
The Moses Tool Suite — A Tutorial

Version 1.2

Robert Esser
Department of Computer Science
University of Adelaide
Adelaide 5005, South Australia
esser@computer.org

Jörn W. Janneck, Martin Naedele
Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
{janneck,naedele}@tik.ee.ethz.ch

The Moses Project
May 2001



Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich

Contents

1	Introduction	3
1.1	Starting Moses	3
2	A Simple Example	5
2.1	Creating a project folder	5
2.2	Creating a Moses Component	6
2.3	Creating the <i>two_to_one</i> component	7
2.4	Creating the <i>two_a_plus_b_sqr</i> component	12
2.5	Creating the <i>testbed</i> component	18
2.6	Animating the <i>testbed</i> component	21
2.7	Correcting the <i>testbed</i> component	24
3	The Graph Editor Commands	29
3.1	The Tool Bar	29
3.2	The Menu Bar	30
4	Conclusion	33

Date	Section	Who	Changes
Nov 1999			Initial Version 1.01
May 2001		JB, SK	Revised for new repository structure

Table 1: Revision History

1 Introduction

Moses is a rich and powerful modelling and simulation environment. Systems in Moses are modelled as cooperating components that can be written in a number of different formalisms. The standard release of Moses supports the following three formalisms:

Time Petri nets A high level Petri net augmented with *time* and *container places*;

Process Networks A very generic form of Kahn's process networks where nodes can contain instances of any component;

Harel State Charts This hierarchical state machine formalism allows control oriented (sub) systems to be defined naturally.

In addition the user can define additional formalisms using the Graph Type Description Language (GTDL). How this is done is beyond the scope of this tutorial, however interested users are referred to the Moses user manual for further information.

This tutorial describes how to start Moses followed by the creation and animation of a simple example.

1.1 Starting Moses

Moses is written in the Java 2 language and requires either the Java runtime environment (JRE), if only predefined models are to be viewed and animated or the Java software development kit (SDK), if models are also to be created. In this tutorial you will be shown how to create models using a number of different formalisms and hence the Java SDK will need to be installed on your machine.

To test whether you have the correct version of Java installed on your machine enter **java -version**. The version information returned should be greater than version *1.2*. To ascertain whether the Java SDK is installed enter **javac**. If this command executes successfully, showing the possible options, then you have the prerequisites required to run Moses.

To start Moses first ensure that your current directory contains the *moses.jar* file and the *Basic.rep* directory. Then in a terminal window enter:

```
java -jar moses.jar -d Basic.rep
```

When Moses successfully starts the splash screen similar to that shown in figure 1 will appear. This will be followed by the Moses desk-top appearing.

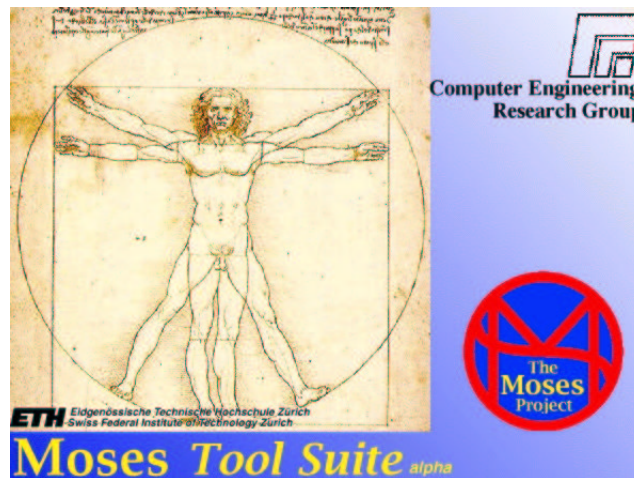


Figure 1: The Moses start-up splash screen

Upon starting Moses the desk-top shown in figure 2 contains just a single window in which all elements in the *Basic* repository are shown. The window is set up in two panes, tools on the left and all Moses objects

contained in the repository on the right. Initially only a folder containing configuration files *config*, tool configurations *Tools* and a library *Lib* is contained in this repository. The library contains *Components* and *Formalisms* with the *Formalisms* folder containing the 3 predefined component formalisms. It can also be seen that the elements in the repository are hierarchically structured. This enables the user to separate repository elements.

The Moses desk-top also contains menu items that are useful in any context within the Moses environment. Perhaps initially the most useful item is found under the **Help** menu. Here the on-line help system can be accessed. This help is in HTML format and can be viewed either using the in-built help viewer or in any external HTML web browser. Also of importance is the **Exit** menu entry found in the *File* menu. **Currently Moses does not check if everything has been stored in the repository.** As a result the user must take care to exit Moses only when all models have been checked into the repository.

In this tutorial you will create a number of models that help illustrate features of the Moses environment. Upon completing this tutorial you will be able to independently model and simulate complex systems using the predefined formalisms.

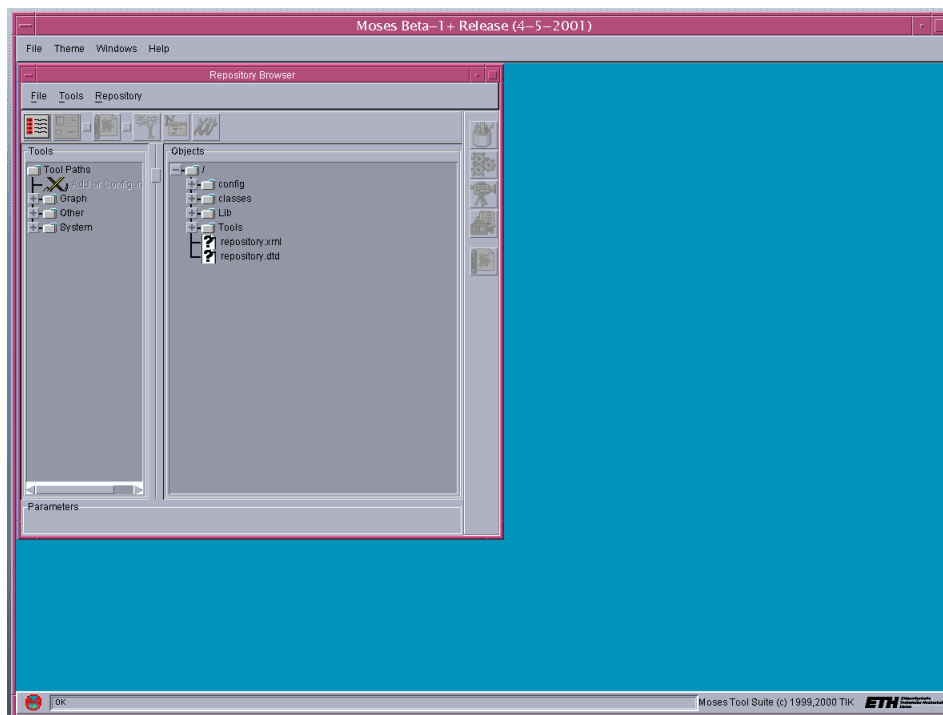


Figure 2: The Moses Tool started using the *Basic* repository

2 A Simple Example

In this section you will build a very simple model of a system that takes two inputs (a and b) and calculates the output function $2a + b^2$. In addition a small test bed will be built to exercise this system for the purpose of demonstrating the simulation/animation capabilities of Moses.

2.1 Creating a project folder

First you will create a project folder in the repository. This is useful for maintaining a degree of order when modeling. The repository is structured as a tree where branches represent project folders and leaves represent elements useful during the modeling process. In the following steps you will create a project folder in the repository.

1. Make sure the root node ($/$) is selected by clicking on it in the *Repository Browser* window.
2. Click on the root node with the right mouse button. A Pop-up Menu appears as shown in figure 3. Choose **Add Directory**, the resulting dialog is shown in figure 4
3. Enter the name¹ of the project **SimpleExample** in the dialog and press **OK** (see figure 4).

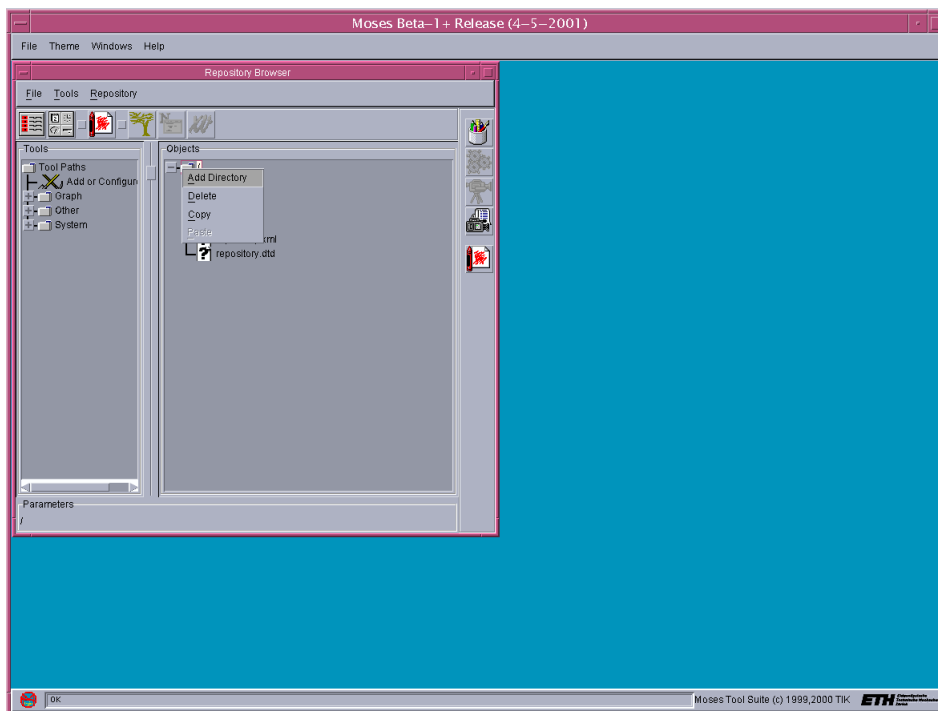


Figure 3: The *Edit* menu with the *Projects* node selected

Figure 5 shows the Moses desk-top after the previous steps have been executed. In the *Project Browser* window you will see that a new project folder *SimpleExample* entry has been created.

¹In *Moses* rules exist for naming projects and components:

- A. The first character of a name must be alphabetic (a..z, A..Z);
- B. The remaining characters must be either numeric (0..9), alphabetic (a..z, A..Z) or (-). Please note that spaces in names are not valid.

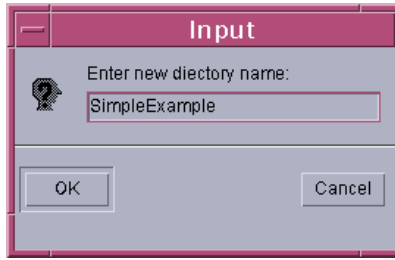


Figure 4: The *Add Project* Dialog

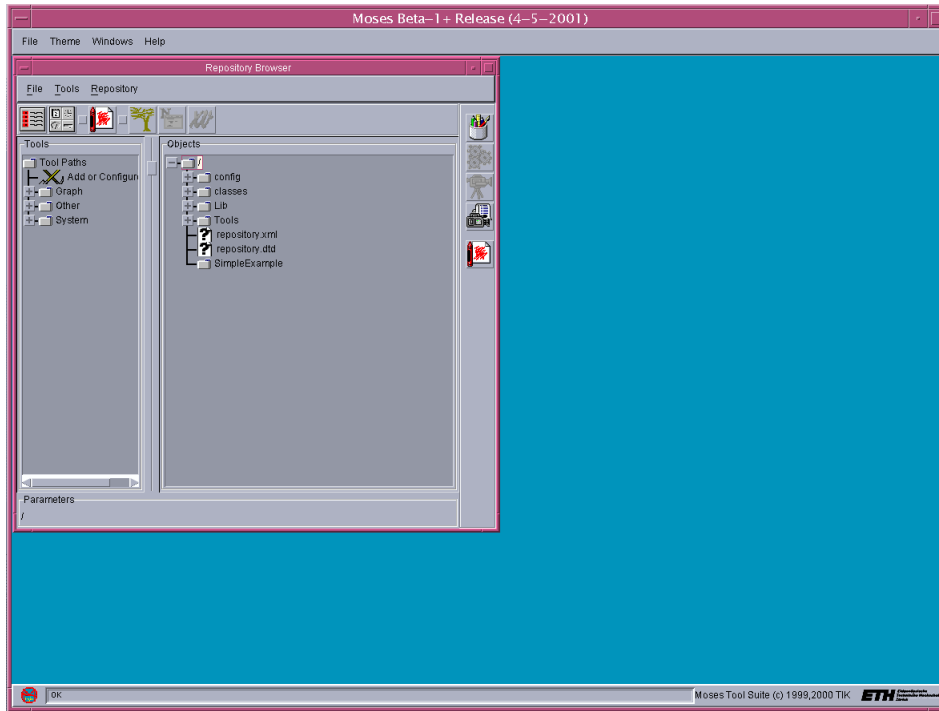


Figure 5: The repository after a new project has been inserted

2.2 Creating a Moses Component

In modeling the function $2a + b^2$ a decision must be made how to best decompose this *system*. Looking at the function you can see that it is made up of 3 binary functions: $a + a$, $b \times b$ and the sum of the results of the two previous functions.

As a result of the observation above you will create a component that applies a function to two inputs producing an output value. To make the system as reusable as possible the function to be applied will be a component parameter. In Moses functions are first class objects and can be used in any situation where an object can be used. In addition the components themselves are also objects so it is possible to build multi-layered systems where the objects being communicated at any particular layer may themselves be components representing (sub) systems.

1. Make sure the *SimpleExample* node is selected in the *Repository Browser* window.
2. Once the Project Folder is selected, click on the Graph Editor button (the uppermost button in the right toolbar).

Moses components are models written in a particular formalism. By default the *Basic* repository contains a number of predefined formalisms. These formalisms are described in a small language *Graph Type*

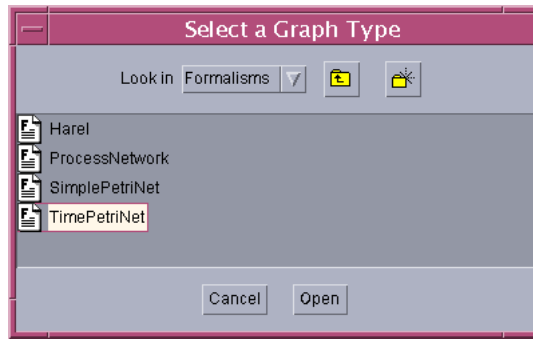


Figure 6: Selecting a formalism for the Moses component

Description Language (GTDL). These formalism GTDL descriptions are also stored in the repository and may be viewed/edited using repository commands. Your first component *two_to_one* will be modeled using the *Time Petri Net* formalism.

1. To choose the formalism select the **TimePetriNet** entry and press **OK** (see figure 6).
2. Press the **OK** button in the component property editor dialog.

As a result the Graph Editor will appear in which the component can be edited using its previously defined formalism (see figure 7).

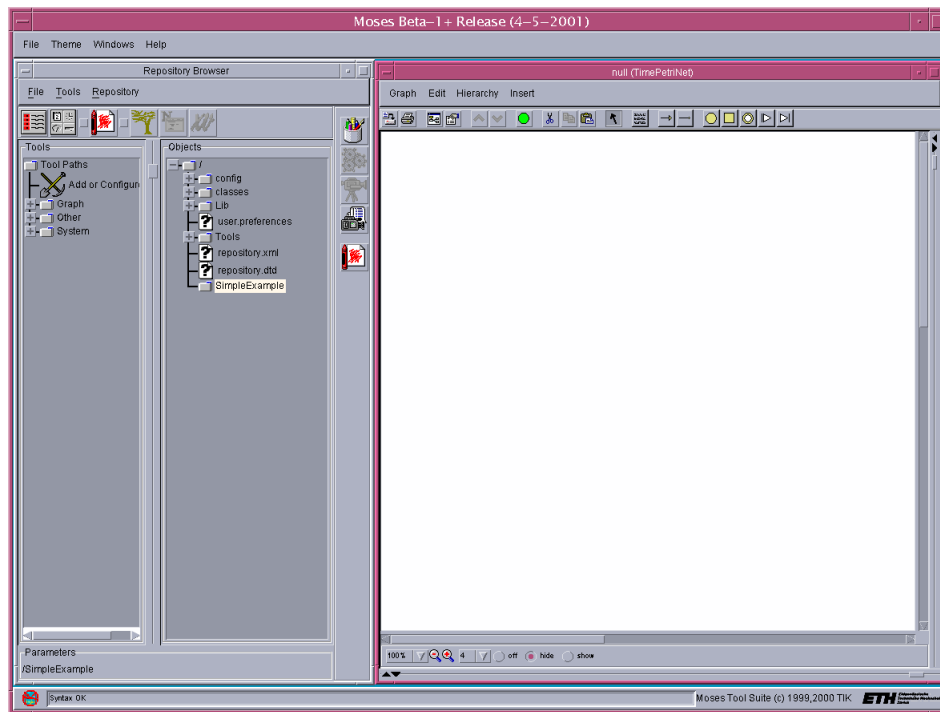


Figure 7: The Graph editor open on the *Time Petri Net* component *two_to_one*

2.3 Creating the *two_to_one* component

In the chapter 3 you will find a description of all menu entries and tool bar buttons. If you have difficulty completing an operation please refer to this. Now you can insert the elements making up the structure of the *two_to_one* component. Figure 8 shows the component consisting of two places and one transition. Petri

net components in Moses are modeled by always connecting input connectors to places and transitions to output connectors. In this way petri net components can be hierarchically or sequentially structured into larger components.

1. In the Graph editor enter and connect all elements as shown in figure 8.

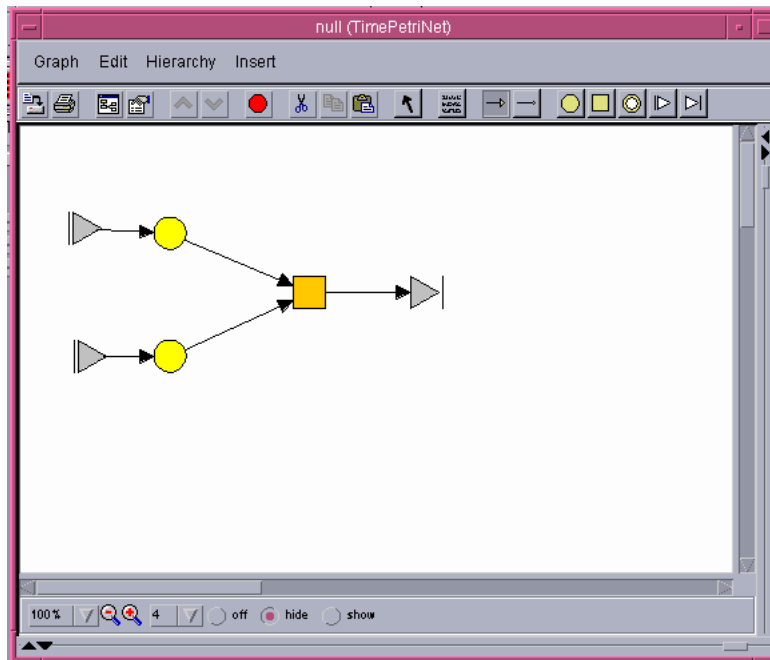


Figure 8: The Graph editor showing the structure of the *two_to_one* component

You may have noticed that the *Check Syntax* tool bar button is now red. This indicates that the component contains a syntax error. This error or errors can be displayed in the syntax error window.

1. Press the **Check Syntax** tool bar button (or select the *Check Syntax Graph* menu menu entry).
As a result the syntax error window will appear showing all syntax errors found (see figure 9).

Errors and warnings displayed in the syntax error window are grouped hierarchically into error types. By pressing on a leaf in the syntax error tree representation the offending element will be selected in the graph editor. If you have faithfully followed the above instructions you will only have three errors indicating that input and output connectors need to be named.

The syntax errors displayed in the *Syntax Error* window all are due to the lack of appropriate names for the input and output connectors. Element names are an example of an attribute. Depending on the formalism elements can have a number of attributes, e.g. describing time, guard and output functions, graphic attributes, type information etc. Naming the connectors will be shown in the following steps.

1. In the Graph editor select the top input connector.
2. Either select the **Open/Close Attribute Editor** menu entry in the *Graph* menu or press the corresponding tool bar button. The Attribute editor will appear in the bottom half of the Graph editor (see figure 10).
3. Select the **Name** tab.
4. Enter the name **a** for the selected input connector and press the **Commit**² button.

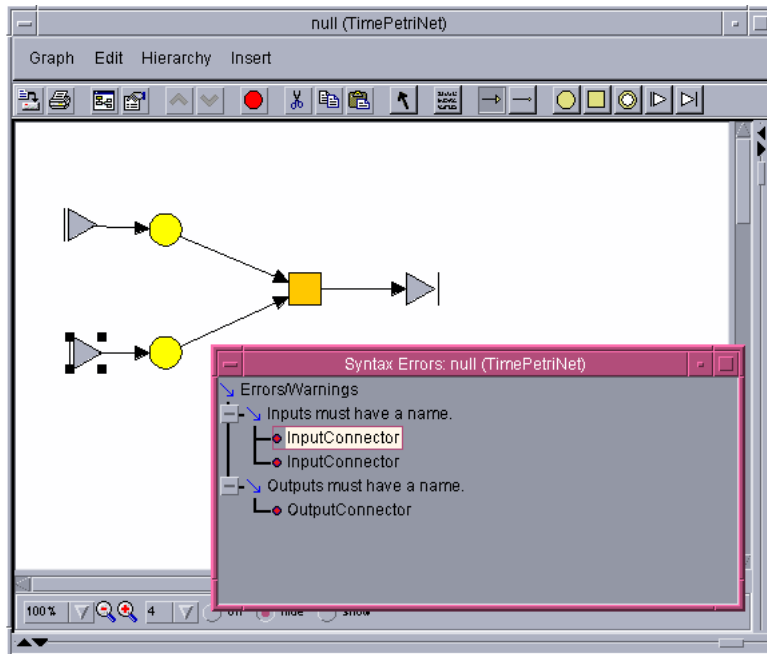


Figure 9: The Graph editor and Syntax Error window showing initial syntax errors

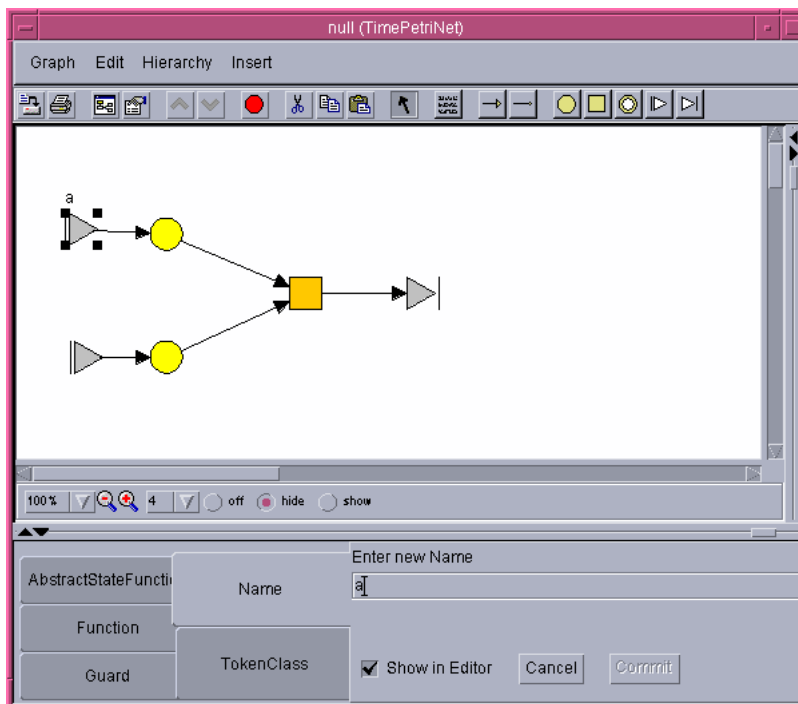


Figure 10: The Graph editor showing the *Input Connector* attribute editor pane on attribute *Name*

The attribute editor allows the attributes of the current selection to be edited. If only a single element is selected then each visible attribute can be edited in an individual tabbed pane in the attribute editor. If for example a number of elements are selected then the common sub set of attributes can be edited. Naturally

²The Commit button will remain disabled until the user modifies the attribute. Upon modification the button will become enabled and can be pressed storing the attribute in the graph. Hence the state of the Commit button indicates whether the attribute displayed in the attribute editor differs from that stored in the graph.

every modified attribute of all selected elements will be changed to the same value. Finally if no element is selected then those attributes of the component itself can be viewed and edited.

The following steps will guide you in naming all necessary elements and in inserting an assignment expression for the transition. In the *TimePetriNet* formalism transition expressions refer to input tokens via the names used on the input transition arcs. Similarly generated tokens are named in the transition assignment expressions and are associated with the names on output arcs. Hence it is possible to have a number of output arcs with the same name receive a token defined by a single assignment expression.

1. Now complete the naming of elements so that the result looks like that in figure 11.
2. Select the **transition** and in the attribute editor the **Function** tab.
3. Enter the assignment expression **out = fn(a,b)**, make sure the **Show in Edit** check box is ticked and press the **Commit** button.
4. Either select the **Open/Close Attribute Editor** menu entry in the *Graph* menu or press the corresponding tool bar button. The Attribute editor will now be closed. The result should look similar to figure 12.

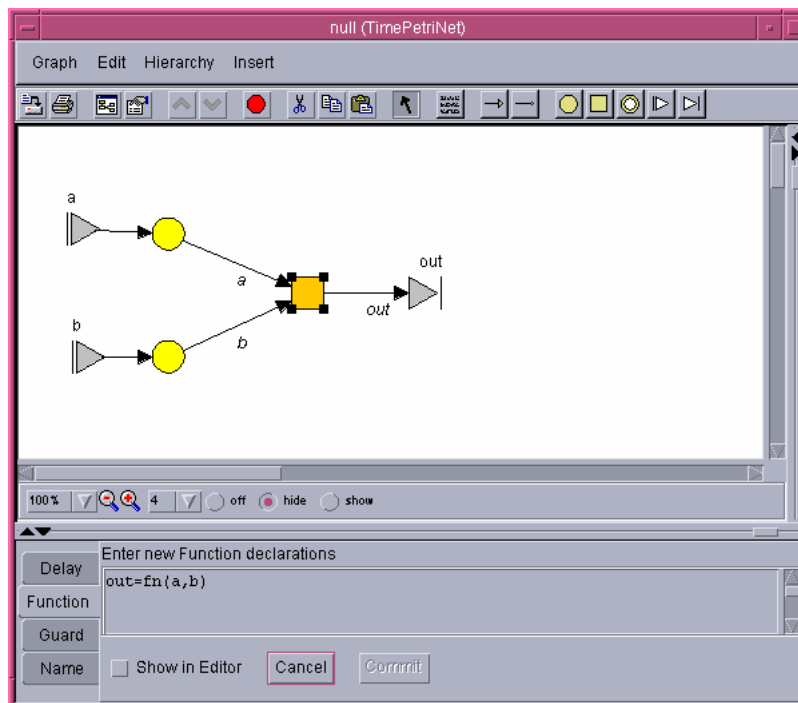


Figure 11: The Graph editor showing the *Transition* attribute editor pane on attribute *Function*

In the steps listed above an assignment expression was defined $out = fn(a, b)$. However the function fn has not yet been defined. As mentioned previously this function will be a component parameter so that individual instances of *two_to_one* can be parameterized with different functions. To insert a component parameter:

1. Make sure no element is selected.
2. Either select the **Open/Close Attribute Editor** menu entry in the *Graph* menu or press the corresponding tool bar button. The Attribute editor will now be visible showing the component attributes.
3. Select the **Parameters** tab.

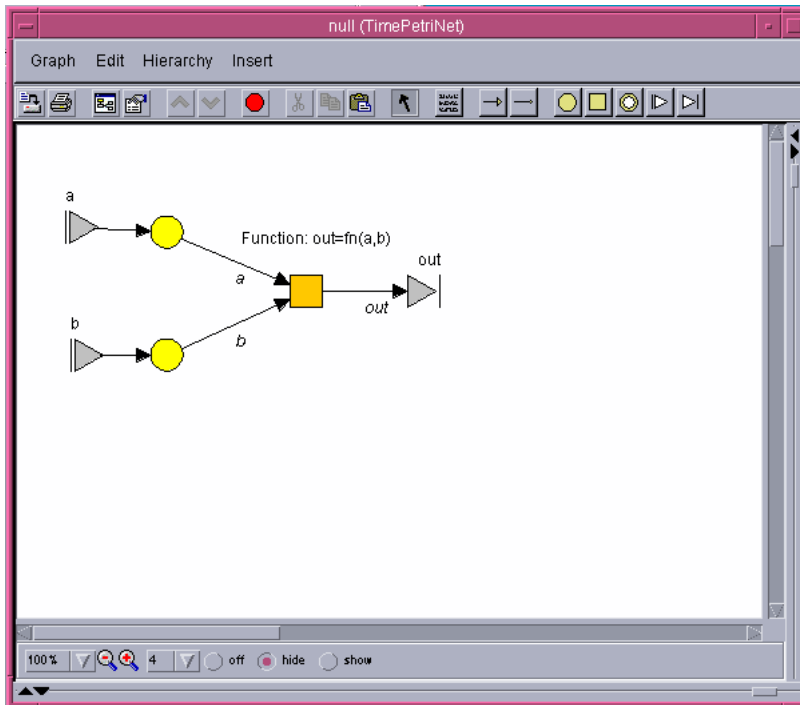


Figure 12: The Graph Editor showing the *Time Petri net* component

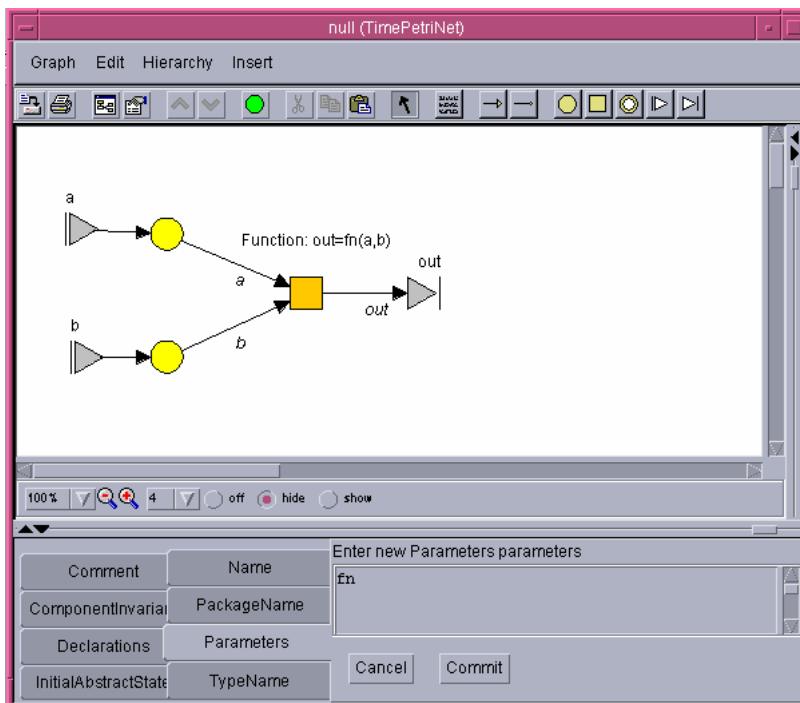


Figure 13: The Graph Editor showing the Component *Parameters* Attribute editor pane

4. Enter the name of the parameter **fn** and press the **Commit** button (see figure 13).

If all is well the Check Syntax tool bar button is now green. Now that the component has been defined it must be checked into the repository.

1. Select the **Save and Exit** menu entry from the *Graph* menu (see figure 14).
2. A **save as** dialog appears. Enter the name of the component **two_to_one** and press save. (see figure 15)

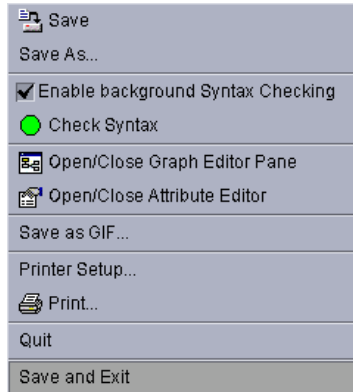


Figure 14: The Graph editor *Graph* menu

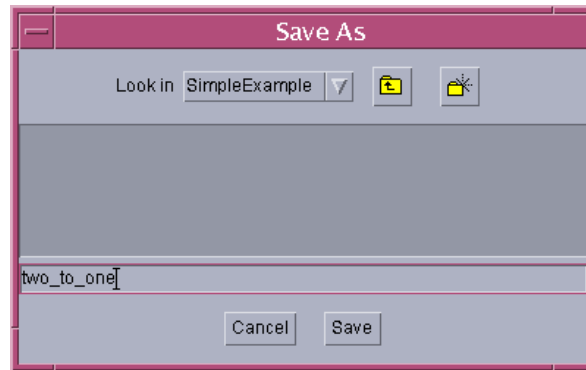


Figure 15: The Save as Dialog

The steps so far have created a component using the *TimePetriNet* formalism. This is equivalent to the source text description of a class written in an object-oriented language. Now the component must be compiled into a form where it can be instantiated and executed.

1. Select the component **two_to_one** in the *Repository Browser* window.
2. Press the **Compiler** button in the vertical toolbar. This is the second button from top (see figure 16).
3. The component should now be compiled successfully.

2.4 Creating the *two_a_plus_b_sqr* component

Now that the reusable component *two_to_one* has been completed a component implementing the function $2a + b^2$ can be created. This component is a composition of three instances of *two_to_one*. Composition can be modeled in the *TimePetriNet* formalism however the *ProcessNetwork* formalism is better suited for modeling this. Process networks simply describe interconnected components. Each block is typed with the type of the component it represents. Arcs represent the interconnection between blocks and describe the flow of objects between components.

The following instructions will create a new component.

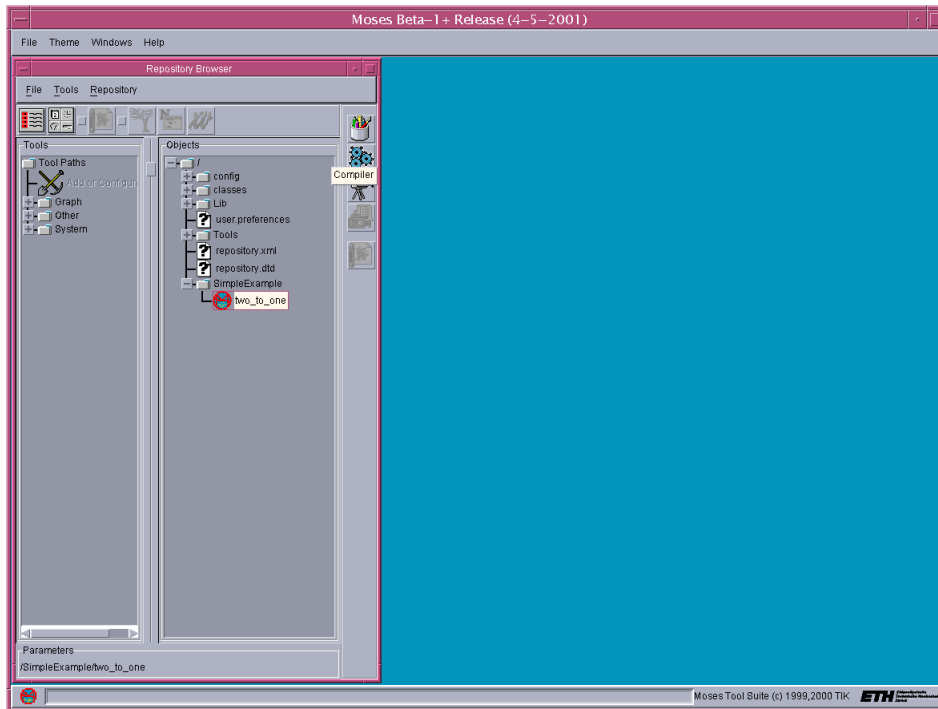


Figure 16: The Compiler button in the toolbar

1. Make sure the *SimpleExample* node is selected in the *Repository Browser* window.
2. Press the Graph Editor Button to create a new component. *Formalism* selection dialog shown in figure 17 will appear.
3. To choose the formalism select the **ProcessNetwork** entry and press **OK** (see figure 17).

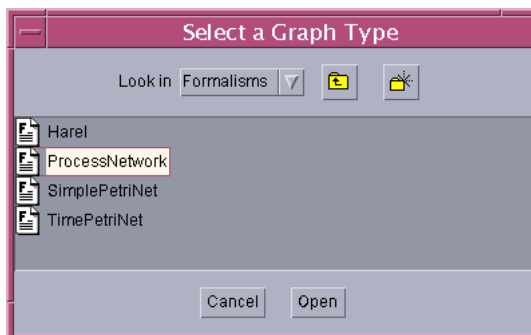


Figure 17: Selecting a formalism for the component *two_a_plus_b_sqr*

Now the Moses component can be edited in the new open Graph Editor window. The formalism used is the one selected before (see figure 18).

The differences between the graph editor shown in figure 7 and the graph editor shown in figure 18 are due to the different formalisms (*TimePetriNet* and *ProcessNetworks*). In this case the differences only lie in the elements of the formalism, e.g. the ability to insert process nodes in addition to comments, arcs, input connectors and output connectors common to both formalisms.

To model the function $2a + b^2$ three process nodes are needed representing the three atomic functions $a + a$, $b \times b$ and the sum of the two previous functions. In addition as this component models a function with two inputs and an output the appropriate number of connectors must be present.

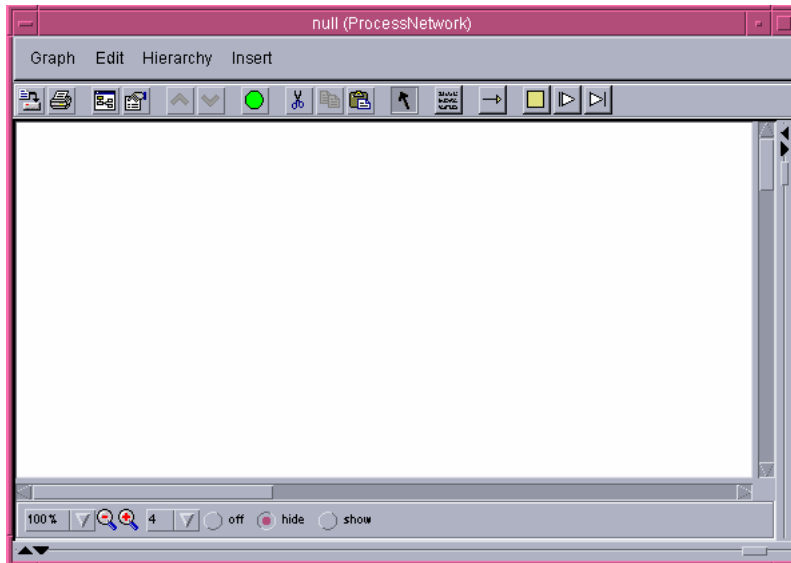


Figure 18: The Graph editor showing the empty *Process Network* component *two_a_plus_b_sqr*

1. In the graph editor insert vertices representing the three process nodes and three connectors as shown in figure 19.

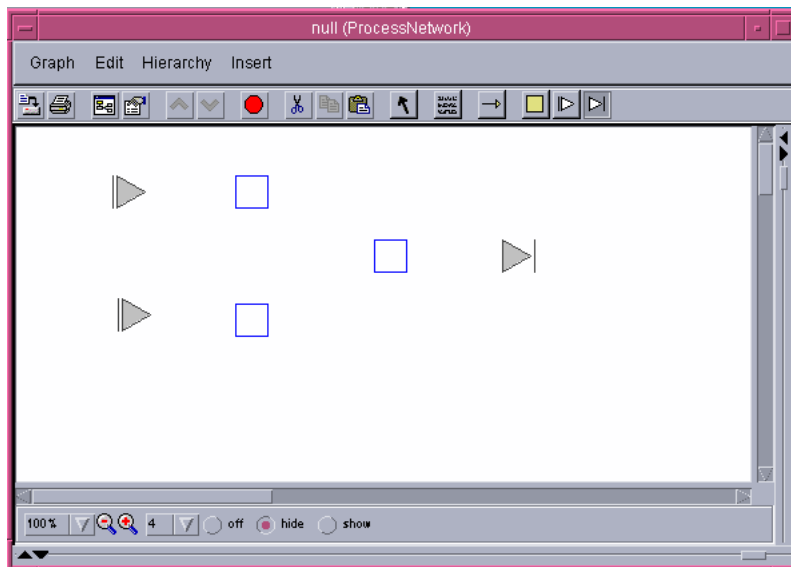


Figure 19: The Graph editor showing vertices of component *two_a_plus_b_sqr*

1. In the Graph editor select the output connector (see figure 20).
2. Either select the **Open/Close Attribute Editor** menu entry in the *Graph* menu or press the corresponding tool bar button. The Attribute editor will appear in the bottom half of the Graph editor.
3. Select the **Name** tab.
4. Enter the name **out** for the selected input connector and press the **Commit** button.

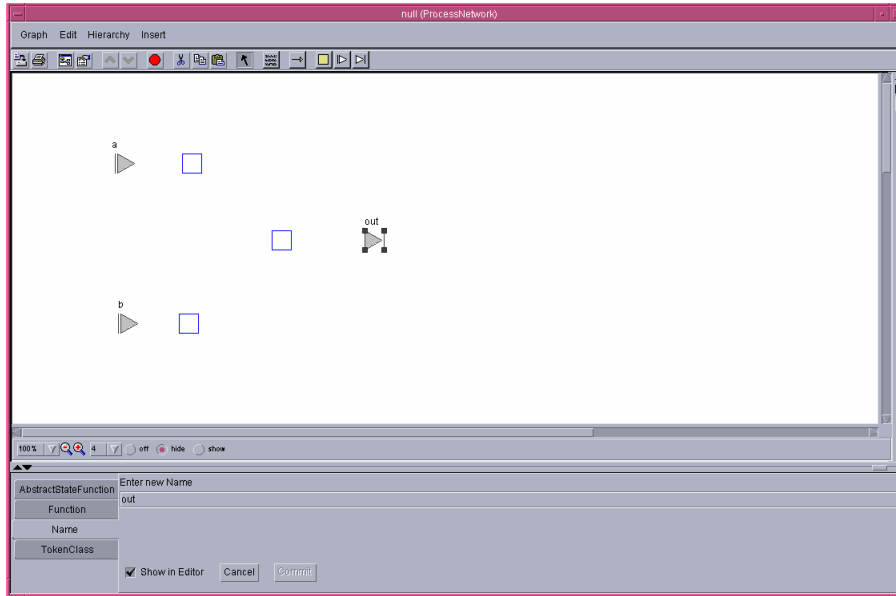


Figure 20: The graph editor showing the output connector *Name* attribute being edited

5. Name the input connectors *a* and *b* (see figure 20).

With the connectors named you now need to configure the process nodes. As mentioned previously process nodes represent a component to be composed within a system. To be able to do this the connections of the component must be made visible so that the component can be wired.

1. In the Graph editor select the top process node (see figure 21).
2. Select the **ComponentClass** tab.
3. Enter the name of the component that the process node is to represent (**two_to_one**) and press the **Commit** button.
4. Select the **InputPorts** tab.
5. Enter the input ports. This is done by the expression ["a","b"]. Do the same with the **OutputPorts**. (there is only one output port defined ["out"], see figure 22). The result of this action is to insert into the process node the connections defined in component *two_to_one*.

With the component class of the process node defined the constructor also needs to be entered. In the process networks formalism the components associated with process nodes are instantiated at the time when the process network container component is instantiated. Typically for components without parameters the constructor expression is of the form *new ComponentName()*. However for the component *two_to_one* it is a little more complex as we also need to define the *fn* parameter. This parameter is a function object which in general is defined as *lambda(x₁, x₂, ..., x_n) expression end*.

1. Select the **Constructor** tab.
2. Enter the constructor expression for this process node. As this node will implement the *a + a* function the expression to enter is **new two_to_one([lambda(x,y) x + y end])**. Press the **Commit** button (see figure 23).

With the first process node configured the remaining nodes must not be forgotten. Remember that the process node that inputs *b* must have a function that multiplies rather than adds. Finally the nodes and connectors must be wired together with arcs.

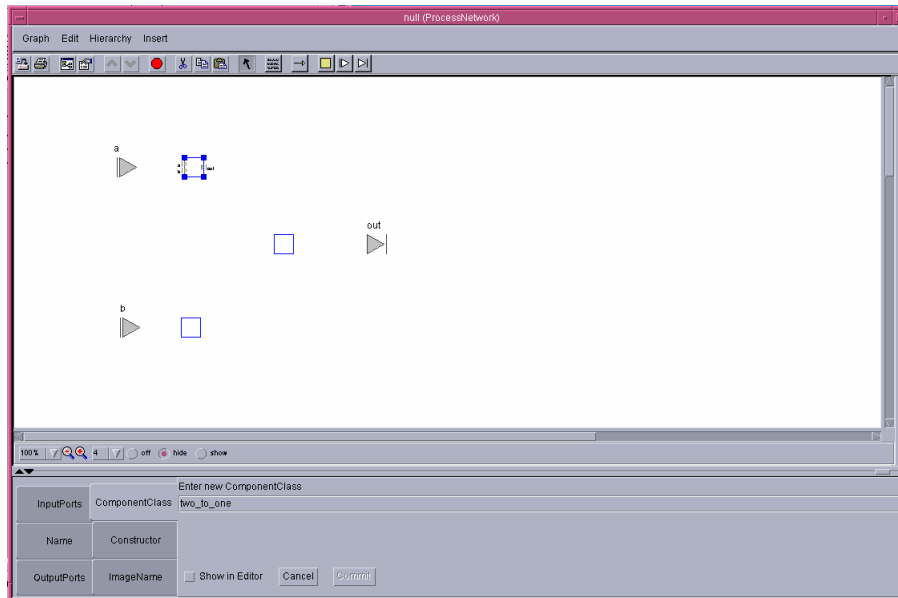


Figure 21: The graph editor showing a configured process node

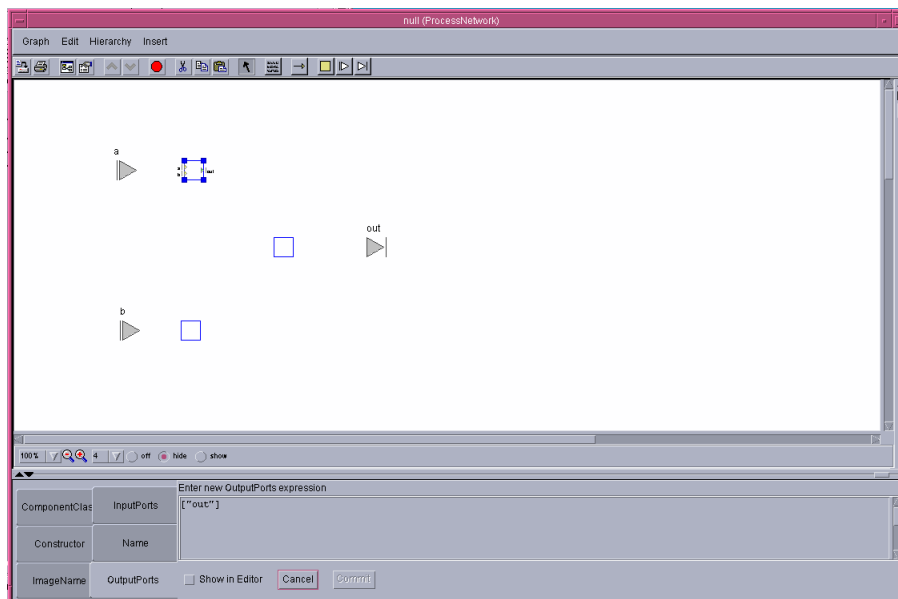


Figure 22: The configuration of output ports

1. Enter the **ComponentClass** attributes for the remaining process nodes.
2. Enter the **Constructor** expressions for the remaining process nodes.
3. Enter the **InputPorts** and **OutputPorts** for the remaining process nodes.
4. Now wire the connectors and process nodes together as shown in figure 24.

Figure 24 shows the component *two_a_plus_b_sqr* containing a *comment* element. As comment elements have no semantic meaning any number may be inserted into a component. Now the component must be compiled into a form where it can be instantiated and executed.

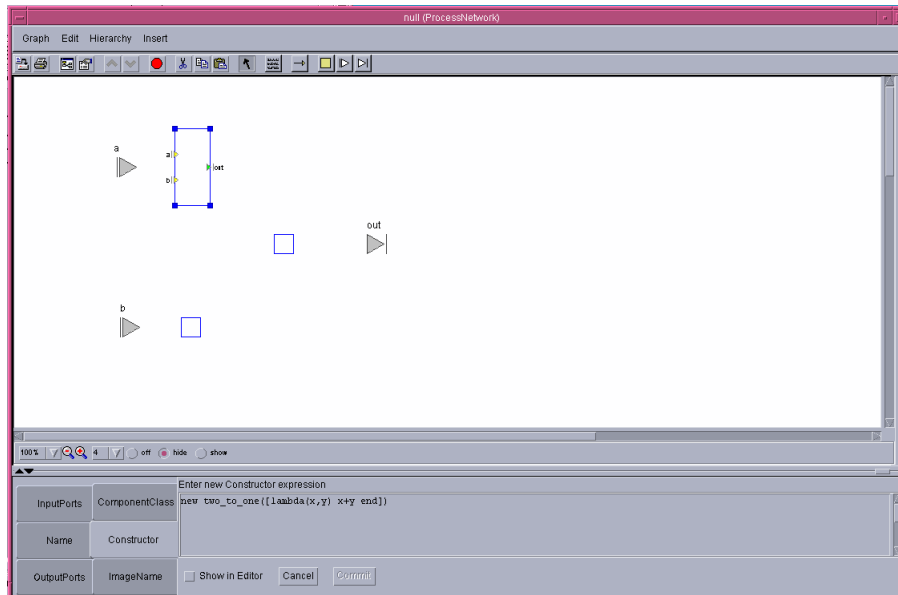


Figure 23: The graph editor showing a process node constructor attribute

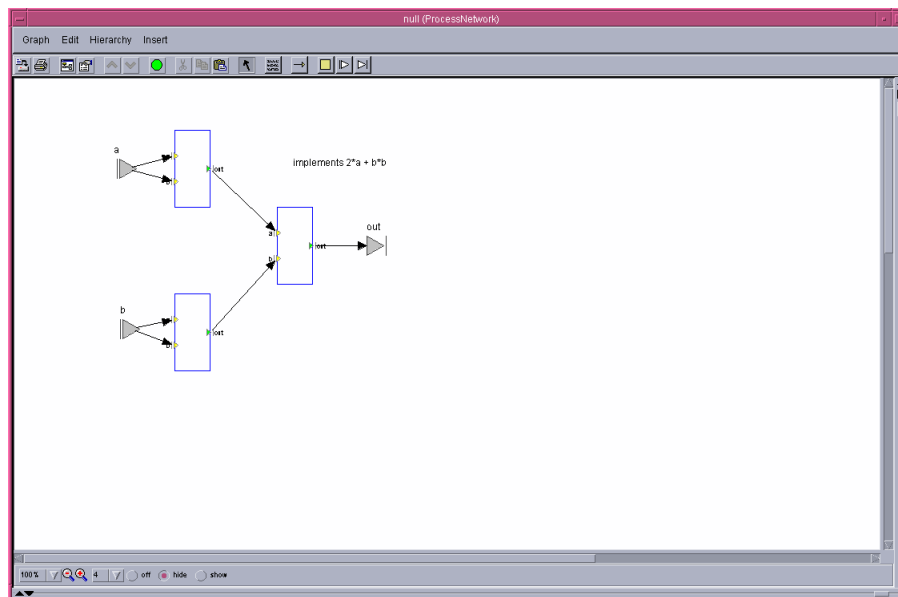


Figure 24: The completed process network component *two_a_plus_b_sqr*

1. Select the **Save and Exit** menu entry from the *Graph* menu. This will cause a **Save As** dialog to appear.
2. Enter the name **two_a_plus_b_sqr** of the component and press **Save**.
3. The component *two_a_plus_b_sqr* will be written into the repository and the graph editor will now close.
4. Select the component **two_a_plus_b_sqr** in the *Project Browser* window.
5. Press the **Compiler** button in the toolbar (see figure 25).

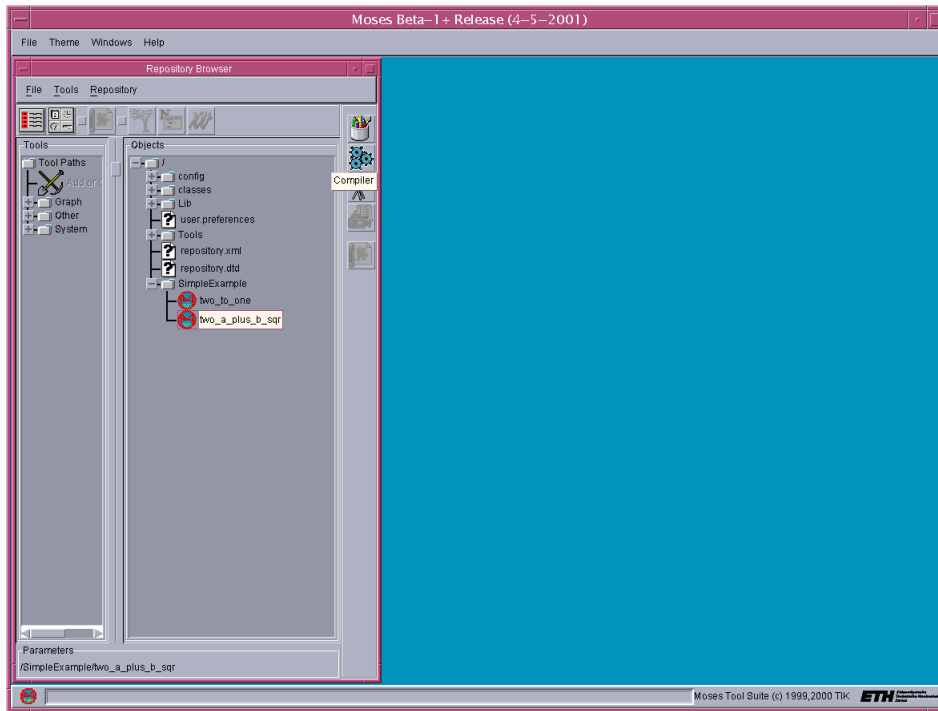


Figure 25: The compiler button in the toolbar

2.5 Creating the *testbed* component

Now that the component *two_a_plus_b_sqr* has been created that implements the desired function it would be sensible to check and see if it implements the function correctly. There are two possible ways of doing this. The first relies on the user generating a text file with the test inputs and checking that the result is correct by either viewing them in the animator or producing an output file containing the results. The second method is to construct a test environment where inputs are generated and the system animated to check that it is functioning correctly. Naturally test environments can be more or less elaborate depending on the degree of automation one desires in the test process. One advantage of the approach taken in Moses is that in creating test environments one has the full Moses modeling environment available. In fact it is possible to create generic environments that implement parametric testing.

In this section the component *testbed* will be created. The following instructions will guide you in this task.

1. Make sure the *SimpleExample* node is selected in the *Project Browser* window.
2. Press the **Graph Editor** button. As a result the *Select a Graph Type* selection dialog will appear.
3. Select the entry **TimePetriNet** as shown in figure 26.

Now a new graph editor window will appear and you can start editing (see figure 27).

The task of the *testbed* component is to feed inputs to an instance of the *two_a_plus_b_sqr* component and to collect the results. As this component uses the *TimePetriNet* formalism we need to insert a *container place* to hold an instance of the *two_a_plus_b_sqr* component.

Places and container places both have attributes defining their initial tokens. As both kinds of places can contain any number of tokens, initial tokens must be defined as a list. In Moses a list is defined as square bracketed comma delimited set of object instantiation expressions, e.g. [0, 1, 2, 3] or [new two_a_plus_b_sqr()].

1. In the graph editor insert a container place as shown in figure 27.

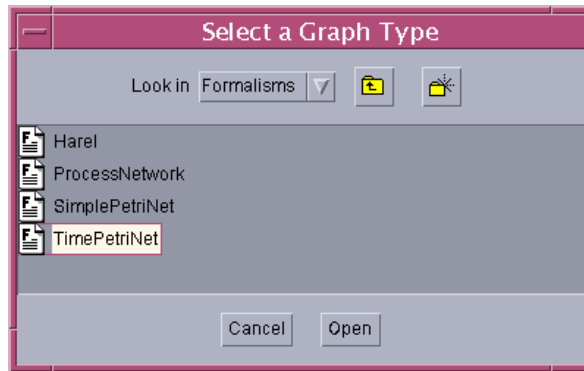


Figure 26: The formalism selection dialog for the *testbed* component

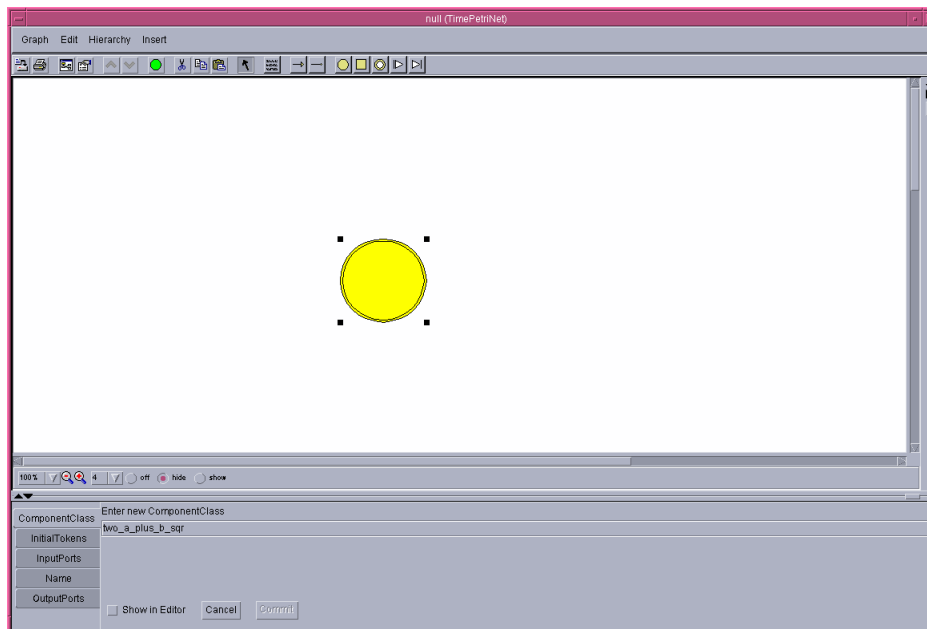


Figure 27: The graph editor showing the *testbed* component with the attribute showing the *ComponentClass* attribute of the selected container place

2. In the Graph editor select the container place.
3. Either select the **Open/Close Attribute Editor** menu entry in the *Graph* menu or press the corresponding tool bar button. The Attribute editor will appear in the bottom half of the Graph editor.
4. Select the **ComponentClass** tab.
5. Enter the name of the component that the process node is to represent (**two_a_plus_b_sqr**) and press the **Commit** button. The result of this action is to insert into the container place the connections defined in component *two_a_plus_b_sqr* (see figure 27).
6. Select the **OutputPorts** tab.
7. Enter the output ports list
8. Select the **InputPorts** tab.
9. Enter the input ports list ([“a”,”b”], see figure 28)

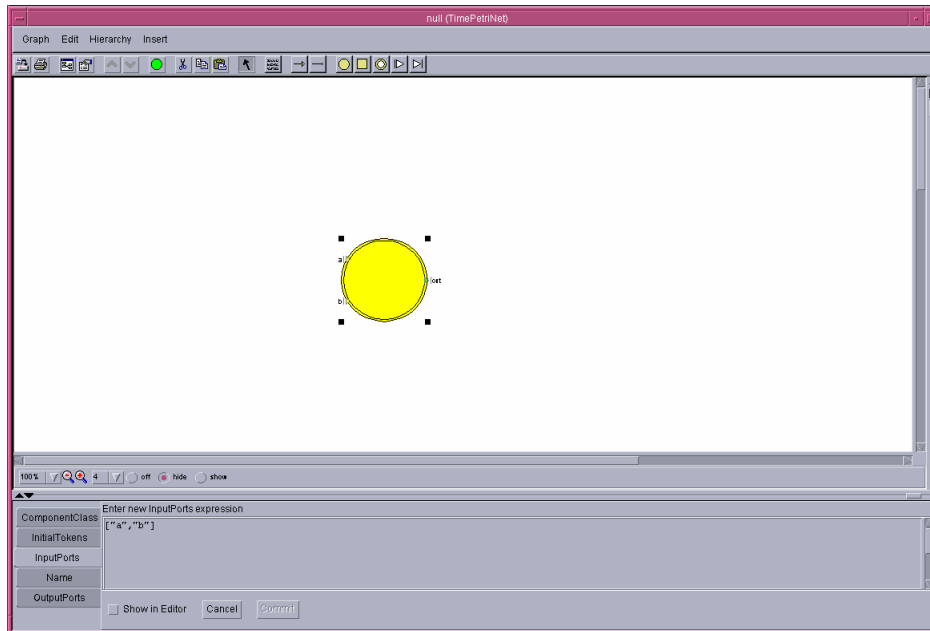


Figure 28: The graph editor showing the *testbed* component with the attribute showing the *InputPorts* attribute of the selected container place

10. Select the **InitialTokens** tab.
11. Enter an expression to define a list of initial tokens for this container place. As this container place will represent the system under test as a token enter [**new two_a_plus_b_sqr()**]. Press the **Commit** button (see figure 29).

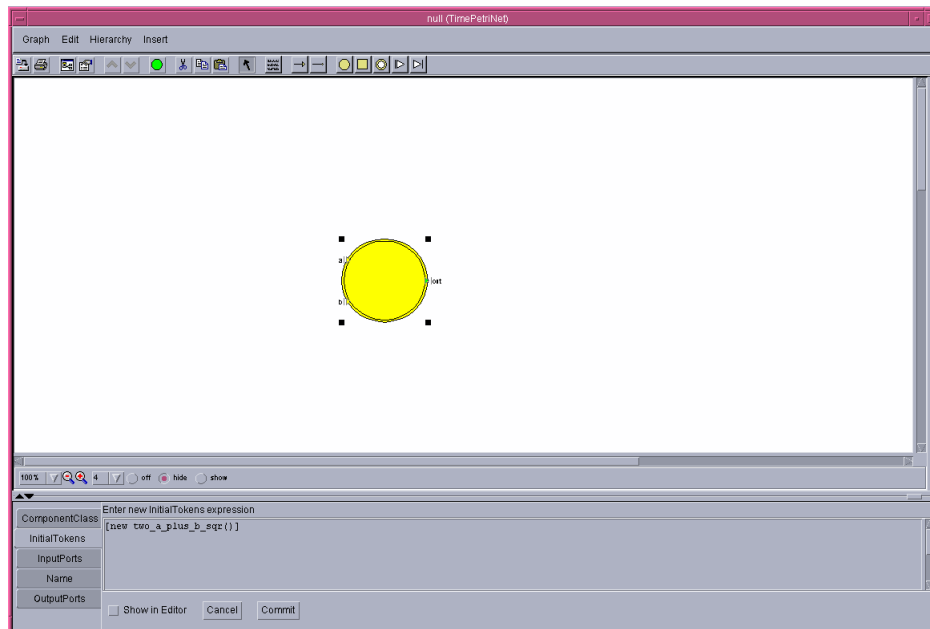


Figure 29: The graph editor showing the *testbed* component with the attribute showing the *InitialTokens* attribute of the selected container place

With the container place now defined the remainder of the *testbed* component must be created. Figure 30

shows the completed component. This models a very simple test environment where a place contains a list of initial tokens that are input into the system under test instantiated as a token on the container place. Objects output from the test system are collected on the rightmost place. The arc names together with the transition function define the value of objects entering the *a* and *b* inputs of the *two_a_plus_b_sqr* component.

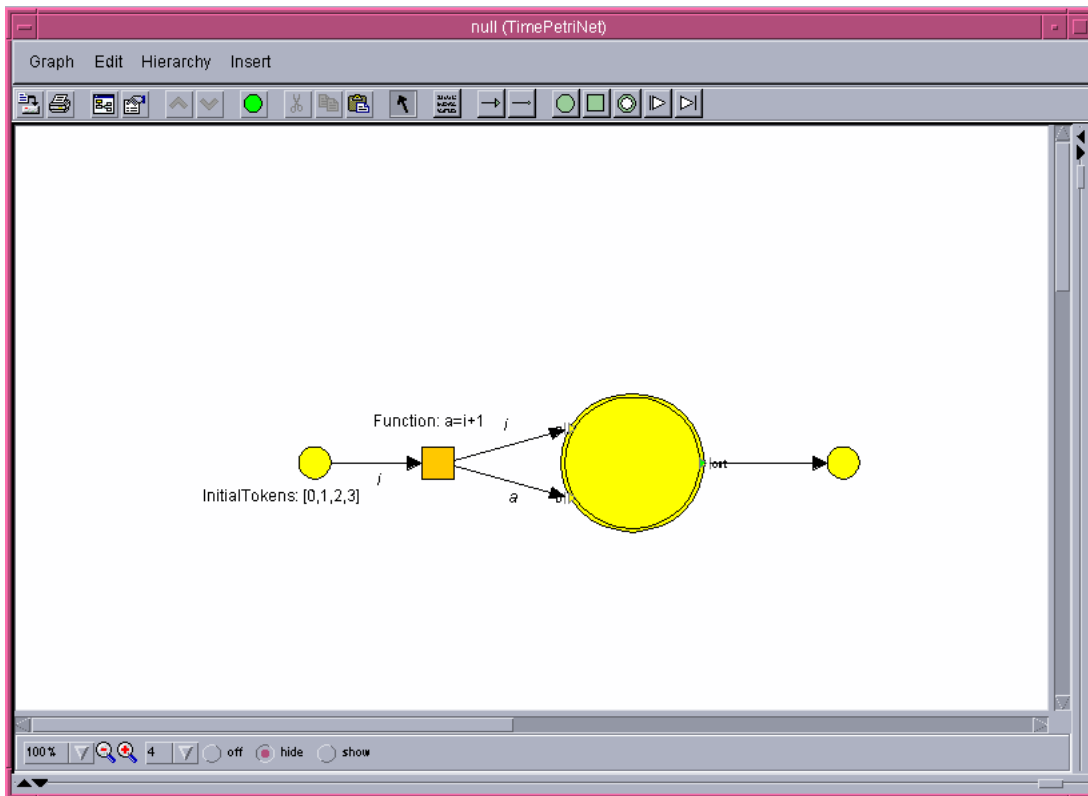


Figure 30: The completed *testbed* component

Now the component must be compiled into a form where it can be instantiated and executed.

1. Select the **Save and Exit** menu entry from the *Graph* menu.
2. In the **Save As** dialog enter *testbed* as the name of the component.
3. Press **Save**. The component *testbed* will be written into the repository and the graph editor will now close.
4. Select the component **testbed** in the *Project Browser* window.
5. Press the **Compiler** button.

2.6 Animating the *testbed* component

In the previous sections you were guided in building a model of a simple system and a small test environment. As the *testbed* component does not have any input or output connectors it will not read input or produce output when run external to the Moses tool as a stand-alone application. Hence executing this component only makes sense when it is animated. In this section you will be shown how a component can be animated and the state of the model investigated.

1. Make sure the *testbed* node is selected in the *Project Browser* window.
2. Press the **Animator** button in the toolbar (the one with a camera on it). As a result the *Simulation Parameter Editor* dialog will appear (see figure 31).

3. As this component has no parameters or input/output connectors just press the **OK** button.

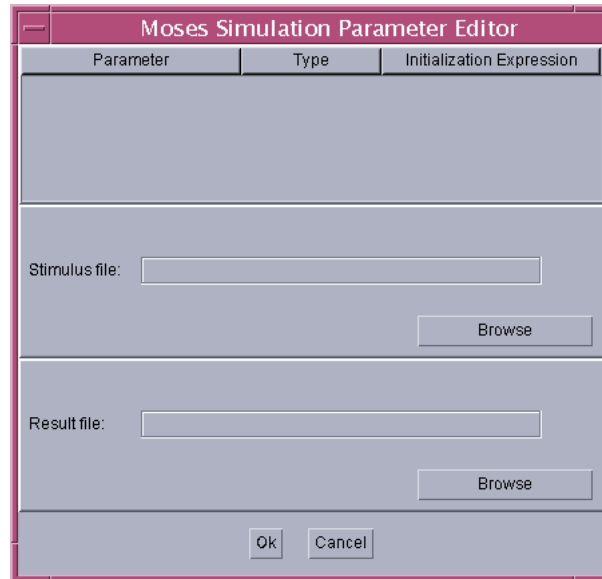


Figure 31: The simulation parameter dialog

The above steps will invoke the animation environment with the *testbed* component. Figure 32 shows that the animation environment is structured as a *desk-top* component. Within this desk-top there is a panel containing the simulation controls and a text status line. In addition there is a window showing the instantiated *testbed* component. The nodes representing the *testbed* component can be expanded by doubleclicking the component showing the current instantiation hierarchy. Figure 32 shows that this component consists of an instance of the *two_a_plus_b_sqr* component which in turn consists of 3 instances of the *two_to_one* component. Associated with every component name shown in the instantiation hierarchy is an object identifier. These identifiers enable objects to be uniquely identified within a model.

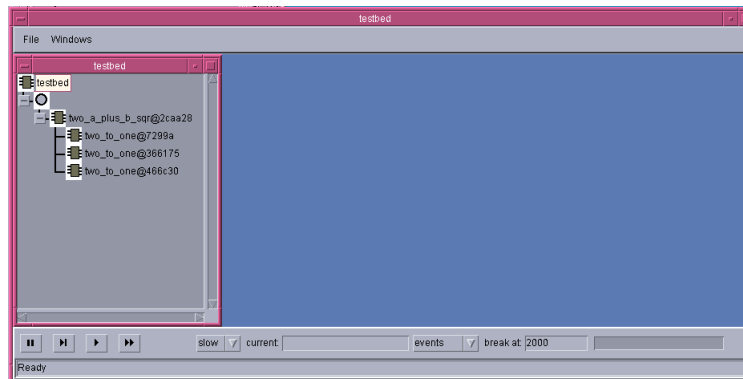


Figure 32: The animation environment with the *testbed* component

Rather than guide you step-by-step in the animation of your model you will be shown how to make a single step in the animator and then to investigate the components that make up your system.

1. In the animation desk-top double click on the *testbed* entry in the *testbed* instantiation window.
2. Continue to double click until all components are visible in the instantiation window (see figure 32).
3. In the instantiation window select the *testbed* component.

4. Press the right mouse button and from the pop-up menu select **Open Animation**. In the Animation desk-top a window will appear displaying the *testbed* component (Figure 33 shows the result).
5. In the simulation controls press the **step** button once. The number of tokens displayed on the input place in figure 33 will decrease from 4 to 3 due to the transition firing.
6. In the instantiation window select the *two_a_plus_b_sqr* component.
7. Press the right mouse button and from the pop-up menu select **Open Animation**. In the Animation desk-top a window will appear displaying the *two_a_plus_b_sqr* component (see figure 34).
8. In the instantiation window select the *two_to_one* component.
9. Press the right mouse button and from the pop-up menu select **Open Animation**. In the Animation desk-top a window will appear displaying the *two_to_one* component (see figure 35).

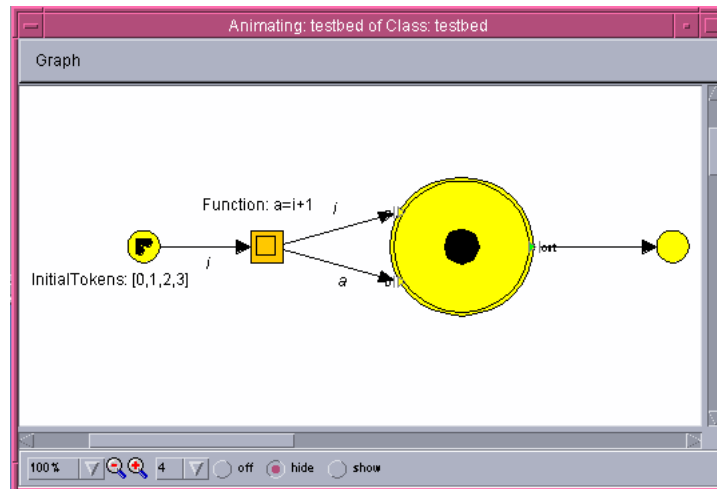


Figure 33: The instantiated *testbed* component after one step

The current state of the system is indicated by the distribution of tokens on the places that make up the system. To view the values of tokens on a place:

1. In the *tesbed* component shown in figure 33 move the mouse cursor over the input place.
2. Press the right mouse button and from the resulting pop-up menu select **Inspect**. A window will appear showing the value of the tokens on the place.

Figure 36 shows the value of the tokens before any any simulation steps have been executed. This corresponds to the *initial tokens* of the place and is defined via the attribute editor when defining the component. One thing to note is that the tokens on a place are not necessarily ordered. Hence the tokens on the place are represented by a list of tuples (*value@creationtime*) $1@0.0, 3@0.0, 2@0.0, 0@0.0$.

The simulation may be executed with animation (single step or continuous) or without animation at full speed. To stop continuous simulation press the stop button. As the *testbed* component can only execute a finite number of times you can execute the component without animation and observe the component when it is in its final state. Naturally this can be preceded by animation steps.

1. Press the **fast continuous simulation** button. In the status line you should see the message *Net is dead...*

Figure 37 shows the *testbed* component after all possible steps have been executed. In figure 38 the token value window is shown for the output place in the *testbed* component. Here the results can be compared with those expected from the system. Do not worry if the results are incorrect the next section will remedy this!

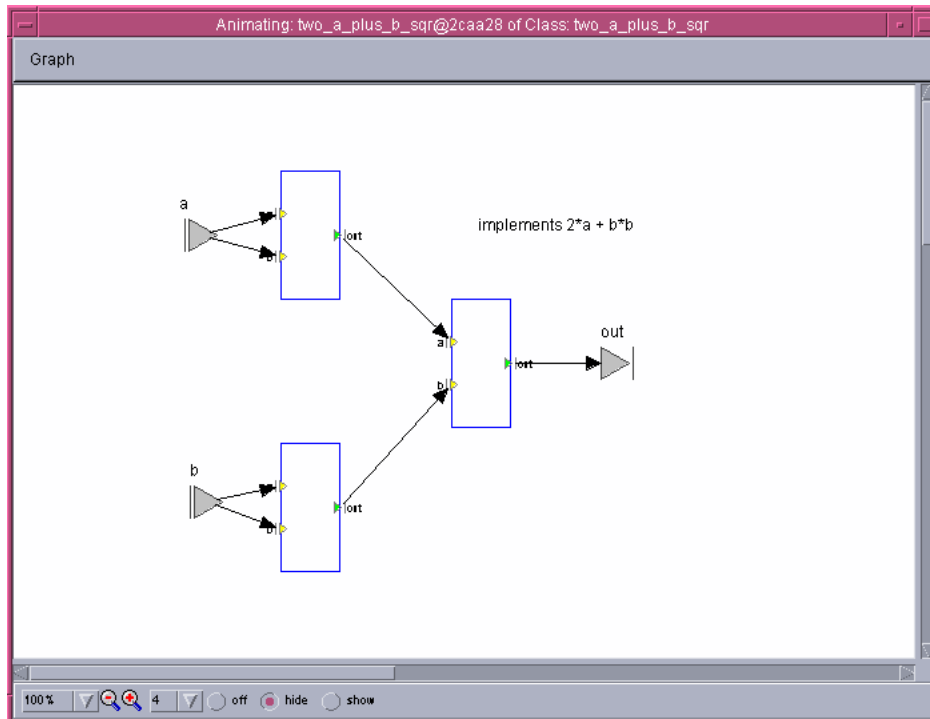


Figure 34: The instantiated *two_a_plus_b_sqr* component

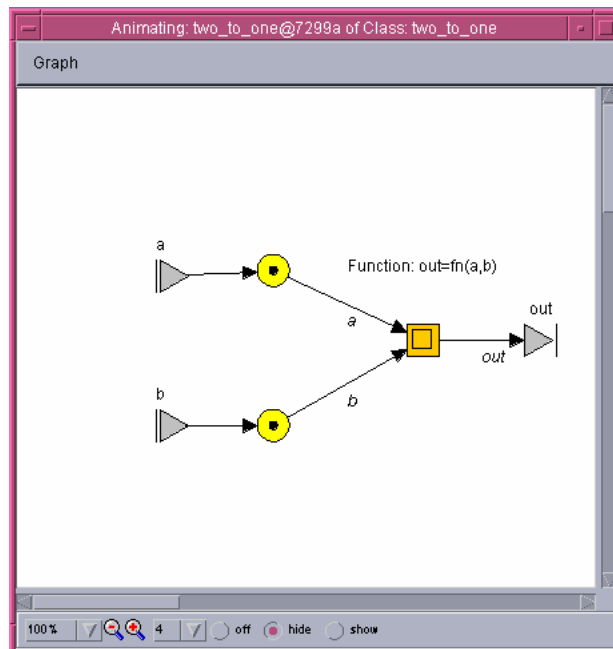


Figure 35: The instantiated *two_to_one* component

2.7 Correcting the *testbed* component

The Petri net shown in figure 37 may not produce the expected results. The behavior that you would like is for a token to be taken from the input place by the transition in *testbed*. This transition generates tokens that represent a and b and injects them into the component *two_a_plus_b_sqr*. Finally a token representing the result will be added onto the output place.



Figure 36: The initial tokens on the input place of the *testbed* component2

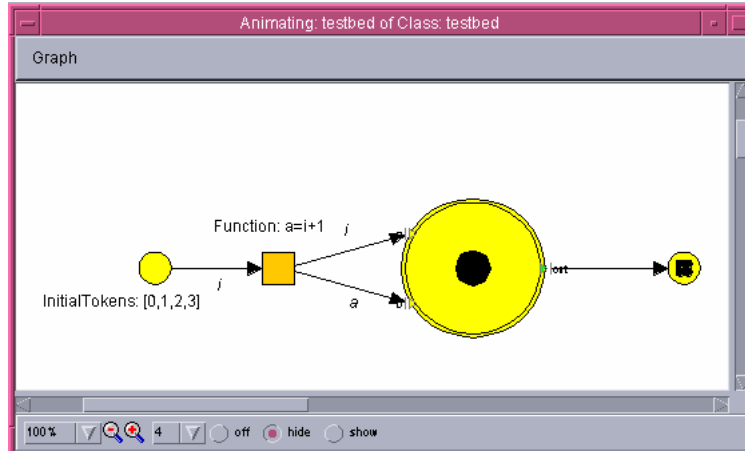


Figure 37: The instantiated *testbed* component showing its state after all possible steps have been executed



Figure 38: The tokens on the output place produced by the *testbed* component

The above scenario will not necessarily always take place. The semantics of Petri nets are such that all enabled transitions have an equal chance to fire. As a result it is possible that tokens in the flow described above might be caught up by tokens that originated from a later firing of the *testbed* transition.

The solution to the problem is to couple the arrival of the result token on the output with the ability of the *testbed* transition to be enabled and to fire again. The following steps will guide you in modifying the *testbed* component so that it exhibits the correct behavior.

1. First close the animation environment by selecting **Close** from the File menu.
2. Select the *testbed* entry in the *Project Browser* window.
3. Press the **Graph Editor** button.

To synchronize the input of a new data token with the output of a result token an additional place needs to be inserted into the *testbed* component. To ensure that the component can start the inserted (feedback) place must have an initial token. Now for the *testbed* transition to fire a token must be present on the

feedback place and the input place. To enable the start of a new cycle a second result token is fed into the feedback place.

1. Modify the *testbed* component to that shown in figure 39.
2. Select the **Save and Exit** menu entry from the *Graph* menu. The component *testbed* will be written into the repository and the graph editor will now close.
3. Select the component **testbed** in the *Project Browser* window.
4. Press the **Compiler** button.

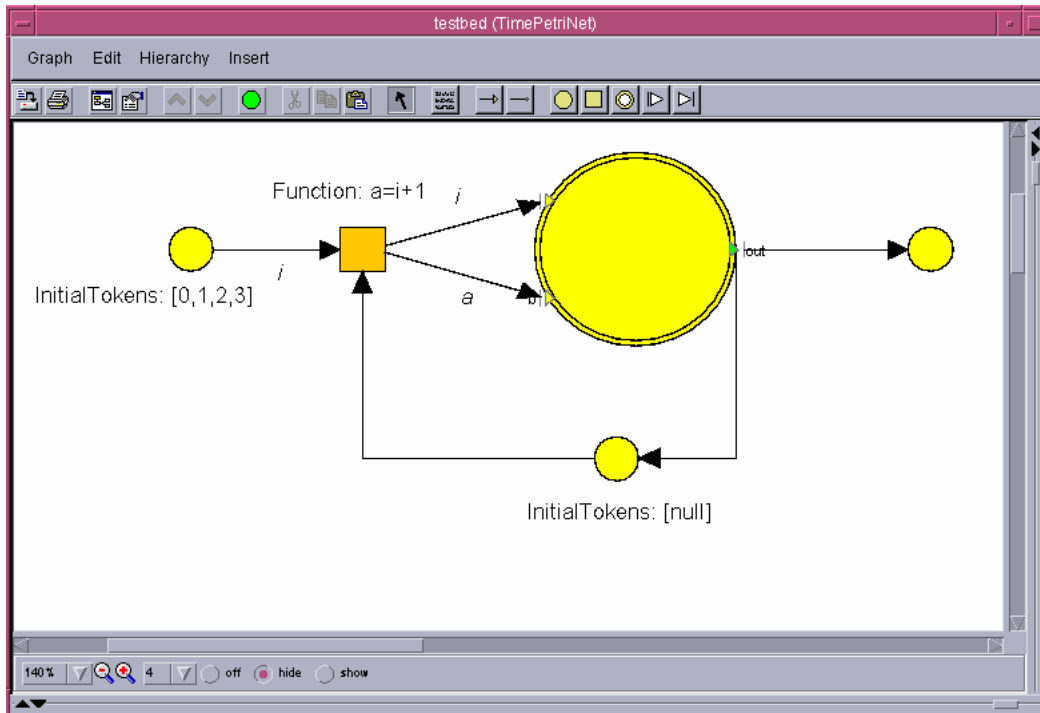


Figure 39: The corrected *testbed* component (shown scaled at 140%)

Now that the *testbed* component has been modified and compiled the system can be animated again.

1. Make sure the *testbed* node is selected in the *Project Browser* window.
2. Press the **Animator** button. As a result the *Simulation Parameter Editor* dialog will appear.
3. As this component has no parameters or input/output connectors just press the **OK** button.

As before the animation desk-top will appear. Rather than guide you through the previous animation steps only the new behavior of *testbed* will be described.

1. In the instantiation window select the *testbed* component.
2. Press the right mouse button and from the pop-up menu select **Open Animation**. In the Animation desk-top a window will appear displaying the *testbed* component (see figure 40).

Figure 40 shows the *testbed* component. As no simulation has taken place the distribution of tokens denotes the initial state of the component. The following steps will guide you to simulate one cycle.

1. In the simulation controls press the **step** button once. The number of tokens displayed on the input place in figure 40 will decrease from 4 to 3 due to the transition firing.

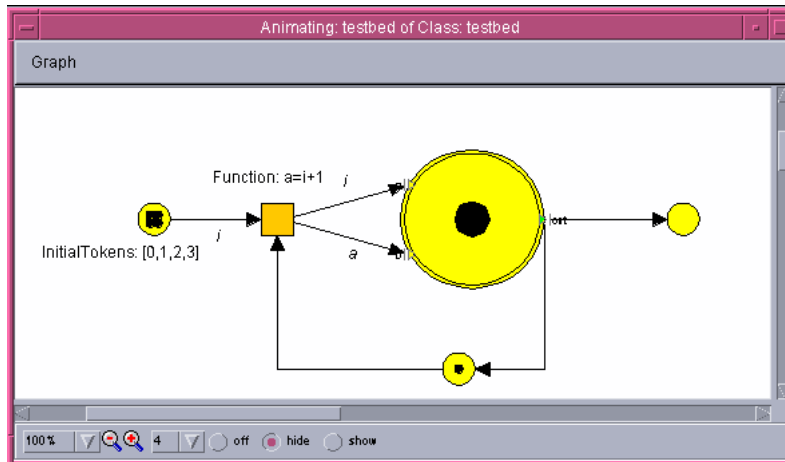


Figure 40: The initial state of the *testbed* component

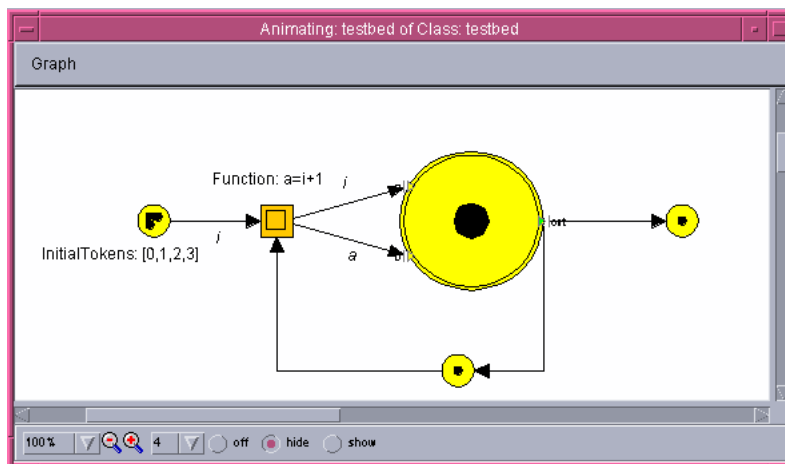


Figure 41: The state of the *testbed* component after one cycle

2. Keep pressing the **step** button until a result token emerges from the *two_a_plus_b_sqr* (see figure 41).

Finally simulate without animation and check that the result tokens are correct.

1. Press the **fast continuous simulation** button. In the status line you should see the message *Net is dead...* (see figure 42).

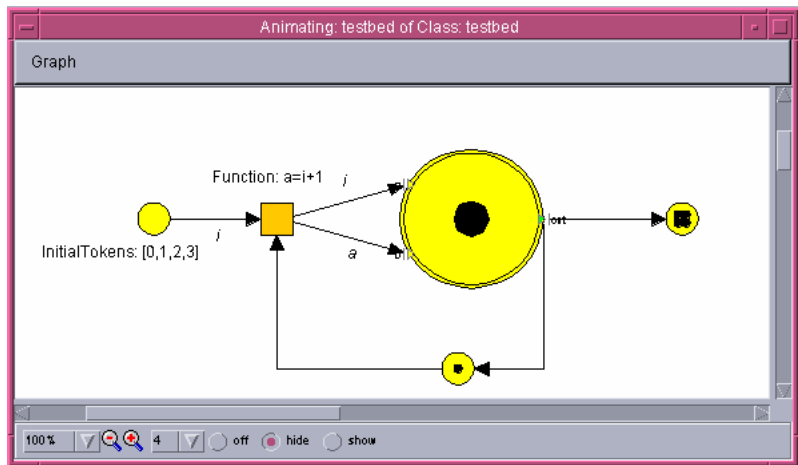


Figure 42: The final state of the *testbed* component

3 The Graph Editor Commands

Like almost all tool components in Moses the Graph editor is very configurable. The Graph editor assumes that all graphics information has been defined in the formalism *GTDL* description. A typical description defines the vertices and edges of a formalism including their graphical representation. This can be considered to define the syntax alphabet of a formalism. In addition predicates can be declared in the *GTDL* file that describe the syntax rules. These predicates are executed during the editing process or explicitly via a menu or tool bar item.

This appendix describes the components and tools of the Graph editor. More detailed information is available in the Moses user manual which also includes information on how the editor menu and tool bar entries can be configured. In this tutorial a brief description will be given sufficient to create the model.

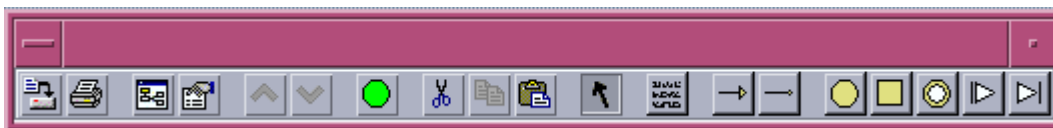


Figure 43: The Graph editor tool bar for *Time Petri Net* components. From left to right: (*Check in, Print, Open/Close Graph Editor Pane, Open/Close Attribute Editor, Move up a model hierarchy, Move down into a model hierarchy, Check Syntax, Cut, Copy, Paste, Selection mode, Insert Comment mode, Insert arc mode, Insert inhibitor arc mode, Insert place mode, Insert transition mode, Insert container place mode, Insert input connector mode, Insert output connector mode*)

3.1 The Tool Bar

Associated with the *TimePetriNet* formalism is a properties file that describes the editor tool bar. Figure 43 shows the tool bar. The following list names and describes each icon in sequence from left to right.

Save Save the component into the repository. This command overwrites the previously saved component;

Print Print the currently displayed model;

Open/Close additional Graph Editor Pane The second editor pane is used in situations where two different portions of the same model are to be viewed simultaneously at perhaps different magnifications;

Open/Close Attribute Editor The component itself and each vertex and edge may have attributes. These attributes are defined in the *GTDL* file and are edited in the attribute editor;

Move up a model hierarchy For formalisms that contain hierarchy such as Harel's Statecharts this button will move the currently visible editable model to the next highest in the model hierarchy;

Move down into a model hierarchy If the currently selected vertex contains hierarchical sub-components then the current editable model can be changed to one of the sub-components. If a vertex contains more than one sub-component then a pop-up menu allows the desired component to be chosen;

Check Syntax This tool bar item will cause the current component to be syntax checked and if errors are found an error window will appear. The color of the button indicates whether the current graph contains syntax errors (*red* \Rightarrow *error*, *green* \Rightarrow *OK*);

Cut Cut the current selection from the component;

Copy Copy the current selection;

Paste Paste the previously cut or copied selection into the component. The newly inserted elements will be selected;

Selection mode Set the editor mode so that component elements can be selected and manipulated;

Insert Comment mode In this mode when the mouse enters the edit pane a *comment* will be shown to float within the component. The component can be inserted by clicking the mouse at the desired position;

Insert arc mode In this mode when the mouse enters the edit pane an *arc* will be shown to float within the component. The source of the arc is defined by clicking the mouse on the desired vertex while the destination is defined by clicking a second time on the destination vertex. If between defining the source vertex the user clicks at a position where no vertices are present a knick point will be inserted;

Insert inhibitor arc mode As above but instead of an arc an *inhibitor arc* will be used;

Insert place mode A *place* element can be inserted when the editor is in this mode;

Insert transition mode A *transition* element can be inserted when the editor is in this mode;

Insert container place mode A *container place* element can be inserted when the editor is in this mode;

Insert input connector mode An *input connector* element can be inserted when the editor is in this mode;

Insert output connector mode An *output connector* element can be inserted when the editor is in this mode.

3.2 The Menu Bar

The following lists describe the Graph editor menus. As *TimePetriNet* is not a formalism that requires hierarchical components the *Hierarchy* menu will be ignored.

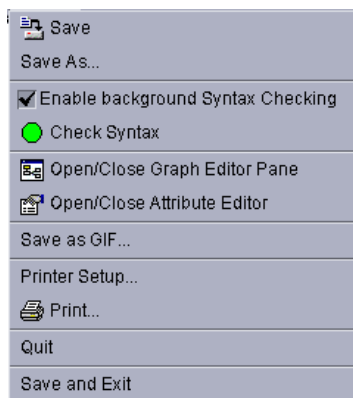


Figure 44: The Graph editor *Graph* menu

Figure 44 shows the *Graph* menu. The following list describes the graph editor *Graph* menu entries.

Save Save the component into the repository. This command overwrites the previously saved component;

Save as Save the component into the repository under a different name;

Check Syntax This tool bar item will cause the current component to be syntax checked and if errors are found an error window will appear;

Open/Close Graph Editor Pane The second editor pane is used in situations where two different portions of the same model are to be viewed simultaneously at perhaps different magnifications;

Open/Close Attribute Editor The component itself and each vertex and edge may have attributes. These attributes are defined in the *GTDL* file and are edited in the attribute editor;

Save as GIF The component is saved as GIF file for your documentation.

Printer Setup... On some platforms this menu entry will cause a system dialog to appear where printer parameters can be viewed and modified;

Print Print the currently displayed model;

Quit Close the editor discarding any changes made to the component since the last check in;

Save and Exit Close the editor after the component has been saved.



Figure 45: The Graph editor *Edit* menu

Figure 45 shows the *Edit* menu. The following list describes some of the graph editor *Edit* menu entries.

Undo Most editor operations can be undone by selecting this menu entry;

Redo Previously undone operations may be redone by selecting this menu entry;

Cut Cut the current selection from the component;

Copy Copy the current selection;

Paste Pastes the previously cut or copied selection into the component. The newly inserted elements will be selected;

Select All Selects all vertices and edges in the current component;

Select None Ensures that no component elements are selected;

Layout Graph Gives a shot at arranging your graph;

Flip Contains a sub menu with menu items to flips the current selection about the vertical and horizontal axes;

Rotate Contains a sub menu with menu items to rotate the current selection;

Align Contains a sub menu with menu items to reposition a number of elements with respect to the first selected element.

Figure 46 shows the *Insert* menu. The following list describes some of the graph editor *Insert* menu entries.

Comment In this mode when the mouse enters the edit pane a *comment* will be shown to float within the component. The component can be inserted by clicking the mouse at the desired position;

Arc In this mode when the mouse enters the edit pane an *arc* will be shown to float within the component. The source of the arc is defined by clicking the mouse on the desired vertex while the destination is defined by clicking a second time on the destination vertex. If between defining the source vertex the user clicks at a position where no vertices are present a knick point will be inserted;

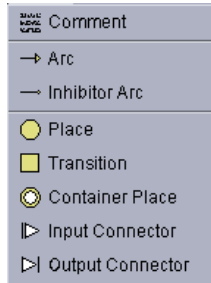


Figure 46: The Graph editor *Insert* menu

Inhibitor Arc As above but instead of an arc an *inhibitor arc* will be used;

Place A *place* element can be inserted when the editor is in this mode;

Transition A *transition* element can be inserted when the editor is in this mode;

Container Place A *container place* element can be inserted when the editor is in this mode;

Input Connector An *input connector* element can be inserted when the editor is in this mode;

Output Connector An *output connector* element can be inserted when the editor is in this mode.

4 Conclusion

The goal of this tutorial was to help you in modeling a system using the Moses tool. You were shown how to model different parts of the system using different formalisms and how to combine them into a consistent model.

The skills you have acquired will hopefully motivate you into modeling more realistic systems. Moses also comes with a repository containing a number of demonstration models. As a next step it would be useful to study these examples as they all highlight particular aspects of Moses that may not have been touched upon by the tutorial. Finally the Moses reference manual contains information about how to configure and extend Moses.

We hope you have enjoyed the tutorial and that it has left you motivated to continue using Moses.

Rob Esser, Jörn Janneck and Martin Naedele, November 1999