Statecharts Environment

Spencer Borland
Documentation
November 2003

Modeling, Simulation & Design Lab
McGill University

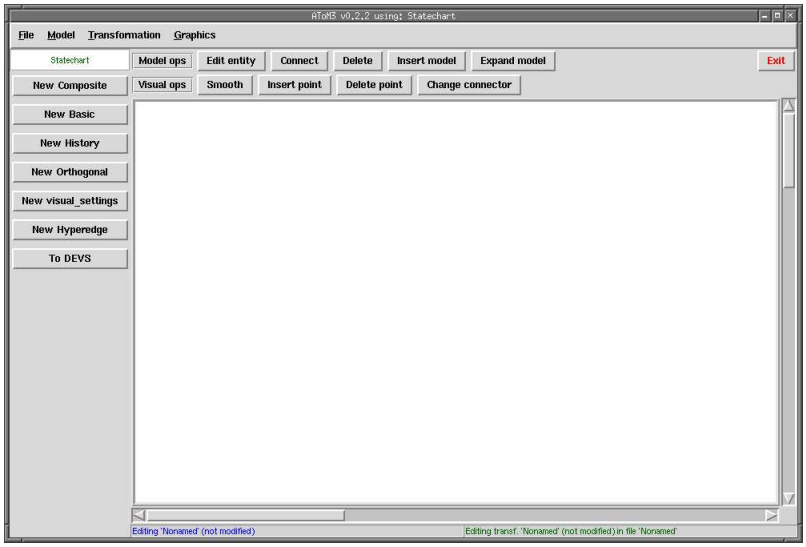

Figure 1: The starting screen.

1 Introduction

This appendix explains the usage of the Statecharts to DEVS environment in AToM^3. It discusses how to build graphical Statechart models, how to transform a Statechart to a DEVS model and then how to execute the DEVS models in real-time.

2 Buttons Overview

When the environment starts you should see the screen in Figure 1. The left-hand toolbar features buttons which are specific to Statecharts while the top toolbar provides more general functions. The following sub-sections explain the usage of each of the Statechart buttons. For more information on the top toolbar, please refer to the AToM^3 website at `http://atom3.cs.mcgill.ca`.

2.1 `New Composite`, `Basic`, `History`, `Orthogonal` and `Hyperedge` Buttons

These buttons do exactly what their names imply. For example, clicking on `New Composite` then clicking on the canvas will create a new composite state. Note that while building Statechart models you will almost never need to use the `New Hyperedge` button. This is present only for the purpose of building Statechart graph grammars.

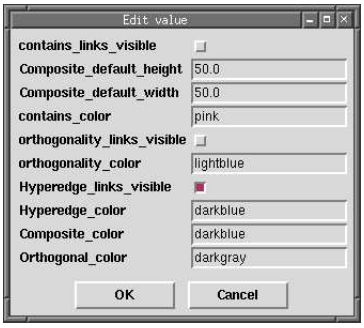

Figure 2: The visual settings dialog.

2.2 `New visual_settings` Button

This button draws a large square on the canvas. When edited (ie. by clicking the `Edit entity` button then on the `visual_settings` object) the modeler will be presented with a series of visual options in a dialog box. This dialog can be seen in Figure 2. Each of the options change visual settings based on their descriptive names.

2.3 `To DEVS` Button

The `To DEVS` button converts a Statechart model to an equivalent DEVS model. The output is written in `python` and is for usage with the `pythonDEVS-RT` package. The output file containing the `pythonDEVS` code is set using the global model attribute `devs_output_dir`. The DEVS model which is output is a subclass of `CoupledDEVS`. This represents the root of the Statechart. It is the responsibility of the modeler to combine this model with the `Executor` and any other desired packages to construct a working application. This will be discussed in more detail.

3 Operational Overview

Now that you know how to put basic components on the canvas, a discussion on how to connect them in various ways, is given. The following sub-sections discuss simple operations and how to perform them.

3.1 Connecting States with a Hyperedge

In order to connect two states with a `Hyperedge`, first click the `connect` button located on the top toolbar. Next click the source state and then the destination state. If the source state is of type `Basic`, then the Hyperedge will be drawn automatically. If the source state is of type `Composite` The modeler will be presented with another dialog as seen in Figure 3.

Choosing the `Hyperedge` option will draw the desired connection between the states. If you wish to create an *n*-ary Hyperedge, simply perform this operation again.

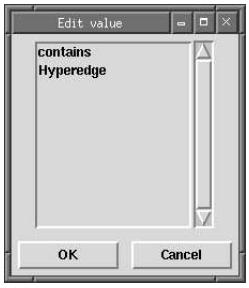

Figure 3: The connect dialog.

However, first click the middle segment of the transition as the source, and the desired state as the destination.

3.2 Nesting States

Nesting a state within a `Composite` state is very similar to connecting them via a `Hyperedge`. However, when presented with the connection dialog, choose the `contains` option. This will nest the destination state within the source state. The container state will automatically resize to envelop everything within itself.

3.3 Destroying a `contains` Relation

If you wish to destroy the `contains` relation between two states, you should use the `visual_settings` component. Put a `visual_settings` on the canvas, and edit it. Set the `contains_links_visible` option and click `OK`. Now you will be able to see the contains links. Simply click the `Delete` button on the top toolbar then on the link you wish to delete.

3.4 Adding History

To add a history state, simply add the history state anywhere on the canvas, then nest it within the desired composite state. You can connect hyperedges to the history state, but outgoing history hyperedges are not supported in this version. Remember that the history state is not a state at all. Rather it represents a section of a transition.

3.5 Editing States

Editing states requires clicking the edit button, then on the component to be edited. When editing a state, the modeler will be presented with the dialog in Figure 4. The options presented in this dialog are explained in table 1.

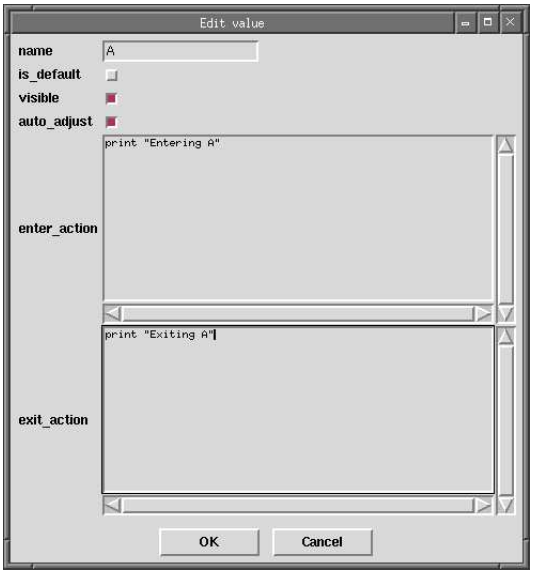

Figure 4: The edit state dialog.

Table 1: Options for the state editing dialog.

Option	Description
`name`	The name of the state.
`is_default`	Indicates if the state is default.
`visible`	Will hide/reveal the interior of a `Composite` state.
`auto_adjust`	Should always be set.
`enter_action`	The action to be executed upon entering the state.
`exit_action`	The action to be executed upon exiting the state.

3.6 Creating n-ary Transitions

To create a transition with more than one source state and/or more than one destination state perform the following steps. To create another source state simply click on the source state, then on the transition. A link will be created. To add a destination state click on the transition, then on the destination state and a link will be created. Recall that these hyperedges move the system from a state, x, which is the product of x' basic states where $x' > 1$, to a new state y which is the product of y' basic states where $y' > 1$. This means that you must ensure that each individual source and destination state resides in a separate orthogonal component.

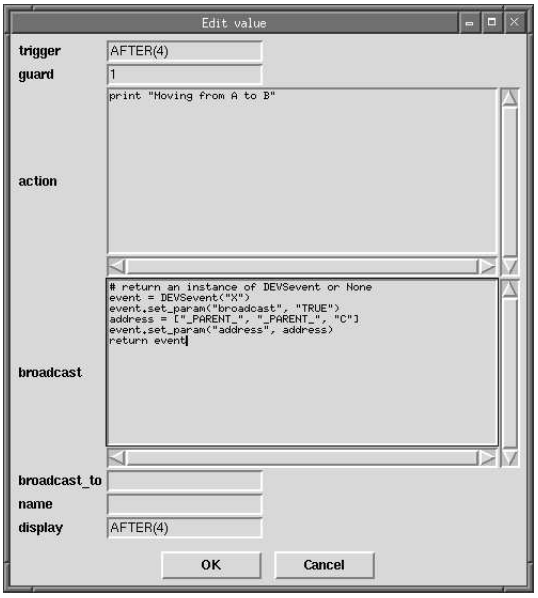

Figure 5: The edit transition dialog.

3.7 Editing Transitions

Editing transitions requires clicking the edit button, then on said transition. The dialog in Figure 5 is presented when editing transitions. Each option is explained in Table 5.

Table 2: Options for the transition editing dialog.

Option	Description
`trigger`	The transition trigger. See section 3.8 for more information.
`guard`	Valid `python` boolean expression.
`action`	The action to be executed when the transition fires.
`broadcast`	Used to broadcast or narrowcast events. See section 3.7 for more information.
`broadcast_to`	Deprecated.
`name`	Deprecated.
`display`	The text to be displayed for the transition.

3.8 Broadcasting

The broadcast field of the transition edit dialog is used to narrowcast and broadcast events. The modeler can write python code to construct an event in a variable, `x`. Then

the last statement should be `return x` which will indicate that `x` is the instance to be broad or narrowcasted.

The `x` instance should somehow be an instance of `RT_DEVS.EVENT` (you can also use `RT_DEVS.DEVSevent` which is an alias for the same class). This object has special methods for setting and getting event fields. These methods are `get_param(name)` which returns the field value associated with `name`. The method `set_param(name, value)` sets the field `name` to `value`.

Any parameters may be added to an event, but there are some special parameters that the modeler should know. These special parameters are listed in table 3.

Table 3: Special Event Fields.

Field	**Description**
`broadcast`	`"TRUE"` or `"FALSE"`. Indicates if the event is to be broadcasted or not. If not specified, "FALSE" is implied.
`address`	List of port names. The list is used by DEVS atomic relays to route events to be narrowcasted. The special name `"_PARENT_"` is used to indicate the containing state. The name `"_ROOT_"` is a direct route to the `"_ROOT_"` outport of the root coupled DEVS.

Note that the modeler may use both the `address` parameter to narrowcast an event coupled with broadcasting. The effect is that once the event reaches its destination via the `address` parameter, it will then be broadcast only within the destination state.

For example, consider the statechart model in Figure 6. The transition from state `PASSED_FF` to `WAIT_SEEK_DONE` generates a `SEEK` event. This indicates synchronization between the `Audio` component and the `Display` component. Note that the only transition which responds to the `SEEK` event is that which emanates from the `SEEKING` state. Thus, broadcasting the event is not needed, since its exact route is known. Thus in order to route the event from its place of origin to the `SEEKING` state, the `address` parameter should be specified as `["_PARENT_", "_PARENT_", "_PARENT_", "_PARENT_", "Audio", "SEEKING"]`.

Notice that this forces the event to travel from `PASSED_FF` to `TIME_PASSED` to `TIME` to `Display` to the `ROOT` to `Audio` to `SEEKING`. Note that if you wish an event to be *heard* without broadcasting you should narrowcast it to the source state of the desired transition.

3.9 Triggers

Triggers can either be a string indicating the name of the trigger or an `after` expression. To specify an event name, simply enter it without quotes. A timed trigger has the following syntax: `AFTER(exp)` where `exp` is a valid python expression evaluating to a number. Be sure to enter `AFTER` in capitals.

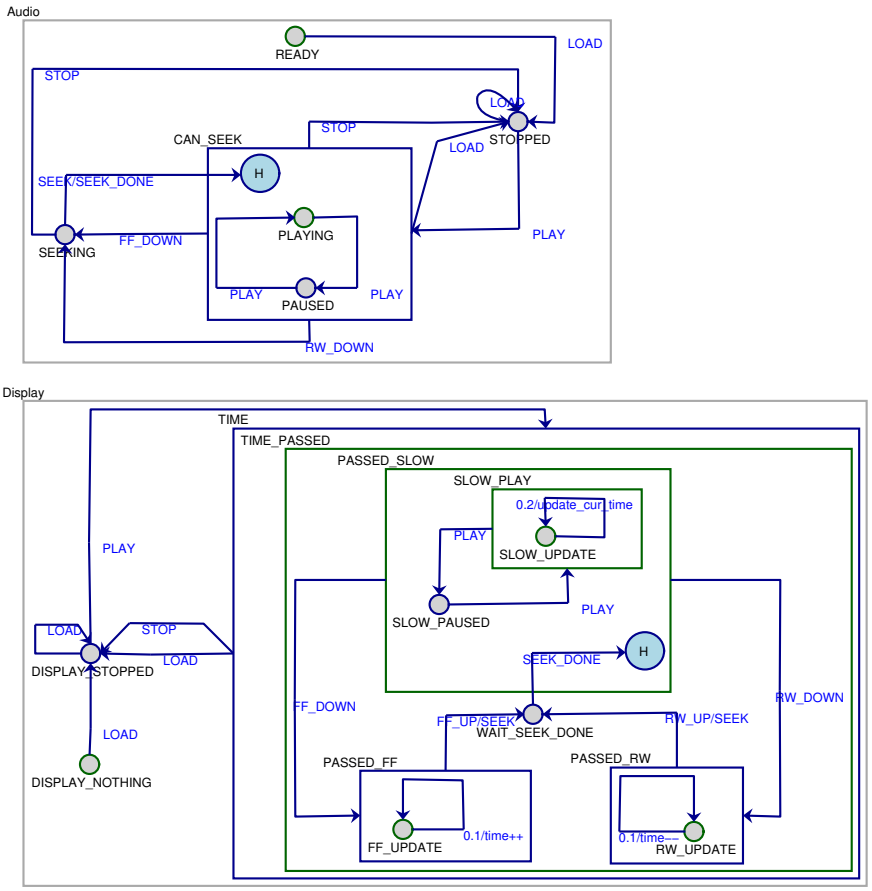

Figure 6: An MP3 Player.

4 Executing a Generated DEVS Model

A pythonDEVS model on its own will not do anything. It must be linked with an Executor in order to put it into motion. This is as simple as passing an instance of your model to the RT_DEVS.Executor constructor. Once you have constructed an Executor instance it can be executed by calling the execute() method. This will execute your Statechart model in real-time.

5 Building Larger Applications

In this section, the example in Figure 7 will be used. This Statechart is meant to be the control layer between the MP3 decoder/audio hardware and the player's graphical interface (See Figure 8. The MP3 decoder and audio hardware as well as the graphical interface are implemented as DEVS components themselves. This makes it very easy to combine generated Statechart models with these DEVS models. All the files used in the application are listed in table 4.

Table 4: Other files used in the DEVSamp application.

File	Description
main.py	The file which glues everything together.
config.py	Contains globals.
MP3AudioDriver.py	MP3 decoder and audio hardware interface.
DEVSampGUI.py	The application's graphical interface.
DEVS_DEVSamp.py	The Statechart pythonDEVS model.

Each of the main components or globally scoped variables may be needed in various places thus it is best to instantiate them in another file, config.py, so they may be included later using the import statement.

```
import DEVS_DEVSamp, DEVSampGUI, MP3AudioDriver
GUI = DEVSampGUI.DEVSampGUI()
DEVS = DEVS_DEVSamp.ROOT()
DRIVER = MP3AudioDriver.Driver()
```

The three main components in the application are each DEVS objects. Thus, they can be coupled within a Main coupled DEVS. This will be written in the main.py file along with the code to execute() the Main model.

```
import config, RT_DEVS
class Main(RT_DEVS.CoupledDEVS):
  def __init__(self):
    RT_DEVS.CoupledDEVS.__init__(self)
    self.addSubModel(config.GUI)
    self.addSubModel(config.DEVS)
    self.addSubModel(config.DRIVER)
```

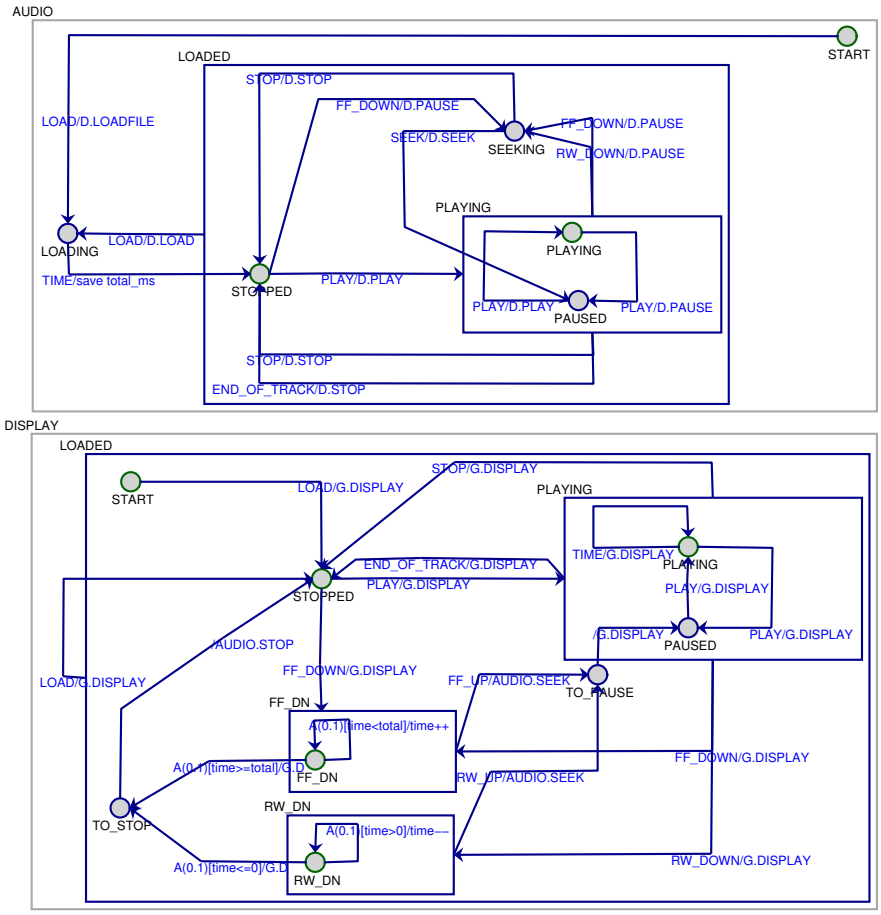

Figure 7: The MP3 Player Control Layer Statechart.

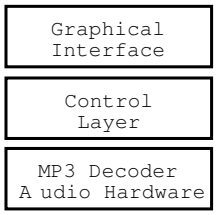

Figure 8: Application layers.

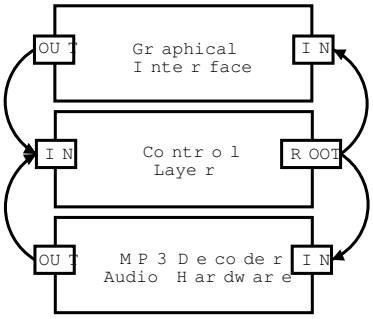

Figure 9: Port connections among the high level components.

```
    self.connectPorts(config.GUI.ports['OUT'], config.DEVS.ports['_IN_'])
    self.connectPorts(config.DRIVER.ports['OUT'], config.DEVS.ports['_IN_'])
    self.connectPorts(config.DEVS.ports['_ROOT_'], config.GUI.ports['IN'])
    self.connectPorts(config.DEVS.ports['_ROOT_'], config.DRIVER.ports['IN'])
E = RT_DEVS.Executor(Main())
E.execute()
```

Each connection is now visible in the high level diagram in Figure 9. Notice that the `_ROOT_` port is connected to the `IN` ports of both the graphical and the driver components. It would probably be better plug the `_ROOT_` port into an atomic DEVS which filters the events so they are not sent to the adjacent models when not needed. This is a non-crucial optimization. Optimizations will be discussed further in the following sections.

Also notice that it is easy to instantiate the three high-level components within the `config.py` file. This makes it easy to access them in various parts of the program. However, beware of breaking the compositional structure of the DEVS semantics. It is easy to import the driver instance at any point of the program and simply call some of its methods. However, if the DEVS model is meant to be executed on a network or in low-level electronic components, you may not have this luxury. You should always use events to communicate between components.

5.1 The `_IN_` Port

The most important port of the top-level DEVS generated from a Statechart is `_IN_`. This is only way to ensure events are heard within the walls of the Statechart model. Remember that events pushed through the `_IN_` port can use the `address` parameter to provide narrowcasting. Otherwise, you may wish to set the `broadcast` parameter to `"TRUE"` which will broadcast the event across the entire Statechart.

5.2 The `_PARENT_` Port

This port refers to the containing composite Statechart state or orthogonal component. The `"_PARENT_"` port will never be routed to a DEVS representing a basic state.

5.3 Substate Ports

There exist an output port for each substate of a particular composite Statechart state. The output port `x` will connect to the input port `_IN_` of the substate (or orthogonal region) named `x`.

5.4 The `_ROOT_` Port

An important port when linking top-level DEVS components is `_ROOT_`. You may want an output for a particular transition to be sent to one of the adjacent top-level components. Instead of fully specifying the path of the `address` parameter, you can simply set it to `_ROOT_`. This is a special port which pushes an event directly to the `_ROOT_` output port of the coupled DEVS representing the root of the Statechart. This is merely an optimization. That is, you can also specify the full address path, if desired.

5.5 Dynamically Changing an Event Path

This is made possible by intercepting a particular event, modifying its address parameter, then ensuring its output. However, this is the equivalent to responding to an event and outputting another event which happens to be the same as trigger event and has a different address.

5.6 The Top-level DEVS Component Interfaces

The modeler must have a full working knowledge of the DEVS components they are using in their application. In this case, the other top-level components are `MP3AudioDriver` and `DEVSampGUI`. A high-level description of their semantics is enough to allow the modeler to successfully interface them in an application.

5.6.1 The `MP3AudioDriver` Interface

The class to instantiate is `MP3AudioDriver.Driver`. This atomic model has one input port `IN` and one output port `OUT`. Table 5 is a description of the events which may be passed to the driver to change its state.

Table 6 is a description of events which may come out of the `OUT` port. A brief description is also given, so the modeler may interpret the event's meaning and rough timing.

5.6.2 The `DEVSampGUI` Interface

The class to be instantiated is `DEVSampGUI.DEVSampGUI` which is a sub-class of `RT_DEVS.AtomicDEVS`. The appearance of this component can be seen in Figure 10. Notice that, each of the widgets used are labeled in red. Table 7 describes the possible inputs you may give to this component. Table 8 lists all the possible outputs from this component. Each output basically corresponds to a button click. Note that the parameters are not specified because each event has the same parameters. These parameters are `broadcast` which is set to `"TRUE"` and `address` which is set to `[]`.

Table 5: Reactive Events for `MP3AudioDriver`.

Event Name	Parameters	Description
`LOADFILE`	`filename`:The name of the file to be loaded.	Loads an MP3 file.
`PLAY`		Starts playback from the current file pointer.
`PAUSE`		Stops playback.
`STOP`		Stops playback and sets the file pointer to the beginning of the file.
`SEEK`	`offset_ms`: The driver will set the file pointer such that the current time will be `offset_ms`.	Move the file pointer freely. If `offset_ms` is less than 0, the driver will move to a STOPPED state. If `offset_ms` is greater than the total track time, the driver will output an `END_OF_TRACK` event.
`CLOSE`		Closes the driver.

Figure 10: The DEVSamp Graphical Interface.

6 Simple Examples

Some simple Statecharts will be examined as well as how to build them in the Statechart modeling environment. The first example is seen in Figure 11.

Start up the AToM^3 that came with the Statechart package you downloaded. Statecharts will be the default meta-model loaded. Next put two Basic states on the canvas. Name the first *A*, the other *B* and make sure that *A* is the default state. Create a transition between the two of them and set the trigger as *X*. You can also set the display as *X*. Your statechart should now look like the one in figure 12. Recall that there is an implicit composite state enveloping this model, named `ROOT`. Next specify some enter/exit actions for both *A* and *B* which simply print something. For example the edit

Figure 11: A Simple Statechart.

Table 6: Output Events for MP3AudioDriver.

Event Name	Parameters	Description
PLAYING_PULSE_TIME	offset_ms:The current time in ms. broadcast: "TRUE". address: [].	Once the driver is put into the playing state, these events are pulsed out approximately every 0.2 s in order to indicate the current time.
LOADING_PULSE_TIME	total_ms:The total track time in ms. broadcast: "TRUE". address: [].	This event is only output right after receiving a LOADFILE event. It indicates the total time of the track just loaded.
END_OF_TRACK		Indicates the end of the current track has been reached. Can be output while playing or seeking.

Figure 12: The Statechart in AToM^3.

dialog for *A* should look something like the one if Figure 13. You can do the same for the transition from *A* to *B*. Its edit dialog should look something like the one in figure 14.

Now, this Statechart model will be transformed to a DEVS model. First the output directory must be specified. Select ”Edit Model Attributes” from the ”Model” menu. Specify a full physical path filename, of a path relative to the current directory in the space for ”devs_output_file”. Finally, click the ”To DEVS” button on the left toolbar. You will see some output on stdout which should look something like the following.

```
START TRANSFORMATION
ABOUT TO BUILD STATECHART
ABOUT TO VERIFY SYNTAX
ABOUT TO BUILD DEVS
ABOUT TO EMIT DEVS
END TRANSFORMATION
```

This means everything went according to plan. If you see any python errors, then something went wrong and you must retrace your steps.

You can now examine the output file, in this case THEMODEL.py, for the presence of the Statechart model as a DEVS model. In the same directory as the output file you should place the RT_DEVS.py package. Finally, a main.py file is constructed where everything is glued together. The main.py file should look something like the following.

```
from RT_DEVS import Executor
```

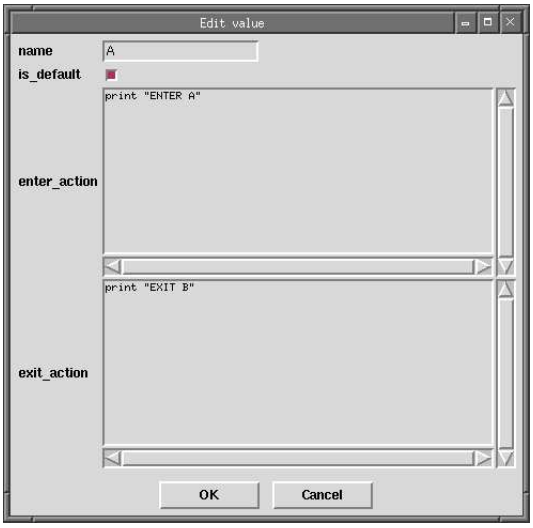

Figure 13: *A*'s edit dialog in AToM^3.

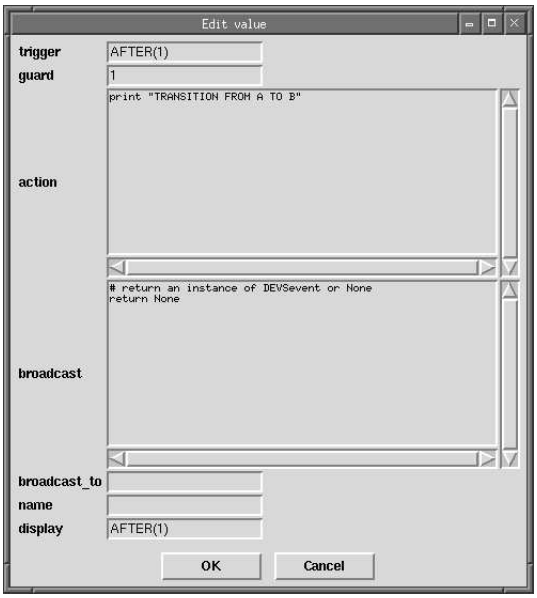

Figure 14: Transition dialog in AToM^3.

Table 7: Inputs to DEVSampGUI.

Event Name	Parameters	Description
DISPLAY	time:A tuple of the form (mins,secs) [7]. info: A string [8]. state: A string which should indicate the state of the player [9]. playpause: A string indicating which to write on the PLAY/PAUSE button.	Sets the various display widgets on the graphical interface.

Table 8: DEVSampGUI Outputs.

Event Name	Description
PLAY	Button 1 click.
STOP	Button 4 click.
LOAD	Button 3 click.
QUIT	Button 6 click.
RW_DOWN	Button 2 down.
RW_UP	Button 2 up.
FF_DOWN	Button 5 down.
FF_UP	Button 5 up.

```
from THEMODEL import ROOT
Executor(ROOT()).execute()
```

Now execute the main file using the command `python main.py`. This will start by producing the following output.

```
ENTER A
```

Then after about a second, you will see the rest of the output.

```
TRANSITION FROM A TO B
EXIT A
ENTER B
```

Now suppose a small modification is made to the original Statechart design (Figure 15). The transition from A to B is now triggered by the event X. The transition edit

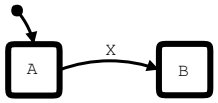

Figure 15: The Simple Statechart slightly modified.

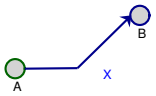

Figure 16: The modified Statechart in AToM^3.

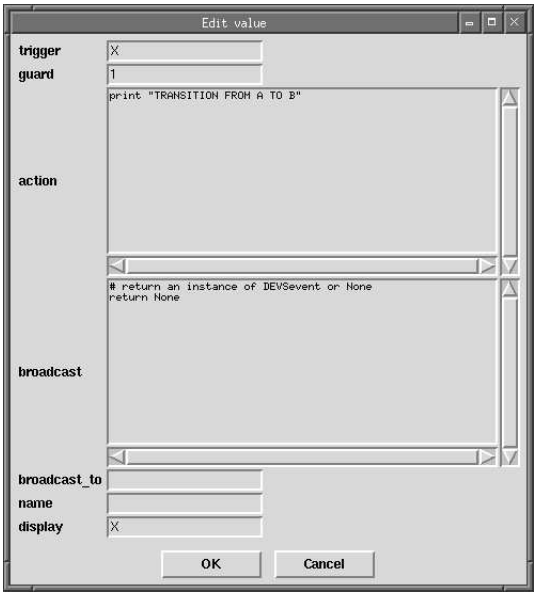

Figure 17: Modified transition dialog in AToM^3.

dialog should looking something like the one in figure 17. Your Statechart should now look like the one in Figure 17.

Again, click the "To DEVS" button and the Statechart model has been transformed to a DEVS model. However, if an attempt is made to execute this model, state *A* will never be exited. This is due to the fact that an *X* event is never generated. In order to generate an *X* event, another AtomicDEVS model could be contructed which would output an event to the `ROOT` Statechart. This model can be placed into the `main.py` specified below.

Notice that after 5 seconds the event *X* is output with a zero length address parameter and a set broadcast parameter. This event is poked out the `OUT` port. Thus, there must be a coupled DEVS which connects this `OUT` port to the `_IN_` port of the `ROOT`. This coupled DEVS will be called `Main` and should also be placed in the `main.py` file specified below.

The final `main.py` file looks like the following.

```
from RT_DEVS import Executor,AtomicDEVS,CoupledDEVS,INFINITY,EVENT
from THEMODEL import ROOT

class Generator(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self)
    self.states = {"A":1,"B":2}
    self.ports = {"OUT":self.addOutPort("OUT")}
    self.state = self.states["A"]
  def timeAdvance(self):
    if self.state == self.states["A"]: return 5
    else: return INFINITY
  def outputFnc(self):
    if self.state == self.states["A"]:
      self.poke(self.ports["OUT"], EVENT("X", [("address", []), ("broadcast", "TRUE")]))
  def intTransition(self):
    if self.state == self.states["A"]: return self.states["B"]
    else: return self.state

class Main(CoupledDEVS):
  def __init__(self):
    CoupledDEVS.__init__(self)
    G = self.addSubModel(Generator())
    R = self.addSubModel(ROOT())
    self.connectPorts(G.ports["OUT"], R.ports["_IN_"])

Executor(Main()).execute()
```

Notice that instead of executing the `ROOT` model, the `Main` model is executed.