

SB-MASE

subsumption-based multi agent simulation environment
version 1.2



INSTITUTE OF KNOWLEDGE & AGENT
TECHNOLOGIES MAASTRICHT, NL

User Manual

Steven de Jong

Benjamin Torben-Nielsen





Institute of Knowledge & Agent Technologies (IKAT)



Universiteit Maastricht, the Netherlands

First edition

08 September 2005




Table of contents

Table of contents	2
Manual conventions.....	5
Background knowledge.....	5
System requirements and startup	5
1 General concepts	6
2 The user interface	7
3 Creating agents	8
3.1 Creating a new agent.....	8
3.2 Creating sensors and actuators.....	8
3.2.1 Creating and using sensors 	8
3.2.2 Creating and using actuators.....	10
3.3 Creating custom processes 	11
3.3.1 The process structure	11
3.3.2 SB-MASE data types.....	12
3.3.3 Specifying custom process behavior	12
Assignment of values to variables	13
Mathematical expressions	13
Lists: the FOR loop, ADD statement and CLEAR statement	13
Conditional programming	14
The PRINT statement	14
The STOP statement	14
3.4 Creating a process hierarchy.....	14
3.4.1 Creating connections.....	14
3.4.2 Using suppressors and inhibitors  	15



3.5	Creating built-in processes	17
3.5.1	The Monitor process 	17
3.5.2	The Factory process 	18
4	Creating simulator worlds.....	19
4.1	Introduction	19
4.2	Creating the canvas.....	19
4.3	Creating classes and objects	20
4.3.1	Creating classes	20
	Color (required)	20
	Shape (required)	20
	Fixed size (optional)	20
	Massive (optional)	21
	Movable (optional)	21
	Static (optional)	21
	Light (optional).....	21
	Flock (optional)	21
4.3.2	Creating object instances	21
	Name (optional).....	22
	Position (required)	22
	Rotation (optional)	22
	Size (optional).....	22
	Shape (only for flock classes)	22
	Items (only for flock classes).....	22
	Distribution (only for flock classes)	22
4.4	Placing agents in simulator worlds	22
4.5	Specifying settings	23
5	Creating external agents	26

5.1	Creating and editing an external agent	26
5.2	The run mode sensor	26
5.3	Loading the SB-MASE Server Control Panel	26
6	Advanced configuration and settings.....	27
6.1	SB-MASE.....	27
6.2	SB-MASE Server Control Panel.....	27
7	Developers' Guide.....	28
7.1	Developing plug-in processes	28
7.2	Developing agent models.....	28
7.2.1	Developing simulator agent models	28
7.2.2	Developing external or hybrid agent models.....	28

Manual conventions

-  Important knowledge items will be denoted with the symbol you see in front of this paragraph.
-  This manual is a practical one; usually, whenever a concept is explained, this is done by providing one or more practical examples from which the concept should become clear.
-  Interface commands in SB-MASE will be denoted as follows: [SB-MASE Tasks: Agents: New agent > Simulator agent > OK]. This reads as: under the interface part labeled *Agents* in the *SB-MASE Tasks* panel, first click on *New agent*, then click on *Simulator agent*, and finally click on *OK*.


Background knowledge

-  In this manual, we assume that you possess sufficient knowledge on the basics of artificial intelligence, subsumption architecture, embodied and situated agents, and the general concepts of computer programs. If this is not the case, you are advised to consult relevant literature or Google.
-  Furthermore, it is assumed that you are using a standard installment of the software. If this is not the case, please contact your administrator.

System requirements and startup

SB-MASE will run on any modern computer (i.e., built in 2003 or after). You need to have the Java runtime installed, preferably the latest version, which can be obtained on <http://www.java.com>.

For the Windows platform, a start script named `start.bat` can be found in the program's directory.

-  On all platforms, the program can be launched by changing to its directory, changing to the `classes` directory and entering `java raar.SBM`.

1 **General concepts**

SB-MASE, or the Subsumption-Based Multi Agent Simulation Environment, enables users to build control software for situated agents in both simulated and real-world environments. Development aims include, but are not limited to:

- High educative value combined with scientific usefulness;
- Intuitive user interfacing, customizable for the audience at hand;
- Limited programming skills required for typical users;
- Reliable and powerful agents;
- Fast engine, capable of running many agents at once;
- Control software development for simulated agents, embodied agents and agents that run in both modes;
- Extendable architecture, capable of dealing with many process- and agent architectures;
- Platform independence.

SB-MASE consists of a framework written in Java, with a carefully designed graphical user interface and straightforward, yet powerful programming languages for the development of feature-rich agents and simulator environments. External agents, such as robots or agents operating in specific simulated environments, can be controlled using a local or network connection. Convenient sensors and actuators allow situated agents to perceive and change their environment.

The basic control architecture used in SB-MASE is the subsumption architecture, which allows a task-based decomposition of agent control. Each task can then be modeled as a process, either a predefined process, a process written in SB-MASE's process programming language, or an external process such as a trained neural network. Processes are coupled to sensors, actuators and each other using a graphical representation.

2 The user interface

Below, we present a screenshot of the main user interface components of SB-MASE 1.2.

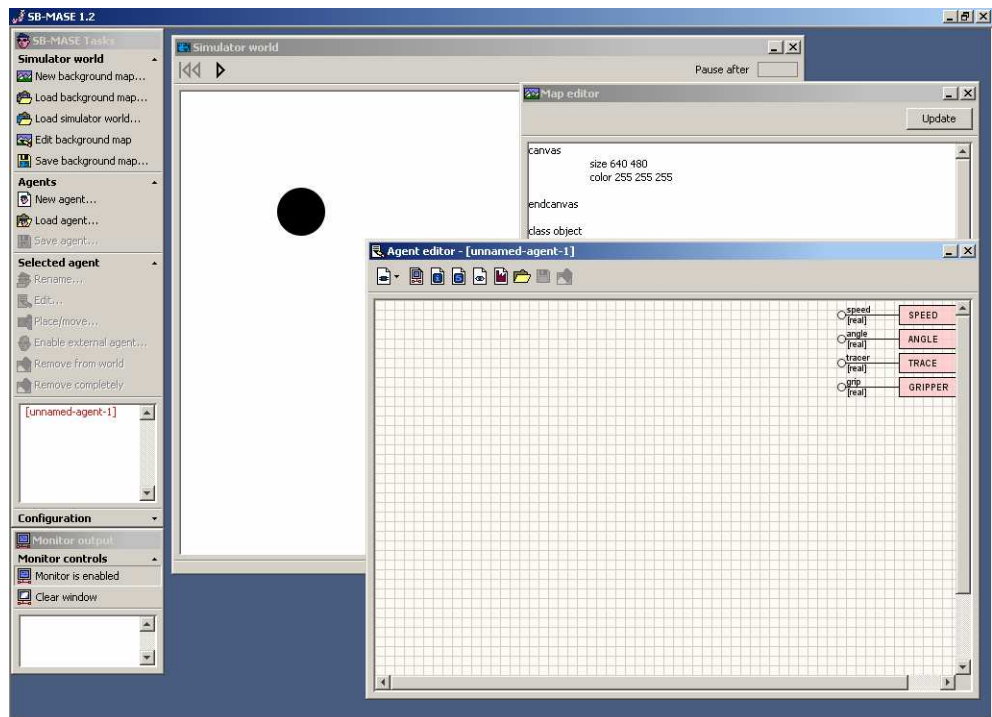


Figure 1 - Some parts of the SB-MASE user interface.

To the left, the *SB-MASE Tasks* panel provides the main controls of the program. Below, the *Monitor output* panel provides debug facilities. The *Simulator world* window presents a graphical representation of the current simulated environment. This environment can be edited using the *Map editor*. Finally, the *Agent editor* provides facilities for building agents.

3 Creating agents

3.1 Creating a new agent

To create an agent, select [SB-MASE Tasks: Agents: New agent...]. A new agent will be created and its initial configuration will be shown in the Agent editor.

- *If several agent models are present on your system, you will see a list of available models. Select the model of your choice and press [OK]. A new agent of this model will then be created.*
- *SB-MASE allows you to save agents to a persistent storage device, and to retrieve agents from such a storage device. To save an agent, select it in the list at the bottom of the SB-MASE Tasks panel, then click [SB-MASE Tasks: Agents: Save agent...] and save your agent. To load an agent, click [SB-MASE Tasks: Agents: Load agent...].*

Agents in SB-MASE consist of four parts, viz. (1) sensors with which the agents perceive the environment; (2) actuators with which they can actively modify this environment, for example by moving; (3) processes that define a transition of (optional) inputs to one or more outputs and finally (4) a coupling between sensors, actuators and processes. In the next sections, we will show how you can add these parts to agents, using the *Agent editor*.

3.2 Creating sensors and actuators

Depending on the type of agent you are editing, it is possible to add a variety of sensors and actuators to it, or to use the sensors and actuators that are present in the agent by default (for example because the agent emulates or controls certain hardware).

3.2.1 Creating and using sensors



Using the *Sensor Wizard*, you can create and configure sensors. Which models are available depends on the type of agent you are currently editing. In the screenshot below, we present the sensors available to the standard simulator agent, as presented by the *Sensor Wizard*.

Hover with your mouse cursor over the short explanation presented under the sensor name (i.e., *Gripper sensor*) to make a longer explanation appear next to the wizard. Pay careful attention to this explanation; it contains useful information on the functionality and output of the sensor. Some details will follow below. Select a sensor model and press [Finish]. Some sensors

require additional configuration. This will be explained later on in this section.

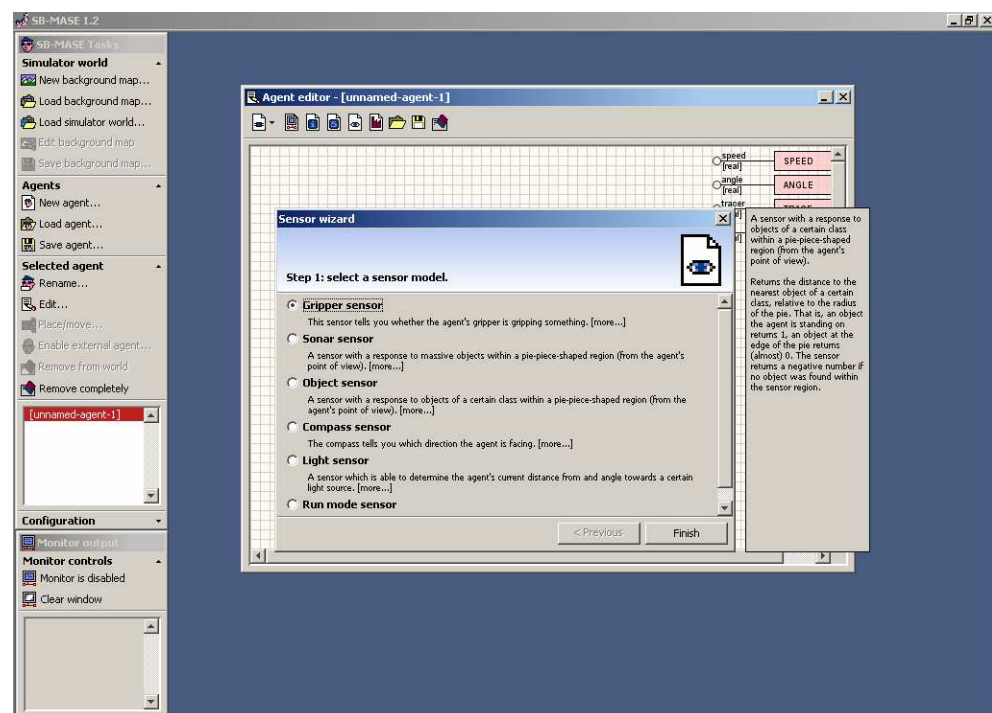


Figure 2 - The Sensor Wizard presents explanations on sensor models.

- The **grripper** sensor reports whether the agent is currently gripping something (output is 1) or not (output is 0).
- The **sonar** sensor reports the distance to the nearest massive object within the range of the sensor. Here, 0 stands for no object, +0 stands for the far most edge of this range, and 1 stands for an object directly in front of the agent.
- The **object** sensor reports the distance to the nearest object of a certain class (for example, islands). The values produced are similar to those of the sonar sensor. The object sensor is actually a high-level sensor; it is assumed that objects can be perfectly recognized. Obviously anyone able to develop such a sensor for real world situations will win a Nobel Prize.
- The **compass** sensor tells you which direction the agent is facing. The direction corresponding to 3 o'clock is 0 degrees and the degrees progress to 360 clockwise. That is, an agent facing North will have a compass reading of -90 degrees.
- The **light sensor** is able to determine the agent's current distance from and angle towards a certain light source. This enables agents to follow the gradient originating at this light source, for example. The sensor returns a type (see *SB-MASE data types*) with a real-valued distance (in pixels) and real-valued angle (relative to the agent's heading, from -1 to +1 for full left

and full right) in it, or NIL (no data) if the light source it should respond to was not found.

- The **run mode sensor** returns 1 if the agent is running in either external mode or simulator mode. Otherwise, it does not return anything. It is therefore very suitable for coupling to suppressors or inhibitors (see *Creating a process hierarchy*) that ensure different behavior in the simulator and external modes. Obviously, it does not have much to do in a purely simulated or external agent.

Some sensors require additional configuration. For example, for the sonar and object sensors, you must specify a range within which these sensors can perceive objects. An example is shown below.

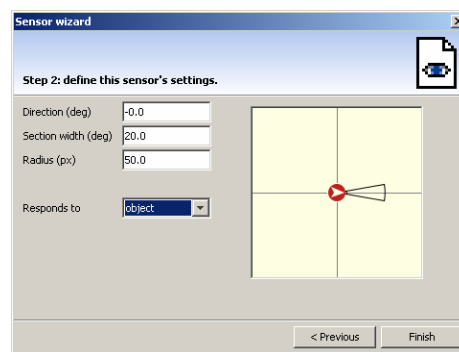


Figure 3 - Additional configuration for an object sensor.

This additional configuration should be self-explanatory due to the clear preview displayed. Press [Next >] for such sensors and after configuring, press [Finish].

3.2.2 Creating and using actuators

SB-MASE agents can use five different actuators.

- The **speed actor** is always present and indicates the speed of the agent, which should be between 0 (do not move) and 1 (move at full speed). The actual distance of movement is determined by program settings.
- The **angle actor** is always present and indicates the rotation of the agent. It enables you to control the steering mechanism of the agent. It accepts values between -1 and 1, for 180 degrees to the left and to the right, respectively. Setting the actor to 0.5, for example, will result in a rotation of 90 degrees to the right.
- The **trace actor** is present purely for visual purposes. It draws a trace behind your agent when it is activated, so that the agent's movement is more clearly visible. Valid input values are: 0 - do not produce trace; 1 - produce trace.

- Using its **gripper actor**, the agent can pick up movable objects. Accepted values are NIL (do not send any data) or 0 for turning the gripper OFF, and 1 for turning the gripper ON.
- The **factory actor** allows the creation and destruction of objects by agents. It is explained in section *The Factory process*.

3.3 Creating custom processes

SB-MASE contains a powerful programming language that allows you to write your own custom processes. We will now briefly introduce this language here.

3.3.1 The process structure

Every custom process has the same structure. When you create a new custom process, this structure is already present in the *Source code editor*, as is shown below:

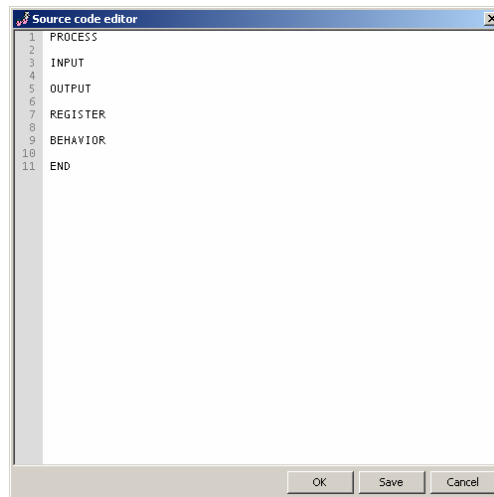


Figure 4 -The *Source code editor* for custom processes shows the standard process structure.

Every part of this structure has its own meaning:

- Behind **PROCESS**, you need to specify a name for your process, for example **PROCESS TEST**. Without a name, the process will not compile.
- Under the **INPUT** header, you will specify the optional input ports of the process. These ports have a name and a data type (see below).
- Under the **OUTPUT** header, you will specify the optional output ports of the process. These ports have a name and a data type (see below).
- Under the **REGISTER** header, you will specify the optional persistent storage slots of the process. These slots have a name and a data type (see below).
- Under the **BEHAVIOUR** header, you specify the behavior of the process, using SB-MASE's built-in programming language, discussed below.

- 🌀 Intuitively, you would say that input port names, output port names and storage slot names can all be used as variables in your process behavior. However, you should use input ports as read-only and output ports as write-only. If you do not do this, processes will have strange and erroneous behavior.

3.3.2 SB-MASE data types

SB-MASE processes can use four different data types for variables, which can be declared as input ports, output ports and storage slots by placing them under the relevant header. These data types are:

- The **REAL** data type. Variables of this type can store a single double precision number. For example, declare such a variable as a storage slot by writing: `a_number: REAL:` below the **REGISTER** header.
- The **STRING** data type. Variables of this type can store any string of characters or numbers. The data type is hardly, if ever, very useful. Declare it the same way as a real.
- The **TYPE** data type. This data type can store compound data in the form of a fixed set of real values and/or strings. Each of these elements will have a name. For example, to declare a type that stores both angle and velocity as one of the inputs of a process, write `a_type: TYPE[angle:real, velocity:real]:` below the input header. If you want to refer to an entire type, you can use its declared name. If you want to refer to an element of the type, use the syntax `<type_name>.<element_name>`, for example `a_type.angle`.
- The **LIST** data type. A list can store any number of items of any data type except for lists. All items must conform to the same specification, i.e., it is not possible to store both reals and strings, nor is it possible to store two differently specified types. For example, to declare a list containing real numbers, write `a_list: LIST[REAL]:` below any of the headers.

3.3.3 Specifying custom process behavior


Using SB-MASE's programming language, it is possible to define process behavior. Please note that all variables used in such a behavior must be explicitly declared under one of the headers (see *The process structure*). With very straightforward examples, we will now discuss all language constructs.

- 🌀 The language compiler is not case-sensitive when it comes to keywords; they are automatically capitalized. However, variables will be treated case-sensitively.

Assignment of values to variables


Assignment of values to variables is straightforward:

```
[...]  
REGISTER  
a_real: REAL:  
BEHAVIOUR  
a_real := 4;
```

-  Using the `RANDOM` keyword, you can assign a random value between 0 and 1 to a variable. The value will be updated every time the process is executed. For example: `a_real := RANDOM;`

Mathematical expressions


The standard mathematical expressions using `*`, `/`, `+`, `-` and `^` can be formed with variables, constants and the `RANDOM` keyword.

-  It is advised that you always use brackets to create at most binary relations, since the parser for mathematical expressions does not deal with expressions such as `4 / 2 * 3` even though they are accepted. Therefore, use `(4 / 2) * 3` or `4 / (2 * 3)`.

Lists: the FOR loop, ADD statement and CLEAR statement

Using this pretty useless program snippet, you first add the element 1 to the end of the list. Then, you subtract 1 from every element in the list. Finally, the list is cleared.

```
[...]  
REGISTER  
a_list: LIST[REAL];  
  
BEHAVIOUR  
ADD 1 TO a_list;  
FOR element IN a_list DO  
  element := element - 1;  
ROF;  
CLEAR a_list;
```

-  Due to the risk of an infinite loop, it is not allowed to add something to a list within a `FOR` loop that iterates over the same list. For example, in the above

program, we would get an error message if we wrote `ADD 1 TO a_list:` somewhere between the `DO` and `ROF`.

Conditional programming

Conditional programming is straightforward:

```
[...]  
INPUT  
a_real_input: REAL;  
  
OUTPUT  
a_real_output: REAL;  
  
REGISTER  
BEHAVIOUR  
a_real_output := 0;  
IF a_real_input > 0.5 THEN  
    a_real_output := a_real_input;  
FI
```

This simple piece of code demonstrates conditional programming. Obviously, the comparators `<`, `<=`, `=`, `=>` and `>` can all be used, and the expressions before and after it can have any valid form, as explained earlier.

The PRINT statement

The `PRINT` statement can be used for debugging purposes; it allows you to display the name and value of any variable to the Monitor output window, if this window is enabled, by typing `PRINT a_variable:`.

The STOP statement

The `STOP` statement can be used to stop running the agent (and all other agents). This statement should obviously be used with care; for example, if you are simulating a race between ships trying to reach the harbor, the last one arriving might call the `STOP` statement, but obviously not the first one arriving.

3.4 Creating a process hierarchy

3.4.1 Creating connections

Communication between processes is a key concept of SB-MASE programming. For example, a sensor can communicate its value to a process that

makes some decision based on this value. You can set up such a communication by connecting the output of a process to the input of another process. To do this in the Agent editor, click in the circle next to the output name you wish to connect. A red line will appear. Click in the circle next to the input name you wish to connect and the red line will turn black, indicating that a connection has been made. For example:

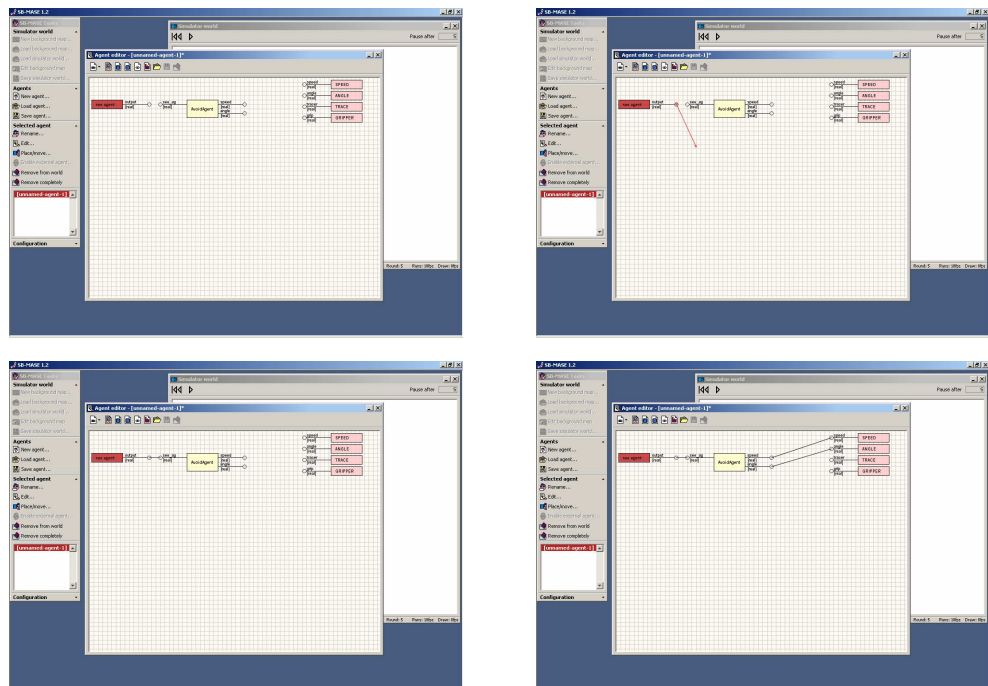



Figure 5 - Connecting two processes.

Using this connection, the process named AvoidAgent can now use the output of the sensor SeeAgent. Similarly, connections have been made between the process AvoidAgent and the SPEED and ANGLE actors of the agent.

3.4.2 Using suppressors and inhibitors



The power of the subsumption architecture stems mainly from the fact that it can use suppression and inhibition mechanisms. These mechanisms mediate between two processes of different priority. For example, a Mars Rover needs to wander around on Mars quite randomly, but it should not bump into obstacles. Instead of modelling this task in one process, we create a process for random walking and a process for obstacle avoidance which is coupled to a relevant sensor. Now we specify that the wandering process should be active unless the obstacle avoidance process becomes active. To achieve this, we could use the code presented below.

 **Observe that the Avoid process only produces output when this is necessary! This is a key concept in subsumption programming.**

PROCESS Wander

INPUT

OUTPUT

angle: REAL:

speed: REAL:

BEHAVIOUR

speed := 1:

angle := (0.002 * RANDOM) 0.001:

END

PROCESS Avoid

INPUT

sensor: real:

OUTPUT

angle: REAL:

speed: REAL:

BEHAVIOUR

IF sensor > 0.4 THEN

speed := 0:

angle := 0.5:

FI

END

Now, the two processes need to be mediated; we want to specify that the output of the Wander process is used *unless* the Avoid process becomes active. This can be done using a Suppressor. In the next screenshot, we demonstrate the usage of this mediator:

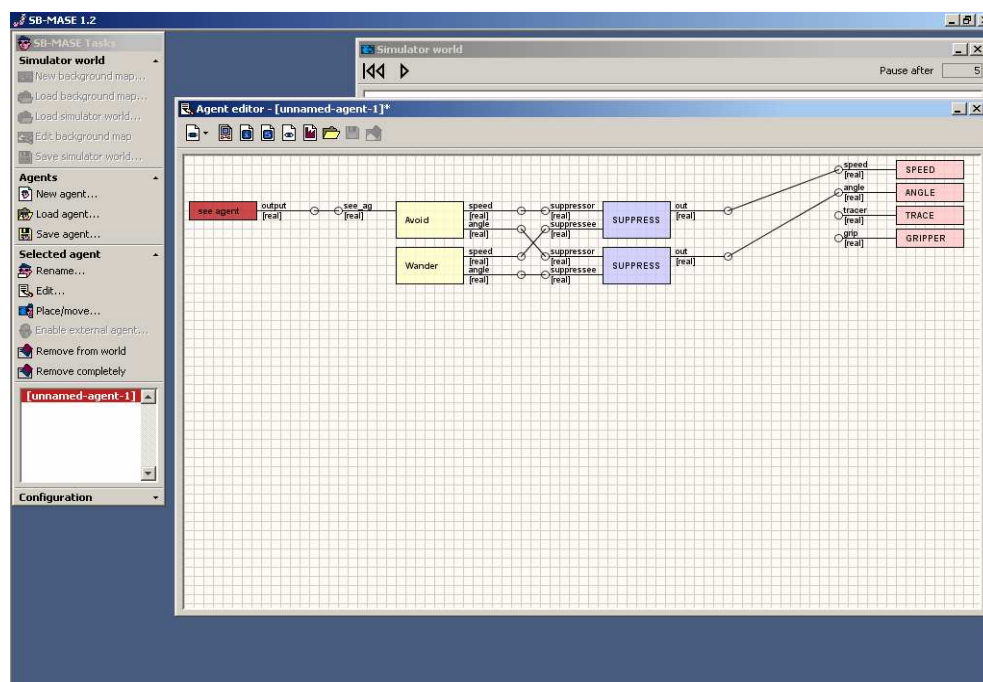


Figure 6 - Mediating between two processes using Suppressors.

Thus, a Suppressor process can be described as follows in pseudocode:

```

IF suppressor = NIL THEN
    output := suppressee;
ELSE
    output := suppressor;
FI

```

The function of an Inhibitor process is very similar, except for the fact that the Inhibitor completely blocks transmission of data:

```

IF suppressor = NIL THEN
    output := suppressee;
ELSE
    output := NIL;
FI

```

Thus, using Suppressors and Inhibitors, you can program your agent with task-based decomposition and priority mediation.

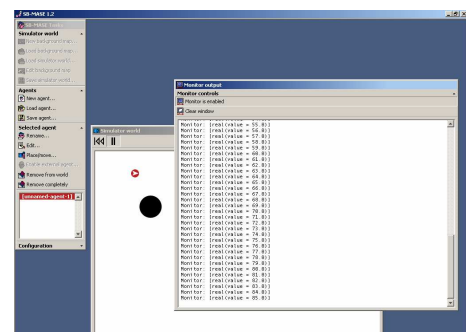
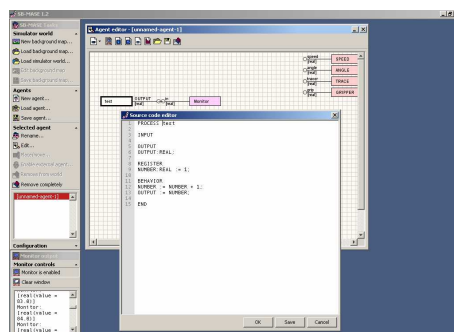
3.5 Creating built-in processes

SB-MASE comes with a large set of built-in processes that can be used in your agents. We will now list these processes. You can create these processes by clicking on the button with the icon corresponding to the icon shown below.

3.5.1 The Monitor process



The monitor provides debug facilities. After you named the monitor, connect its input to the output port of another process. For example, below we create a simple process called test and connect a monitor to it.



Now, we place the agent into a simulator world, enable the monitor by clicking [Monitor output: monitor is disabled] and press [>] in the *Simula-*

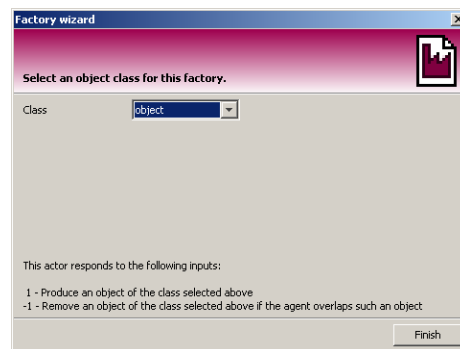
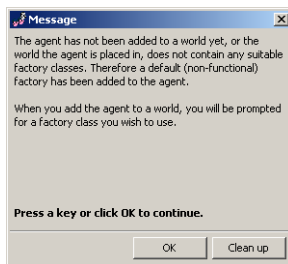
tor world window. The output produced by test is now displayed during runtime in the *Monitor output*.

3.5.2 The Factory process



Certain simulator worlds allow agents to add and remove their own objects. For example, in a simulated Mars Rover environment, robots might perform path finding by dropping and picking up radioactive grains.

Using the *Factory wizard*, you can define a device (a Factory process) that produces and consumes objects of a certain class. If the agent you are editing is not placed in a simulated environment that supports producing and consuming objects, a dialog box informs you that the Factory will not be functional until you add the agent to a suitable environment (below; left). If the environment is already suitable, the wizard allows you to define which type of object the Factory will produce and consume (below; right).



Using the Factory process is straightforward. It responds to only three values, viz.

- 1: Produce an object of the class selected
- -1: Remove an object of this class, if the agent overlaps such an object
- NIL: Do nothing at all

In section *Creating objects*, we discuss which objects are suitable for production or consumption by the Factory process.

4 Creating simulator worlds

4.1 Introduction

SB-MASE simulator worlds are created by specifying the content of such a world in a feature-rich text-based format. This allows you to introduce various interesting features, such as:

- Objects belonging to a certain class, with each object having a unique name;
- Objects that can be massive or flat;
- Objects that agents can pick up;
- Objects that agents can produce and consume;
- Large quantities of the same object distributed over a specific area;
- Objects that can emit light.

In the following sections, we will discuss how you can create your own simulator worlds.

4.2 Creating the canvas

Start your new simulator world by choosing [SB-MASE Tasks: Simulator world: New background map...]. This will create a default simulator world.

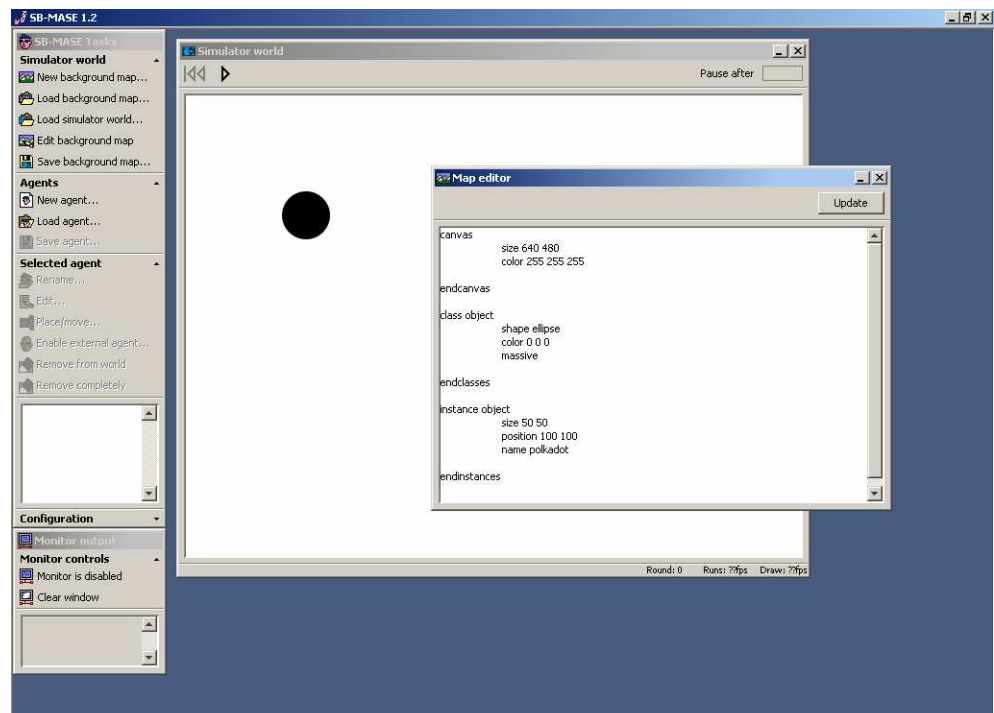


Figure 7 - SB-MASE created a default simulator world.

As you can see, the canvas (as defined between `canvas` and `endcanvas`) has two properties, viz. a size (in pixels, respectively width and height) and a color (specified in RGB values). You can separate these values by spaces, comma's, or both. Adapt the properties and press [Update] when ready.

4.3 Creating classes and objects

With the Object sensor and the Sonar sensor, simulator worlds can be perceived by agents both in a high-level way (perceiving objects) and a low-level way (reading sonar values). The way simulator worlds are defined facilitates this. You define your objects in two steps. First, you create classes to which these objects will belong (for example, rocks, islands, enemies), then you create instances of these classes to make actual objects (for example Pebble1, Hawaii or Super Mario). We will now explain how this is done.

- 🌀 The language for defining simulator worlds is entirely case-insensitive.
- 🌀 SB-MASE comes with some example simulator worlds that can help clarify the theory explained here.

4.3.1 Creating classes

Below the canvas definition, there is room for class definitions. Each class definition should start with “`class <classname>`”, followed by the properties of this class. End the class definitions section with the `endClasses` statement.

- 🌀 Each object sensor can perceive all instances of *one* specific class within its own region, regardless of the properties of the class or the instances.

We will now discuss the various properties classes must and can have.

Color [required]

Using the property `color`, followed by three RGB values, you define the color of all instances of the specific class.

Shape [required]

The `shape` property determines which shape all instances of this class will have; valid options are `rectangle` and `ellipse`. Rectangles can also be used to draw lines (see below).

Fixed size [optional]

The optional property `fixed size` indicates that all instances of the class in question will have the same size. This comes in handy in case of classes that will have many instances, for example grains of radioactive sand. Provide two arguments for width and height respectively.

- 🌀 The property `size` can also be used separately. Then, it is optional and provides a default value for the size.

Massive [optional]

The optional property `massive` distinguishes flat (default) objects from objects that have height (i.e. are massive). If an object is flat, it is not perceived by sonar sensors.

- ✎ If an object is massive, this does not mean that they cannot be crossed – the behavior of agents defines whether or not agents can enter certain massive objects. For example, islands in an ocean are definitely massive, but some agents can easily walk on them.

Movable [optional]

Using the property `movable`, you specify that instances of the class can be moved by agents, for example by using the Gripper actor.

Static [optional]

The optional property `static` indicates that agents cannot create instances of a class. If the property is not provided, agents are allowed to instantiate the class and remove instances of this class from the simulator world using a Factory.

- ✎ For a class to be usable by a Factory, it must be massive and not static.

Light [optional]

Objects with this property can be perceived by a Light source sensor. They are very convenient for beacons or the origins of a gradient.

Flock [optional]

Flock classes enable you to define an area in which there is a large number of similar objects. In the class definition, you specify the shape, color and size of these objects. Later on, in the object definition, you specify the area in which these objects are distributed, their distribution, and the number of objects generated. Thus, objects with the `flock` property must have the `size` property set as well, in addition to color and shape.

4.3.2 Creating object instances

The actual objects in your simulator world are defined by instantiating the classes. Each instance definition should start with “`instance <classname>`”, followed by the properties of this instance. End the instance definitions section with the `endinstances` statement. We will now discuss the various properties instances must and can have.

Name [optional]

This property specifies a name for an instance and is optional except for light source objects. Specify the name as a single argument – other symbols than [A-Z, 0-9] are not allowed!

Position [required]

This instance property, followed by an x- and an y-coordinate, determines the position of the object, in computer coordinate system (i.e., a higher y-value leads to lower placement; a higher x-value shifts the object to the right) and in pixels.

Rotation [optional]

This instance property is optional and defines the rotation around the center point of the object, in degrees, where a positive rotation is clockwise. Provide a single real-valued argument.

Size [optional]

This instance property is optional, unless (1) the instance's class does not define a default size, or (2) in flock classes, where this size determines the size of the region in which the flock is generated. Specify two numerical arguments, for width and height in pixels.

Shape [only for flock classes]

This property is ignored for all but flock classes. It specifies the shape of the region in which the flock is generated.

Items [only for flock classes]

This property is ignored for all but flock classes. It specifies the number of objects that are generated within the flock region. Specify one real-valued argument.

Distribution [only for flock classes]

This property is ignored for all but flock classes. It specifies the distribution used for distributing the flock objects in the region. Valid options are `normal` and `uniform`.

4.4 Placing agents in simulator worlds

Before an agent can be used in simulation, it must be added to a simulator world. In order to do this, first make sure that you have an agent and a simulator world (obviously). Then, click on the agent you wish to place in

[SB-MASE Tasks: Selected agent]. Click [SB-MASE Tasks: Selected agent: Place/Move...]. A dialog box will pop up as follows:

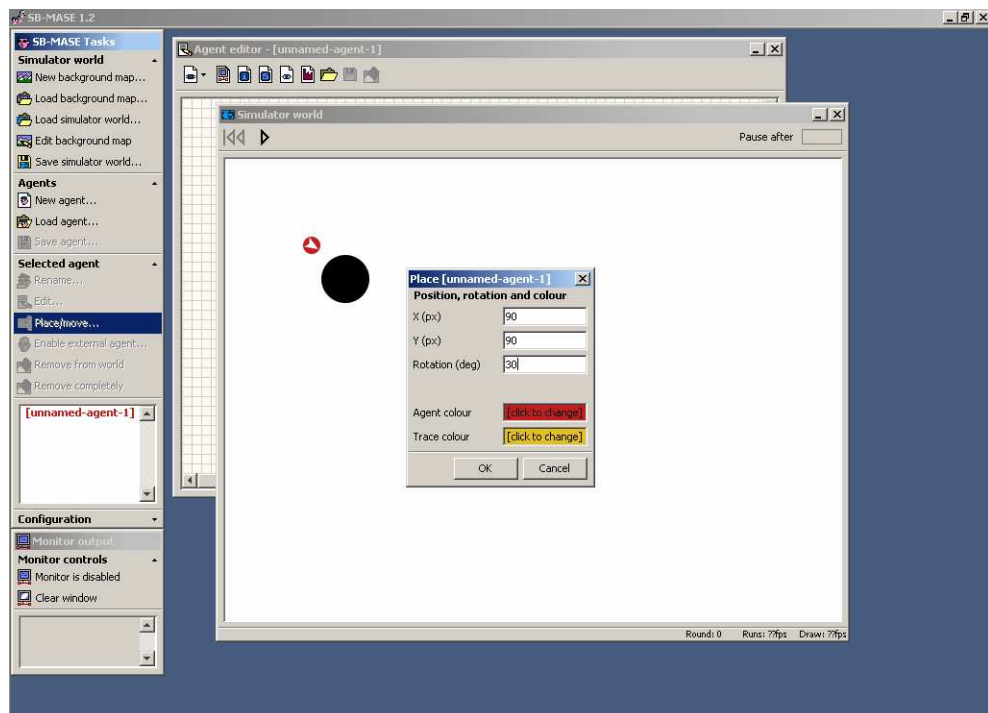


Figure 8 - Placing agents in a simulator world

As you can see, it is possible to specify the X-coordinate and Y-coordinate as well as the rotation. For your own reference it is also possible to give each agent a unique color (visible in both the simulator and the agent list to the left) and a unique trace color (if the agent has a trace actor and this actor is enabled).

- 🔗 The same dialog can be used to change the position of an agent that already has been placed.
- 🔗 Agents that are placed in a simulator world will remain editable unless the simulation is currently running. Pause or rewind the simulation before editing your agent.

4.5 Specifying settings

Under [SB-MASE Tasks: Configuration: Configure plug-ins: SB-MASE Simulator] you can find various configuration options for the simulator. We will briefly discuss them here.

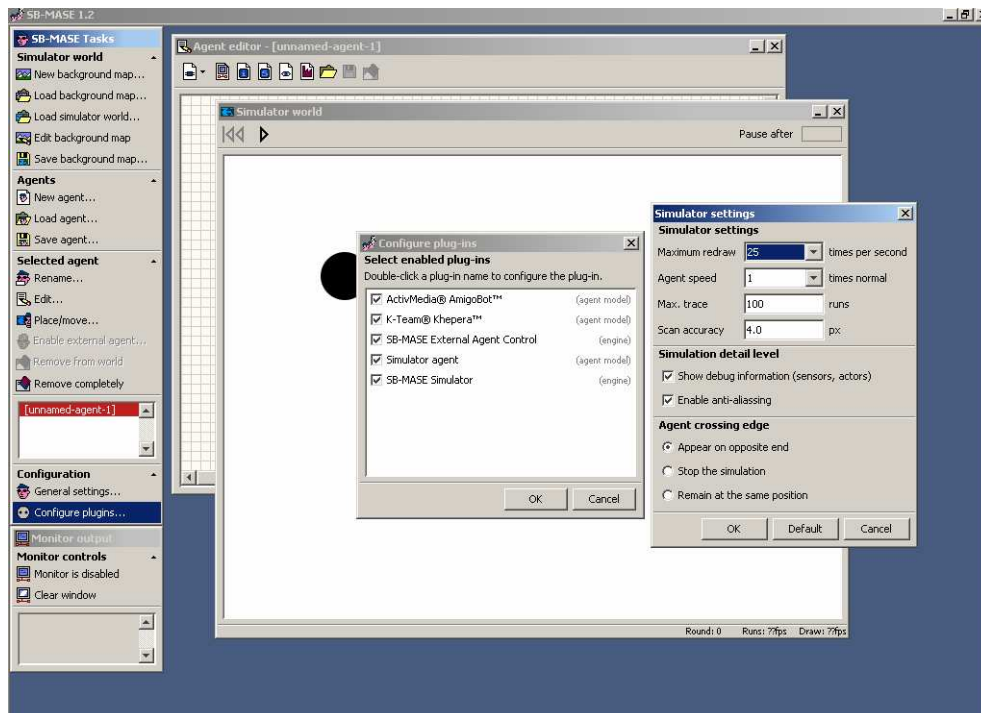


Figure 9 - Simulator settings.

Maximum redraw – Specifies the number of frames per second drawn in the simulator world window. This is the maximum – in complex worlds or on slow machines, the frame rate might be lower.

Agent speed – A ‘normal’ agent speed corresponds to 1 pixel per round if the speed actor is set to 1.0. Higher speeds make your agents move faster because they make bigger steps.

Max. trace – For performance reasons, traces produced by the trace actor can have a maximum length. More length gives you better overview of the path an agent has followed, less makes the simulation progress faster.

Scan accuracy – This effects sonar and object sensor precision. Unlike rectangular objects to which the distance can be calculated exactly, the distance to circular and ellipsoid objects must be detected by an iterative algorithm. More accuracy is slower here. If you use only rectangles, the setting obviously will have no effect.

Show debug information – With this visual effect enabled, you can see while simulating what the agents’ sensors perceive and what their gripper actors are doing (agents get a little gripper in front if they are gripping).

Enable anti-aliasing – With this visual effect enabled, the image presented by the simulator world window is smoothened. This requires some extra processing power and can be turned of because of that.

Agent crossing edge – There are three options for agent crossing the edge of the map, viz. (1) appear on opposite end, which turns the square simulator world into a circular one (or more precisely, a donut world); (2) stop the simulation, which does exactly that; and (3) remain at same position, which makes the agent halt at the edge.

5 **Creating external agents**

5.1 **Creating and editing an external agent**

This part of the manual is under construction.

5.2 **The run mode sensor**

This part of the manual is under construction.

5.3 **Loading the SB-MASE Server Control Panel**

This part of the manual is under construction.

6 **Advanced configuration and settings**

6.1 **SB-MASE**

This part of the manual is under construction.

6.2 **SB-MASE Server Control Panel**

This part of the manual is under construction.

7 **Developers' Guide**

7.1 **Developing plug-in processes**

This part of the manual is under construction.

7.2 **Developing agent models**

7.2.1 **Developing simulator agent models**

This part of the manual is under construction.

7.2.2 **Developing external or hybrid agent models**

This part of the manual is under construction.