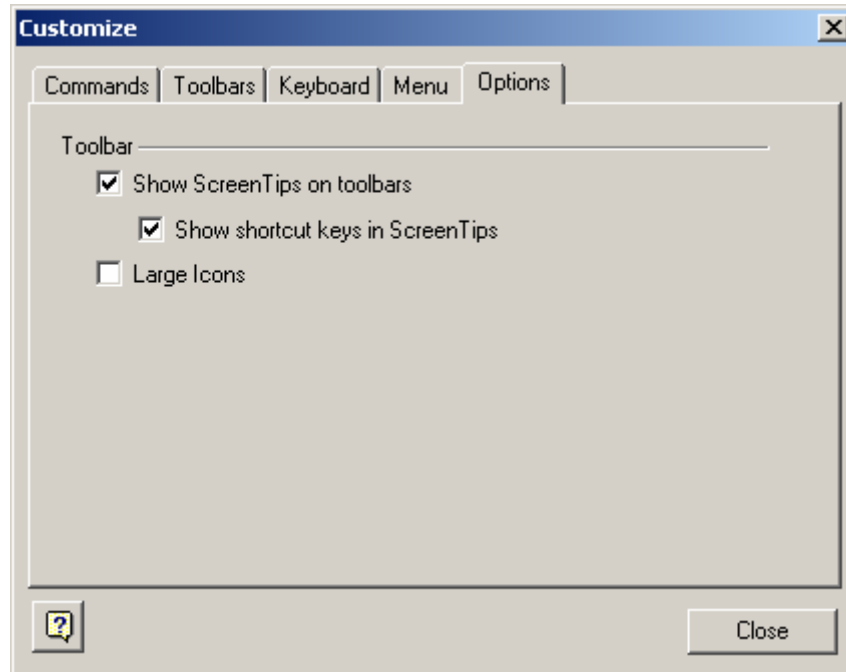


Figure 183. Customize dialog box. Options page

► **To show/hide tooltips for toolbar buttons**

1. Select/clear the *Show ScreenTips on toolbars* check box.

► **To show/hide shortcut keys in the tooltips**

1. Select/clear the *Show shortcut keys in ScreenTips* check box.

► **To display buttons with large/standard icons**

1. Select/clear the *Large Icons* check box.

21.2 Customizing colors

Colors of the model elements on AnyLogic diagram editor windows are customizable. To customize colors, you use the *Colors* page of the *Options* dialog box.

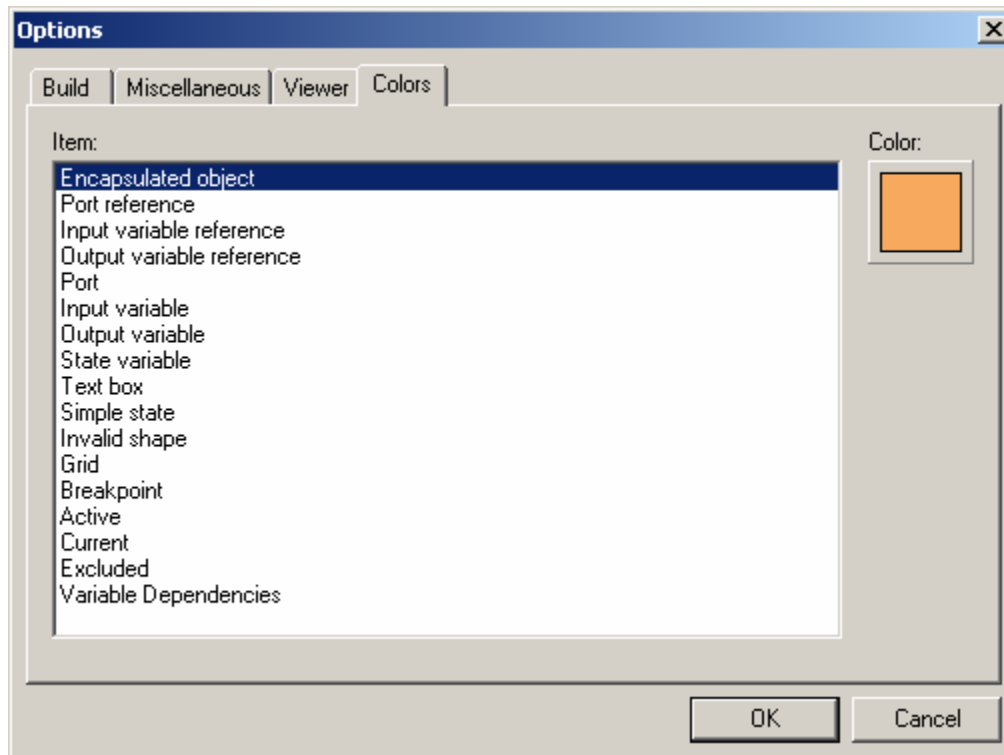


Figure 184. Options dialog box. Colors page

► **To set the color of an item**

1. Choose *Tools | Options...* from the main menu.
The *Options* dialog box is displayed.
2. Click the *Colors* tab.
3. Select the item in the list.
The *Color* button has the color of the currently selected item.
4. Click the *Color* button.
The *Color* dialog box is displayed.
5. Choose color you want to set.
6. Click *OK*.

22. Project building. Model initialization and termination order

The information presented in this chapter might be useful for better understanding of how an AnyLogic project is built (see section 22.1, “Building a project”) and how a model initializes and terminates in terms of ordering of various operations (see section 22.2, “Model initialization and termination order”).

22.1 Building a project

22.1.1 Project building stages

AnyLogic builds the project in three stages: code generation, compilation, and archiving (the latter is optional).

On the first stage, AnyLogic generates Java code for all classes defined in the project. The code is put into `.java` files. Files are placed into temporary folders whose structure mimics the structure of packages of the project. External files included into the project are also copied to those folders. You can examine the generated code each time you want to see how the visual representation of the model is mapped into source code. To specify the folder the generated files should be stored in instead of temporary folders, use the *Folder for generated files* property of the project. To set the project property, select the project item in the workspace tree.

On the second stage, a Java compiler is invoked. The compiler produces `.class` files containing bytecode to be executed on Java virtual machine. The `.class` files are placed into the same folder as `.java` files. The execution of the byte-code is the execution of the model.

In case the *Target file* property of the project is specified, the third stage – archiving – takes place. On the third stage, AnyLogic builds an archive by packing bytecode into the file specified in the *Target file* property of the project. For that purpose, a packager is used. The resulting file contains the complete bytecode of the model and usually has `.jar` (Java

archive) extension. Note that in most cases you do not have to generate an archive. The rationale for creating an archive is that it is smaller than unpacked bytecode. So, if you put the model on a Web site as an applet, the archive is preferable. However, if the project being developed is a library, it must be packed into an archive; otherwise it will be unusable.

You can customize the process of compiling and archiving the model. You can tell AnyLogic which Java compiler and packager to use and specify compiler options (see section 22.1.3, “Project building options”).

The files involved in the process of building depend on each other. The example of dependencies is shown in Figure 185.

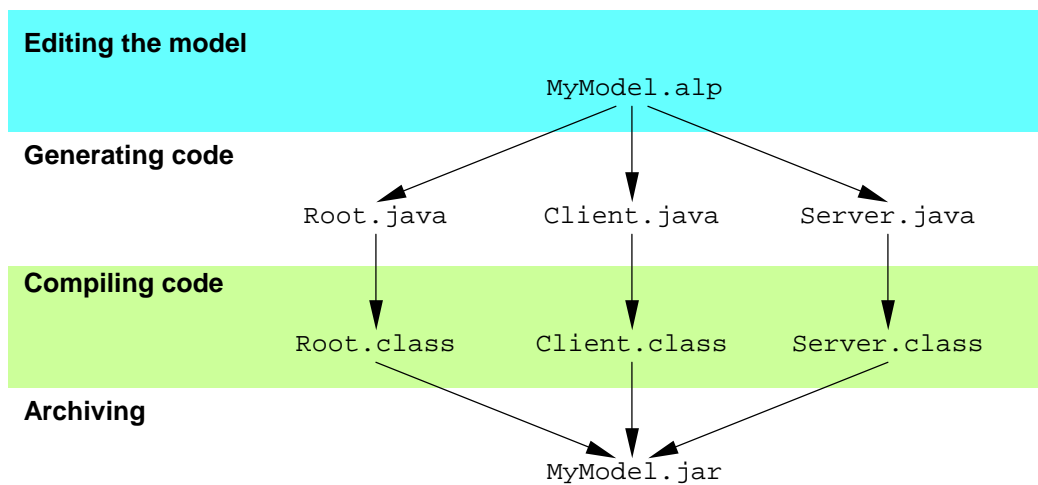



Figure 185. Example of file dependencies

A file is considered out-of-date if it is older than any of the files it depends on. To build the project, you either bring it up to date by building out-of-date and missing files, or you rebuild the whole project, starting with code generation for all classes.

22.1.2 Building a project


► To bring the project up to date

1. Click the *Build*  toolbar button, or
Choose *Model|Build* from the main menu, or
Press F7.


► To regenerate code for all classes and rebuild the project

1. Click the *Rebuild*  toolbar button, or
Choose *Model|Rebuild* from the main menu.

► To bring the project up to date and create the model (without starting it)

1. Click the *Step*  toolbar button, or
Choose *Model|Step* from the main menu, or
Press F10.

► To bring the project up to date and run the model

1. Click the *Run*  toolbar button, or
Choose *Model|Run* from the main menu, or
Press F5.

22.1.3 Project building options

Project building options are set on the *Build* page of the *Options* dialog box, see Figure 186.

► To open the Build page of the Options dialog box

1. Choose *Tools|Options...* from the main menu.
The *Options* dialog box is displayed.
2. Click the *Build* tab.

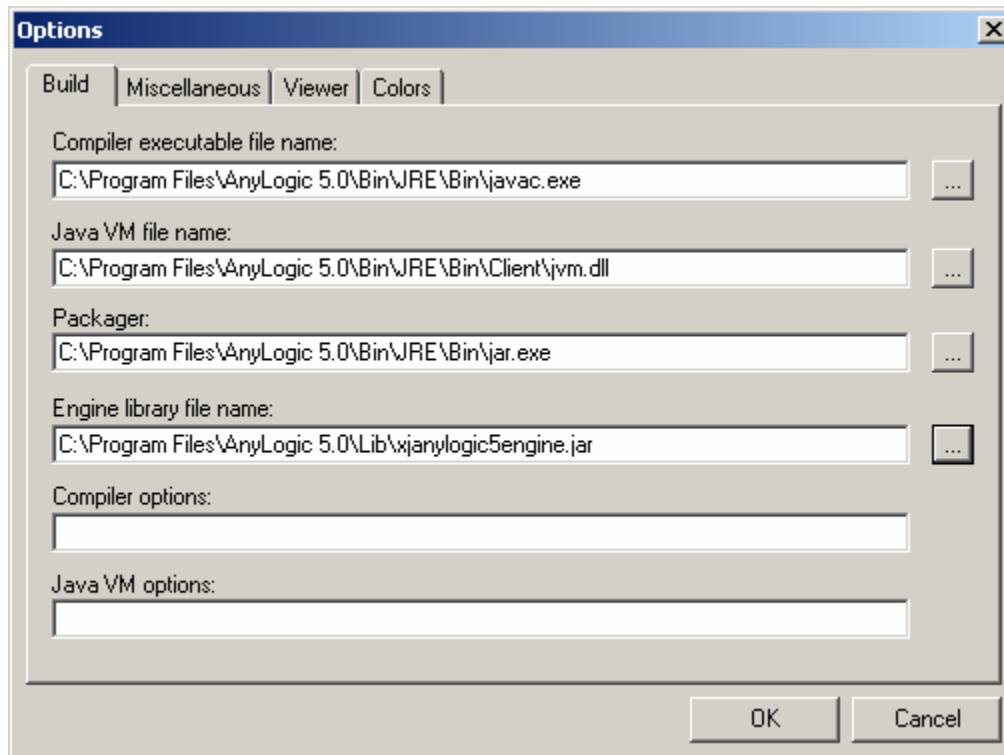


Figure 186. Options dialog box. Build page

On the *Build* page of the *Options* dialog box, you specify paths to the Java compiler, Java virtual machine and packager used and optionally some compiler and Java VM options.

Compiler executable file name - the full path to Java compiler,

e.g. `C:\j2sdk1.4.0\bin\javac.exe`.

Java VM file name – the full path to Java virtual machine dynamic link library

(e.g. `C:\j2sdk1.4.0\jre\bin\client\jvm.dll`).

Packager – the full path to Java packager

(e.g. `C:\j2sdk1.4.0\bin\jar.exe`).

Engine library file name – the full path to AnyLogic simulation engine library

(e.g. `C:\Program Files\XJ\AnyLogic5.0\Lib\xjanylogic5engine.jar`).

Compiler options – [optional] options passed to Java compiler. If JDK compiler is used, compiler options are usually not needed.

Java VM options – [optional] options passed to Java virtual machine (e.g. “-Xmx80M” sets Java heap size to 80MB). If JDK compiler is used, virtual machine options are usually not needed.

22.2 Model initialization and termination order

The information presented in this section might be useful for better understanding of how a model initializes and terminates in terms of ordering of various operations.

22.2.1 Model initialization order

The order of operations during initialization is the following:

1. Time is set to 0
2. The root object of the model is constructed, its parameters are being set, and its method `create()` is called. The method `create()` of each active object does the following steps for each encapsulated object:
3. Constructs the encapsulated object
4. Sets up the encapsulated object parameters
5. Calls its method `create()`
6. Calls its animation method `setup()`, if any

At this point creation of the model and the animation is completed.

7. The method `startup()` for each object is called. The method is called on the innermost objects first and then up along the active object tree. During the method `startup()` each object creates and initializes its statecharts (each statechart enters its initial state) and timers
8. The global equation set is evaluated for the first time
9. The animation method `update()` is called for the first time.

22.2.2 Model termination order

The model destruction is performed in the following order:

1. The model destruction starts at the root object, where the method `destroy()` is called. For each active object the method `destroy()` performs the following steps:
2. The user-defined method `cleanup()` is called
3. All activities (statecharts, timers, etc) are stopped
4. The method `destroy()` is called for all encapsulated objects recursively
5. All active object elements (ports, variables, activities, animation, etc) are deleted
6. The method `onDestroy()` is called for the active object.

Since the method `cleanup()` is called prior to the recursion, it is first called for root object and then for all object down the tree. On the other hand, the method `onDestroy()` is called after the recursion, therefore it is first called on leaf objects and then up the tree.

23. Threads

You can implement an activity in a Java method and run it in a separate thread within an active object. Threads are less visual than statecharts or timers, but in rare cases, they may provide for more natural representation of some algorithms. In general, however, if the activity has a set of states with different reactions to events, make it a statechart.

If there are no synchronization operations in the behavior (it is a pure computation), then you can call it from other places; e.g., from ports or from actions of states and transitions, and you do not need to run it in a separate thread.

To run an activity implemented in a Java method, you need to:

- Define method of an active object class with the return type `void`. This method will run as thread. The method can be defined e.g. in the code section *Additional class code*.
- Call method `startThread()` of the active object supplying the name of the defined method as a string.

The thread runs concurrently with all other activities (statecharts, timers). If it does not wait for anything, it terminates taking zero model time.

AnyLogic provides the following API to work with synchronization and time (for more information please consult AnyLogic Class Reference):

Related methods of `ActiveObject`

`void startThread(String methodName)` – creates a new simple thread based on the method name of this `ActiveObject`.

`void delay(double timeout)` – suspends the thread for the specified amount of time.

Related methods of `PortQueuing`

`Object waitForMessage()` – suspends the thread until there is a message in the port queue. Extracts the message and returns it.

Object `waitForMessage(double timeout)` – same as above, but with a timeout. On timeout returns null.

Example

In the example shown in Figure 187 the thread performs an infinite loop: it waits for the message arrival, then makes 10 time units delay.

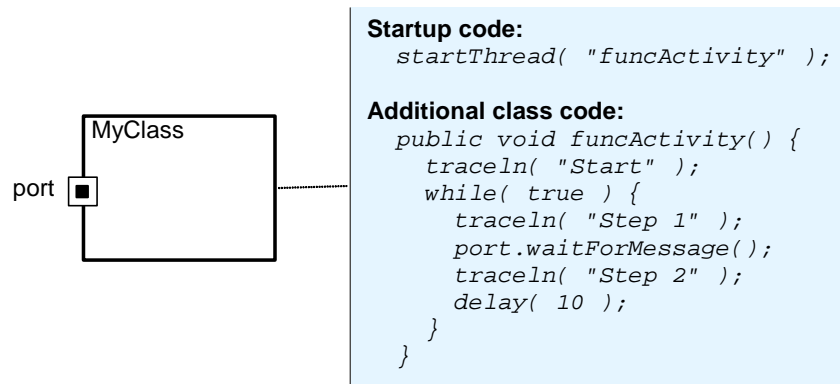


Figure 187. Thread