More than a simple Tutorial about **SeSAm**

June 13, 2003

Contents

This is a tutorial for **SeSAm**. You will learn how to implement a multi-agent simulation from the very beginning to sophisticated high end models. **SeSAm** was and is further developed at the Universität Würzburg and can be freely distributed. It is academic software, that means for us not, that it some preliminary non-working prototype, but we won't spend month of our time writing manuals (that are always requested, but never read). On the other side, we want you to use **SeSAm**, as it is worth using, therefore we see the need for some introductory material. Thus is is more than a tutorial: it contains an index that leads to the location where the explanation can be found. In addition you will find more general explanations in smaller text size between two lines. Thus this tutorial is almost a complete documentation – at least when it is finished.

We wish you successful modelling and simulation and a lot of fun using **SeSAm**

The **SeSAm**-Team
Franziska Klügl
Christoph Oechslein
Rainer Herrler
Steffen Glückselig
Clemens Mühlberger

1 What this Tutorial is about?

1.1 SeSAm

This is a tutorial about using **SeSAm**. Before going on, we should make clear what **SeSAm** is, and what it is not:

SeSAm stands for Shell for Simulated Agent Systems. Its provides a generic environment for modelling and experimenting with agent-based simulation. We specially focused on providing a tool for the easy construction of complex models. Agents consist of a body that contains a set of state variables and a behavior that is implemented in form of UML-like activity diagrams. Activities are seen as simple scripts that are initiated and terminated based on rules. The actions and conditions can be composed of a extensive number of primitive building blocks, a user is able to design visually a simulation without programming in a traditional language. User functions, features (set of variables and functions) and user defined data types provide a possibility to construct higher level building blocks. A component for evolutionary adaptation of parameters is also integrated. All is implemented visually in form of quasi declarative descriptions. These are executable in the same environment. Dynamics of this simulation may be observed in form of animation or user configured, autonomously updating curves. As there are freely configurable instruments for gathering data and scripting options for constructing simulation experiments, **SeSAm** is a highly valuable tool for MAS simulations especially for complex models with flexible agent behavior and interactions.

1.2 Further Material

Further material on **SeSAm** can be acquired on the **SeSAm**-Webpage: *www.simsesam.de.* You can find news about courses we offer or download several example models. There you can always get the most recent version of **SeSAm**. However they won't change any more with this frequency like before (less bugs :-).

There is also a mailing list for **SeSAm**-related problems. We are all listening and will answer all questions in reasonable time.

For earlier **SeSAm**-versions (the Lisp) there is a quite extensive documentation, however useless! A recommendable introduction to multi-agent simulation in general (and into that old **SeSAm**-version) is the book "Multi-Agent Simulation" by Franziska Klügl, however in German.

2 First Steps

Download and installation of **SeSAm** is quite simple and explained in all necessary detail on the web-pages *www.simsesam.de*, where you can also access the most recent **SeSAm**-versions. If you are not sure about already having the correct Java Virtual Machine Version - also download it. With an installation there comes also this tutorial and the model we will implement in all phases.

2.1 Getting Started

After opening **SeSAm** (double click on the link *runSeSAm*) you will see, what is shown in figure 1

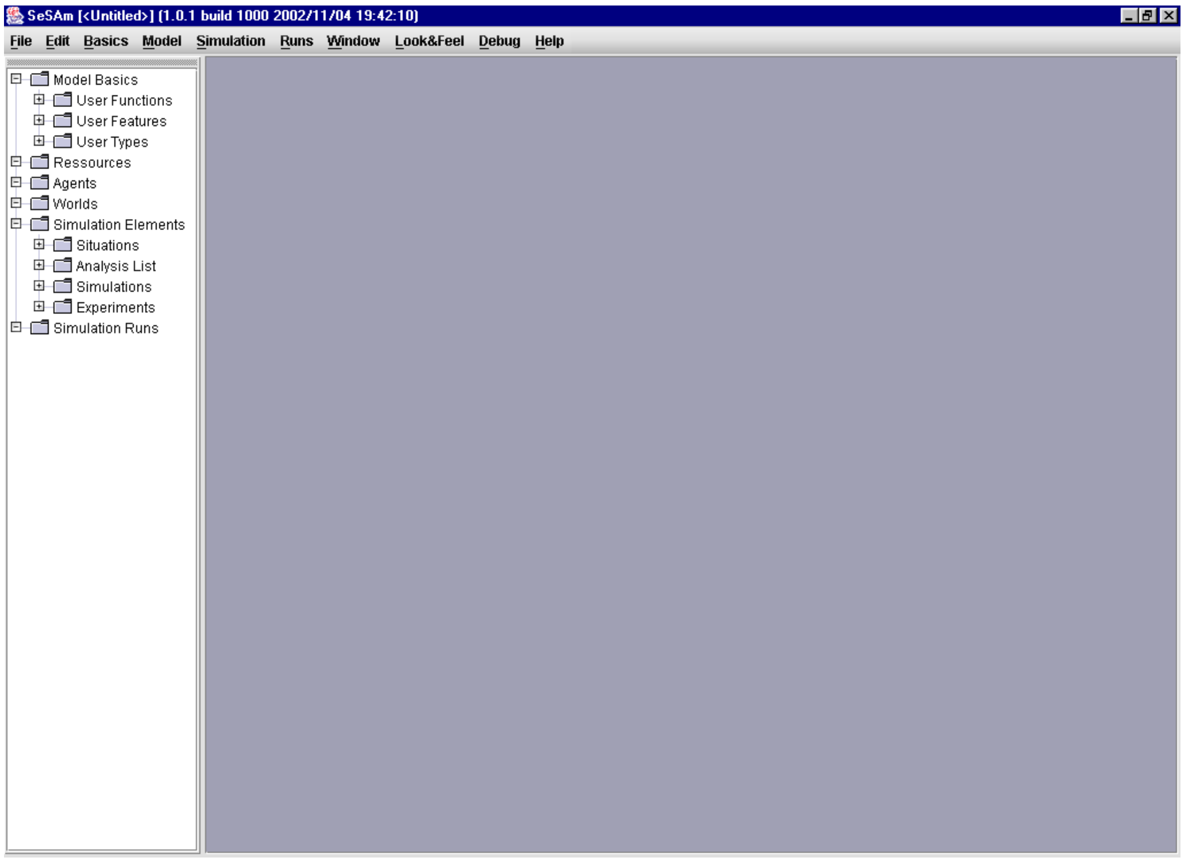


Figure 1: **SeSAm** directly after starting.

Except the *Look&Feel* , which is exclusively for switching between different ways windows and dialog items are presented, all menus in the menubar will be described when necessary. On the left you find the so-called *DeclarationTreePanel* where one can also navigate through all model contents (this may replace the third to seventh menu in the menubar).

DeclarationTreePanel

allows to access all edit-actions of the model. Just click with the right mouse button onto an item and you will find Menu-items like *New*, *Properties*, *Remove*, etc... It is located normally at the left side of the screen. You can close it by clicking with the right mouse button onto the upper frame of it, or track it to any position you want. However for restoring the fixed position on the left you have to restart **SeSAm**.

2.2 The First Agents

We want to start with the first agent class in our food web model. You have three possibilities for doing this: Either open the selection box for agent classes (double-click on the Menu *Agents* or select *Model→Agents* from the menubar) and select the button *New*, of click with the right mouse button onto the *Agents* in the DeclarationTree on the left and also choose *New*. After you did that you find a situation as depicted in figure 2 or figure 3. It shows that a new agent class with the name "untitled" was generated.

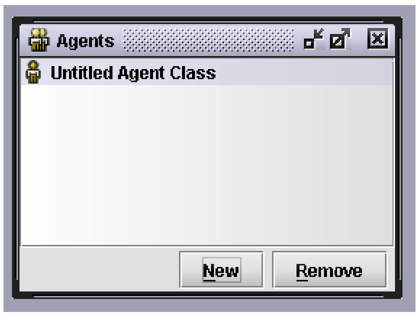


Figure 2: Newly generated Agent Class in the Agent Class Selection Box

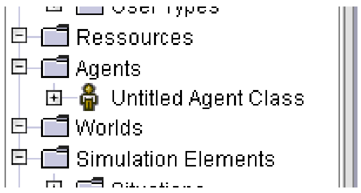


Figure 3: Newly generated Agent Class in the DeclarationTree.

2.2.1 Agent Class

This new agent class can be edited by double-clicking on the name - either in the selection box or in the DeclarationTree. This is, what we are going to do next and we see an editor as presented in figure 4

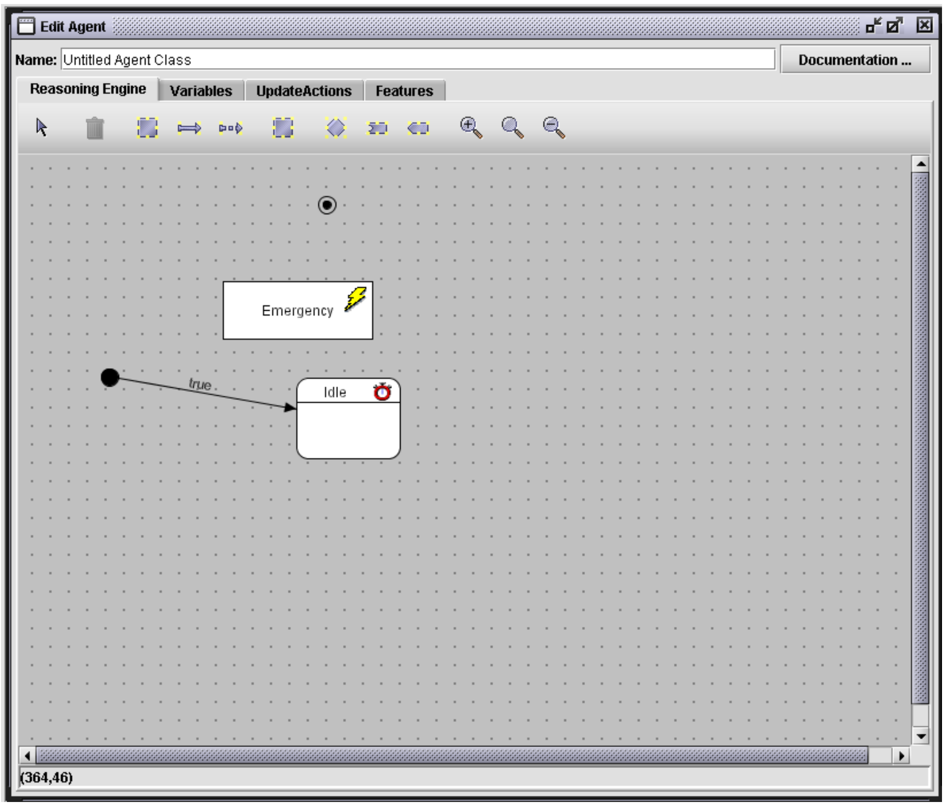


Figure 4: Editor for Agent Classes – here opened with a newly generated agent class.

In this window you see different dialog items: At the top you see on the left the *Name* field, here we exchange the text "UntitledAgentClass" by "MouseClass". At the top right you see a button where you can edit documentary strings . For complicated models this is highly recommended! The rest of this window is filled by four tap views: *Reasoning Engine*, *Variables*, *Features* and *Update Function*. In this section we will only tackle the first two. *Features* are explained in section **??**, *Update Functions* play an important role for debugging, thus they are explained in section **??**.

SeSAm Agent Concept

The most important parts of an agent are its reasoning engine and its variables. The first is used to specify the agents behavior the second is used specifying the agents state based on a set of state variables. There are different reasoning engine imaginable, yet until now there is just implemented: the Activity Reasoning Engine is based on UML-like activity diagrams. The Agent is always in one activity - which's actions the agent performs. Activities are terminated and selection by rules.

Our MouseAgent will behave like the following: It just runs around. The velocity and the agile-ness is depending on the inside temperature (in the next section, we will of course extend this model with cheese and cats). During that time, the mouse will age.

2.2.2 Agent Body

For implementing this very simple agent model we will proceed by clicking on the *Variable*-Tap and generating the variables of temperature and age. Therefore we click on the *New* in the left button of this view. After this in the left box there occurs a text "UntitledVariable" and the complete right box of the window becomes active.

Variable Form

Editing Variables is always the same in **SeSAm**. The view for this contains:

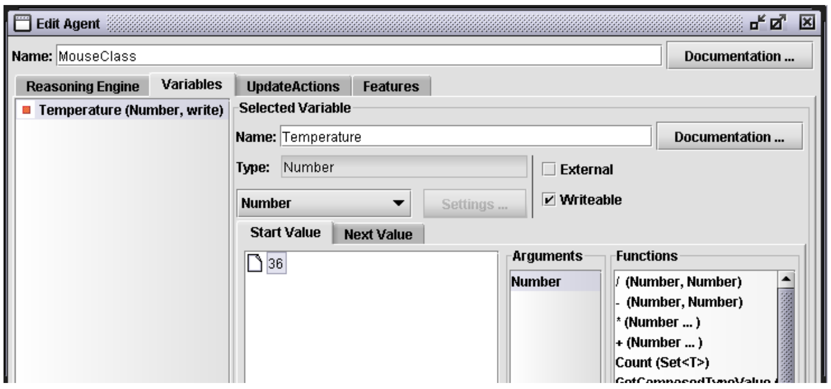
- a *Name*-field, where you can specify the name of the variable (name doubles are allowed, but can be confusing). A newly generated variable has the name "UntitledVariable".
- A documentation button invites you to input explanatory texts, that e.g describe the ideas that you had for introducing this variable.
- a *Type*-field, where you have to determine the type of the values that the variable will contain (down by the pop-up menu directly below the type field. A complete list of currently available variable types is given in annex A Types that need further information are marked using $< T >$. The default type is number.
- The checkbox for *External* specifies whether the content of this variable will be accessible by other agents. At the beginning this checkbox is not marked
- The checkbox for *Writeable* determines whether the variable is manipulable or constant (default is unselected)
- In the bottom of this part of the window is filled by two taps and a function selection list. There the start (default) value for this variable is determined. For an absolute value double click on that start value and specify a value in the new dialog or select a function in the right list. For specifying a variable with automatic dynamics select the *Next Value*-Tap and select the "Next value" checkbox and give the functions that is called for computing the new value in every round.

To implement the internal temperature of our mouse we fill these fields in the following way:

- Enter "Temperature" in the *Name*-field
- Select the *writable*-checkbox
- Double-Click on the "0"-Entry in the *Start Value*-Tab and change the "0" in the up-popping number entry dialog into "36". Click on *Ok* to insert this value.

The default values of the *Variable Type* (number) and the *External*-checkbox are ok for our `Temperature` variable. After these actions the *Edit Agent* Dialog looks like depicted in figure 5.

Now we will give the attribute `age` to our mouse agents. This will be a dynamic variable, that means a variable which's value is changing independently from any actions. For this purpose we generate a new variable (by selecting the *New* in the lower left corner of the current *Edit Agent* dialog. After entering the name "Age" (The variable type `number` is



Name: MouseClass
Temperature (Number, write)
Name: Temperature
Type: Number
Number
☐ External
☑ Writeable
36
Number
/ (Number, Number)
- (Number, Number)
* (Number ...)
+ (Number ...)
Count (Set<T>)

Figure 5: Editor for Agent Classes – after entering the `Temperature`-variable.

correct here, too) and selecting both, the *External* – the age of our mouse will be observable for other agents – and *Writable* – the age changes –, we select the *Next Value*-Tap. Here we also select the *Next Value* checkbox to confirm that this will be a dynamic variable. What you see below the *Next Value* checkbox is an area for editing functions. Like the variable edit forms, these areas are repeated in many places in **SeSAm**, it is the only way to specify a function.

Function Edit Area

The area to edit a function consists of three fields that are always ordered from left to right in the same way:

1. The field most on the left contains the current value of the function specification. This field is also called the function tree as the function specification and the arguments belonging to it are organized as a tree or hierarchy. Every argument can be a tree-like function specification again.
2. The field in the middle contains a list of possible atomic arguments. If you double click on a type name, like `number` a new dialog occurs where an absolute value can be specified.
3. On the most right a list of appropriate functions is offered. This list is adapted to what is expected for the argument or function that is selected in the current value field on the left. That means, if you don't find a function in that list then you might have not selected the right thing in the function tree on the left. Positioning the mouse pointer on one functions opens a tool tip that describes what this function will do.

The procedure for specifying a function is always the same. Select on the left field the entry that you want to change, then the argument and function lists will change to contain the possible entries. Double Click on a function on the left (sometimes additional type information may be requested, e.g. when selecting `GetVariable` then the type of the variable will be asked for). After that in the left field a new sub-tree occurs where you can specify the leafs in the same way. If you want to edit an atomic value, double click on the appropriate entries in the arguments list. Copy and Paste are possible, as well as moving entries, if the types of the sources and destinations are the same.

The `Age` of the mouse agents should be increased by one degree in every time step. In the function specification for the next value we have to specify a function call that returns a new number that is set as the new value of the variable `Age`. Therefore we double click on the `+ (Number)` entry in the function list on the right. On the left current specification area a small hierarchy occurs that shows on the upper level a ”+” with two arguments with the atomic value 0. We continue with the first of these two arguments, selecting it and choosing in the right function list the entry `GetVariable (Var <Number>)`. This is a function that supplies the value of a variable of the agent that calls it . The first argument of the `+` is now a sub-tree with the `GetVariable` and the argument value ”null”. As here a variable name is required the Arguments and Functions lists are adapted. If you select the variable entry ”null” and double click on the entry `Variable<Number>` in the Arguments list then a dialog will be opened where you can select one of the numeric variables of the mouse agent. Now you have to select `Age` and close the dialog by clicking on the OK-Button. Double clicking on the second argument of the `+` – the second ”0” – a dialog opens where you can change the 0 to a 1. Now we have finished the specification of the *next value* of the attribute `Age`. The next value is determined by adding the number 1 to the current value of the variable `Age`. In figure 6 this procedure is illustrated.

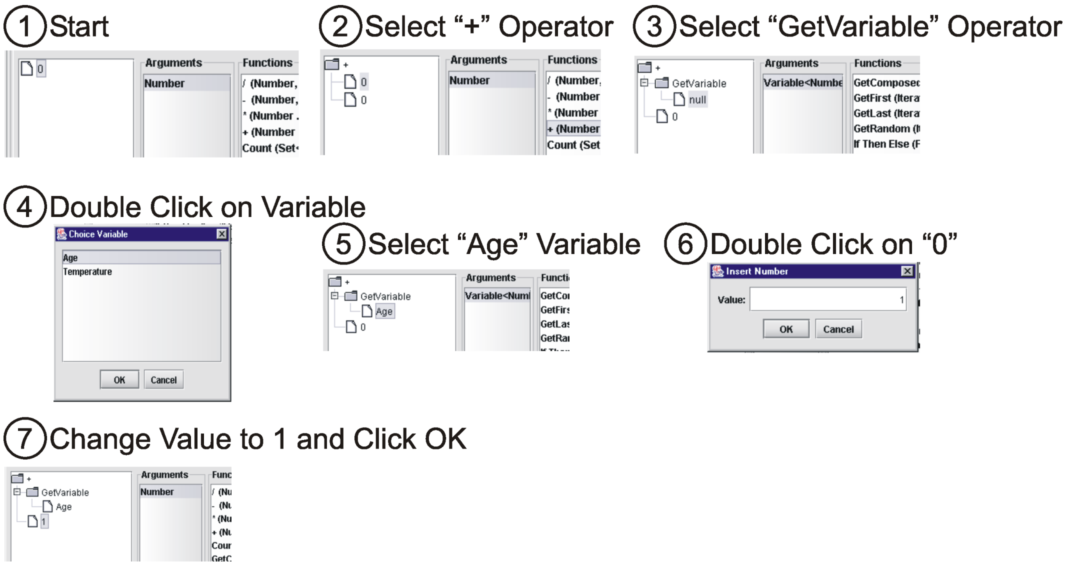


Figure 6: Specification of the new values for the variable `Age`

With this variables we have finished the specification of the agent body. Now we can continue with the specification of the mouse behavior.

2.2.3 Agent Reasoning Engine

In the agent reasoning engine the behavior of an agent is implemented. Currently one type of reasoning engine, namely the Activity Reasoning Engine is available. When opening an *Edit Agent* dialog or selecting the *Reasoning Engine* tap in an already opened *Edit Agent* dialog a panel for editing the reasoning engine becomes visible. At the beginning a default behavior as depicted in 4 is provided. One can see four types of nodes:

- Start Activity is represented by a filled black circle. At the beginning a rule that always fires (condition is equal to ”true”) connects this activity to the next normal one.
- State Activity is the box with the rounded edges. At the beginning the name of this

activity is `Idle`. The stop watch shows that this activity consumes at least one time step. During the interpretation of the activity diagram an activity with a stop watch can only be changed until the next update round. The shape of the activity node denotes that this is a state – that means the agent is in a more quiet conditions and stays passive.

- End Activity is small filled black circle with the second frame. If a rule ends at this activity, the reasoning engine is terminated and the agent is erased from the world - it dies.
- Emergency Activity is the box with the flash. Rules that start in this activity are tested in every time step (see also section ??).

The running around behavior of our mouse agent will contain tree activity nodes. At the beginning we specify an `Init`, where we set the velocity of our mouse according to its temperature and a second activity where the actual (random) `Movement` happens, during that the mouse becomes warmer. A third activity is responsible for `Sleeping` - some relaxing and cooling down activity.

For preparation we have a look onto two menus: The operator selection menu on the top of the reasoning engine panel. Figure 7 explains the different operators. For executing the associated action, select the operator and click either on the panel or the node that this operator should be applied to.

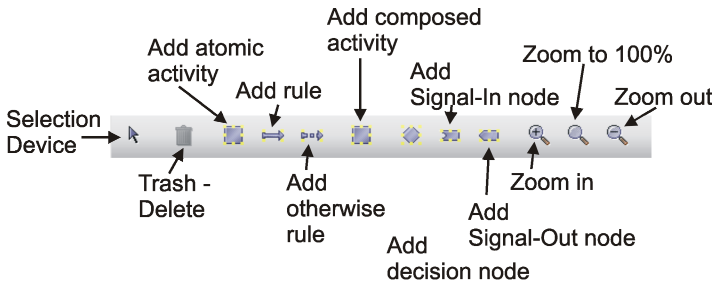


Figure 7: Operators on the activity reasoning engine editor

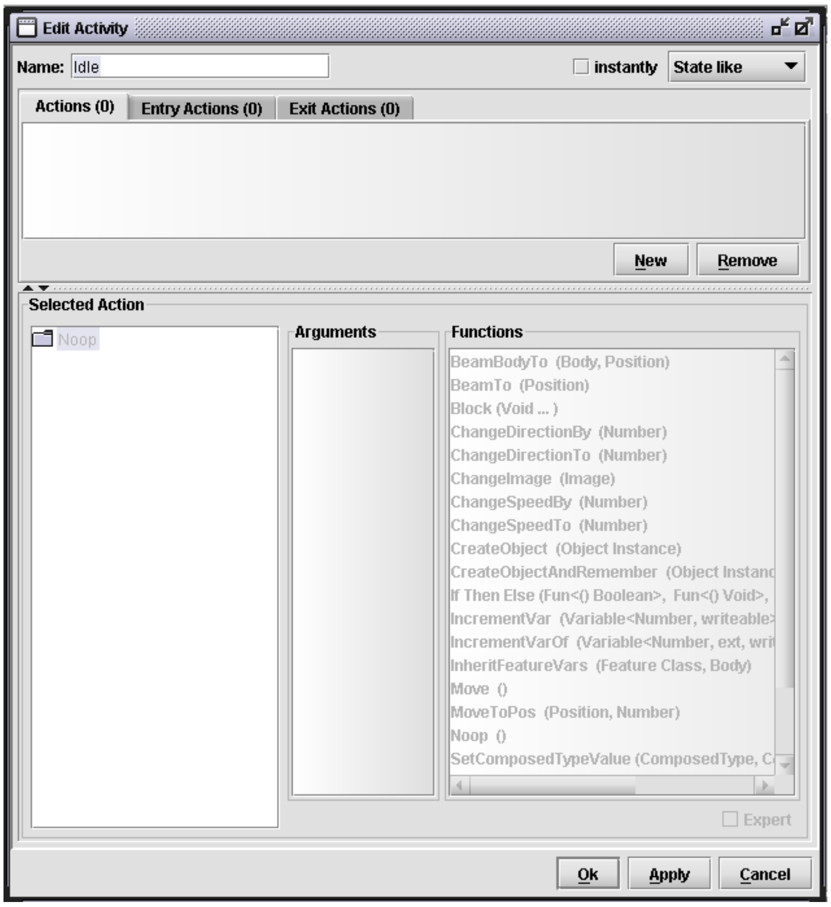
Based on this operators a lot of activity nodes with different shapes can be inserted into the activity graph definition. These shapes are explained in section ??. The shape itself makes no difference, it is just different for transparency reasons.

Now we can start editing the mouse behavior. For specifying the `Init` we re-use the default `Idle`. We make a double click onto that box. An dialog *Edit Activity* opens. This dialog is depicted in figure 8

Activity Form

The editor for specifying an activity contains the following fields

- *Name*-field contains the name of the activity. Like variables names need not to be unique, however it might be confusing.
- *instantly*-checkbox specifies whether the activity is instantly or not. An instant activity may be terminated and a new one selected during one time step, non instant activities at least consume one time step. In the activity graph they are depicted by a small stop watch that is crossed



Name: Idle

instantly ☐

State like

Noop

Expert ☐

BeamBodyTo (Body, Position)
BeamTo (Position)
Block (Void ...)
ChangeDirectionBy (Number)
ChangeDirectionTo (Number)
ChangeImage (Image)
ChangeSpeedBy (Number)
ChangeSpeedTo (Number)
CreateObject (Object Instance)
CreateObjectAndRemember (Object Instanc
If Then Else (Fun<() Boolean>, Fun<() Void>,
IncrementVar (Variable<Number, writeable>
IncrementVarOf (Variable<Number, ext, wri
InheritFeatureVars (Feature Class, Body)
Move ()
MoveToPos (Position, Number)
Noop ()
SetComposedTypeValue (ComposedType, C

Figure 8: Operators on the activity reasoning engine editor

out (instant) or not . Instant activities always require an termination rule, as they are executed again and again until a rule may fire (or the interpreter detects an endless loop, then an error message is produced and the simulation is terminated).

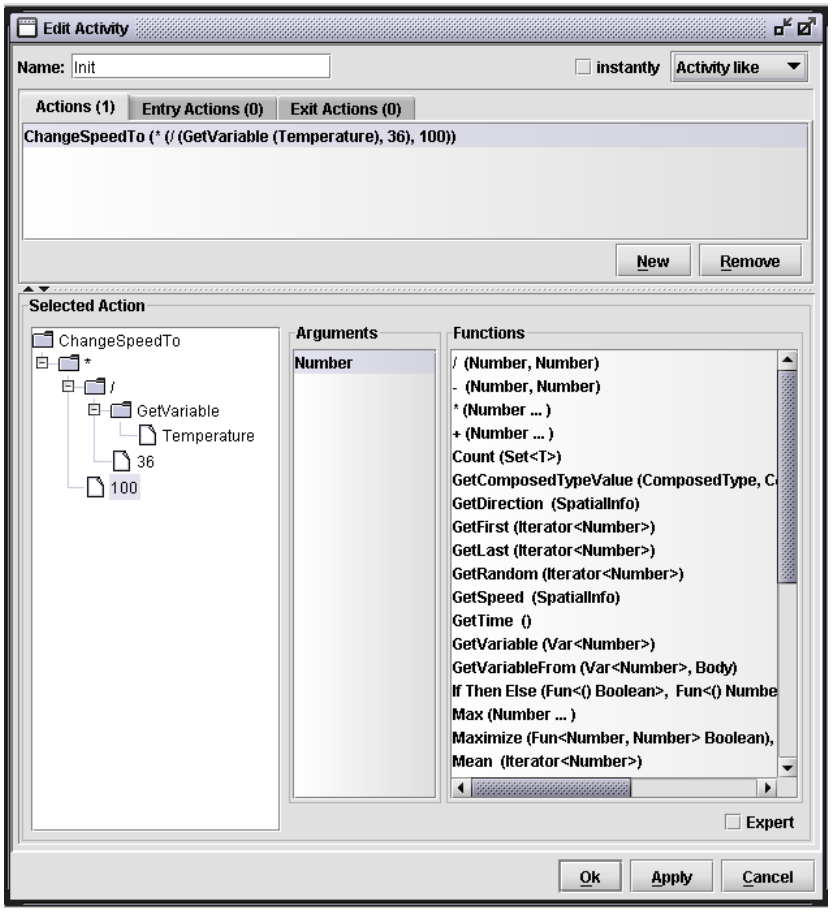
- *Shape*-Pull-down-menu. The default `Idle` is "state-like", all other newly added activities are either "activity-like" or according to the selected panel operator.
- *Action*-subview contains three taps with the same types of fields. All have a sequence of actions and *New* and *Remove*-Buttons. The title of the taps shows whether in the list there are
  - *Actions* – that are the actions of the activity that are in every activity update executed (once of non-instant activities per time step, potentially more than once for instant activities)
  - *Entry Actions* – this sequence of actions is executed when the activity is newly selected to be the current activity of the agent.
  - *Exit Actions* – this sequence of actions is executed when the activity is terminated as another activity is selected.

  The number in brackets behind the tap title denotes how many actions are already specified in this sequence.
- *Selected Action* Specification fields is a function edit area like we learnt at the new value of a dynamic variable specification. After selecting an action to change in the sequences above, you can proceed like described on page 8. The only difference to the next value specification is that in the *Functions* List there is a list of actions instead of a list of Functions that return numbers.

For editing our `Init`-Activity we proceed – after giving it the correct name – by clicking on the *New* to generate a new action (in the *Actions*- tap view). In the action sequence a new action `Noop` occurs. `Noop` means "Do nothing", Selecting this action the lower part of the function edit area becomes active with the `Noop`-Action in the left part. We select the action `ChangeSpeedTo(Number)` in the right list by double clicking on it. This action requires one argument, namely a value for the new speed of our agent. This value should be depending on the temperature of the agent. We choose the following formula for computing the speed: $\frac{GetVariable(Temperature)}{36} \times 100$. This means, if the mouse has the temperature 36 then it moves with a speed of 100. If the temperature is higher, then the movement is faster and vice versa. For this aim we first select the function `*(Number...)`. As the first argument in this subtree we select the function `/(Number, Number)`. Here the first argument will be the function `GetVariable` with `Temperature` as the selected variable (for a more detailed description in a different context see page 9). The second argument of the `/` is the absolute number 36. The second argument of the `*` is the absolute number 100. After these inputs the window for editing the `Init` looks like depicted in figure 9.

The activity form has to be closed using the *OK*-Button for confirming the changes made in this editor (*Apply* confirms the changes without closing the window and *Cancel* closes the window discarding the changes made in it.

The activities `Movement` and `Sleeping` are specified in an analogous way. First select the *Add Atomic Activity* operator then click on some place in the panel, a new activity node occurs with the name `Untitled...` (full name: `Untitled Atomic Activity`). We replace this name by `Movement` and generate three new actions. The first of these actions is `ChangeDirectionBy (Number)` with the degree (max. 360) of change in the movement direction $\frac{GetVariable(Temperature)}{9}$. As we do not want them always to turn left we input as the number the following construct: `If (RandomBoolean(0.5)) Then` $\frac{GetVariable(Temperature)}{9}$ `Else`



Edit Activity

Name: Init

☐ instantly Activity like

Actions (1) | Entry Actions (0) | Exit Actions (0)

```
ChangeSpeedTo (* (/ (GetVariable (Temperature), 36), 100))
```

New Remove

Selected Action

```
ChangeSpeedTo
⊟ *
   ⊟ /
      ⊟ GetVariable
            Temperature
         36
      100
```

Arguments

Number

Functions

```
/ (Number, Number)
- (Number, Number)
* (Number ... )
+ (Number ... )
Count (Set<T>)
GetComposedTypeValue (ComposedType, C
GetDirection  (SpatialInfo)
GetFirst (Iterator<Number>)
GetLast (Iterator<Number>)
GetRandom (Iterator<Number>)
GetSpeed  (SpatialInfo)
GetTime  ()
GetVariable (Var<Number>)
GetVariableFrom (Var<Number>, Body)
If Then Else (Fun<() Boolean>,  Fun<() Numbe
Max (Number ... )
Maximize (Fun<Number, Number> Boolean),
Mean  (Iterator<Number>)
```

☐ Expert

Ok Apply Cancel

Figure 9: Edit Activity form for the Init-Activity.

$-\frac{GetVariable(Temperature)}{9}$. In the function tree the resulting specification looks like depicted in figure 10.

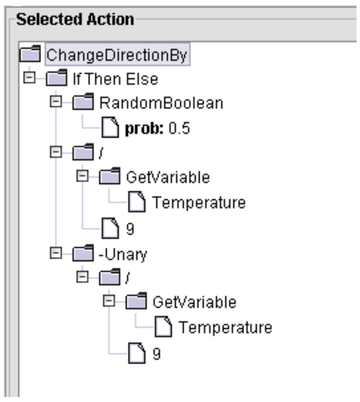
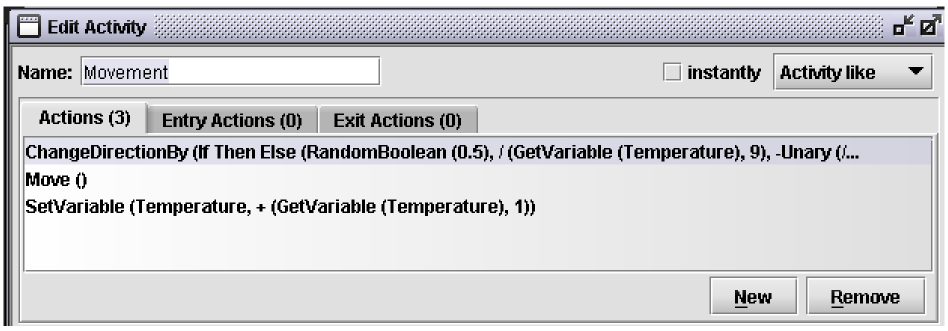


Figure 10: Function specification for a temperature-dependent random change in movement direction

The second action in the `Movement` is a simple `Move` action without any arguments. This action lets the agent move with the stored speed into the given direction. In the third action we have to specify the effects of the movement unto the temperature of the mouse agent. Therefore we have to change the value of the `Temperature`. We make a double click on the action `SetVariable` and select the type `Number` in the proceeding dialog where one has to enter the data type of the variable that one wants to change. The first argument in the `SetVariable` tree is the variable, where we select `Temperature` after making a double click on to the default value "null". The new value is determined by computing the current value of the variable `Temperature` plus 1. Alternatively one could also use the action `IncrementVar` with the variable `Temperature` as first and 1 as second argument. After these inputs the upper part of the activity form will look like depicted in figure 11.



Name: Movement

```
ChangeDirectionBy (If Then Else (RandomBoolean (0.5), / (GetVariable (Temperature), 9), -Unary (/...
Move ()
SetVariable (Temperature, + (GetVariable (Temperature), 1))
```

Figure 11: Actions in the `Movement` activity.

Next we implement the `Sleeping` activity. Again we generate a new "Untitled Atomic Activity", open the *Edit Activity Form* and change the name. While sleeping the agent is not very busy, the mouse just cools down. That means we need one action that decreases the value of the variable `Temperature` by 2 degrees (cooling down is faster than warming up).

This is done in analogy to the increasing temperature of the `Movement`. The resulting window of the `Sleeping` specification looks like in figure 12

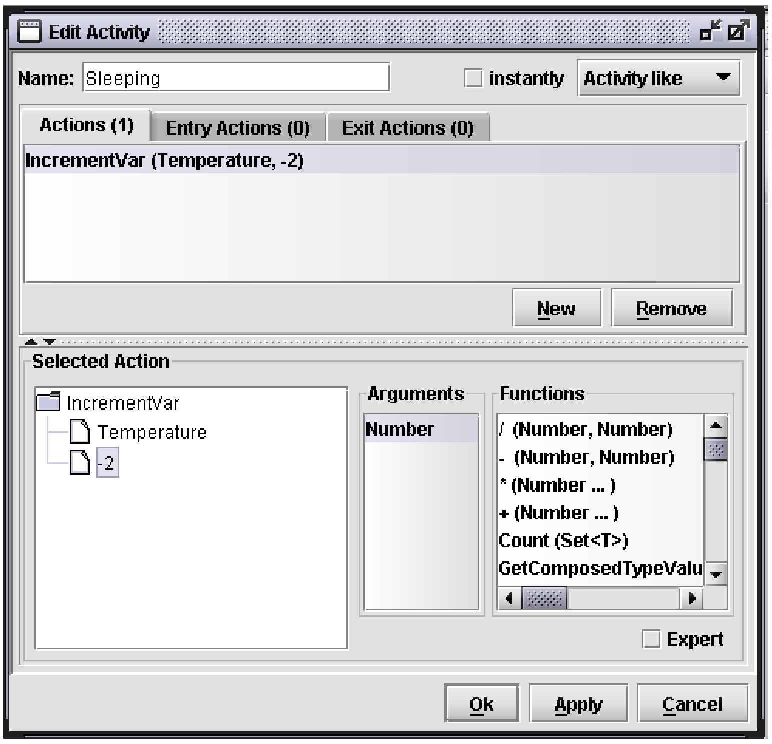


Figure 12: Specification of the `Sleeping` activity.

Now we have specified the activities of our mouse class. However, if we would try to run a simulation now, the mouse would stay in its activity and recompute its speed and nothing else. There are just unconnected activities as depicted in figure 13.

That means, what is now left to do is to connect the activities using rules. We want to specify the following transitions: If the temperature is higher than a certain threshold then stop movement and start sleeping. On the other side, if the temperature is lower than a certain threshold then stop sleeping and start moving. During the movement the speed should be adjusted in every time step to the current temperature. To add a rule, we are selecting the *Add new rule* operator in the operator panel, move the mouse to the starting activity until a green frame around this activity occurs, click and drag - while the mouse left button is pressed - a line to the new activity that should be selected, when the rule fires. To determine, whether you reached this activity, again a green frame is drawn. If the following situation (figure 14, left) occurs, you can release the mouse and a new rule with the condition `true` is generated. This rules always fires. Figure 14 shows this process.

The rule that provides the transition between `Init` and `Movement` is correct with this boolean condition of always true, as it should happen in every time step. The next step is to connect `Movement` and `Sleeping`, for specifying the transition, when the temperature threshold for sleeping is exceeded. We again drag a rule from `Movement` and `Sleeping` and make a double click on the arrow in order to change the condition. A new editor is opened: *Edit Activity Form*. This dialog for a new rule is depicted in figure 15.

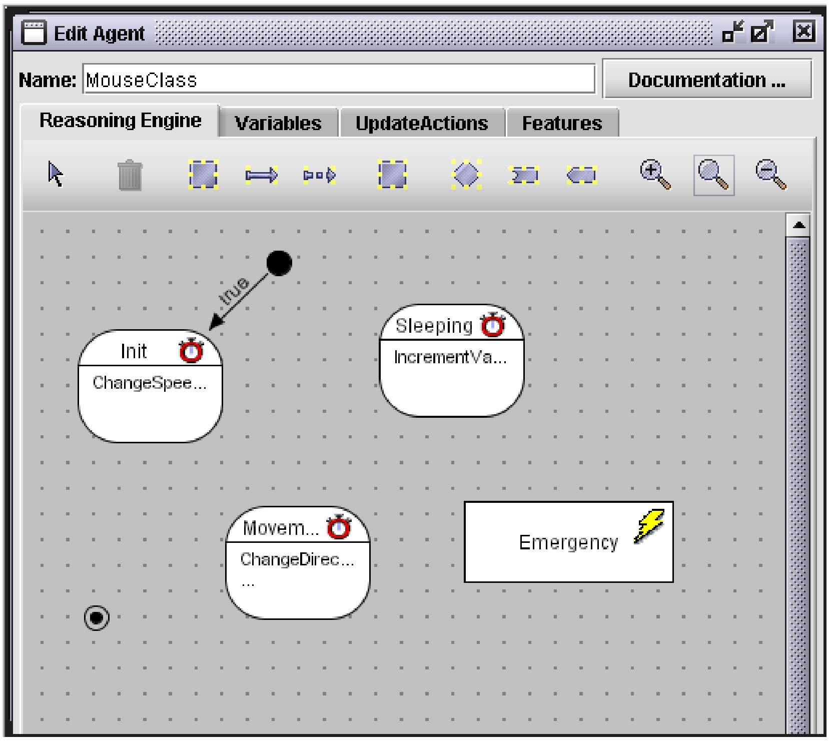


Figure 13: Activity graph without rules.

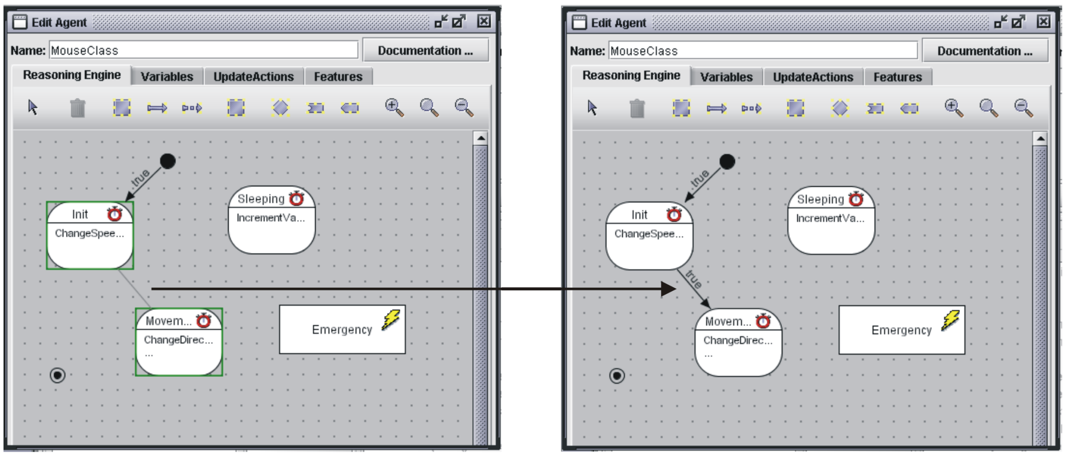
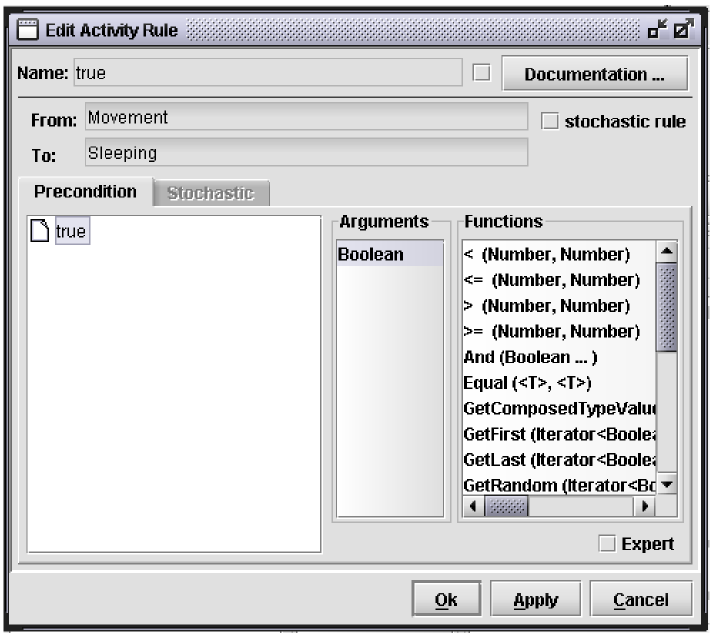


Figure 14: How to generate a new rule.


Edit Activity Rule

Name: true

From: Movement

To: Sleeping

Precondition

true

Arguments

Boolean

Functions

< (Number, Number)
<= (Number, Number)
> (Number, Number)
>= (Number, Number)
And (Boolean ...)
Equal (<T>, <T>)
GetComposedTypeValu
GetFirst (Iterator<Boole
GetLast (Iterator<Boole
GetRandom (Iterator<Bo

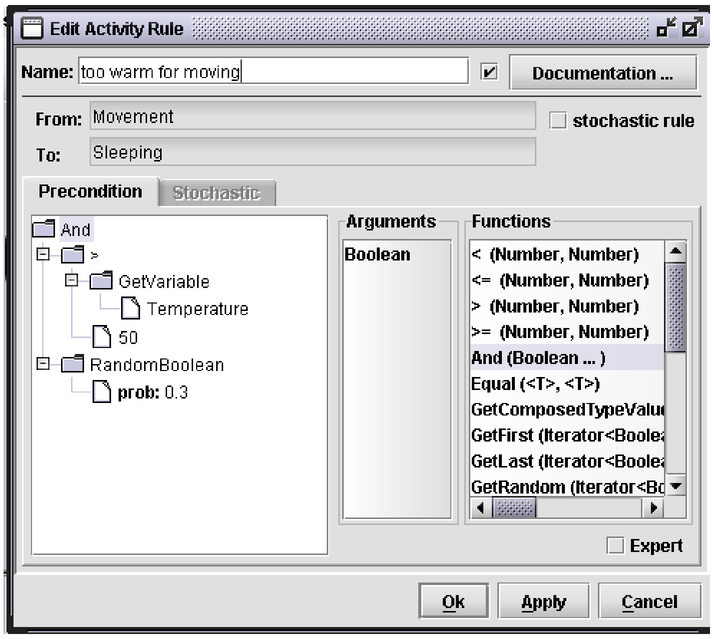
Figure 15: Form for specifying rules.

Editing Rules

After making a double click on a rule arrow in the activity graph a window is opened in which the properties of a rule can be edited. This form contains several fields containing different information:

- *Name*-field contains normally a verbalization of the rule condition. This name is also written at the arrow in the activity graph. Therefore one might want to give a more readable name to the rule. To change the name, one has to mark the checkbox directly to the right of it. After that the *Name*-field becomes edit-able.
- *Documentation* button invites you to input documentary texts to the rule
- Two not writable fields show the origin and destination of the rule. One may interpret them also as an additional condition (The agent has to be currently in the *From:* activity) and the actual action of the rule (When the condition is true then select the *To:* activity as the next activity of the agent).
- *stochastic rule* checkbox enables the *Stochastic* tap. There an additional probability with which the rule may fire can be specified. To use this possibility is unnecessary, as the function `RandomBoolean` allows to formulate stochasticity in a more elegant and transparent way directly in the standard precondition. Therefore, this way of adding stochasticity will be obsolet in future versions of **SeSAm** (later than 1.0.2)
- *Precondition* area that covers the lower parts of the dialog is again a function specification area, like described previously. Here a boolean value has to be computed. If this expression becomes true the rule may fire.

For specifying the transition between the activities `Movement` and `Sleeping` we want to use the following condition: `If GetVariable(Temperature)>50 Then Sleeping with 30% probability`. For implementing this we can adapt the condition to `(GetVariable(Temperature)>50) AND (RandomBoolean(0.3)`. This can be edited directly by making a double click on `And (Boolean ...)`. The first argument is `> (Number,Number)`,... At the end the function specification tree will look like depicted in figure 16, where we also changed the name of the rule to some more readable text.



Name: too warm for moving

From: Movement

To: Sleeping

```
And
  >
    GetVariable
      Temperature
    50
  RandomBoolean
    prob: 0.3
```

Arguments: Boolean

Functions:

```
< (Number, Number)
<= (Number, Number)
> (Number, Number)
>= (Number, Number)
And (Boolean ... )
Equal (<T>, <T>)
GetComposedTypeValu
GetFirst (Iterator<Boolea
GetLast (Iterator<Boolea
GetRandom (Iterator<Bo
```

Figure 16: Specification of the rule that controls the transition from `Movement` to `Sleeping`

Next we enter a rule that ends `Sleeping` and selects again the `Movement` activity. We generate a new rule and change its precondition to `(GetVariable(Temperature)<20) AND (RandomBoolean(0.3)`. The name of the rule can be something like "too cold for sleeping". Opening the newly generated rule may be a little bit tricky as the two rules between the two activity are at the beginning drawn at the same places. You can change this by selecting an arrow and dragging it a little bit to the side until a sufficient space is between the two arrows. Then you may make a double click without any ambiguity.

At this point we have completely specified the mouse class. At latest you should save your model now. The save command can be found in the menu *File* in the menubar. This tutorial comes with some example xml-files. The file with the name "mouseClass.xml" contains the model we have implemented so far.

2.3 Running a First Simulation

Before being able to test our definition of mouse agents we have to define some aspects of their environment and the world they live in.

2.3.1 World Class

A world provides the basic environment the agents are "living" in. A world can also have variables and behavior, defined in form of activity graphs. The role of a world is to determine the outside influences, like abstract weather or processes that generate orders, etc... Until now, our mouse agents do not interact with something from outside. However, for generating a simulation run, we have to define some form of "world", some situation and simulation definition. This is done by making a right mouse click on the item *Worlds* in the *Declaration Tree Panel* on the left (or by choosing *Worlds ...* from the *Model*-Menu in the menubar.). Selecting *New* generates a new, empty world class named `Untitled World Class`. Making a double click on this entry, an *Edit Agent* dialog is opened, where we can substitute the name "Untitled World Class" by an appropriate other name, like "MouseWorldClass". This is all we want to do now with the environment of the mouse agents, just ensuring that there is one. Thus the *Declaration Tree Panel* will look the following way (figure 17

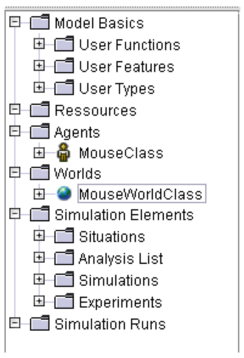


Figure 17: *Declaration Tree Panel* after the generation of a world class "MouseWorldClass"

World Agents

It should not be strange that the title of the dialog for editing a world is *Edit Agent*. A world is able to store variables and to have activity-based behavior. Thus in **SeSAm** the world classes are also agent classes. However, this is an implementation detail, from the conceptual point of view a world is something different from an agent of the agent system. It is quite useful to define global behavioral parameters, e.g. the threshold for freezing as a world variable. Another very useful thing is a world that generates a start distribution of agents and resources. During the simulation, the world cannot only monitor the agents, but generate new resources, new agents or kill them. Sometimes it is also quite useful to shift some aspects of behavior to the world for synchronization purposes. We will see later that there is no update order for the agents, but there is a defined sequence that first the world is updated and then the agents.

Actually we did not need to define a world class for the purpose of simply testing our mouse agents. If we generate a situation then a world class would be needed and automatically generated by **SeSAm**. However, as we want to formulate a more sophisticated environment we took some care for the environment.

2.3.2 Simulation Elements

Before starting a simulation run, one has to define some "simulation elements". In the menubar they can be found in the *simulation* menu.

Simulation Elements

The model elements consist of the description of resources, agents and worlds – both a set of variables as body and a behavior declaration based on activities and rules. For finally running a simulation one has to define some more aspects:

- *Situations* are configurations of a world with given variable settings and some agents that are positioned and configured in a way that a simulation run can be started. One can see it as a starting situation.
- *Analysis List* is the definition of possible instrumentation of the model. One *Analysis* contains data about what selections of data (e.g. number of agents ...) has to be exported to what medium (file, table, graph) and under what conditions (every time step, in given intervals, in certain situations).
- *Simulations* are combinations of a situation, a set of analysis items and a definition of a situation, when the simulation run should be terminated. That means a *simulation* is a complete declaration of a single simulation run.
- *Experiments* are necessary when simulation runs should be automated. Here you may describe combinations of single runs. Based on actions like change parameter of situation in a simulation and run it n times one can generate a form of experimental scripts useful for systematic experimentation.

Finally, only when a simulation run is generated, all descriptions are instantiated and optimized using concepts from compiler theory. When generating a run, you may choose whether what simulation element to use. However, if you want to change something in the definitions, you have to generate a new run for using the changed declarations.

2.3.3 Situation

For testing our mouse agent definition we first have to generate a new situation. This is again done by a right mouse click on the *Situations* and the selection of *New* in the pop-up-menu (if you don't see the new "Untitled Situation", click on the cross mark in front of *Situation* to open the sup-tree below it). A double click on "Untitled Situation" opens a dialog *Edit Situation* like presented in figure 18.

Edit Situation Dialog

In this dialog a situation can be manipulated. The dialog has four main parts after opening it.

- *Name*-field, where you can give a specific name to the concrete situation. To the right of it a documentation buttons allows to enter additional information about this simulation element.
- *Map*-tap view contains the definition of the spatial configuration of the world (only available, when the world class possesses the `SpatialMap` feature).
- Top right to the map there is a list called *Selected Objects in Map*. This list is directly connected to the small positioning square on the map. All objects – resources and agents that are positioned inside this square are listed here.
- Below this list there is an object editor panel called *Selected Object in List* where the object selected in the list above can be manipulated.

Besides the *Map* tap view there are two additional taps: *World*, where you can manipulate the variable settings of this instance of the world class (you even may change the world class) and the *Objects* tap, where you have a list of all objects on the left and a *Selected Object in List* editor panel on the right to change the selected object.

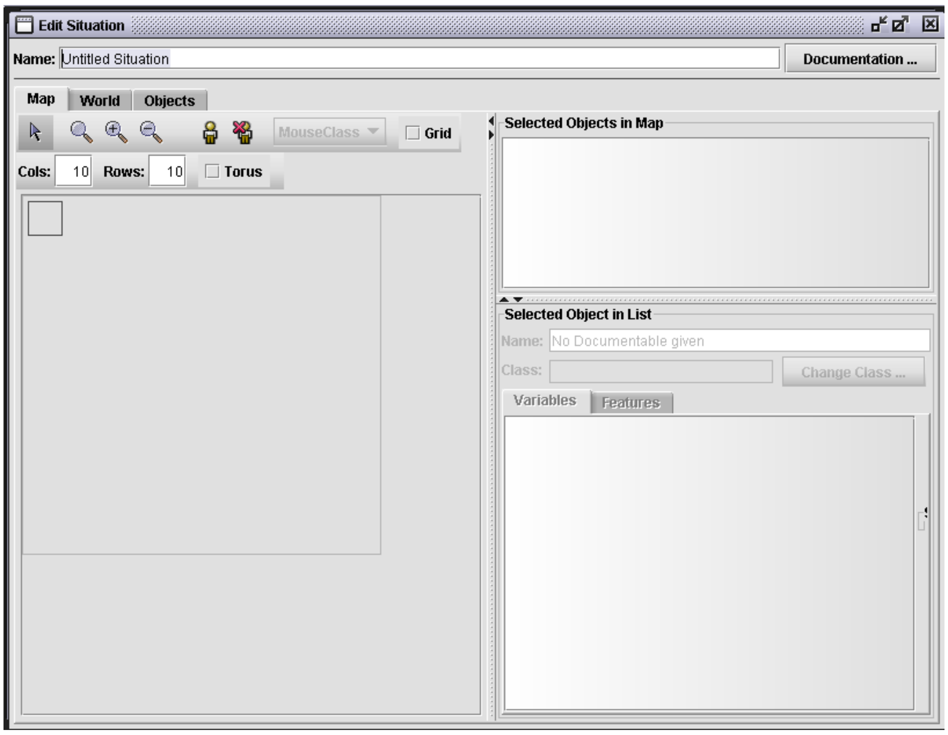


Figure 18: Window for editing a new situation

Manipulating the Map

Extend and Configuration of can be changed in a way similar to the activity graph editor. There are a set of operators and settings, that can be selected and via a click on the map executed. The meaning of the map manipulation options is explained in figure 19.

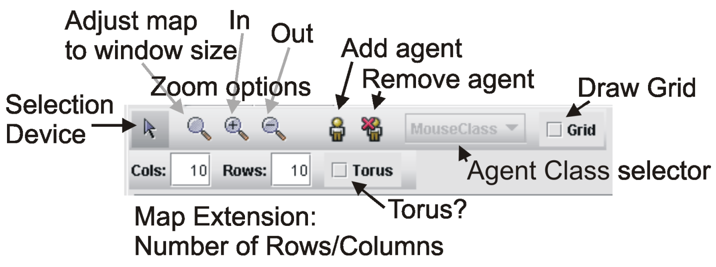
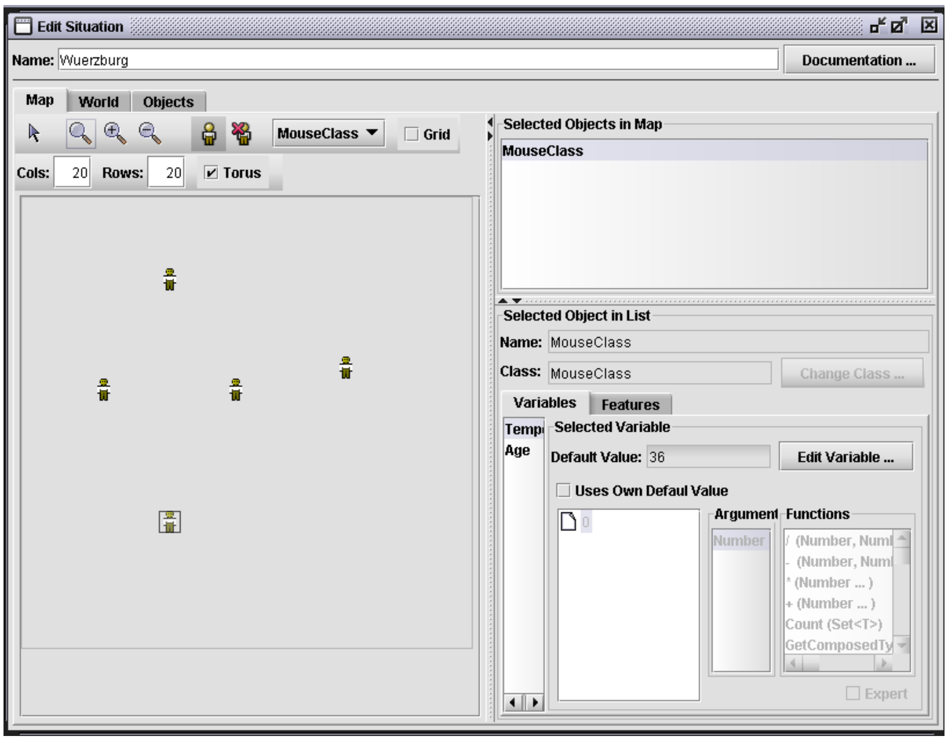


Figure 19: Options for manipulating a map.

These options for manipulating a map are we using now to configure our situation. First we change the name of situation to ”Wuerzburg”. Then we enlarge the map by increasing the number of rows as well as the number of columns to 20 (One click on the icon for the size adjustments changes the zoom in a way that we can overview the complete map without scrolling). We also select the *Torus* checkbox. After that we select the *Add Agent* option – the small man icon. Then the *Agent Class Selector* becomes active. We just have defined one agent class, namely `MouseClass`, thus we don't have any options to change the class of the agents that we will position on the map. Now we can generate agents by making double clicks on some positions on the map. This is what we will do now five times for generating five agents at different positions. After that our situation will look the following way, as depicted in figure 20.



Name: Wuerzburg
MouseClass
☐ Grid
Cols: 20 Rows: 20 ☑ Torus
MouseClass
Name: MouseClass
Class: MouseClass
Temp
Age
Default Value: 36
☐ Uses Own Defaul Value
0
Number
/ (Number, Numl
- (Number, Numl
* (Number ...)
+ (Number ...)
Count (Set<T>)
GetComposedTy

Figure 20: The situation "Wuerzburg" after generating five mouse agents. Notice that the small positioning device is located at one agent. This instance is listed in the object list and may be manipulated in the object editor.

One after the other we select our agents and give them individual starting temperatures between 20 and 50 degrees. This is done by selecting first the agent icon using the positioning square on the map, then selecting the agent in the list and then selecting the variable `Temperature` in the variable list of the object editor panel. The *Default Value* 36 is set, however we select the checkbox *Uses Own Default Value* and the function edit area below this checkbox becomes active. By making a double click on the number 0 and opening the number input dialog we can give the new individual start value for `Temperature` as we like (We choose 10, 20, 30, 40, 50). If you have problems to meet the agents on the map, change the tap view to the *Objects* tap and select the agents directly in the object list that contains all agents. After that we have finished the definition of a test situation. Now we want to run it.

However, you might miss some explanations about the spatial concepts in **SeSAm**.

Space in SeSAm

After defining a map and positioning agents on that map, one might wonder how this discrete world is related to the speed and direction values we have used before. The grid of a map in a situation is just a tool for visualization. The grid has no meaning in the interpretation and simulation of the model. The entries of *Rows* and *Columns* refer to cells, however a single cell in the grid always contains 100×100 points where an agent can be positioned. Thus the maps are discrete, but much finer than the grid configuration seems to. A world of 20×20 cells has actually 2000×2000 possible positions. Speed may take every positive number, not just numbers that are divisible by 100.

Torus

Another interesting feature of maps in **SeSAm** is the possibility to make them to a torus by selecting just one checkbox. A torus means that the world has no edges, but continues on the other side. An agent that moves upwards beyond the upper edge will occur on the bottom of the map. Perception, Movement, etc. is changed automatically due to that configuration.

2.3.4 Simulation Run

Now we want to see how our mouse agents behave in a simulation run. Therefore we generate one by making a double click on the entry *Simulation Runs* in the *Declaration Tree Panel* and selecting the button *New* (or select *New* from the right mouse menu...). As there is only one possibility for generating a simulation run, no selection dialog will be opened, but directly a run generated. This run is called "Sim[Wuerzburg]". The small icon in front of the name shows that this simulation run is currently not running. Making a double click on it the *Show Simulation* dialog is opened for this simulation run. It is presented in figure 21.

The options and taps are the same as in the *Edit Situation* dialog, however information is readable, yet not manipulate-able. In the upper part of that dialog there is information about the current time step, the current status of the simulation and the current number of objects (agents and resources) is given. There are four buttons for controlling the simulation run. *Play* starts a simulation run as long as an end situation is occurring or the user stops the simulation by clicking on *Pause*. *Reset* resets all values and objects to the situation after initializing it. The simulation run can be restarted by hand. An important control is the *Step* with configurable *Step Size*. On the right side of the dialog there is a button *Analysis*, with which you may access analysis graphs, if you define some. There might be a problem when there is a too small number of agents, then the simulation process may ignore events from the user or the animation. The simulation seems to freeze, however in the background the simulation is continuing. Its best just to wait a little time and you will see that the simulation is now simulating some hundreds of time steps later. If you want to test a simple model like the mouse agent, its better using *step*. Selecting an agent and looking at the information given in the *Selected Object* area, you can understand how the agents change their activity and their temperature develops.

This model is saved under the name "MouseClassSimulation.xml".

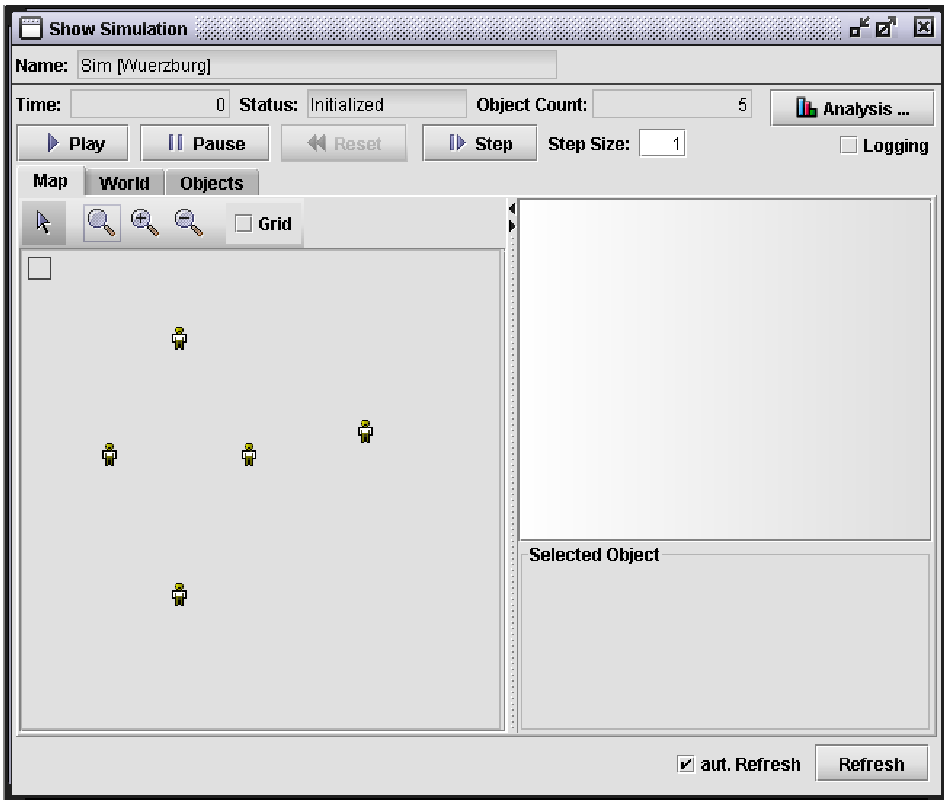


Figure 21: The *Show Simulation* for the "Wuerzburg" situation.

2.4 Two simple improvements

Before going on in extending the model with different forms of interactions we will integrate two small extensions that improve the performance and looks of our mouse agents.

2.4.1 Temperature and Speed

You might notice that a mouse agent never adapts its speed to its internal temperature. This is due to the fact that there is no rule that selects the activity `Init` from the activity `Movement`. It is quite simple to change this. We simple add an *otherwise* rule to connect these two activities (from `Movement` to `Init`). .

Otherwise Rule

Sometimes one may want to formulate something like: $if < condition > then < activity1 > Else < activity2 >$. Basically these are two rules. One rule $if < condition > then < activity1 >$ and a second rule $ifNOT < condition > then < activity2 >$. There there are more than two rules potentially terminating $< activity1 >$ then the formulation of a rule for the *else*-case of all rules is quite sophisticated and expensive. The *Otherwise Rule* is a short way for formulating this overall-*else*-case. If no condition of the other rules is true then the otherwise rule is fired.

In our case, this means, if it is not too warm, so that the mouse starts relaxing, then adapt speed to the current temperature. Thus the name of the activity `Init` is not really apt any more. If you click with right mouse button pressed on the activity node, a pop-up-menu occurs where you can select the item *Rename* and give a new name "AdaptSpeed" to the activity. Now we have another problem. A mouse agent behavior consumes one timestep for one movement and another one for the adaptation of behavior. This is as long ok, as long there are only mice in our world. However, we are now changing the `AdaptSpeed` to an instant activity – this can be done again via the right mouse button menu. The activity stop watch of `AdaptSpeed` is now crossed out. After these changes the complete behavior graph now looks like depicted in figure 22.

2.4.2 Mouse-like images

Until now the mouse agents on the map look like small men as they use the default icon. This is the next thing we want to improve. First you have to draw a new icon, or you can use the abstract mouse icon we prepared. These images are always saved in separate files in the .jpg or .gif format. There are two possibilities, to assign the mouse icon to the agents.

The first is that you set for every agent instance in the *Edit Situation* a specific image file. This is done in the *Selected Object* edit area. If you select the tap *Features* – instead of *Variables* you can change all aspects of spatial information, namely *Position*, *Direction*, *Speed* and the *Image*. For manipulating the latter you have to open a file-select dialog by clicking on the button *Browse...* and e.g. selecting the image file *mouse.jpg*. The result can be seen in figure 23. This you may repeat with every single agent. You also may like to give individual images to the individual mouse agents.

The second possibility consists of integrating the assignment of the image into the mouse behavior. Every mouse agent could execute at the beginning an action that changes its icon. Therefore we introduce a new – this time true – `Init` activity and insert it between the `Start`

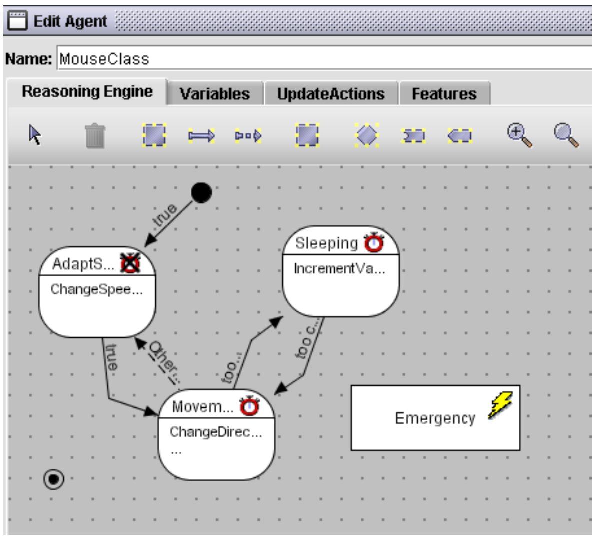


Figure 22: The activity graph of a mouse agent after changes related to the speed adaptation.

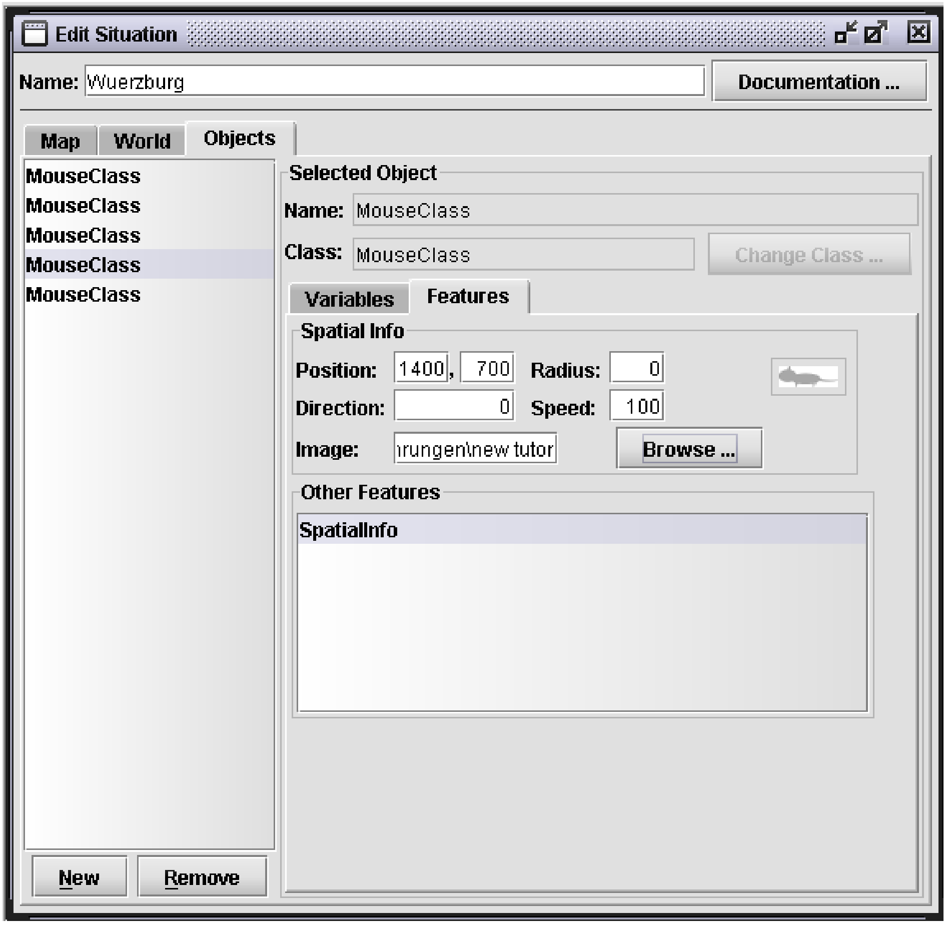
`Activity` and the `AdaptSpeed` activity. Therefore we have either to delete the rule between the `Start Activity` and the `AdaptSpeed` or to re-direct the rule (right mouse pop-up menu and select in the menus *From* or *To* the new correct source or destination). Don't forget to insert also new rule leading from the new `Init` activity to the `AdaptSpeed` activity. After generating the new activity, a double click opens the *Edit Activity.* We adjust the name of the new activity and generate one new action. For specifying this action in the intended sense, select the action `ChangeImage (Image)` in the *Functions* list. The required argument for this function is a image file. By making a double click onto the entry "null" or the item "Image" in the *Arguments* list, an image selection dialog is opened, like depicted in figure 24.

By clicking on the button in the middle of the dialog – here marked with "No Image" – a file select dialog is opened, where you may select the appropriate image file. Then the image is presented as content of the button. If you click again on it, the file select dialog is again opened and you may change your selection. Changing images during the execution of the activities is the appropriate mean for adapting the looks of an agent during the simulation to its internal state. If you want to have a special image for an agent depending on its activity, you have to execute the action `ChangeImage` in the start actions of every activity.

Situation and Behavior Settings

If you choose both options – setting an individual icon in the situation and changing it in the activities, you should not be surprised, if during the simulation the individual picture is replaced by the other. The values specified in the situation are just start values which are overwritten by the changes in the behavior. This is also true for every configured property.

The model with this two extensions is saved under the name *mouseClassExtension.xml.* You may test the adapted behavior of the mouse-like agents in the same way as described above.



Name: Wuerzburg

MouseClass
MouseClass
MouseClass
MouseClass
MouseClass

Selected Object
Name: MouseClass
Class: MouseClass

Spatial Info
Position: 1400, 700 Radius: 0
Direction: 0 Speed: 100
Image: ırungen\new tutor

Other Features
SpatialInfo

Figure 23: Specification of individual images for the agents in the *Edit Situation*

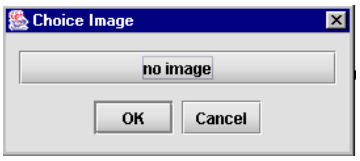


Figure 24: Image select dialog

3 Interactions

Multi-agent models are especially interesting when there are some feedback loops like in predator- prey models or some relations between two of more levels of aggregation like effects of global to local properties. We are now adding some of these features to our mouse agent model. This will show you how interactions may be formulated. At the end of this section there will be given some remarks on interfaces and interactions. We will also sketch some ideas for further extensions.

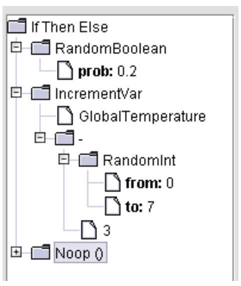
3.1 Between Environment and Agents: Weather-Dependent Activity

We will start with local effects of global properties. The temperature of a mouse is until now just depending on the `Movement` of the mouse agents. We now want to add that the inside temperature is also depending on the outside temperature.

3.1.1 Global World Variable

On page 18 we already generated a new world class and called it "MouseWorldClass". We did not change anything beside the name of this world class. A double click on the entry in the *Declaration Tree panel* on the left opens the *Edit Agent* for the world class. In analogy to the `MouseClass` variable `Temperature` (see page 8) we now generate a new variable and name it `GlobalTemperature`. This global temperature will be accessible for the mouse agents therefore we mark the checkbox *External*. It will also be dynamic, therefore also mark the checkbox *Writable*. Insert a starting value of about 20 degrees. You can - like with the `Age` of the mouse agents, specify the dynamics of this `GlobalTemperature` in the *Next Value* function. However we here try something different, we integrate it into the world behavior.

Therefore we select the *Reasoning Engine* tap and make a double click onto the default `Idle` to open the *Edit Activity* dialog as before. We change the name to "WeatherUpdate" and generate a new action in the *Actions* list. Our global temperature will randomly fluctuate. The value will change with a random number in random intervals. This can be formulated in the following way: With a probability of 20% add to the variable `GlobalTemperature` a random number between -3 and 3. The completely specified action that executes this can be seen in figure 25.



```
If Then Else
  RandomBoolean
    prob: 0.2
  IncrementVar
    GlobalTemperature
    -
      RandomInt
        from: 0
        to: 7
      3
  Noop ()
```

Figure 25: Random Fluctuation of the `GlobalTemperature`

This means: With the probability of 0.2 the condition of the `if-then-else` functions becomes true (`RandomBoolean`) and the `IncrementVar` function is executed. `IncrementVar` has two arguments: the variable to change – here `GlobalTemperature` – and the number that should be added to the current value of the variable. This delta-value is computed by generating a uniformly distributed integer random number between 0 and 7 (the latter is excluded) (`RandomBoolean`). This random number is shifted by decreasing it with 3, so that the random number actually is between -3 and 3. As an action in the only activity of the `MouseWorldClass` this action is executed in every time step and the change in the `GlobalTemperature` happens in the mean in every fifth time step.

This changes can be tested by simulating it and observing the development of the variable in the *World* tap of the *Show Simulation* dialog.

3.1.2 Relation between global and local temperature

We now want to formulate that the inside temperature of the agents is influenced by the global outside temperature. We want to use the following formula for this influence: $\texttt{Temperature} \leftarrow \texttt{Temperature} + \frac{\texttt{GlobalTemperature}-20}{10}$ The formula is chosen in this way so that every degree in global temperature has an effect of a tenth fraction onto to inside temperature. The effect might be positive or negative compared to the normal temperature of 20.

There are several possible ways to implement this interaction between global and local properties. The first thing one has to decide, is, whether the agents should execute the changes or the world should manipulate the agents. If the adaptation of temperature is done by the agents then this can be integrated into one activity or into a dynamic function of the `temperature` variable. Here we want to implement a two step process that mixes these options: We introduce a new variable `PerceivedTemperature` at the `Mouse` agents which is set by the `MouseWorld`. Then we define a `Next Value` function for the inside `Temperature` that implements the formula given above using the `PerceivedTemperature`. This way of implementing costs the generation of an additional variable, yet makes the interface between the global and local temperature transparent. It is a little bit expensive, but clear. There is the possibility to directly access either the global temperature of the inside temperature, these can be seen as sometimes more efficient, however rather low level interactions.

1. New Variable `PerceivedTemperature` at the `MouseClass` You already know, how to generate a new variable for the agent class `MouseClass`. The `PerceivedTemperature` has the type number and must be both, *external* and *writeable*. As a start value you may set the normal global temperature of 20. Thus the definition of the variables of the `MouseClass` looks now as depicted in figure 26

2. External Setting of `PerceivedTemperature` We now change to the `MouseWorldClass`. In the *Reasoning Engine* tap we add a new atomic activity. We open the the *Edit Activity* for this new "Untitled Atomic Activity" and and rename it "InformMice". We mark the *instantly* so that the activity won't consume a time step and the intervals for the changes of the global temperature are unaffected by this new activity. In a new action in the *Actions* list we formulate the following: `For every object in <All Objects of the class <mouseClass>> do: set the value of PerceivedTemperature to the value of GlobalTemperature`. First you have to search for the action `ForElements (Fun<(T) Void>, Set<T>)`. If you don't find it in the *Functions* list then select the *Expert* checkbox below the list, then additional entries in the *Functions* list can be accessed. After making a double click onto this `ForElements ...` you have to specify what type the elements of the set have. We select here the entry

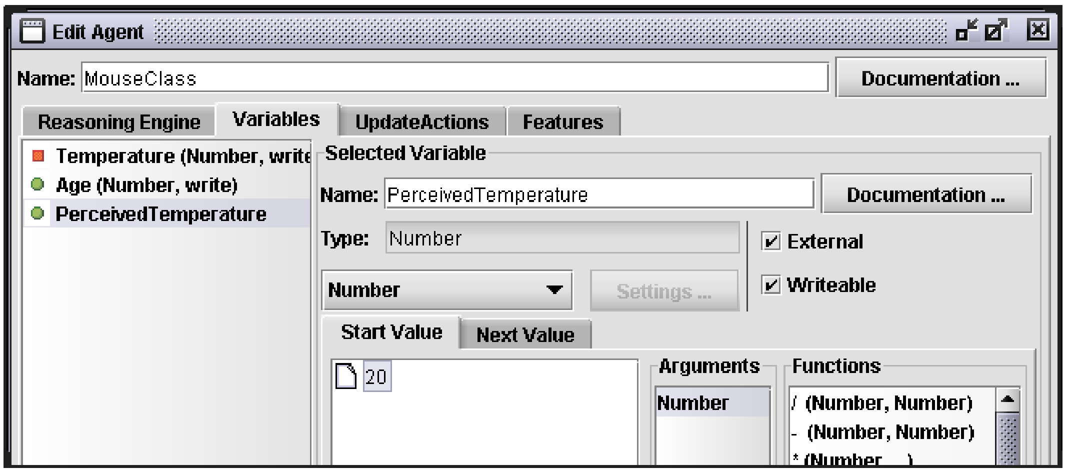


Figure 26: List of Variables of the `MouseClass` after introducing the `PerceivedTemperature`

`Body` (In **SeSAm**-versions later than 1.0.2 select `SimObject`). After that you have to further specify two arguments in the *Function Tree* below the entry `ForElements`. The first one is the function that should be executed with the elements of the set and the second one is the specification of the set itself. You may start with the second one and by selecting it, in the *Functions* list a number of appropriate functions becomes accessible. You select the function `GetAllObjectsOfType (Object Class)`. This function needs the specification of the type of agents that is should collect. Double clicking on either "null" or the "Object Class" entry in the *Arguments* list opens a dialog with which you may select the class `MouseClass`. At this point it is also possible to use `GetAllObjects`, as all agents in our world are mice, however this we will change later and may lead to problems when the instances of the other object classes do not possess this variable. Then we continue with specifying the first argument of the `ForElements`. A double click on this argument opens a function specification dialog, like illustrated in figure 27.

Unnamed Functions

These are functions that are used for example in actions like `ForElements` or `Select` to represent the action that is executed on the elements of a set or the condition that shows which elements have to be selected. These function definitions have no additional name and their definition is not accessible outside of their calling construct. There are specified using the *Create Function* dialog which is always a modal dialog. This dialog contains two parts. The upper part contains the description of the arguments of this function. Besides the names for the arguments nothing can be changed here (it is highly recommended to give expressive names to the arguments). The lower part contains the description of the function call in the same way as we know from for example the *Next Value* call for a dynamic variable or the specification area of actions in activities. A unnamed function is automatically defined in the appropriate way with the correct number and types of input arguments and the correct output type. For example the function executed in `ForElements` has one input argument with the type of the elements of the set that is run through. The output argument is in this case `void` that denotes an action.

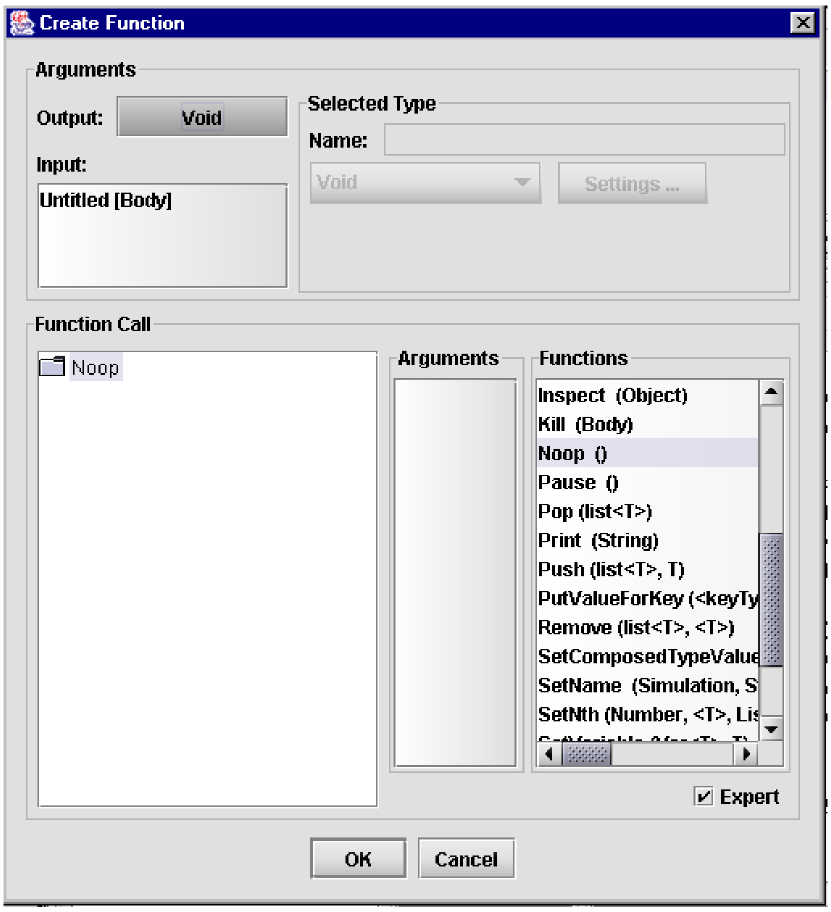


Figure 27: *Create Function* dialog for the function that should be executed on every element of a set of `Body`s.

Here we specify the action that the `MouseWorld` sets a variable at the `MouseClass` agents. We know that the elements of the set – thus the argument of our unnamed function – is a mouse, therefore we can change the name of the single input argument. First you have to select the "Untitled[Body]" and then in the area titled *Selected Type* replace this string by "AMouse" or something related and confirm this change of text by pressing the enter key. Now we continue with the *Function Call* area. We replace "Noop" by the action `SetVariableOf(Var<T>,T,Body)`. After double clicking on this action, a dialog pops up, where you have to select the data type of the variable that should be set. Here `Number` is required. `SetVariableOf` needs three arguments (from top to buttom)

1. First argument is the variable which's value should be manipulated. A double click opens a *Select Variable* dialog which contains a tree of all classes and variables that are marked as external ones. We open the sup-tree below `MouseClass` and select `PerceivedTemperature`. When you don't see the variable you actually want to select, then either the type of the variable to be set is wrong or you did forget to mark the variable as *external*.

2. The second argument is the new value for the variable. Here we select `GetVariable` and specify the variable argument with `GlobalTemperature`.

3. Third argument is the object that possesses the variable that should be changed. Here it is the input argument, as we want to set that variable of all `MouseClass` agents. Therefore we choose `AMouse(Input Argument)` from the *Arguments* list. Sometimes two or more input arguments with the same typ are given, then both are named "Untitled". Therefore, its better to change the name of the input arguments.

Thus the definition of the unnamed function looks after the specifications done like in figure 28.

Closing this function definition with *Ok* the definition of `InformMice` is finished. Confirming this *Edit Activity* dialog you may return to the definition of the `MouseWorldClass` behavior. We now have to integrate this new activity into the graph. We choose the simplest way and generate two rules with conditions that are always "true". One of them makes the transition from `WeatherUpdate` to `InformMice`, the other makes the return transition. This is not very efficient as in every time step the `MouseClass` agents receive a new `PerceivedTemperature` value, even when nothing has changed. The resulting *Reasoning Engine* is illustrated in figure 29

3. Reaction onto the `PerceivedTemperature` For specifying the reaction of the `MouseClass` agents onto the changing perceived temperature we return to the *Edit Agent* dialog for this agent class. We select the tap *Variables* as the adaptation of the inside temperature should be automatic and not depending on same activity. Therefore we define it as a *Next Value* function of the variable `Temperature`. We select the *Next Value* tap and mark the *Next Value* checkbox for this variable. You remember the formula? The new value is determined by $GetVariable(Temperature) + \frac{GetVariable(PerceivedTemperature) - 20}{10}$. This can be directly formulated using the *Funktions* list in this *Function Specification Area*. Thus the complete dialog with the *Variables* tap looks like depicted in figure 30.

This definition ends the specification of the interaction between global and local properties. Our inputs now have to be tested. This can be done again be generating a simulation run

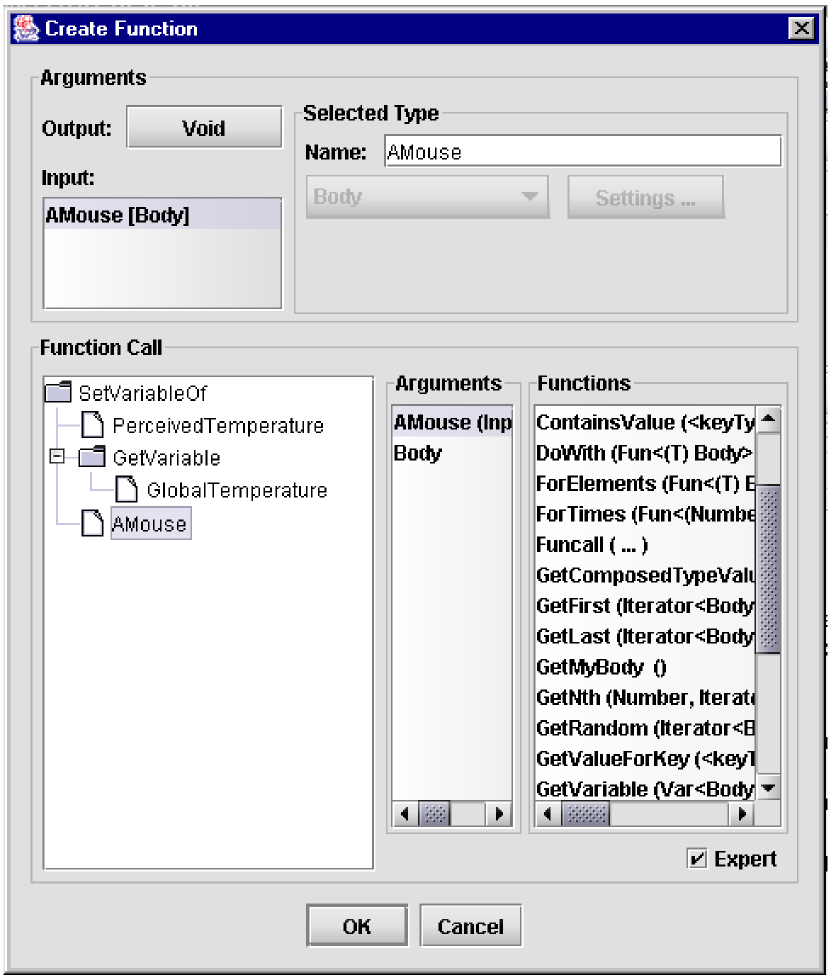


Figure 28: *Create Function* dialog after finishing specification of `PerceivedTemperature` setting.

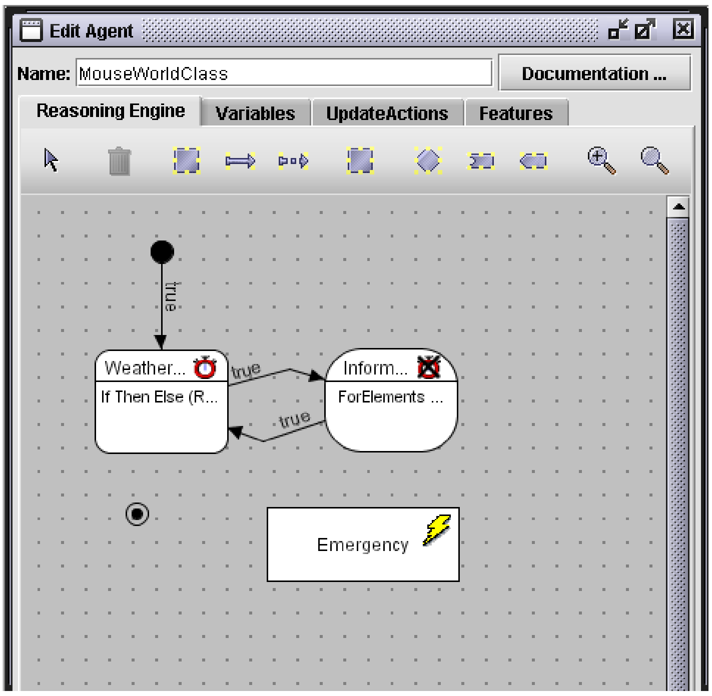


Figure 29: The activity graph of the `MouseWorldClass` after inserting the activity `InformMice`

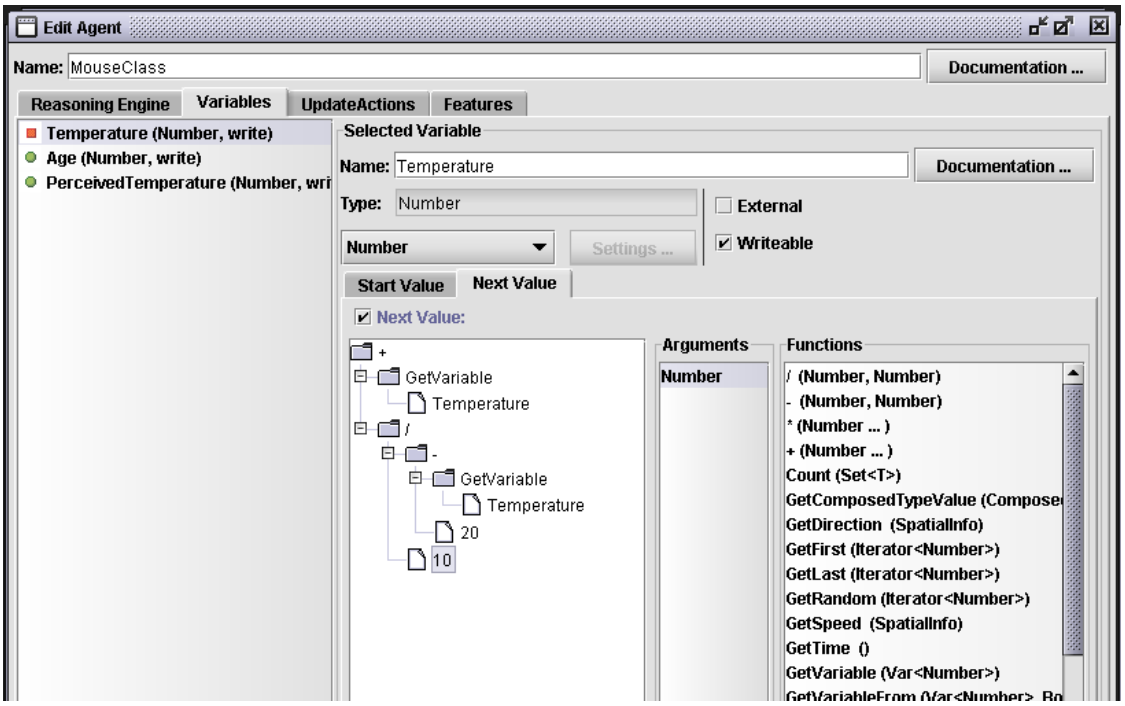
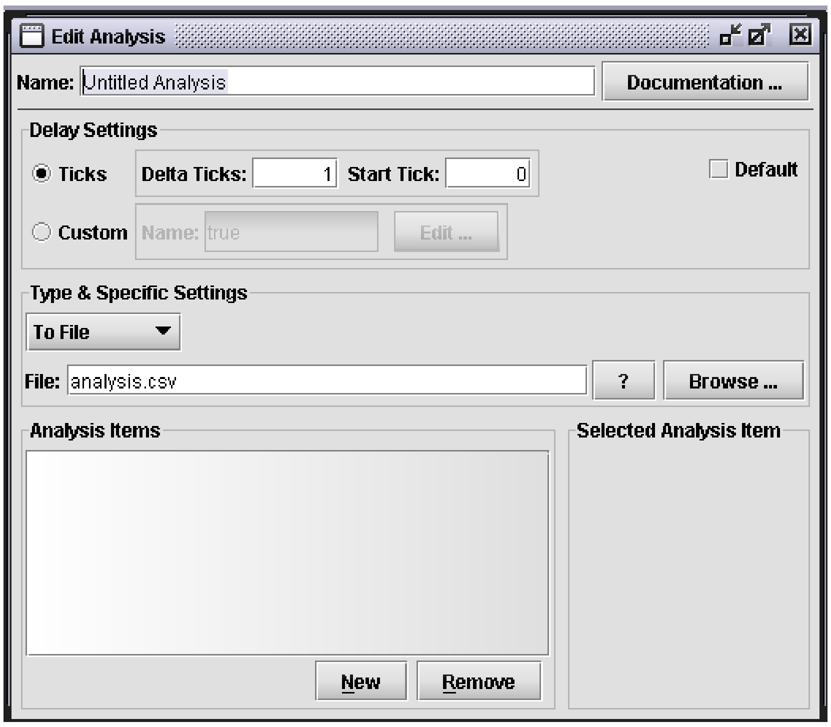


Figure 30: The *Variables* tap with `Temperature` as selected variable showing the definition of the interactive *next value* function.

based on the old situation "Wuerzburg". New variables are added automatically with their default values. However observing all the temperature changes becomes more and more sophisticated. Therefore we now start using the *Analysis* feature.

3.1.3 High Level Observation using *Analysis*

We already learnt about *Analysis List* and *Analysis* on page 20. Now we try to generate a new one by selecting the entry *New* in the right mouse button pop up of the simulation element *Analysis List*. A new *Analysis* named "Untitled Analysis" is generated (if you don't see it at once, open the sub-tree). Double clicking opens the *Edit Analysis* dialog, which is depicted in figure 31.


Edit Analysis

Name: Untitled Analysis

Delay Settings
- (●) Ticks — Delta Ticks: 1, Start Tick: 0
- ( ) Custom — Name: true
- [ ] Default

Type & Specific Settings
- To File
- File: analysis.csv

Analysis Items

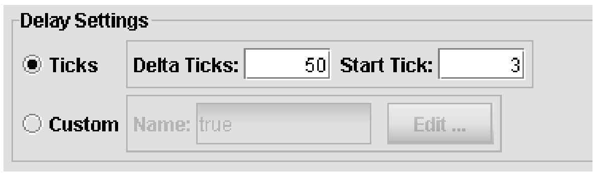
Selected Analysis Item

Figure 31: The *Edit Analysis* for a newly generated analysis.

Edit Analysis Dialog

This dialog for editing instrumentation of a simulation – namely data gathering – contains four parts. The uppermost part is a well-known *Name*-field, where the name of this analysis can be specified. A *documentation* button invites for explanatory texts. The other three parts are the following

- *Delay Settings* refer to the interval in which the data is collected. The standard configuration is that data is treated in every time step without an additional condition. The *Ticks* line refers to absolute intervals with an absolute starting time step. If you want that the analysis updates in every 50th time step starting in time step 3, you have to specify it as depicted in figure 32. The second line in the *Delay Settings* refers to a user defined situation. Every time the condition specified here using the well-known function specification area becomes true, data is collected and the Analysis is updated.
- The area in the middle of the dialog is entitled *Type& Specific Setting*. This refers to the way the Analysis is treating the data. First you have to select between "To File", "Block Chart",


Delay Settings

● Ticks Delta Ticks: 50 Start Tick: 3

○ Custom Name: true Edit ...

Figure 32: Example for definition of analysis update scheme

"Series Chart" and "Table". When the option "To File" is selected you have to give a file name, as the data is written into a file in a semicolon separated form. The first line of the data file are the names of the single analysis items. The first column always is the time step in which the data was collected. The file name may integrate several forms of information form the situation and simulation. This is explained in detail in the *?* button. The other important form of data presentation is "Series Chart". These are curves, that are observable during a simulation run. Here you have additionally to specify the window size of the data that is concurrently visible. "Block Chart" and "Table" just present current values of the analysis items, and thus are rarely used.

- The lowest part of the dialog is the definition of the *Analysis Items.* A can be a column in the file or a single curve of the series chart. This part consists of two sub parts. On the left there is a list of all already defined analysis items, on the right you can edit a selected one. The right part consists of a *Name*-field that forms the title of the item and a *Value Function Call* that is a *Function Specification Area* for specifying a function that returns a value that should be collected under the given name. In the configuration of "Series Chart" or "Block Chart" you may also select a color for the curve or the block.

For actually activating an analysis it has to be assigned to a simulation.

Something like "Series Chart" analysis is what we want to use now for an high-level observation of the temperatures of the world and our agents. This analysis we want to call "TemperatureCharts". The configuration of updating the analysis in every time step is ok here. We select "Series Chart" in the *Type & Specific Settings* area and set the window size to 100. Then the next task is to define the single curves. One item is surely the "GlobalTemperature". The appropriate *Value Call Function* is `GetVariableFrom` with `GlobalTemperature` as first argument and GetWorldBody as second. `GetVariableFrom` is the primitive to read a value from an external variable of another object, here the world. A color is automatically assigned, if you don't like it, click on the color and choose a better one. The next analysis items should be the maximum, average and minimum temperature of our agents. Therefore we have to access the variable `Temperature` that is now not yet possible as it is not marked as external, therefore we have to shortly return to the *Edit Agent* and the *Variables* tap for the `MouseClass`, select the variable `Temperature` and mark it as *external.* After that we can continue in defining new analysis items. Lets go on with the maximum temperature of all `Mouse` agents. Generating a new item and giving it an appropriate name like "Max.Temperature" need not to be explained more. First we select the function `Maximize(Fun<Number,Number>Boolean,Iterator<Number>)`. This function requires two arguments: the first is a function that compares two numbers and returns a boolean. This function denotes the direction of the maximization. We select "¿". The second argument must be a list of numbers. We want to read the temperature value from every mouse agent, therefore we select the primitive `Map(Fun<(T) Number>,Iterator<T>)`. We are asked for specifying the input set of our mapping in a type selection dialog. We choose `Body` (in later version of **SeSAm**, select `SimObject`). The first argument of the map call is a unnamed function (see page 30) that receives a `Body` (type of the elements of the source set) and returns a number (type of the elements of the destination set). We specify the unnamed function by `GetVariableFrom` with the arguments `Temperature` and the input argument for the object which's variable is read. The second argument of the `Map` sub-tree is the set that contains the elements that have to be reduced to numbers. Here we input `GetAllObjectsOfTypes` with `MouseClass` as argument. The specification for this analysis item looks like illustrated in figure 33

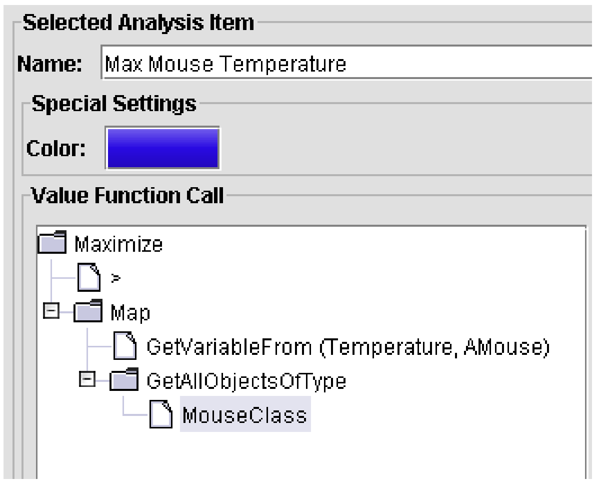

Selected Analysis Item
Name: Max Mouse Temperature
Special Settings
Color: 🟦
Value Function Call
Maximize
>
Map
GetVariableFrom (Temperature, AMouse)
GetAllObjectsOfType
MouseClass

Figure 33: Specification of analysis item value call that returns the maximum value of the temperatures of all `mouse` agents.

The other mentioned analysis item – minimum temperature and mean temperature can be specified in analogy (for the mean temperature use the primitive `Mean` instead of `Maximize`).

3.1.4 Editing Simulation

In order to use this analysis we have to generate a simulation and assign it to it. Therefore we select the entry *New* in the right mouse button pop-up-menu on *Simulations* and open the *Edit Simulation* dialog. This window is depicted in figure 34.

Edit Simulation Dialog

The dialog for editing Simulations gives the possibility to configure a single simulation run description. You may give the run a certain name – like in the *Edit Rule* (Page 17) – after you select the checkbox right to the *Name*-field. There is a name automatically generated using "Sim[" ... "]" and the name of the situation. You are invited to give documentary texts for this simulation run. In the middle of the dialog there is a pull-down-menu that contains all situations previously defined. You may select one or open the *Edit Situation* dialog for the selected situation. About half of the dialog is used by two tab views. One for editing the terminating condition, the so called *End Functions.* Here you may specify a set of conditions (combined using `OR`). If one of these conditions becomes true, then this simulation run is terminated. . Another way of terminating a simulation run is a rule of in the world class that ends at the end activity. The specification of the selected conditions is again done be a *Function Specification Area.* . You may give documentation for any of these end functions.

The second tap view is entitled with *Analysis.* Here you have the possibility to select a set of single analysis like defined on page 35. On the left of this tap view there is a list of all previously defined

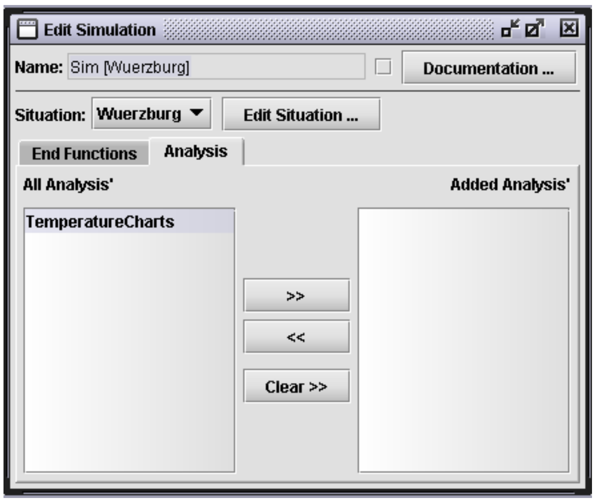


Figure 34: Dialog for combining a situation, termination conditions (”End Functions”) and analysis to the description of a single simulation run. When opening it, the tap *End Functions* is selected. This is the dialog as it looks like after selecting the tap *Analysis*.

analysis, on the right there is the set of selected ones (entitled with *Added Analysis'*). As depicted in figure 34, there are three buttons between these lists. With the upper two you can copy selected Analysis from all to added list, with the other one you may delete selected ones from the added list. The button in the bottom clears all added analysis. You may mix all kinds of analysis, ”to file”, ”series charts”, etc...

Until now we just have defined one analysis, namely `TemperatureCharts` for observing the development of the temperatures during a running simulation. This is what we select now. With the >>-Button you copy it to the simulation description. If we start now this simulation run, the analysis will automatically start with it.

Now we have defined every thing for a simulation run with aggregated temperature information. This we may now generate a new simulation run, like we did for our first tests. After selecting the *New* button or menu entry, a selection dialog opens, as we now have more than one options for running simulations. We may directly run the situation or the defined simulation – that means the situation with defined analysis. The latter is selected per default. This selection dialog is depicted in figure 35.

We select ”Sim[Wuerzburg] ” and open the *Show Simulation* window for this simulation run. Before starting the run by clicking on *Play* we select the button *Analysis*. After that a empty chart with the title ”TemperatureCharts” – the name of our analysis – is opened. In figure 36 this empty chart is shown.

Try what is happening now, when you start the simulation run (if the *Show Simulation* is in the foreground and you cannot see the analysis window, use the *Window* menu in the menubar to select the analysis window again) . In this window you will see, how the mean temperature the mouse develops, etc. You may also see, if you have made any mistakes in the implementation. The curve should look after 100 time steps similar to the one depicted in figure 37 – not too similar according to the completely random global temperature.

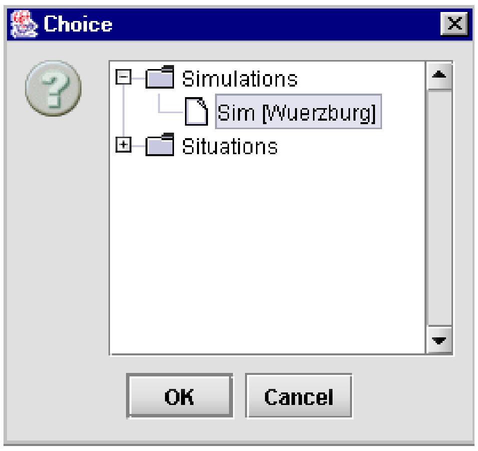


Figure 35: Dialog for selection for preparing a running simulation.

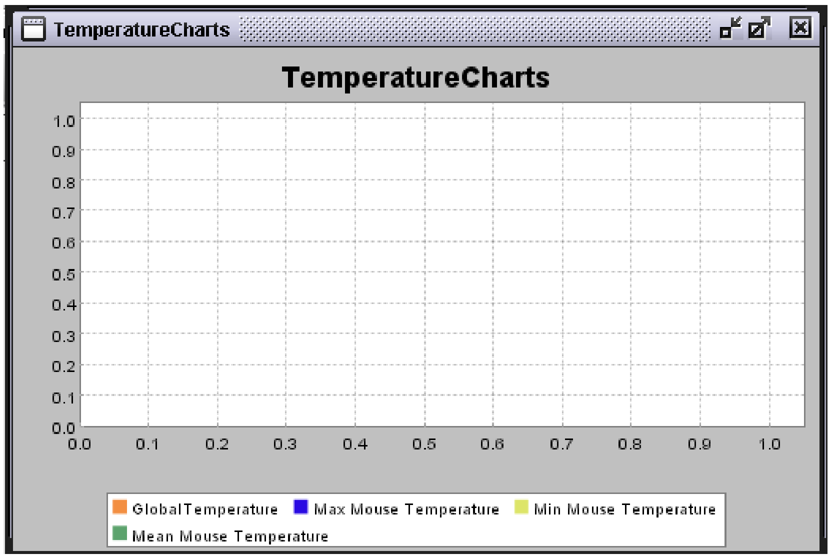


Figure 36: Analysis (Series Chart) opened before starting the simulation (no data available).

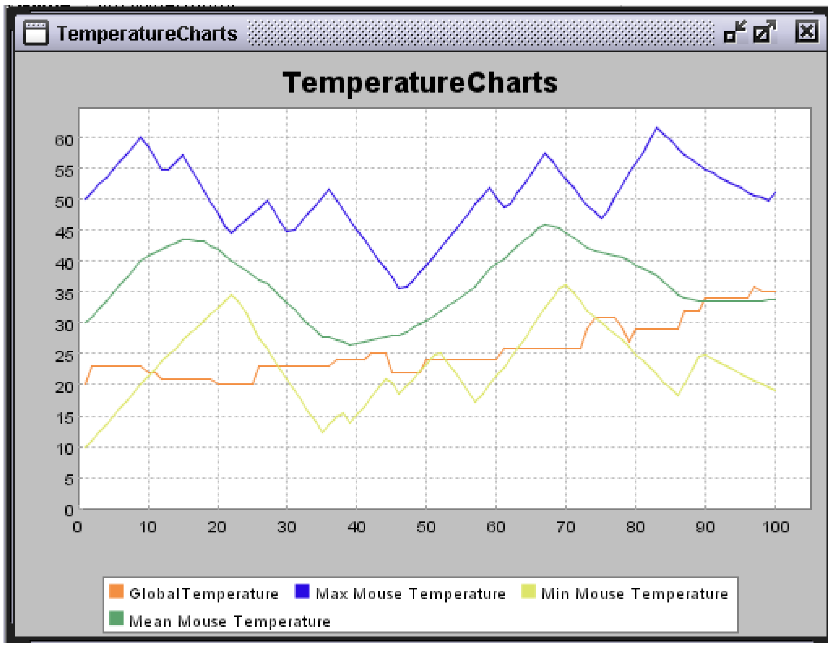


Figure 37: Analysis (Series Chart) after 100 time steps in the model with global and local temperature.

Remember that after any change in any part of the model, including analysis, you have to generate a new simulation run. This model is saved with the name *mouseAndGlobalTemperature.xml*. However, it might be more interesting to track the temperature of individual mouse agents. The changes necessary for it, are left to you on your own. You would need to define a new variable like `Name` and use this name to select the right mouse from the set of all mice for fetching its personal temperature. It might also be interesting to play with the thresholds for moving or sleeping.

3.2 Between Agents and Resources: Cheese Eating and its Effects

Another form of interaction may happen between two local objects. In the following we start by adding `Cheese` resources to our world. `Mouse` agents that encounter a cheese eat it and fill up their energy level. If two `Mouse` agents meet and both have enough energy they will produce a new `mouse` agent.

3.2.1 `CheeseClass` Resources

We start by adding cheese to our world. For this aim we generate a new resource and name it `CheeseClass`. A new resource class can be made in the same way like generating new agent classes. After opening the *Edit Resource* dialog for editing the class, we change the name of "Untitled Resource Class" to "CleeseClass".

Resources

The *Edit Resource* can be – like the *Edit Agent* – opened by making a double click unto the class entry in the *Declaration Tree Panel.* It is pretty much the same like the other dialogs where you may manipulate agent classes or world classes. The main difference is that resources do not possess active behavior, that means they do not have a reasoning engine nor update date functions. Their only form of dynamics are next-value computations in variables.

Our `CheeseClass` will have two state variables: `fat` and `age`. The value of `fat` will be set to a random value at its generation. The `Age` is increased in the same way, also the age of the `Mouse` agents increases. `Cheese` cannot be eaten, if its too old, and will vanish, if its still older. That means we have two additional parameters that are also implemented as variables that represent these thresholds. Of course we also can use absolute values directly in the action definition, like we did e.g. with the threshold for the `Movement` → `Sleeping` transition. However defining in this way the parameter is made explicit (see also section 4.1). This is what we are going to implement next:

- `Fat` is a variable that has a constant value, but must be externally readable, as our mice will consume this fat. The interesting part is the *Start value* that is set to `RandomInt` with the two numeric arguments 0 and 100 for the thresholds between them a uniformly distributed random number will be selected.
- `Age` is defined in analogy to the `Age` of the mice. It has a *Next Value* call that computes a new value that is 1+ the old value (remember you must mark the *Next Value* checkbox before being allowed to specify a next value function.
- `Threshold for eating` is a constant number that is externally readable (our mice may notice whether the cheese is spoilt.
- `Threshold for existence` is also a constant number that must be external, as the would is responsible to erase `cheese` items that are older than this threshold.

After defining these four variables the dialog for editing the resource `Cheese` looks like depicting if figure 38.

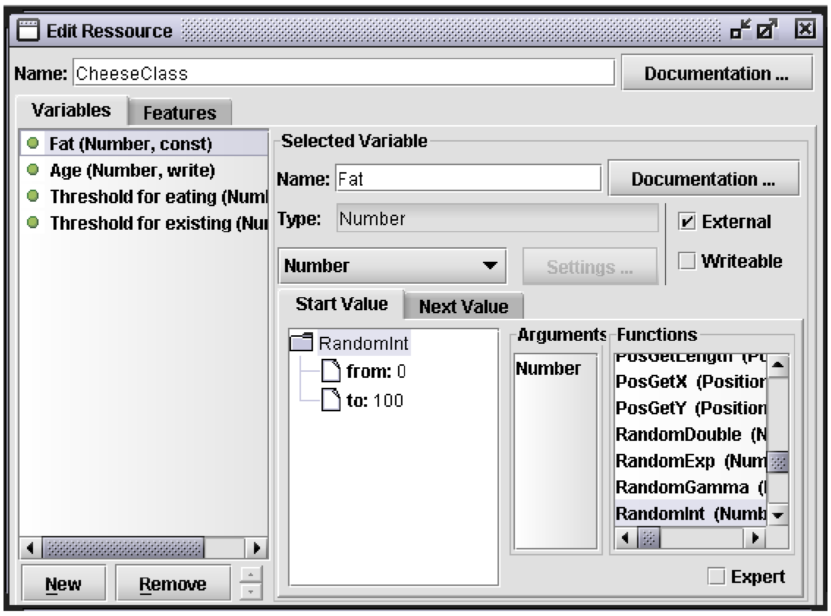
We have now two possibilities to bring `Cheese` objects into our simulation. Either we again distribute them by hand, like we did to test the mouse behavior, or we let the world generate them. As the world also has to erase too old cheese, it is quite obvious to also let it distribute them. This is what we want to do next.

3.2.2 World-managed Resources

As described above we want our world to generate new `Cheese` objects and erase too old ones. Therefore we have to extend the behavior declaration of the world.

Figure 29 illustrates the current configuration of the world activities. Here we have to insert another new activity into the reasoning engine. Of course, you may add the following actions to an action list in an existing activity. However it is a lot more transparent to create new activities for actions that deal with other aspects of behavior than to hide them in existing activities. Thus you may proceed according in the following plan:

1. Generate new activity node using the *New Atomic Activity* option in the behavior graph of the world.


Edit Ressource

Name: CheeseClass

Variables: Fat (Number, const); Age (Number, write); Threshold for eating (Numl; Threshold for existing (Nu

Selected Variable — Name: Fat; Type: Number; Number; ☑ External; ☐ Writeable

Start Value: RandomInt, from: 0, to: 100

Arguments: Number

Functions: PosGetLength (P; PosGetX (Position; PosGetY (Position; RandomDouble (N; RandomExp (Num; RandomGamma (; RandomInt (Numb

☐ Expert

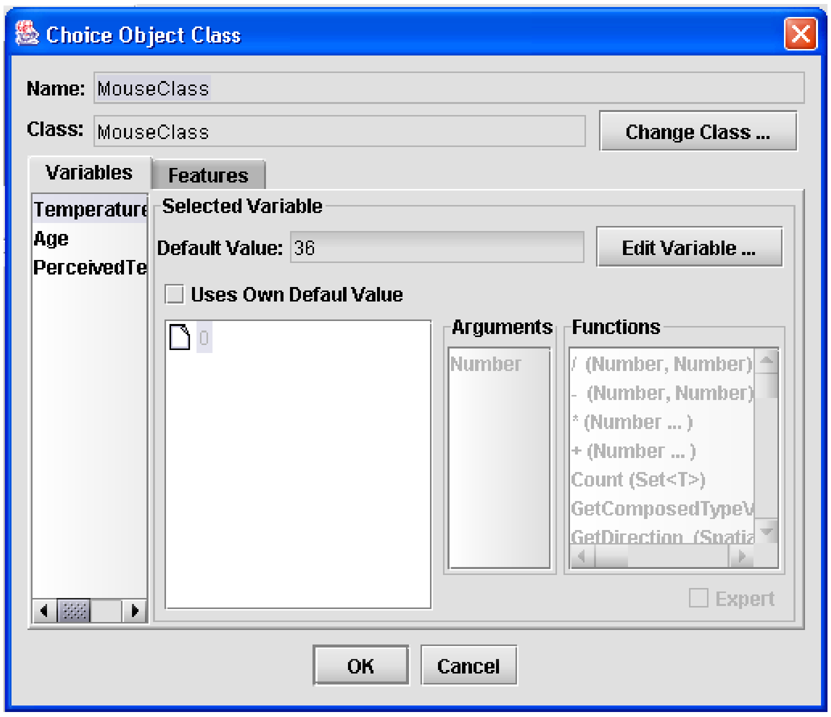
Figure 38: *Edit Resource* dialog for `Cheese` resources.

2. Set activity to "instantly" (e.g. via the right mouse button menu). The stop watch in the node is now crossed out.

3. Open *Edit Activity* dialog for this new "Untitled Atomic Activity" and change the name to something like "TackleCheese".

4. Generate a new action for randomly generating and distributing new cheese objects. This can be done using the following primitives: `If ... then ... else` with `RandomBoolean` as the condition. You may give a rather low probability, however the probability must be adapted to the size of the world map! We choose 0.1 for our small test map. The `then`-part of the `if` construct consists of a `CreateObjectAndRemember` call. with two arguments, namely the class that an instance should be created of and an action that should be executed with that new instance. Double clicking on the class argument the following dialog is opened where class and variable settings of the new object may be manipulated. This dialog is shown in figure 39.

This default selection is wrong here. Therefore we change the class selection to `CheeseClass` using the button *Change Class ...*. As we do not need to influence the starting values for the new instance, we simple close the dialog after that. The second argument – the action – is responsible for setting the object to a random position. If we would not do this, then all new objects would be positioned on the position $< 0,0 >$. We make a double click on the argument and have to edit an unnamed function as in figure 27. The context dependent input argument of this function is the new instance. The world should now re-position the new cheese object. This can be done with the action `BeamBodyTo` that sets another object to a new position – without any movements "by feet". `BeamBodyTo` requires two arguments: the first is the object that should be re-positioned, the second is the new position. For the


Name: MouseClass

Class: MouseClass

Default Value: 36

Figure 39: Edit the class and variable settings during for the generation of a new simulation object.

first we choose the input argument that we have renamed to ”NewCheese”. For the second we have to compute a random position. Therefore the primitive `CreatePos(Number,Number)` exists that creates a position from two numbers that will play the role of the x-coordinate and the y-coordinate, respectively. For these numbers we generate random numbers that range between 0 and the maximum extension of our situation - for the latter we use the primitive `GetMapDimensions`. We have to take either the x-coordinate (`PosGetX`) or the y-coordinate (`PosGetY`). Figure 40 shows how the function tree of this action looks like.

5. Define an action that erases the `Cheese` objects that are too old For this aim we have to look at every single cheese resource, test wether it is too old and if this condition is true, ”kill” it – that means erase it from the current situation. We already know the primitive `ForElements...` from page 29. (Do you remember that `ForElements` is hidden in the *Expert* -List?). The first argument of `ForElements` is the function that is executed with every entry of the list. Make a double click onto this entry and a dialog for a new untitled function is opened like in figure 27. The first change we make in this dialog is to rename the fixed input argument of the type `body` (or `SimObject` in versions later than 2002). Here we rename it to ”Cheese” as all objects that we want to deal with in this function are `cheese` resources. Now we input an `if` primitive with the condition of `>= GetVariableFrom(Age,Cheese), GetVariableFrom(ThresholdForExisting,Cheese),` for testing whether this piece of cheese is elder than its threshold. If this condition is true, then one action has to be executed: `kill` with the argument set to the input argument `Cheese`. The second argument of the `ForElements` call is set to `GetAllObjectsOfType` with

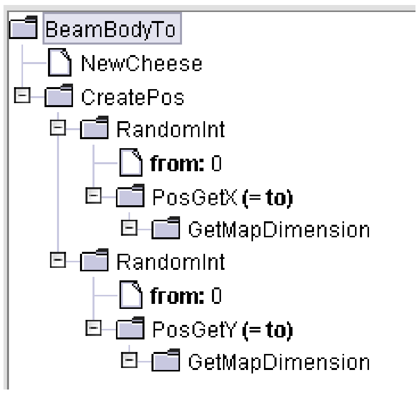


Figure 40: Action for random positioning of cheese

the argument `ChesseClass`. The implementation of this action is done in the same way like all action specifications before, the result should be clear.

Killing and Erasing

There are two possibilities to remove an agent or an object from the simulated situation. Either the world executes the primitive action `kill` on the item that should be removed, or a rule fires in the activity graph that selects the end activity of that agent ("suicide"). However in both of these ways the agent or object is actually deleted at the end of the update cycle! That means, a resource object that is already killed can still be perceived by other agents. You should take this into account. Here, it is not necessary as our `cheese` resources possess a lower threshold for being attractive for hungry mice than for being deleted by the world.

---

6. Connect new activity node to the original activity graph The last step is to integrate the new activity node `TackleCheese` into the behavior graph of the world. For this reason we erase the arrow that goes from `WheatherUpdate` to `InformMice` and draw two new rule-lines: one from `WheatherUpdate` to `TackleCheese` and another one from `TackleCheese` to `InformMice`. All these activities have to be executed in every update cycle therefore it is good to leave the default condition "true". The resulting activity graph is depicted in figure 41.

3.2.3 Mice and Cheese: Interactions between Resources and Agents

Why did we introduce `Cheese` resources? We want our mice to eat them for acquiring new energy. What they can do with that energy will become clear in the next steps. We now will modify the `Mouse` behavior so that a mouse that perceives a cheese piece moves towards it and eats it, if it is young enough to be still eatable. For this reason we generate two new activity nodes in the same way as we've already done before. One is called "RunToCheese" and the other "EatCheese". Before we add new connecting rules, we also generate a new variable for our `MouseClass`: "PerceptionRadius". This is a constant (not writable), not

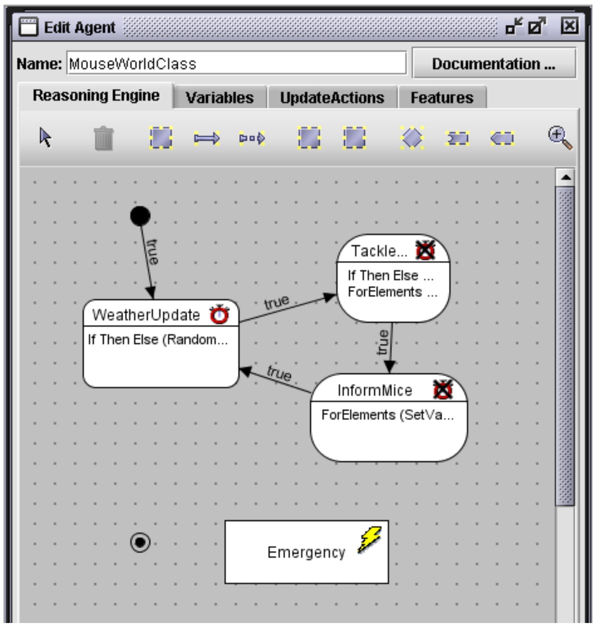
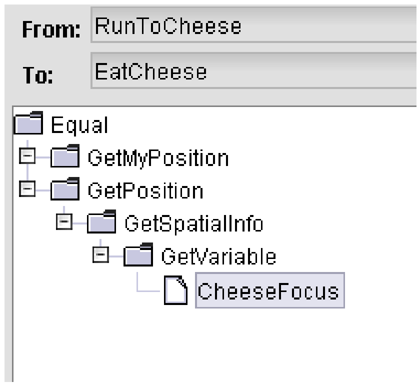


Figure 41: World activities with node for the treatment of cheese pieces

external variable with the type `numeric` and the start value of 200. After that we may add the following connections between our new activities and nodes of the old activity graph:

- Rule between `Movement` and `RunToCheese`: The random movement should terminate when a mouse perceives a cheese object in a distance less than its `PerceptionRadius`. After double Clicking on the new arrow between these two nodes, formulate the following conditions (using the primitives in brackets): the rule should fire, if the list of `CheeseClass` objects - selected (`Select`) from all – within the `PerceptionRadius` – perceivable objects (`ObserveObjectsOnPosition`) is not empty (`Not` and `IsEmpty`). You should start with "Not(IsEmtpy....)".

- Rule between `RunToCheese` and `EatCheese`: A mouse is able to eat only if it is on the same position as the cheese object. This is the rule that should go from `RunToCheese` to `EatCheese`. We have several possibilities: Either we use something analogous above – There is a `CheeseClass`-Object within a very small distance (using `ObserveObjectsOnPosition`) or we implement something like a focus on a certain Cheese object and test whether this is on the position like the mouse. I prefer the second possibility. It is more efficient, as the spatial perception primitive is quite expensive. A more important reason is that – if there are more than one cheese objects or if the cheese could move, the cheese would not get out of sight or the cheese object that the mouse is running to, changes. An advantage of the first way of formulating would be that the movement towards a cheese object stops when any is on the same position and not a special one (this would be important if the cheese could move...). As our cheese is not able to move (and we can focus on the cheese object that is the nearest to our mouse), we use the second way. Therefore we have to define a variable `CheeseFocus` that is writable and of the type `Body` (`SimObject`). After that one can use this variable in the condition of the intended rule testing if my position (`GetMyPosition`) is equal (`Equal <Position>`) to the position of the object in `CheeseFocus` (`GetPosition ( GetSpatialInfo ( GetVariable CheeseFocus)))`. The function specification tree should look like depicted in figure 42



From: RunToCheese
To: EatCheese

```
Equal
├─ GetMyPosition
└─ GetPosition
   └─ GetSpatialInfo
      └─ GetVariable
         └─ CheeseFocus
```

Figure 42: Rule Condition: Comparing my position to the focussed cheese position

- The last rule connects the `RunToCheese` with the random movement `Movement`. This should be always the case, that means that the condition is the default `true`.

Spatial Perception

The primitives `ObserveObjectsOnPosition` and `ObserveObjectsInDirection` are the two central function for spatial perception. `ObserveObjectsOnPosition` provides a list (iterator) of `Bodies` (`SimObjects` after **SeSAm**-version 1.0.2) that are positioned around the specified position (1. Argument) in a maximum distance specified by the second number. This number is in points (not in cells – see page 23). The third argument is a boolean flag that informs whether the percieving agent itself is perceived or not - that means, if the perceiving agent will be returned in the list or not. `ObserveObjectsInDirection` is quite analogous. It requires five arguments:

1. *Position* specifies the start "null" position of the perception.
2. *Direction* specifies the direction of the perception.
3. *Angle* refers to the opening angle in degrees. An angle of 360 refers to an all-around sight.
4. *Radius* is the maximum distance of the perception
5. *Including Me* again denotes whether the perceiving agent should be able to perceive itself.

After naming and connecting the activities for cheese eating, we have to specify what is actually means to run to a cheese object or to eat one. We start with `RunToCheese`: As this activity is only selected when there is cheese observable, we can start with focussing the cheese object that is the nearest to our mouse agent. We set the action of setting the focus variable on that resource object as the start action list. We use the `Maximize` construct for determining the nearest cheese object of all perceivable ones. The `Maximize` primitive returns the entity of the given list (second argument) that maximizes the specified function (first argument).

3.2.4 Producing new agents – and other ”Emergencies”

3.2.5 Evolving the decision to use energy for cooling down...

3.3 From Mouse and Cheese to Foodweb: Adding Cats

3.3.1 The energy feature

3.4 Complex behavior: The simulated human mouse-protector

3.4.1 Data types

3.4.2 Composed Activities

4 From Testing to Systematic Experimentation

4.1 Preparing the model

4.2 Defining Experiments

4.3 Debugging and Calibration

4.4 Remote Control

A Variable Type List

The following list contains all data types available in **SeSAm**.

- `Activity`: an activity object as specified in the reasoning engines
- `Body` – 1.1.2003 replaced by `SimObject`: an simulation entity, can be an agent, a resource or a world object.
- `Boolean`
- `Composed <T>`: a user defined composed data type. It must be specified, which one.
- `Enum <T>`: a user defined symbolic enumeration data type.
- `Hashtable <T,T>`: Traditional hashtable data type. The types of the keys and the values that are stored in the table have to be specified.
- `Image`: a picture that is used for showing an entity in the animation. Variables of this type contain a file name of a picture in .gif or .jpg format.
- `List <T>` (formerly `set`): a list of entries of the same type - that has to be specified.
- `Number`
- `Position`: Position on a world map
- `String`

Index