MIMOSA user's manual
(Draft version 1.2.1beta)

Jean-Pierre Müller[1]
CIRAD-ES-GREEN
jean-pierre.muller@cirad.fr

January 29, 2008

[1]Associated researcher to LIRMM

Contents

List of Figures

Chapter 1

Introduction

Mimosa[1] is an extensible modeling and simulation platform ([8]). It is aiming at supporting the whole modeling and simulation process from the conceptual model up to the running simulations.

The modeling process is assumed to be constituted iteratively of the following stages:

The conceptual modeling stage: it consists in elaborating the ontology of the domain as a set of categories, their attributes and their relationships, either taxonomic or semantical.

The dynamical modeling stage: in order to describe the dynamics of the categories defined in the first phase, one must decide on the choice of paradigm (differential equations, straight scripting, agent-based, etc.) for each category. The paradigm is described using a built-in meta-ontology. Given the choice of dynamical paradigm, one must specify the possible states and state changes according to the chosen paradigm.

the concrete modeling stage: the previously described stages define the vocabulary in which the concrete model(s) can be described as a set of individuals linked to each other and with given attribute values.

the simulation specification stage: apart from the structure of the model to simulate as described in the previous stage, an important work consists in deciding which attributes can be considered as fixed parameters, which ones can be manipulated by the user, how to output the states of the model (plots, grids, databases, statistical tools, etc.)

the simulation stage: it consists in running the simulations themselves by creating the simulation model to run as a set of entities linked through ports by connections, by associating the means to specify the input parameters and to handle the outputs of the simulations and by actually simulating it.

In the following, we shall describe these stages in turn. At the time of writing, only the simulation specification stage is missing. Consequently, only the other

[1] It is the french acronym for "Méthodes Informatiques de MOdélisation et Simulation Agents": computer science methods for agent-based modeling and simulation

stages shall be described in detail. But before, we shall shortly introduce how to run the system.

Mimosa is also implemented to be multi-lingual. For the time being only english and french are provided; spanish is coming soon. Even if the user's manual is only in english, do not be surprised if the user interface appears in the language of your operating system (or in english if the corresponding language is not available). Most of the explanations still apply even if the menus and titles are not the same.

Chapter 2

Running Mimosa

2.1 Downloading Mimosa

Mimosa is a free software under LGPL license and CIRAD copyright. The source and code is available on SourceForge.

If you are only interested in the program itself, you can go to the Mimosa site on SourceForge: `http://sourceforge.net/projects/mimosa`. You just have to follow the link "dowload" to go to the page where you can download the software. How to run it is explained in the next section.

If you are interested by the software itself (or even want to contribute), feel free to access it via the CVS server at:

`pserver:anonymous@mimosa.cvs.sourceforge.net/cvsroot/mimosa`.

The latest version is currently under the branch: `version2006-04-19`. The tagged versions `version101beta` and `version110beta` can be dowloaded but, of course, are not fully up to date. If you want to be a developer, just create an account on SourceForge and send a message to:

`jean-pierre.muller@cirad.fr`

to give the name of your account and to explain what you want to do.

2.2 Launching Mimosa

Mimosa is written in Java 1.5 and can be run on any platform (both hardware and operating system) as long as at least Java JRE 1.5 is installed.

For the time being, Mimosa is provided as a folder containing:

- mimosa.jar which is the main program to be launched by typing: `java -jar mimosa.jar` or by double-clicking on it if your OS has Java integrated in it.
- a `libs` folder containing the libraries necessary for running Mimosa.
- an `example` folder containing some examples to load within Mimosa for exploring its functionalities.
- a `documentation` folder for the documentation (it should be soon or later a user's manual (this one), a programmer's manual and the full javadoc hierarchy).

- a `plugins` folder contains so-called plugins which are either hard-coded examples or additional dynamical specification paradigms. Most of Mimosa is assumed to be sooner or later distributed in this form.

When launching Mimosa, a first window is opened with an editor for conceptual modeling (see 3). The next section describes the menus in detail.

2.3 The menus

Three menus are provided:

File: this menu provides access to all the functionalities related to the window which is active or to open new windows:

New: is used to open any of the following new windows:

Conceptual model editor: opens a window for editing conceptual models;

Mereology editor: opens a window for editing more sophisticated conceptual models (in particular with whole/part relationships, what mereology is all about!). Currently, it is not yet fully operational;

Concrete model editor: opens a window for editing concrete models (as instances of conceptual models);

Scheduler: opens a scheduler control window for running the simulation models.

Open...: loads the content of a file depending of the selected window. The file must contain an appropriate XML representation. The kind of content which can be loaded depends on the active window. If it is a conceptual model editor, only a saved conceptual model can be loaded. If it is a mereology window, only a saved mereological or above model can be loaded. If it is a concrete model editor, only a saved concrete model can be loaded. In the last case, be sure that the conceptual models used by the concrete model have been loaded beforehand. Finally, if it is a scheduler window, only the XML files especially generated from the concrete model editor for this purpose can be loaded[1].

Save: saves the model currently edited in the active window in the associated file (the last file it was saved to). If it was never saved before, a file chooser dialog opens.

Save as...: saves the model currently edited in the active window in a file to specify regardless of the last save (or open).

Print...: prints the content of the current window if applicable (it is applicable when a graph is displayed).

Restore...: this item is only used if you defined a new meta-ontology in a so-called plugin and you want to dynamically reload the plugins definitions for further use without relaunching Mimosa.

[1] This possibility is provided to create stand-alone models without the associated conceptual models.

Edit: this menu provides the contextual editing functionalities provided for the selected window or object. Any editor provides at least the following functionalities in addition to the usual cut, copy and paste:

- **Add:** to add a new object (categories, individuals, states, etc.);
- **Change:** to change the name of an object when there is an associated name;
- **Edit:** to edit the structure of an object (the structure depends on the object and, sometimes, includes the associated behavior description);
- **Delete:** to remove the selected object(s);
- **Delete all:** to remove all the defined objects.

Window: this menu provides quick access to the opened windows. One of these is always accessible even if not shown by default:

- **Output:** to display the output window which is a console containing: a panel for user specific output, a panel where error are displayed and a panel where the traces are displayed.

Help: this menu gives access to a number of tools for debugging:

- **Statistics:** displays in the output window some statistics about the data structures used by the scheduler: number of created entities and usage of the influences;
- **Predefinitions:** displays in the output window the predefinitions as defined in the scripting mechanism;
- **Show content:** displays in the output window the content of the tables created by the various editors which are the data strcutures behind the scenes;
- **Script interpreter:** displays a window in which the user can enter expressions in any of the provided scripting languages in order to test the code. The results are displayed in the output window when pushing the `eval` button.

2.4 The editor windows

Each editor window has the same structure (see the figure 2.1). It is divided in two vertical panels.

The left panel contains the list of existing models (either conceptual or instantiated) referenced by their names. In Mimosa, these models are also referred to as ontologies. One can select an existing model (in the model editor) or ontology (in the ontology editor) by left-clicking on its name. By right-clicking on the panel, one accesses a popup menu where it is possible to add a new ontology, change its name or delete it. It is highly recommended to create a new ontology each time one is describing a different structure for modularity and reuse reasons.

The right panel is editor specific and usually allows multiple views of the same ontology or parts of it. In most cases a graphical view is provided. In the

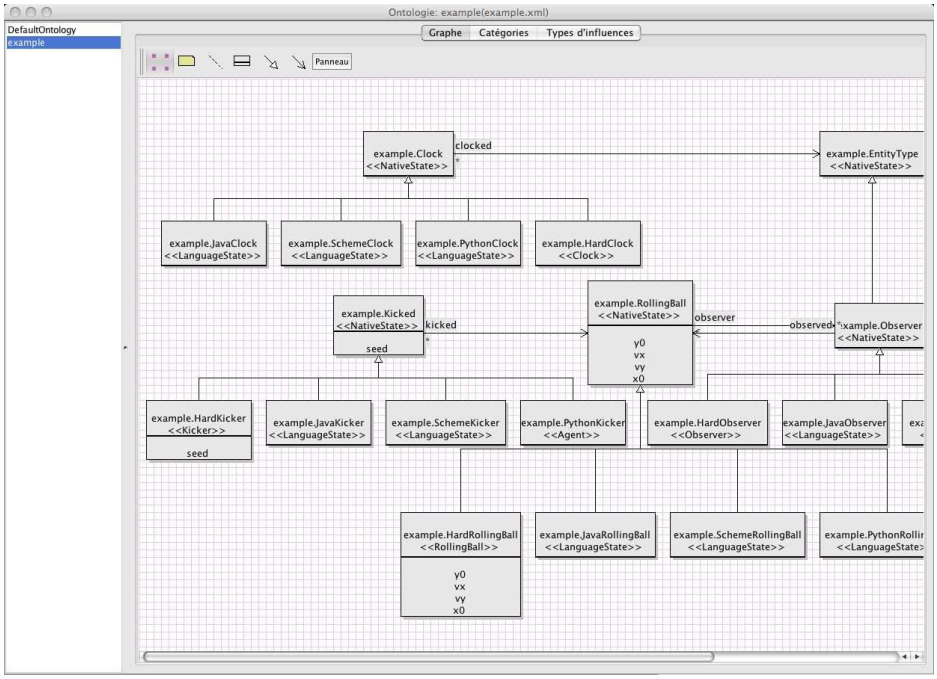

Figure 2.1: The conceptual model editor as an example of an editor window

figure 2.1, there are three editor panels. The shown one is the graphical editor panel. The other two are used to edit categories and influence types (see 5.2.3) as lists. The figure 2.2 shows the list editor where it is also possible to add, change the name, edit and remove categories.

On the top of any drawing view, there is a toolbar with a number of model specific buttons. These buttons are specific and shall be described in the related chapters. These editing buttons are also available as a popup menu when right-clicking in the drawing area. The last button is a drop down menu to manipulate the editor window (zooming in and out, reducing, enlarging or hidding/showing the grid for objects alignment). The figure 2.3 shows the buttons for editing mereological conceptual models.

Any created object can be edited by double-clicking on it. On right-clicking on an object, one can access a popup menu for editing (same as double-clicking) or deleting the object.

2.5 The scheduler window

On the top of the scheduler window (see 2.4), the list of existing models is provided for inclusion within the list of available models to the scheduler. It is also possible to add additional models to simulate by loading them from scheduler specific files. This possibility is used when delivering turn key models.

The bottom of the scheduler window is divided in two vertical panels.

In the left panel, there is the list of existing models (as added from the model

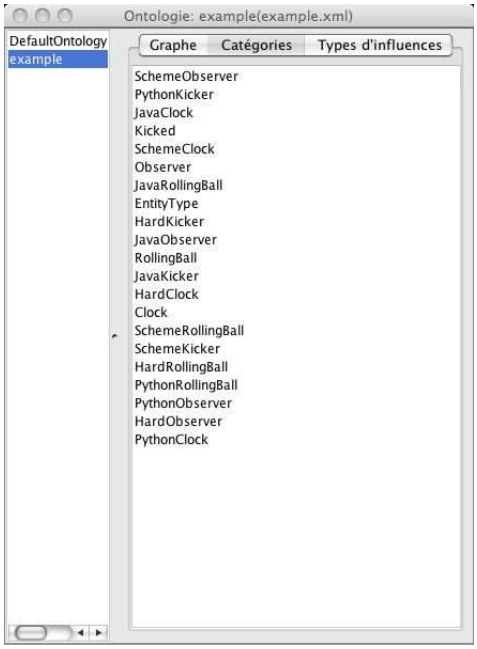

Figure 2.2: The category list editor of ontologies

Figure 2.3: The graphical editor buttons

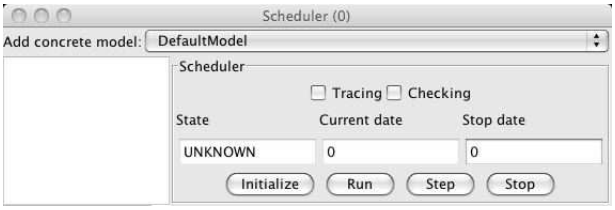

Figure 2.4: The scheduler window

editor or from files). Exactly one model must be selected to be run.

The right panel is divided in three horizontal panels:

1. the top panel has check boxes for debugging:

   Trace: to turn tracing on and off. If the trace is on, the influences posted and sent are displayed in the trace window.

   Verify: to turn verifying on and off. If the verify is on, all the declarations (names, types and cardinality) are checked during simulation. It slows down the simulation quite a bit but it is very useful for checking whether the behavior is consistent with the declarations.

2. the middle panel displays the state of the simulation (unknown, initialized, running or stopped) and the current date (in global time). An end date can be entered to specify when to stop the simulation. The core simulation system being event-based, this is NOT a number of steps but really an end date.

3. the bottom panel has buttons for controlling the simulation:

   Initialize: for putting the model in its initial state. The current date becomes always 0.

   Run: to run the simulation until the provided end date is reached. If the end date is les or equal to the current date, nothing happens.

   Step: to run one cycle of the simulation. All the influences scheduled at the next date are executed.

   Stop: to stop the simulation before the end date is reached. The current cycle is always completed (and cannot be interrupted).

Each scheduler window is associated to its own thread, so there is a possibility of having several scheduler window opened to run several simulations simultaneously.

Chapter 3

The ontologies

In modeling and simulation, the structure is often understood as a composition of models, each model computing a function to produce outputs (outgoing events or values) from inputs (incoming events or values). Of course, this composition reflects the structure of the system one wants to model but no discourse on how to describe a system structure is explicitly given. On the other hand, Artificial Intelligence has focused part of its theories on how people describe the reality. This part of Artificial Intelligence evolved, partly under the pressure of the web developments (both about its contents and its services), into what is called today the description of ontologies.

The term *ontology* has its origin in philosophy, where it is the name of a fundamental branch of metaphysics concerned with existence. According to Tom Gruber at Stanford University, the meaning of ontology in the context of computer science, however, is "a description of the concepts and relationships that can exist for an agent or a community of agents." He goes on to specify that an ontology is generally written, "as a set of definitions of formal vocabulary".

Contemporary ontologies share many structural similarities, regardless of the language in which they are expressed. Most ontologies describe individuals, categories, attributes, and relations. In this section each of these components is discussed in turn as well as the related Mimosa specification. More descriptions can be found in [9, 10].

3.1 Individuals

Individuals are the basic, "ground level" components of an ontology. The individuals in an ontology may include concrete objects such as people, animals, tables, automobiles, molecules, and planets, as well as abstract individuals such as numbers and words. Strictly speaking, an ontology need not include any individuals, but one of the general purposes of an ontology is to provide a means of classifying individuals, even if those individuals are not explicitly part of the ontology. In Mimosa, the model editor is provided for defining the individuals, out of the defined categories. Only the individuals can actually behave and therefore be simulated. In figure 3.1, we have three plots (p1, p2 and p3) and two people (John and Paul). The name of the individual is optional and indicated before the ":". The name after the semi-colon shall be explained in the

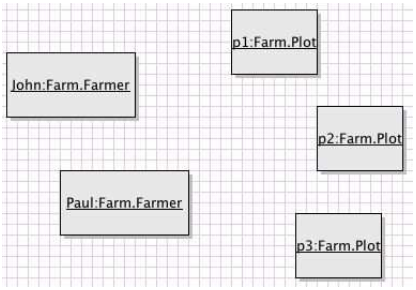

Figure 3.1: Farmer and plot individuals.

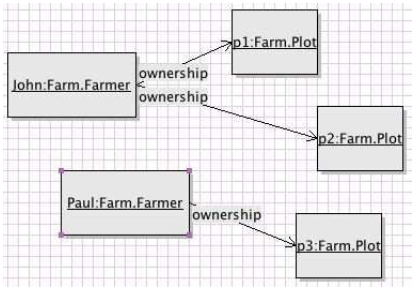

Figure 3.2: Farmers owning plots.

following. It actually is the name of the category the individual belongs to.

3.2 Links

For the model to be properly called a structure, these individuals usually are linked to each other in some meaningfull way. In our example, the figure 3.2 shows some links between the individuals describing that John is proprietary of p1 and p2, while Paul is proprietary of p3. The proprietary link is indicated by the name `ownership`.

3.3 Attributes

Individuals in the ontology are described by specifying their attributes. Each attribute has at least a name and a value, and is used to store information that is specific to the individual it is attached to. For example the p2 individual has attributes such as:

surface 20

cover tree

The value of an attribute can be a complex data type; in this example, the value of the attribute called `cover` could be a list of values, not just a single value. In the figure 3.3, some of the attributes are represented.

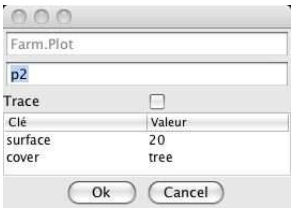

Figure 3.3: The description of the plot p2.

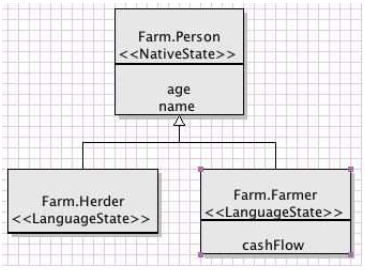

Figure 3.4: A category hierarchy of plots and people

3.4 Categories

Categories are the specification of the common features of groups, sets, or collections of individuals. They are abstractions over sets of concrete individuals. Some examples of categories are:

Person : the category of all people (describing what is common to all people);

Molecule : the category of all molecules (describing what is common to all people);

Number : the category of all numbers;

Vehicle : the category of all vehicles;

Car : the category of all cars;

Individual : representing the category of all individuals.

Importantly, a category can subsume or be subsumed by other categories. For example, `Vehicle` subsumes `Car`, since (necessarily) anything that is a member of the latter category is a member of the former. The subsumption relation is used to create a hierarchy or taxonomy of categories, with a maximally general category which is called `Individual` in Mimosa, and very specific categories like `MaizeFarmer` at the bottom. Figure 3.4 shows such a hierarchy of categories.

Usually what is common to a collection of individuals is that they share the same attributes. In the figure 3.4, all the people have a name and an age. We also assume that each farmer has a cashflow (but not a herder!). By subsumption, any farmer and any herder has also a name and an age because they are particular case of Person. In Mimosa an attribute has a name, a type

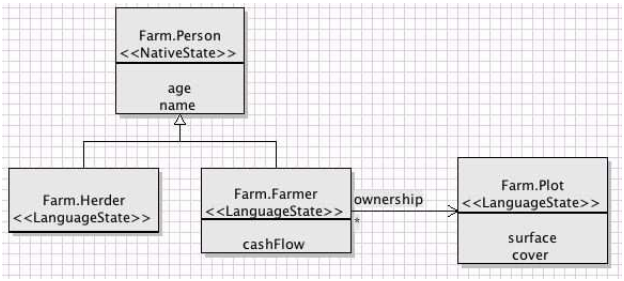

Figure 3.5: A category hierarchy of plots and people with a relationship

which can be only a single type (short, integer, long, float, double, string and color) and a cardinality to have list of values. If an attribute refers to another category, it is a relationships and no longer an attribute.

3.5 Relations

An important use of relations is to describe the relationships between individuals in the ontology. In fact a relation can be considered as an attribute whose value is another individual in the ontology, or conversely an attribute can be considered as a relationship with another individual (a number is also an individual, instance of the category of numbers). For example in the ontology that contains the Farmer and the Plot, the Farmer object might have the following relation:

ownership Plot

This tells us that the Plot can be owned by a Farmer. Together, the set of relations describes the semantics of the domain. In the figure 3.5, a relation has been added accordingly. In addition, we have also declared that a person can be proprietary of any number of plots. One can see that the individuals described in figure 3.2 appear to be instances of the categories described in 3.5 and that their links appear to be instances of the related relations.

In Mimosa, a relation is uni-directional and links a category to another, with a cardinality.

The most important type of relation is the subsumption relation (is-superclass-of, the converse of is-a, is-subtype-of or is-subclass-of) already mentioned in the previous section.

Another common type of relations is the meronymy relation (written as part-of) that represents how objects combine together to form composite objects. For example, if we extended our example ontology to include objects like Steering Wheel, we would say that "Steering Wheel is-part-of Ford Explorer" since a steering wheel is one of the components of a Ford Explorer. If we introduce part-of relationships to our ontology, we find that this simple and elegant tree structure quickly becomes complex and significantly more difficult to interpret manually. It is not difficult to understand why; an entity that is described as 'part of' another entity might also be 'part of' a third entity. Consequently, entities may have more than one parent. The structure that emerges is known

as a Directed Acyclic Graph (DAG). This aspect is not introduced in the ontological level of Mimosa but will be further discussed in the mereological level where, precisely, a stronger account of meronymy is introduced (but not yet implemented at this stage).

The part of the ontology consisting of the categories, attribute descriptions and relations (either taxonomic or semantical) shall be called the *conceptual model*. The part of the ontology consisting of the individuals, their attribute values and their links shall be called the *concrete model*. In the following the editor to create the conceptual model shall be described. In addition, we shall describe how to specify the dynamics associated to each category. Thereafter, we shall introduce the concrete model editor.

Chapter 4

The conceptual model editor

4.1 The editor

The conceptual model editor is made of three panels for editing the conceptual model:

- the graph panel for graphical editing.
- the list panel for editing the ontology as a list of definitions (a kind of dictionary).
- the list panel of influence types to be explained in the section 5.

The list panel is the reference to know all the categories defined in the edited conceptual model. In effect, a category may not appear in the graph panel. Conversely, a category may appear several times in the graph panel as well as categories from other conceptual models. The rational behind this behavior is that the drawing (hence the graph panel) must have an explanatory power (not only a definitory one) and therefore any drawing clarifying the explanation should be possible.

We shall concentrate on the graph panel which is nevertheless easier to use for defining categories. The starting point is the tool bar in the upper part of the window as illustrated in the figure 4.1 where seven buttons appear:

- the first one is the grabber for selecting an object (category or relations) in the drawing and is always selected by default;
- the second is the note object to write down documentary comments to associate to categories;
- the third is the link to associate a comment with a category;

Figure 4.1: The buttons of the ontology editor.

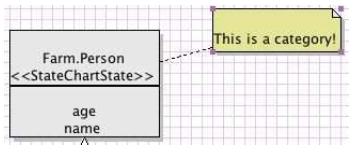

Figure 4.2: An annotated category.

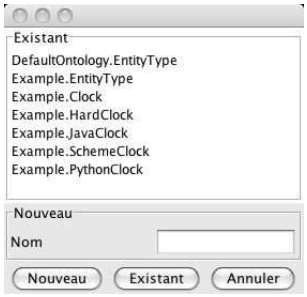

Figure 4.3: The creation dialog for a category.

- the fourth is for creating or selecting categories to draw;
- the fifth one is the taxonomic relationship;
- the sixth is the semantic relationship;
- finally, the seventh is the button to access the push down menu for manipulating the grid behavior as already described in 2.4.

The first three buttons as well as the last one are always present for each graph editor, so it shall not be explained again. The figure 4.2 shows the use of a note.

4.2 Category edition

4.2.1 Drawing a category

To draw a category in a given place it is enough to click on the corresponding button and then at the desired place, or to right click at the desired place to show up the same toolbar as a popu menu. A new dialog is opened as illustrated in the figure 4.3.

This dialog is composed of two parts:

- the upper part lists all the categories available in all the opened ontologies. Selecting one of these and typing either return or pushing the `Existing` button shall draw the corresponding category at the selected place;
- the down part is used to create a new category with a name field to enter a new name (which must be unique within the current ontology).

A rectangle with either one or two subparts shall be drawn at the selected place 4.4:

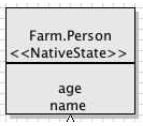

Figure 4.4: The category graphical form.

- the upper part has two lines:
  - the first line is the name of the category prefixed by the name of the ontology;
  - the second line is the name of the way to define the dynamics for this category[1]. `NativeState` is chosen by default and does nothing.
- the down part is the list of attributes (for the time being, the attribute specifications are not shown).

4.2.2 Editing a category

A category can be edited by double-clicking on it, or by selecting it and selecting `Edit...` from the `Edit` menu, or by right-clicking on it and selecting `Edit...` in the popup menu. The category editor dialog (4.5) shows up with the following parts:

- the name of the category, which cannot be changed;
- an "abstract" check box to specify whether the category can have instances or not (e.g. most probably, in our example, there shall not be direct instances of Person, but only of Farmer and Herder);
- the super type, i.e. the category subsuming this category;
- a panel where one can specify either the attributes, the relations and the behavior (see chapter 5).

In figure 4.5, one shows the attribute panel where the local attributes can be added or deleted through a popup menu. Additionally, one can see the list of inherited attributes as shown in figure 4.6, but this list cannot be edited. Only the locally defined attributes can be edited, the inherited list being computed.

4.2.3 Deleting a category

A category can be deleted by selecting it and selecting `Remove...` from the `Edit` menu, or by right-clicking on it and selecting `Remove...` in the popup menu. It is asked whether the category must be removed from the ontology:

- if yes, the category is removed both from the drawing and the list of categories defined in the ontology;
- otherwise, only the drawing is removed but the category remains as an existing category.

[1] For UML literates, it looks mike a stereotype, and in fact it has a related semantics with respect to the MDA specifications.

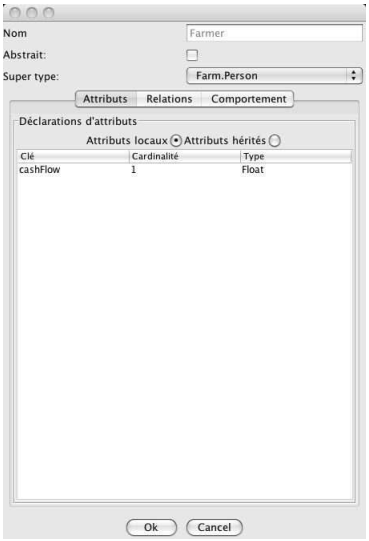

Figure 4.5: The category editor with the attribute panel.

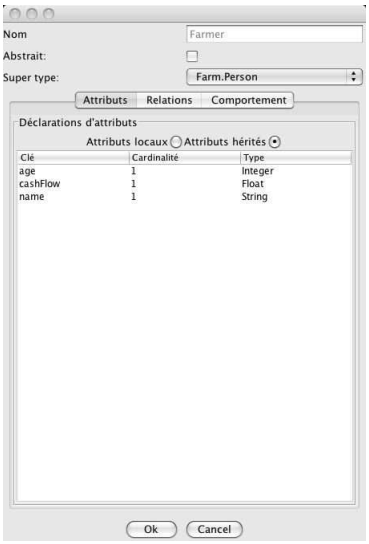

Figure 4.6: The category editor with the inherited attributes.

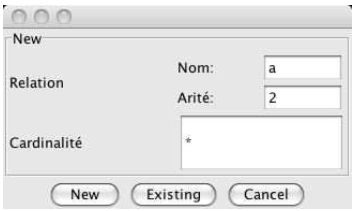

Figure 4.7: The creation dialog for a relation.

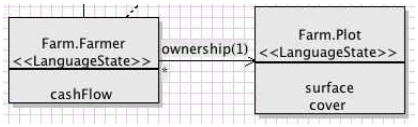

Figure 4.8: The example of a relation.

4.3 Relation edition

4.3.1 Drawing a relation

To draw a relation in a given place it is enough to click on the corresponding button and then from a category (called the source category) to another one (called the target category), or to right click at the desired place to show up the same toolbar as a popu menu. A new dialog is opened as illustrated in the figure 4.7.

This dialog is also composed of two parts even if in the figure 4.7 only one shows up:

- the upper part lists all the existing relations available between the two selected categories. Selecting one of these and typing either return or pushing the `Existing` button shall draw the corresponding relation between the two categories.
- the down part is used to create a new relation with three fields:
  - a name field to enter a new name (which must be unique within the source category);
  - an arity field to possibly create a family of relation names indexed by the given number of indices;
  - a cardinality field to specify whether the relation can reference one, several or any number of objects of the given target category.

The arrow from the source category to the target category is annotated by all the relevant information as shown in the figure 4.8. The arity is written between parenthesis. It means that the links between individuals can be named: `ownership(0)`, `ownership(1)`, and so on. Additionally, the "*" means that each of these links can be drawn with any number of plots.

The list of defined relations for a category also appears in the relation panel of the category editor as shown in the figure 4.9. A relation can be added or

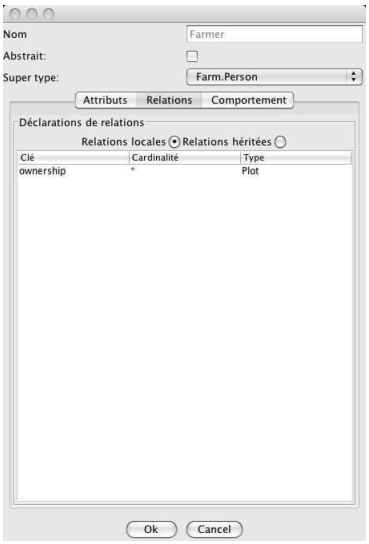

Figure 4.9: The relations of a category.

removed directly from this panel but the added relations shall be drawn only if requested as an existing relations.

The subsumption or taxonomic relationships is a particular case where nothing need to be specified but the source and target categories.

4.3.2 Editing a relation

A relation can be edited by double-clicking on it, or by selecting it and selecting `Edit...` from the `Edit` menu, or by right-clicking on it and selecting `Edit...` in the popup menu. The same editor appears as for creating it.

4.3.3 Deleting a relation

A relation can be deleted by selecting it and selecting `Remove...` from the `Edit` menu, or by right-clicking on it and selecting `Remove...` in the popup menu. It is asked whether the relation must be removed from the model:

- if yes, the relation is removed both from the drawing and the list of relations defined for the source category;
- otherwise, only the drawing is removed but the category remains unchanged.

A relation can also be removed from the relation panel of the source category editor. If it is deleted this way, all the drawings of the relation shall disappear as well.

Chapter 5

The dynamics

5.1 Introduction

For each category, one can associate a specification of the dynamics of the corresponding individuals. Basically, it is made by selecting a way to specify the dynamics (a state machine, a markov process, a differential equation, and the list is extensible at will) and then by specifying the dynamics according to the selected way (the states and transitions for a state machine, the states and transition matrix for a markov process, etc.). The way to specify the dynamics shall be called the *state* (although it does not only defines the state but also how the state changes) and the associated specification the *state specification*. The state can be specified directly when creating a new category as explained in the section 4.2.1. Otherwise, it is enough to open the category editor as shown in figure 5.1 and to select the behavior panel.

The behavior panel is itself made of four subpanels:

- the probe panel is used to define what can be observed from the individuals. It is used for displaying what happens during the simulation or saving it to any media for further processing (statistics, etc.). The probes shall be described in the section 5.2.4.
- the incoming influences panel is used to specify the events the individuals are able to react to. They shall be explained in the sections 5.2.1 and 5.2.3.
- the outgoing influences panel is used to specify the events the individuals are producing. They shall also be explained in the sections 5.2.1 and 5.2.3.
- the behavior panel (sorry, it has the same name as the upper level panel) is used to specify the state and its specification. In the figure 5.1, one can see the drop-down menu in the upper part to select the state (i.e. the way one wants to specify the behavior) and the associated specification. Here, the chosen way to specify the behavior is through a state chart as specified in UML called `StateChartState`. Therefore a corresponding state chart editor is shown.

In addition, at the top, there is the specification of the multiplier between

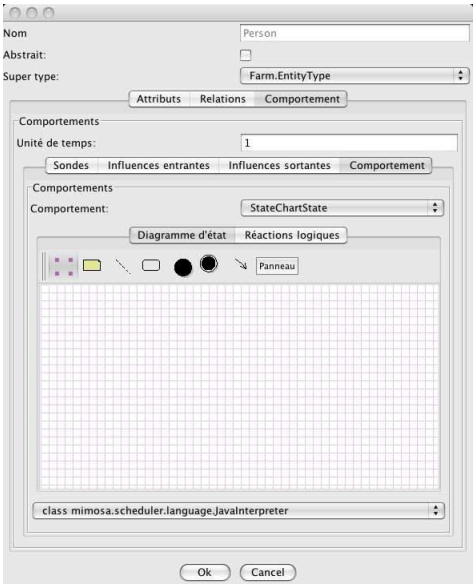

Figure 5.1: The behavior panel of the category.

the global time grain and the local time grain[1]. Further explanation on the representation of time in Mimosa can be found in section 5.2.5.

To understand the offered possibilities and, more importantly, the behavior one can expect from these various states and state specifications, it is necessary to go down to the ground and expose a little bit of the underlying machinery. This is done in the following section. Thereafter, we shall introduce some already existing states and their corresponding specifications.

5.2 The operational semantics

Globally, the underlying machinery is nothing but a discrete event simulation system. The running model is made of entities sending time stamped events which are delivered to their target entities at the specified dates, possibly generating new time stamped events and so on. The scheduler is in charge of ordering the events by their time stamps and to execute them in order. The only thing to specify is how each entity behaves, i.e. generates new time stamped events and reacts to incoming events. It is the purpose of the next section. In the following the events are called influences for obscure (another name for historical) reasons [7].

5.2.1 The model

The underlying simulation semantics is based on an extension of //-DEVS (see [11]) called M-DEVS as a shorthand for Mimosa-DEVS. Therefore, one must

[1]The grain is the smallest difference between any two time measure which can be distinguished.

understand how M-DEVS works in order to master the behavior of the models although most details are assumed to be hidden by higher level of abstractions as suggested in the introduction of this chapter.

A M-DEVS entity is a tuple:

$$< X, Y, O, init, \delta_{ext}, \delta_{int}, \delta_{log}, \delta_{con}, \lambda_{ext}, \lambda_{int}, \lambda_{log}, \lambda_{str}, \tau >$$

with an implicit state space on which no hypothesis is made, where:

X: is a set of incoming influences;

Y: is a set of outgoing influences;

O: is a set of output ports the elements of Y are sent to;

$init$: is a function to set the model in its initial state;

δ_{ext}: is a function to specify the reaction to a set of incoming influences (all the influences occurring at the same time are given simultaneously);

δ_{int}: is a function to specify the internal change (when it occurs is specified by t_a and what occurs is specified by λ_{int});

δ_{con}: is a function to specify the reaction to the occurrence simultaneously of an internal change and the arrival of a set of incoming influences;

δ_{log}: is a function to specify the reaction to a set of logical influences, possibly producing further logical influences;

λ_{ext}: is a function to provide the outgoing influences (when it is called is also specified by t_a);

λ_{int}: is a function to provide the internal influence to occur after a duration of t_a;

λ_{log}: is a function to provide the logical influences to occur after each transtion;

λ_{str}: is a function to provide the structural changes to occur also after each transtion;

τ: is a function which provides the duration until the next internal influence;

For all functions but $init$, the duration since the last cycle (see below) is given as an argument. Therefore the internal logic of any atomic model is based on durations.

Although complicated at the first sight, the logics is very simple:

- λ_{ext} and δ_{ext} are the functions to issue the events (λ_{ext}) and to handle them (δ_{ext}). It corresponds straight away to the intuitive event based mechanism as explained in introduction. The time of the events is when τ elapsed since the last transition;
- λ_{int}, τ and δ_{int} are the functions for specifying the spontaneous behavior, i.e. what the "box" does (λ_{int}), when (τ) and how (δ_{int});
- λ_{log} and δ_{log} is used to propagate information (λ_{log}) and make computations based on this information (δ_{log});

- λ_{str} specifies the possible modifications in the interconnection topology (see below).

Mimosa implements a unique so-called M-DEVS bus which is a set of M-DEVS entities with interconnected ports. More precisely, a M-DEVS bus is a pair $< E, links >$ where:

E**:** is a set of M-DEVS entities;

*links***:** is a mapping from $M \times O$ into E specifying a mutable interconnection topology;

For simulation, the M-DEVS bus runs in cycles. Each cycle corresponds to a certain date where everything happening at that date is propagated through out all the M-DEVS atomic models. At each cycle:

1. each model is asked for its t_a. Let min_{ta} be the smallest value;
2. the global time is advanced by min_{ta}. Let:
   - C be the set of models with the same min_{ta};
   - $C' \in C$ be the set of models producing outputs;
3. λ_{ext} is called for each model in C' and the outgoing influences are gathered and their destinations are identified using $links$;
4. for each model m in C:
   - if m has simultaneous incoming influences and an internal change, δ_{con} is called;
   - if m has only an internal change, δ_{int} is called;
   - if m has only incoming influences, δ_{ext} is called;

   and all the outgoing logical influences are gathered;
5. all the logical influences are dispatched via $links$ by calling λ_{log} and δ_{log} until there is no logical influences left (be careful about possible loops which are not detected).
6. all the structural changes are dispatched by calling λ_{str}.

For each individual, MIMOSA shall generate a corresponding entity which shall be initialized from the list of its attribute values in a state specific way.

5.2.2 The ports

A port provides a way to connect M-DEVS entities together. In Mimosa we distinguish port descriptions and ports.

A port description is defined by a name and an arity:

- a name is simply a `String`. This name must be unique within a category.

- an arity defines the number of indexes which can be given to specify a port. When an entity has to address a large number of other entities, rather than providing a range of distinct ports, it is easier to use a single name with one or more indices. Having more than one index allows to induce a topology on the addressed entities. For example, an entity defining a two dimensional cellular automata can address it cells with two indices, hence with an arity 2 port description.

Externally, the syntax of a port description is `<name>['('<int>')']`. The arity is optional if it is 0.

A port is an instance of a port description. When the arity is 0, there is only one possible instance. When the arity is 1 or more, any number of ports are possible, distinguished by the value of their indices. Externally, the syntax of a port is `<name>['('<int>{','<int>}')']`. The indices are optional if the arity is 0.

If the reader perceives some relationship between a port and a link (and between a port description and a semantic relation), it is right. We are here using the vocabulary used in the modeling and simulation community which is unrelated to the vocabulary used in the ontology community. As for individuals generating M-DEVS entities, the links are used to produce the initial interconnection topology.

5.2.3 The influences

An influence is an event which is transmitted between two M-DEVS entities. In Mimosa we also distinguish between influence types and influences as instances of influence types.

The influence types are just names but must be declared. These names are unique in a given conceptual model (or ontology). This type level is not really useful at this stage but provides a provision for further typing (like the declaration of the arguments) to be used for connectivity with other buses like HLA or CORBA where the type of transmitted information has to be declared.

The influences are instances of influence types. For the time being they have:

- a name which is the name of the corresponding influence type;
- a content which is either empty or a collection of elements.

For ensuring communication between entities possibly written in various languages, and in particular, in various scripting languages, a standard and limited format is imposed for the content. A content is necessarily a collection (at the implementation level an instance of Java `ArrayList`) of:

- collections, allowing recursive structures;
- simple types: shorts, integers, longs, floats, doubles, booleans and strings (respectively implemented internally in Java as instances of Short, Integer, Long, Float, Double, Boolean and String).

No other kind of data can be send through the influences.

5.2.4 The probes

It is possible to associate to any individual (therefore to any M-DEVS entity), a visualization window for displaying any information evolving over time (e.g. the entity state changes). Having no hypothesis on the nature of the entity states, there is NO automatic synchronization between the model and its visualization. To perform this visualization, one has to declare a list of probes given by their name, type (only simple types are allowed) and cardinality. When specifying the behavior, i.e. the various transition functions, the user has to explicitly send probe values whenever he wants to signal a change. The probe value is propagated to the visualization window which can perform whatever one wants: drawing or saving the data for further processing.

5.2.5 The time

The underlying time for the whole system is considered discrete (regardless of the grain which could be as fine as picoseconds) and therefore mapped on natural numbers. As already mentioned, an M-DEVS entity only considers durations. In addition, these durations can only be expressed as integers.

When simulating an M-DEVS entity, a local time is deployed. The creation of an M-DEVS atomic model either at the start of the simulation or during it, defines the origin of the local time (0). All the durations are added up, generating a local date as an integer. In particular, this local time is used to compute the durations transmitted to the M-DEVS entity.

A step further, the M-DEVS bus defines a global time. The origin of the global time (0) is the start of the simulation (initialization always occurs at the global time 0). The M-DEVS entity local times are mapped to the global time in two ways:

- the origin of the local time is situated in the global time at the (global) time of creation of the M-DEVS entity;
- the ratio between the local time grain and the global time grain is given. The global time grain is assumed to be the smallest possible grain able to take into account the grain of any other atomic model as an integral multiplier of the global grain.

Still at this stage, the time is a natural number without dimension (without unit). The correspondence between this time and the real time where the origin of simulation corresponds to a real date and the global grain has a unit (picosecond, hour or week) shall be specified externally. It is foreseen to be able to declare this information to the scheduler and use this reference to define in an easier way the time units of the entities. It is not yet completely implemented at this stage.

In summary, any M-DEVS entity has

- a grain (the smallest undistinguishable time difference) defined implicitly by having durations expressed with integers and explicitly by a multiplier with the global grain;
- an origin defined implicitly by having the entity life starting at 0 and explicitly by a position of this origin with respect to the global time.

5.3 The behavior specification

In order to describe the behavior of an entity, the user must expect to have to specify each of the mentioned function for proper functioning of the model, hence the importance to understand the underlying operational semantics as described before. However, higher level specifications can be made as various kind of state machines, petri nets, directly specified differential equations with various means of integration as long as there execution can be mapped in the previously described functions. These extensions can be added at will to the system in a way which is described in the programmer's manual.

When editing a category behavior, a number of panes are dedicated for specifying the behavior (see the figure 5.1):

- the incoming influences to declare the list of incoming influences;
- the outgoing influences to declare the list of outgoing influences;
- the probes to declare which information is dynamically provided during entity simulation;
- the behavior pane to describe the behavior itself. At the top of this pane, there is drop down menu of available ways of specifying the behavior.

The available means for specifying the behavior are as follows:

- by writing a piece of Java program and declare it to the Mimosa system to make it available in the user interface;
- by specifying the behavior of each of the mentioned function using a scripting language. Several scripting languages are available: java, scheme, jess (unavailable due to a need for a license), python and prolog (not fully tested yet);
- with a state/transition diagram where the conditions and actions can be specified in one of the scripting languages mentioned before;
- with any higher level mean of specification as markov processes, etc. depending on the availability of the corresponding plug-in.

These various technics shall be described in turn in the next sections.

5.3.1 Programmatic specification

This section is more appropriate for the programmer's manual but is included here to introduce the basics which are made available in the other ways of specifying the model behavior. With your favorite Java IDE (for example Eclipse (http://www.eclipse.org)), create a new project with a package (let's call it example) in which you have to create a class as a subclass of `mimosa.scheduler.NativeState`. The result is a file with the following content:

```
package example;

import mimosa.scheduler.NativeState;

public class MyExample extends NativeState {
}
```

`NativeState` defines nine (10) methods doing nothing by default:

- `public void doInstanceInitialize() throws EntityException;`: which is called only once when the entity is created (for example in the model editor). Use it to create the initial content of state variables.
- `public void doInitialize() throws EntityException;`: equivalent to the *init* function. It is called each time the model is initialized by the scheduler. As a principle, each time a model is initialized, exactly the same initial state should result. If you are using random generators, try to reinitialize it with the same seed.
- `public void doExternalTransition() throws EntityException;`: equivalent to δ_{ext}.
- `public void doInternalTransition() throws EntityException;`: equivalent to δ_{int}.
- `public void doLogicalTransition() throws EntityException;`: equivalent to δ_{log}.
- `public void doConfluentTransition() throws EntityException;`: equivalent to δ_{con}.
- `public void doGetExternal() throws EntityException;`: equivalent to λ_{ext}.
- `public void doGetInternal() throws EntityException;`: equivalent to τ and λ_{int} together.
- `public void doGetLogical() throws EntityException;`: equivalent to λ_{log}.
- `public void doGetStructural() throws EntityException;`: equivalent to λ_{str}.

If something is going wrong, just throw an `EntityException` with the entity and a message as parameters. The exception will be taken into account by the architecture in an appropriate way. Do not forget to catch any possible exception and raise an `EntityException` accordingly for securing the model execution. Because they are predefined for doing nothing, you can only define the methods you actually need.

BE CAREFULL, until a next release where these methods shall probably disappear, do not use the methods `doGetExternal`, `doGetLogical` and `doGetStructural` unless you perfectly know what you are doing. Sending the corresponding influences anywhere else is enough in most cases.

When calling each method, this variable is defined and appropriately bound in the context:

time: contains the duration since the previous transition (remember that these methods are called in a given cycle and the M-DEVS bus advance time from a cycle to another);

The following methods are defined for accessing the incoming influences:

- `getAllInfluences()`: to get the list of incoming influences in any order;
- `getInfluence(String name)`: to get the list of incoming influences with the given name. It is used to control the order in which to handle the incoming influences;
- `getInternalInfluence()`: to get the incoming internal influence.

To program each functionality, a number of methods are defined by categories:

- to manipulate random generators[2]:
  - `public Random newRandom(long seed);`
  - `public boolean newBoolean(Random rand);`
  - `public int newInt(Random rand,int max);`
  - `public double newDouble(Random rand);`
- to easily create ports and port references:
  - `public Port port(String name,int args...);`
  - `public Port portRef(Port port...);`
- to manipulate the influence content:
  - `Object contentOf(Influence influence)`: which returns either `null` if there is no content or a `Collection` of objects (as defined in 5.2.3).
  - `List list(Object... objects)`: to create a list of objects as a content or sub-content.
  - `Object object(T i)`: where T is one of the Java simple types (short, int, etc.) to encapsulate them within the corresponding class instance (Short, Integer, etc.).
  - `T toT(Object o)`: where T is one of the Java simple types (short, int, etc.) to unbox them from the corresponding class instance (Short, Integer, etc.).
- to get the initial value of a parameter:
  - `public Object getParameter(String name)`.
- to post an influence at a given port:
  - `void sendExternal(String portName,String influenceTypeName)`,
  - `void sendExternal(Port portName,String influenceTypeName)`,
  - `void sendExternal(String portName,String influenceTypeName,Object args)`,
  - `void sendExternal(Port portName,String influenceTypeName,Object args)`.

[2]it is necessary to hide which kind of generator is used. Currently the Mersenne Twister random generator is known as one of the best and provided in Mimosa.

  - `void sendLogical(String portName,String influenceTypeName)`,
  - `void sendLogical(Port portName,String influenceTypeName)`,
  - `void sendLogical(String portName,String influenceTypeName,Object args)`,
  - `void sendLogical(Port portName,String influenceTypeName,Object args)`.
  - `void sendInternal(int duration,String influenceTypeName)`,
  - `void sendInternal(int duration,String influenceTypeName,Object args)`,
  - `void reply(LogicalInfluence influence,String influenceTypeName)`,

  These methods can be called in most methods.

- to signal a state change by a probe:
  - `public void sendProbe(String name,Object args...)`.
- to destroy itself:
  - `public void die()`.

  It removes the entity from the scheduler, removes of the link references as well as all the scheduled incoming influences.

In addition, a number of methods are defined to dynamically create and link entities during the simulation:

- `void addPort(PortReference name, String categoryName, boolean traced, Map<String,Object> parameters)`: creates an entity as an instance of the given category, whether it is traced or not and the map of attribute values;
- `void addPort(String name, String categoryName, boolean traced, Map<String,Object> parameters)`: same as above when there is a simple syntax for the port reference;
- `void linkPort(PortReference portRef1, PortReference portRef2)`: links the port reference to the entities referenced by the second port reference, creating new links;
- `void linkPort(String portRef1, PortReference portRef2)`: same as above;
- `void linkPort(PortReference portRef1, String portRef2)`: same as above;
- `void linkPort(String portRef1, String portRef2)`: same as above;
- `void removePort(PortReference portRef)`: removes the entities from the given port, without destroying the referenced entities (they kill themselves using `die`).
- `void removePort(PortReference portRef)`: same as above.

For example, if we want to program the behavior of a clock which sends a `tick` influence to its `clocked` port at `interval` time, we could have the following code:

```
package example;

import mimosa.scheduler.NativeState;

public class MyClock extends NativeState {

   private int interval;
   /**
    * @see mimosa.scheduler.NativeState#doInitialize()
    */
   @Override
   public void doInitialize() throws EntityException {
      interval = getParameter("interval");
   }
   /**
    * @see mimosa.scheduler.NativeState#doGetInternal()
    */
   @Override
   public void doGetInternal() throws EntityException {
      sendInternal(interval,"tick");
   }
   /**
    * @see mimosa.scheduler.NativeState#doGetExternal()
    */
   @Override
   public void doGetExternal() throws EntityException {
      sendExternal("clocked","tick");
   }
}
```

in which we declare a variable to cache the parameter value (the interval between two ticks), the function to get the parameter value, the t_a function which signals an output after the given interval and λ_{ext} where a single influence is sent to the port.

Of course, it is not enough to write the code. This code has to be known from Mimosa. In order to do that, you have to create an XML file in which Mimosa can read the following declarations:

```
<?xml version="1.0"?>
<mimosamodule name="Example" package="example">
    <behaviour notion="EntityType" implementation="MyClock">
        <parameters>
             <parameter name="interval" cardinality="1" type="java.lang.Integer"/>
        </parameters>
        <outInfluences>
             <influenceType name="tick"/>
        </outInfluences>
```

```
        <outPorts>
             <port name="clocked" entityType="EntityType"/>
         </outPorts>
    </behaviour>
</mimosamodule>
```

This XML file contains everything you would have declared through the user interface and additionnaly defines through the `package` and `MyClock` attributes where to find the corresponding class.

You then have to create a folder called `example`, to put the `.jar` containing the complied class, to define a file called `example-config.xml` and to put the whole folder in the plugins subdirectory of Mimosa. By trying this example, the behavior `MyClock` will appear in the list of available behaviors.

In general, any new behavior (or way of defining behaviors) can be added to Mimosa by putting in the `plugins` directory a folder called `xxx` with a file called `xxx-config.xml` in it with the related XML content and as many `.jar` as necessary. Further details as well as the complete syntax of the XML file shall hopefully be presented in the programmer's manual.

5.3.2 Scripted specification

The previous procedure being relatively heavy but necessary if one wants either an efficient piece of code or to use Java to encapsulate a legacy simulation software, we provide the same functionality by using scripting languages directly through the user interface. The basic principles are the same and we are using the same names for the variables and functions or equivalent for consistency. For using this functionality, you have to select `LanguageState` in the drop down menu of the `behavior` pane. Immediately below, you will have another drop down menu to select the desired scripting language.

In a model, any combination of scripting languages can be used because all the specific data structures are translated into a standard Java format and back to the specific data structures. So feel free to use any one you find most appropriate for your usage. Of course, it requires to be multi-lingual!

Java scripting

Java scripting makes available the full Java language by using the bean shell library (see [4] for getting the related documentation). In particular, all the methods defined in the section 5.3.1 are readily available. However to call them, a new variable is defined: `self`. The methods can be called by addressing them to `self`. For example, for the λ_{ext} function, the code is:

```
self.sendExternal("clocked","tick");
```

There is one drawback in using Java scripting: all the Java types have to be prefixed explicitly by the package name (for example `java.lang.Integer` instead of simply `Integer`).

Scheme scripting

The Scheme language is a kind of pure functional language (based on lambda-calculus). The facilities for manipulating symbols and lists make it particularly

useful for qualitative and symbolic manipulations, much less for numerical computations. We are using the Kawa library ([5]) for providing Scheme. The documentation for the language itself can be found on the same web site. The appendix A provides a short reference to the Scheme language as well as the list of provided functions for calling Mimosa.

Jess scripting

Jess is a rule base language with a forward chaining semantics (see [2]). The behavior is described as a single set of rules of the form `<conditions> => <actions>`. Whenever the conditions are met, the corresponding rule is fired and the actions executed. In our case, each M-DEVS function introduces the time, the influences and the function name in the fact based and the rules are fired accordingly until no rule is applicable. The example of the clock looks like this:

```
(defrule initialize1
    (initialize)
    =>
    (make (interval (getParameter "interval"))))
(defrule getExternal
    (getExternal)
    =>
    (sendExternal "clocked"  "tick"))
(defrule getInternal
    (getInternal)
    (interval $value)
    =>
    (sendInternal $value "tick"))
```

It is no longer maintained because Jess requires a licence which is free for academics but costly for others. The library is not provided with the distribution for that reason but can be downloaded from [2].

Python scripting

The implementation uses the Jython library whose documentation can be found on [6]. We are using the possibility in this version of Python to call Java objects with the standard python syntax. Accordingly, the variable `self` is defined as well as all the variables as in Java and the correpsonding methods can be called directly. So, there is not much difference with Java.

Prolog scripting

Prolog ia a rule base language with a backward chaining semantics. The behavior is described as a single set of rules of the form `<conclusion> :- <conditions>`. The program is run by asking for a conclusion and the program tries to find the possible proofs. As in Jess, each M-DEVS function introduces the time, the influences and the function name in the fact based and the rules are fired accordingly until no rule is applicable. The `run` predicate must be defined. The example of the clock looks like this:

```
run :- initialize,
          X is getParameter(interval),
          asserta(interval(X)).
run :- getExternal,
          sendExternal(clocked,tick).
run :- getInternal,
          interval(X),
          sendInternal(X,tick).
```

Implemented but not yet fully tested. The implementation uses the tuProlog library whose documentation can be found on [3].

Smalltalk scripting

Implemented but not yet fully tested. The implementation uses Athena (see the we site [1]) which is a lightweight implementation of Smalltalk for embedded applications. The resulting scripts look awfull so it would probably not be explored further.

5.3.3 State charts

Coming soon.

5.3.4 Further extensions

This level being extensible at will by adding further meta-ontologies, this chapter shall only describe some of them as provided in the first versions of Mimosa. How to define new meta-ontologies is described in the programmer's manual. In this chapter, we shall introduce the meta-ontologies for object, space, cellular automata and multi-agent systems.

The objects

Most categories have very simple behavior corresponding roughly to what is available in objet-oriented programming. For the categories, it is not necessary to provide the full M-DEVS functionality (although object-orientedness can be mapped in a subpart of M-DEVS). We have provided two versions corresponding to most needs:

- `StaticObject` is used when the only functionality is around state variable values being set and get;
- `SimpleObject` is an extension of `StaticObject` where external and logical influences are considered as method calls: the external influences when the `SimpleObject` will change state in response, and the logical influences when only information updates and requests have to be handled.

`StaticObject` contains a set of state variables to choose among the attributes[3]. The following incoming influences are expected:

[3]It is assumed that a state variables always has an initial value to be set from the corresponding attribute.

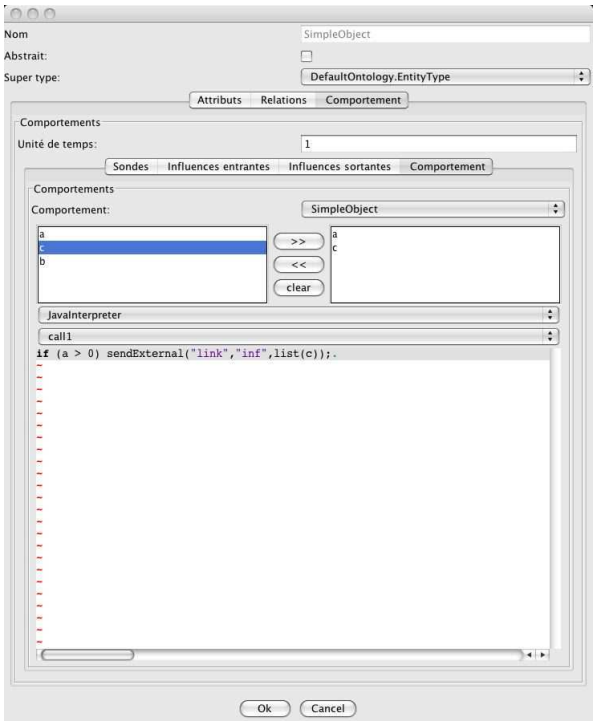

Figure 5.2: The behavior panel of a simple object.

- `setState name value`: as an external influence to change the value of one of the variables;
- `getState name`: as a logical influence to ask for the value of one of the variables.

The following outgoing influences are issued in response to the `getState` influence:

- `state name value`: as a logical influence to communicate the value of the requested state variable;
- `undefinedState name`: as a logical influence to communicate the state variable has no value.

`SimpleObject` has the same semantics as `StaticObject` and as such provides to the same incoming and outgoing influences. In addition to defining the state variables, the modeler can add as many additional incoming and outgoing influences as he wants. `SimpleObject` allows to associate a piece of code to execute to each incoming influence. In figure 5.2, the upper part shows on the left the list of defined attributes and on the right the list of attributes which have been chosen as state variables. In the bottom part, one can see the chosen scripting language, the chosen incoming influence and the associated code. The arguments of the influence if any are stored in the variable `arguments` as a list.

The spaces

Coming soon.

The cellular automata

Coming soon.

The multi-agent systems

Coming soon.

Chapter 6

The concrete model editor

At this stage, the conceptual model has been completely defined both with its structural part (the ontology properly speaking) with the categories, their attributes and their relations, and its dynamical part by specifying in a way or another the dynamics and the individuals specified by each category. The concrete model editor shall use these definitions for providing the user with the possibility to describe as many concrete models as he wants as a set of individuals, attribute values and links. These individuals, attribute values and links are nothing but the instances of the corresponding categories, attribute descriptions and relations.

The concrete model editor is made of two panels:

- on the left pane, there is a list of existing models. These models can be created or removed by double-clicking in this pane.
- on the right pane, there is a graph panel very similar to the one used for creating conceptual models. The top of the panel is occupied by a drop down menu to select the conceptual model from which one wants to instantiate the individuals and links. A concrete model can be drawn from several conceptual models combining various sources of knowledge.

Apart from the conceptual model drop down menu, The starting point is the tool bar in the upper part of the window as illustrated in the figure 6.1 where six buttons appear:

- the first one is the grabber for selecting an object (individual or links) in the drawing and is always selected by default;
- the second is the note object to write down documentary comments to associate to individuals;
- the third is the link to associate a comment with an individual;

Figure 6.1: The buttons of the model editor.

Figure 6.2: The creation dialog for an individual.

Figure 6.3: The individual graphical form.

- the fourth is for creating or selecting individuals to draw;
- the fifth one is the link;
- finally, the sixth is the button to access the push down menu for manipulating the grid behavior as already described in 2.4.

6.1 Individual edition

6.1.1 Drawing an individual

To draw an individual in a given place it is enough to click on the corresponding button and then at the desired place, or to right click at the desired place to show up the same toolbar as a popu menu. A new dialog is opened as illustrated in the figure 6.2.

This dialog is composed of two parts:

- the upper part lists all the individuals available in the selected model. Selecting one of these and typing either return or pushing the `Existing` button shall draw the corresponding individual at the selected place.
- the down part is used to create a new individual with two fields:
  - a drop down menu from which to select the category one wants to create an individual from;
  - a name field to enter a name which is optional but can be used for documentation purpose.

A rectangle is drawn as illustrated in the figure 6.3 with a name which composed of the optional name of the individual, a semi-colon and the category name which is itself composed of the ontology name and the category name.

Figure 6.4: The individual editor with the attribute panel.

6.1.2 Editing an individual

An individual can be edited by double-clicking on it, or by selecting it and selecting `Edit...` from the `Edit` menu, or by right-clicking on it and selecting `Edit...` in the popup menu. The individual editor dialog (6.4) shows up with the following parts:

- the name of the category, which cannot be changed;
- the name of the individual which can be changed at will;
- a "trace" check box to specify whether the individual has to be traced. This allows to trace the M-DEVS function calls specifically for one individual;
- a panel where one can specify the attribute values.

6.1.3 Deleting an individual

An individual can be deleted by selecting it and selecting `Remove...` from the `Edit` menu, or by right-clicking on it and selecting `Remove...` in the popup menu. It is asked whether the individual must be removed from the model:

- if yes, the individual is removed both from the drawing and the list of existing individuals defined in the model;
- otherwise, only the drawing is removed but the individual remains as an existing individual.

6.2 Link edition

6.2.1 Drawing a link

To draw a link in a given place it is enough to click on the corresponding button and then from an individual (called the source individual) to another one (called the target individual), or to right click at the desired place to show up the same toolbar as a popu menu. A new dialog is opened as illustrated in the figure 6.5.

This dialog is composed of the list of available relations between the two selected individuals as defined in the corresponding category of the source individual. Depending on the arity of the relation (i.e. the number of indices to fully specify the relation), as many text fields are displayed underneath to enter the indices values. In the figure 6.5, the relation is of arity 1, so only one index must be specified.

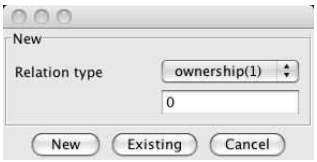

Figure 6.5: The creation dialog for a link.

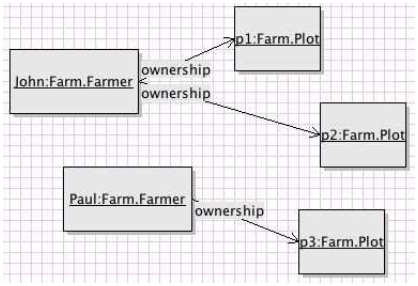

Figure 6.6: The example of a link.

The arrow from the source category to the target category is annotated by the relation name as shown in the figure 6.6. The index values are written between parenthesis.

6.2.2 Deleting a link

A link can be deleted by selecting it and selecting `Remove...` from the `Edit` menu, or by right-clicking on it and selecting `Remove...` in the popup menu. It is asked whether the link must be removed from the model:

- if yes, the link is removed both from the drawing and the list of links defined for the model;
- otherwise, only the drawing is removed but the link remains unchanged.

Chapter 7

Some examples

7.1 The rolling ball example

As an example, we shall model a simple system composed of one rolling ball and a kicker. This example allows the illustration of a combination of continuous and discrete time:

- the rolling ball is submitted to uniform movement described by the following equations:

$$x(t) = x_0 + v_x \times t; y(t) = y_0 + v_y \times t$$

- at random time, the kicker computes a random two-dimensional vector $< k_x, k_y >$ which is sent to the ball to change its trajectory in the following way:

$$v_x = v_x + k_x; v_y = v_y + k_y$$

7.1.1 Defining the conceptual model

The conceptual model will be composed of two categories: `RollingBall` and `Kicker`. The RollingBall is characterized by four attributes: two for the initial position (`x0` and `y0` corresponding to x_0 and y_0) and two for the speed (`vx` and `vy` corresponding to v_x and v_y). The Kicker is characterized by one attribute: the seed of its random generator used for the time of kicking the ball and the generation of the random vector[1].

If we want to visualize the position of the ball, the event-based nature of the simulation will only be able to provide state changes when the ball is kicked. To see the ball rolling between two successive kicks, we have to sample the trajectory. In order to do that, a third category is added to the model to sample the trajectory by asking at each fixed time step to the ball its position. The resulting ontology in shown in figure 7.1.

In addition, you have the definition of three relations:

- `kicked` which a relation of `Kicker` to send a kick to a `RollingBall`. Note that a Kicker can kick simultaneously any number of balls.

[1] To put the seed as a parameter is recommended if one wants to control the outcome of the simulation, i.e. to produce exactly the same result for each simulation.

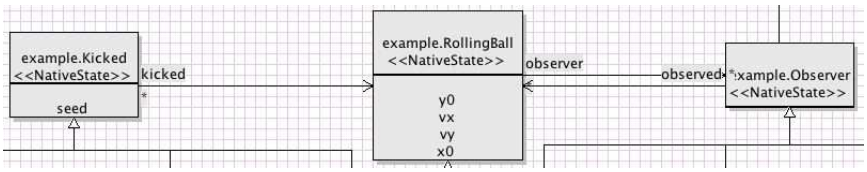

Figure 7.1: The conceptual model for a kicked and observed rolling ball.

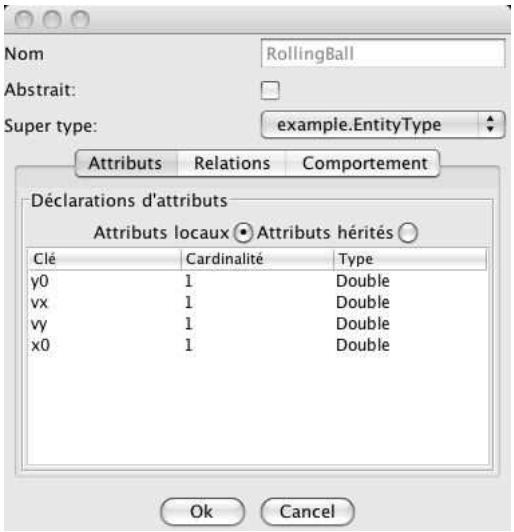

Figure 7.2: The conceptual model for a rolling ball with the attribute panel.

- `observer` which is a port of `RollingBall` to send its position to an observer (and it can have as many observers as it wants).
- `observed` which is a port of `Observer` to send a request for position (it will always be a logical influence, of course).

The parameters can be edited (added, changed or removed) through the category editor as shown in the figure 7.2.

The relations (i.e. the definition of the relation name, cardinality and type) can be either drawn through the graphical editor or entered in the category editor dialog as in figure 7.3. If the relation are defined by the category editor, they will not show up in the graphical editor. They can be visualized by drawing an arc and specifying an existing link as shown in figure 7.4.

At that stage, the structure of the conceptual model (i.e. the ontology) is entirely defined: the categories, attributes and relations.

7.2 Defining the dynamics

For defining the behavior, you have to define:

- the incoming and outgoing influences;

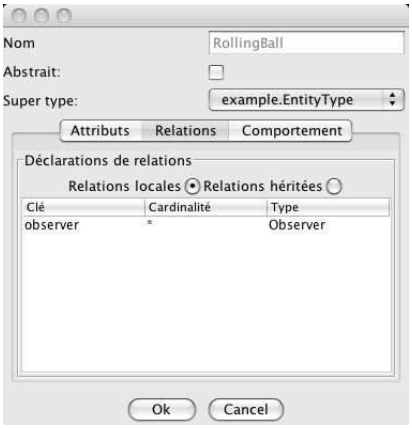

Figure 7.3: The rolling ball category with the relations panel.

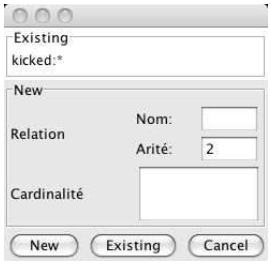

Figure 7.4: Definition of an arc from an existing relation definition

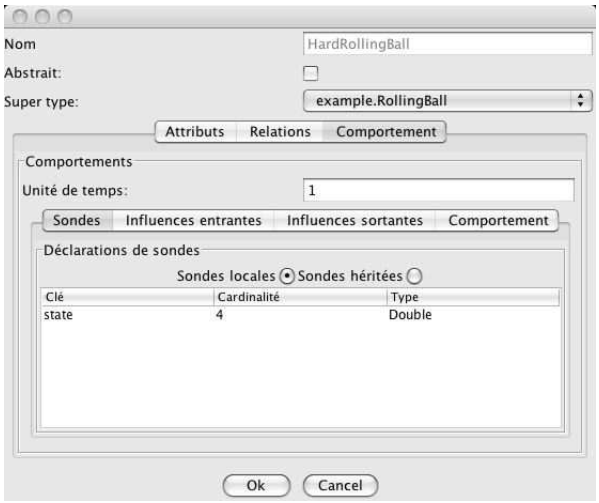

Figure 7.5: The rolling ball category with the probes panel

- the probes;
- the M-DEVS functions.

We assume that `RollingBall` receives kicks and observation requests and sends positions, `Kicker` sends kicks and the Observer sends observation requests and receives positions. The checking of the consistency between what is sent or received is currently very loose but can be reinforced by selecting the "verify" check-bon in the scheduler. In a future release the possibility to check for model consistency when defining the conceptual model will be enforced (at least optionally).

We shall define two identical probes: one for the `RollingBall` to signal the state change (new x0, y0, vx and vy, see figure 7.5) and one for the `Observer` for the ball position, each time it receives the actual coordinates.

These declarative parts of the dynamics being made, we have to focus on specifying each of the function of the corresponding M-DEVS model. The figure 7.6 shows how to define the initialization of the rolling ball. In the shown panel, the `LanguageState` behavior has been selected, which allows to specify the behavior with script languages. In this case, the Java scripting language has been selected (`JavaInterpreter`).

Note that we distinguish the attributes and the state of the model. The attributes define the structure of the ball for an external observer and corresponds semantically to the specification of its initial state. The state itself changes continuously, spontaneously or in response to incoming influences. In this case the state is created and initialized from the parameters.

The figure 7.7 shows the code for handling incoming external influences. The principle is to loop through the set of influences (put in the variable `externalInfluences`), to check its type for each one and compute the state change accordingly. Note that after the state change, a probe value is issued to update all the possible visualization windows.

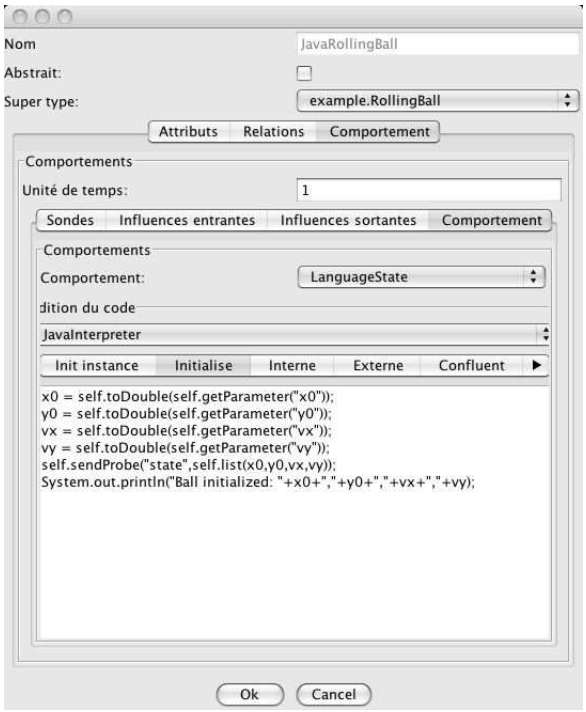

Figure 7.6: The rolling ball category with the initialize panel

The user is asked to further explore the model which is available as an example, to see how the behaviors are defined in the various scripting languages.

7.2.1 Defining the concrete model

As said before, the definition of the structure and dynamics is part of the conceptual model and cannot be run directly. From the conceptual model, a concrete and simulatable model can be instantiated. You have to open a concrete model editor. At the top of the right panel, you have a list of conceptual models you can take your definitions from. The figure 7.8 shows a window in which a model has been built by creating an instance of each of the categories (an instance of clock has been added to define the time rate at which the observer samples the rolling ball). In this figure, each port is linked to the proper entity. The drawing panel uses a modified UML object diagram. The links are named (which is not the case in UML). As in UML, the name of the instances is optional and for documentation purpose only.

The actual structure of an individual is not only composed by its links but also by the values of its attributes (interpreted as the specification of the initial state of the simulation). By editing an individual, the dialog of the figure 7.9 appears where you can change the name of the individual (optional), trace or untrace the individual[2], define or change the attribute values.

[2]while tracing in the scheduler traces the posted and sent influences, tracing an individual traces the call to the M-DEVS functions.

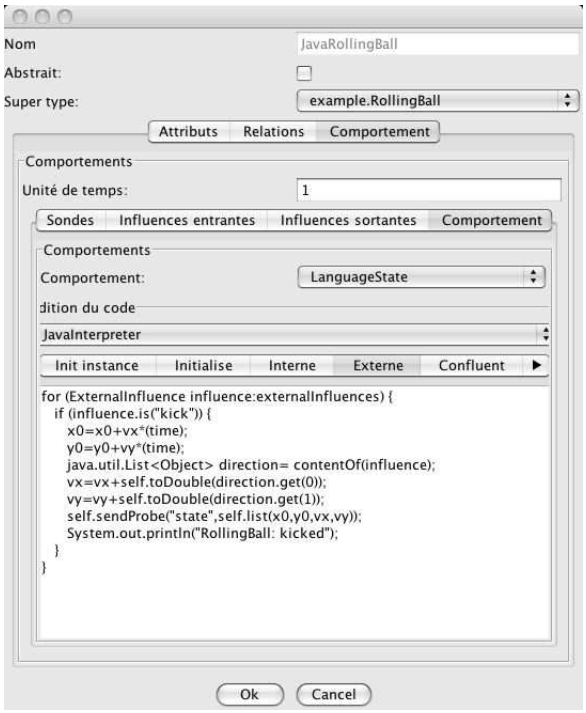

Figure 7.7: The rolling ball category with the external transition panel

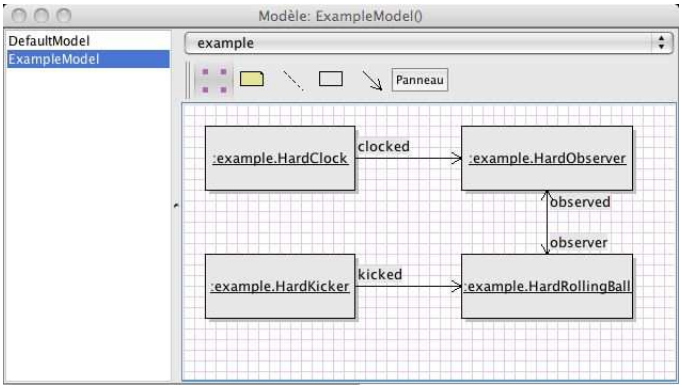

Figure 7.8: The concrete model as an instance of the conceptual model.

Figure 7.9: The edition dialog for an individual.

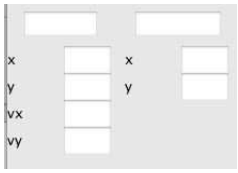

Figure 7.10: The view on the rolling ball state

Once all the model has been instantiated and all the parameters defined (a further version should also check for the model completeness), the user can open the scheduler, select the model to run, initialize and run it, either step by step or in a single run until the end date is reached as described in more details in the chapter 8.

In addition, a visualization window can be opened. For example, a possible view looks like the figure 7.10 and is updated each time the individuals change[3]. The top left panel displays the clock value, the top right panel displays "KICKED" for some time each time the kicker is issuing a kick, the bottom left panel displays the rolling ball state (updated only when kicked) and the bottom right panel displays the actual position of the ball at each time step.

Such a display cannot be created interactively yet. A number of visualization items can be created, positioned within a control board and linked to the individuals receiving its probes and using them to update the visualization. An editor for such a control panel (including the possibility to change the parameters shall be available in a near future.

7.3 The stupid model

Coming soon.

[3]Sorry if we did not program a panel to visualize trajectories yet.

Chapter 8

The scheduler

This chapter is really about running simulations. The concrete models one wants to run are available from the drop down menu on the top of the scheduler window (see 8.1). All the models defined in the concrete model editor are shown in this drop down menu to select from. Additionally, files can be loaded within the scheduler if saved in the scheduler format from the concrete model editor. This possibility is offered to deliver turn key models to be run independently of all the previously described editors.

A concrete model has to be selected from the list on the left. The initialize button shall actually generate the simulation model out of the concrete model description. The first step shall initialize the simulation model (the time shall remain at 0). Further steps shall advance the time depending on the closest scheduled next date.

In the scheduler window, the first item opens an inspector to visualize the list of all created entities (see 8.2). This list is updated during the simulation to reflect the current list of entities. Clicking on an entity opens an entity inspector to monitor what is going on in the given entity (see figure 8.3). The panel is divided in four panes:

- the first pane lists the current parameters of the entity and their values;
- the second pane is the list of current ports with the list of entities their are associated to;
- the third pane is used for managing the probe observers;
- finally the fourth pane displays the warning messages when necessary.

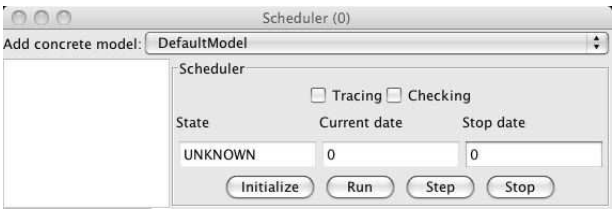

Figure 8.1: The scheduler window.

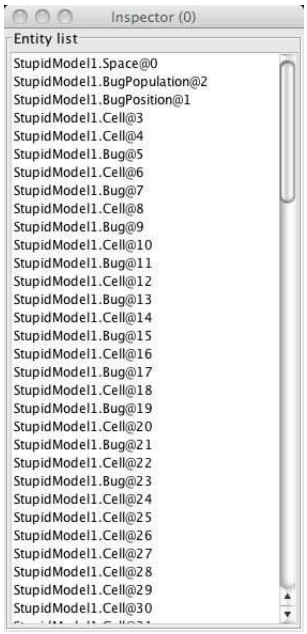

Figure 8.2: The main inspector window.

The most important pane certainly is the third pane because it monitors what is going on inside of the inspected entity. It is composed of a drop-down menu for selecting a probe observer and a panel to display the probe observer when it is displayable. By default, two probe observers are available:

- the probe view which displays the probes when received one after the other. A button to clear the display is available if necessary;
- the probe output which send the probes to a file. When selecting the probe observer, a file name as well as a separator string is asked. The resulting file can be loaded in excel or any similar tool.

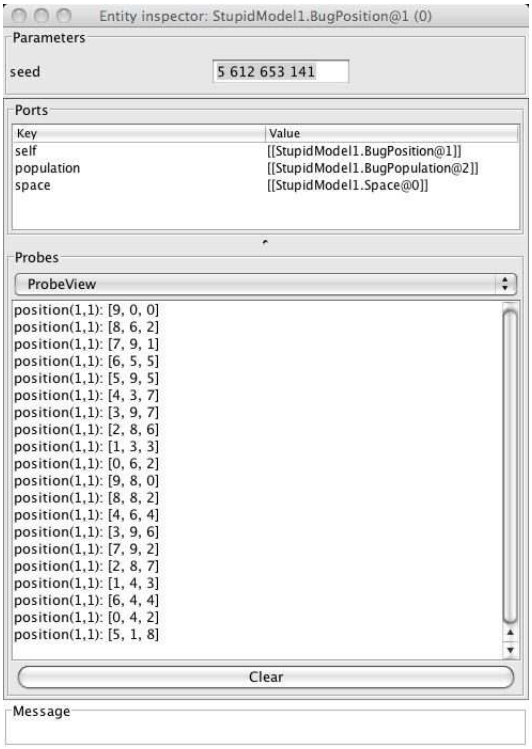

Figure 8.3: The entity inspector window.

Appendix A

Introduction to Scheme

Scheme is a functional language close to Lisp but with a purer semantics. Roughly speaking only two constructs are provided in scheme:

- the *function* (called procedure in the Scheme community) written: `(lambda <parameters> <body>)` where `parameters` is a list of parameter names and `body` is a sequence of expressions.
- the *application* written `(<function> <arg`$_1$`>...<arg`$_n$`>)` where `function` is a function as defined before and `arg`$_i$ are expressions.

Of course, an *expression* is either a function or an application. This seems overly simplistic but it has been shown that it is enough to express any computation one could dream of. Nevertheless, the resulting syntax would become unreadable for any reasonable computation. The simplest way to overcome this problem is to provide the possibility to associate names to expressions with the form: `(define <name> <expressions>)`. A number of names have been predefined in Scheme for all the current arithmetic operations as well as the operations on very common data structures.

By the way, `define` is not a function name but the name of a syntactic form which is transformed behind the scene in a proper application. The set of possible syntactic forms can itself be extended, parameterizing the Scheme interpreter with high level constructs at will (not explained in this introduction).

A structure or *object* is also called a literal expression is of the form: `(quote <something>)` or (alternatively) `'<something>`. The *something* is either:

- a number
- #t and #f
- a character #\.
- a string `"..."`
- a symbol
- a pair `(<something`$_1$`> . <something`$_n$`>)` or a list `(<something`$_1$`>...<something`$_n$`>)`
- a vector `#(<something`$_1$`>...<something`$_n$`>)`

The first four categories do not need the quote because they self-evaluate, i.e. their value is themselves.

Finally, an additional power is acquired by the relationships between structures (or objects) and expressions. Of course, expressions transform structures into structures (it is what functions or all about). The nice thing is that `(eval <exp>)` transforms the structure produced by the expression into an expression...and computes its value as well. Therefore, one can write programs producing programs which are further executed.

This appendix is not suppose to give a full course on Scheme but just provide a summary of the most common definitions for reference, including the definitions introduced for use within Mimosa.

A.1 Control syntax

As in any language, there are some constructs for the usual control structures: the sequence, the conditional and the loop.

`(define <symbol> <exp>)`	the definition
`(set! <symbol> <exp>)`	to change the definition
`(begin` $\langle exp_1 \rangle \ldots \langle exp_n \rangle$`)`	the sequence of expressions
`(if <exp>` $\langle exp_{true} \rangle$ $\langle exp_{false} \rangle$`)`	the conditional
`(cond (<exp`$_1$`> ...)...(else ...))`	the multiple contitional
`(or` $\langle exp_1 \rangle \ldots \langle exp_n \rangle$`)`	sequence until true
`(and` $\langle exp_1 \rangle \ldots \langle exp_n \rangle$`)`	sequence until false

The loop is more complicated with the form `(do (<iter`$_1$`>...<iter`$_n$`>) (<cond> ...) ...)` where `iter`$_i$ is a variable of iteration of the form `(<var`$_i$`> <exp`$_{init}$`> <exp`$_{step}$`>)` with a variable name, an initialization expression and a step computation expression, the condition expression must be true for stopping the iteration and the corresponding expressions are computed accordingly.

Finally, one must introduce the binding construct to create local variables for various purposes:

`(let ((<sym`$_1$`> <exp`$_1$`>) ...) <exp`$_i$`>...)`	parallel binding
`(let* ((<sym`$_1$`> <exp`$_1$`>) ...) <exp`$_i$`>...)`	sequential binding
`(letrec ((<sym`$_1$`> <exp`$_1$`>) ...) <exp`$_i$`>...)`	complete binding

The main difference is that the association of values to symbols are avalaible from the body alone in the first case, directly after the definition (and then for the next definitions) in the second case and from the start in the third (allowing self reference).

A.2 Booleans

There are two booleans #t and #f which are two symbols which evaluates to themselves. Apart from `and` and `or`, we also have the following functions:

`(boolean? <exp>)`	tests if boolean
`(not <exp>)`	the negation
`(eq? <exp`$_1$`> <exp`$_2$`>)`	strict equality
`(eqv? <exp`$_1$`> <exp`$_2$`>)`	slight extension of strict equality
`(equal? <exp`$_1$`> <exp`$_2$`>)`	recursive (or structural) equality

A.3 Numbers

Scheme recognizes the integers (e.g. 51236457), rationals (e.g. 6235645/23672573), reals (e.g. 4.6565e-3) and complex numbers (e.g. 3+5i). The main distinction is between exact and inexact representations of these. The predefined functions are:

`(number? <exp>)`	tests if number
`(complex? <exp>)`	tests if complex
`(real? <exp>)`	tests if real
`(rational? <exp>)`	tests if rational
`(integer? <exp>)`	tests if integer
`(exact? <exp>)`	tests if exact
`(inexact? <exp>)`	tests if inexact
`(zero? <exp>)`	tests if zero
`(positive? <exp>)`	tests if positive
`(negative? <exp>)`	tests if negative
`(odd? <exp>)`	tests if odd
`(even? <exp>)`	tests if even
`(=` $x_1 \ldots$`)`	equality
`(<` $x_1 \ldots$`)`	monotonically increasing
`(>` $x_1 \ldots$`)`	monotonically decreasing
`(<=` $x_1 \ldots$`)`	monotonically non decreasing
`(>=` $x_1 \ldots$`)`	monotonically non increasing
`(abs` x`)`	the absolute value of the number
`(min` $x_1 \ldots$`)`	the min of the numbers
`(max` $x_1 \ldots$`)`	the max of the numbers
`(+` $z_1 \ldots$`)`	the sum of the numbers
`(-` $z_1 \ldots$`)`	the difference of the numbers
`(*` $z_1 \ldots$`)`	the product of the numbers
`(/` $z_1 \ldots$`)`	the quotient of the numbers
`(quotient` n_1 n_2`)`	the quotient of the numbers
`(remainder` n_1 n_2`)`	the remainder of the numbers
`(modulo` n_1 n_2`)`	the modulo of the numbers
`(gcd` $n_1 \ldots$`)`	the greatest common divisor of the numbers
`(lcm` $n_1 \ldots$`)`	the lowest common multiple of the numbers
`(numerator` q`)`	the numerator of the rational
`(denominator` q`)`	the denominator of the rational
`(floor` x`)`	the floor of the real
`(ceiling` x`)`	the ceiling of the real
`(truncate` x`)`	the truncate of the real
`(round` x`)`	the round of the real
`(real-part` z`)`	the real part of the complex
`(imag-part` z`)`	the imaginary part of the complex

As well as most transcendant functions.

A.4 Dotted pairs and lists

The most common data structure in Scheme is the dotted pair written `(<left> . <right>)`. A list `(<elt`$_1$`> <elt`$_2$`> ... <elt`$_n$`>)` is nothing but `(<elt`$_1$`> .`

(<elt$_2$> . ... (<elt$_n$> . ())...) where () is the empty list. We have the following functions:

(pair? <exp>)	tests if dotted pair
(null? <exp>)	tests if empty list
(list? <exp>)	tests if empty list or dotted pair
(car <exp>)	left of dotted pair or first element of list
(cdr <exp>)	right of dotted pair or rest of list
(set-car! <pair> <obj>)	modifies left of dotted pair
(set-cdr! <pair> <obj>)	modifies right of dotted pair
(list <obj$_1$> ... <obj$_n$>)	creates a list
(length <list>)	length of a list
(reverse <list>)	reverse of a list
(list-tail <list> <k>)	the k-th rest of a list
(list-ref <list> <k>)	the k-th element of a list
(append <list$_1$> ... <list$_n$>)	append of lists
(memq <object> <list>)	member using eq?
(memv <object> <list>)	member using eqv?
(member <object> <list>)	member using equal?

An additional structure is the so-called a-list which is a list of pairs whose car is considered as a key and the cdr as the associated value. The related functions are:

(assq <object> <list>)	has key using eq?
(assv <object> <list>)	has key using eqv?
(assoc <object> <list>)	has key using equal?

and returns the found pair if any, #f otherwise.

A.5 Mimosa primitives

For Mimosa, we added three very common control structures for better readability:

(when <cond> <exp$_1$>...<exp$_n$>)	executes if #t
(unless <cond> <exp$_1$>...<exp$_n$>)	executes if #f
(for (<var> <list>) <exp$_1$>...<exp$_n$>)	a simple loop over a list
(times (<var> <nb>) <exp$_1$>...<exp$_n$>)	a simpler loop repeated nb times

Some functions are provided to access the Mimosa random generator:

(newRandom <seed>)	creates a random generator
(nextBoolean <random>)	generates a boolean randomly
(nextInt <random> <n>)	generates an integer from 0 to n
(nextDouble <random>)	generates a real from 0 to 1

Finally, the access to the DEVS entity functionalities are provided as follows:

- the variable self is linked to the current Java state;
- for each parameter, the variable with the same name is defined with the associated value within the global context. It can additionally be accessed through the function (getParameter <sym>);
- when a script for a DEVS function is called, the global variable time is linked to the duration elapsed since the last internal or external transition;

- each influence is a Java object whose structure can be accessed by the following functions:

`(is <influence> <name>)`	#t if the influence has the given name
`(contentOf <influence>)`	the list of arguments
`(getAllInfluences)`	the list of incoming influences
`(getInfluence <name>)`	the list of influences of the given name
`(getInternalInfluence)`	the internal influence

- the various events can be posted with the following functions:

`(port <sym>` $n_1 \ldots n_n$`)`	creates a port
`(sendExternal <port> <sym> <exp`$_1$`>...<exp`$_n$`>)`	post an external event
`(sendInternal` n `<sym> <exp`$_1$`>...<exp`$_n$`>)`	post an internal event
`(sendLogical <port> <sym> <exp`$_1$`>...<exp`$_n$`>)`	post a logical event
`(reply <influence> <sym> <exp`$_1$`>...<exp`$_n$`>)`	reply to an influence
`(sendProbe <sym> <exp`$_1$`>...<exp`$_n$`>)`	post a probe

  A port can be a string or a symbol when there is no indices.

Finally the structure changes can be made through the following functions:

`(portRef <port`$_1$`>...<port`$_n$`>)`	creates a port reference
`(pair <sym> <exp>)`	creates a pair for the parameters
`(parameters <pair`$_1$`>...<pair`$_n$`>)`	creates parameters from the pairs
`(addPort <portref> <category> <traced> <parameters>)`	creates a new entity
`(linkPort <portref`$_1$`> <portref`$_2$`>)`	links referenced ports
`(removePort <portref>)`	removes a references port

Bibliography

[1] http://bergel.eu/athena/.

[2] http://herzberg.ca.sandia.gov/jess/.

[3] http://www.alice.unibo.it:8080/tuprolog/.

[4] http://www.beanshell.org/.

[5] http://www.gnu.org/software/kawa/.

[6] http://www.jython.org/.

[7] Jacques Ferber and Jean-Pierre Müller. Influences and reaction: a model of situated multiagent systems. In Mario Tokoro, editor, *Proceedings of 2nd International Conference on Multi-Agent Systems*, pages 72–79, Kyoto, Japan, December 1996. AAAI.

[8] Jean-Pierre Müller. The mimosa generic modeling and simulatiion platform: the case of multi-agent systems. In Herder Coelho and Bernard Espinasse, editors, *5th Workshop on Agent-Based Simulation*, pages 77–86, Lisbon, Portugal, May 2004. SCS.

[9] Jean-Pierre Müller. Mimosa: using ontologies for modelling and simulation. In *Proceedings of Informatik 2007*, Lecture Notes in Informatik, September 2007.

[10] Jean-Pierre Müller and Pierre Bommel. *An introduction to UML for modeling in the human and social sciences*, volume Agent-based Modelling and Simulation in the Social and Human Sciences, chapter 12. Bardwell Press, 2007.

[11] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, 2000.