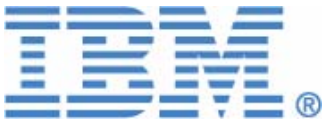
Rational. Statemate

Simulation Reference Manual

Rational Statemate
Simulation Reference Manual



Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF file available from **Help > List of Books**.

This edition applies to IBM® Rational® Statemate® 4.6 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Getting Started with the Simulation Tool

This section introduces you to the Rational Statemate Simulation tool. It describes how to start simulation and takes you through the different stages of simulation by using a simple example.

Before you begin, it is assumed that you have Rational Statemate installed on your system and you can access a project and a workarea. If you are not familiar with window and mouse operations, you may want to refer the *Rational Statemate User Guide*.

The section explains how to:

- Start the Simulation tool.
- Setup a Monitor window to examine and change values of different system elements.
- Use various commands to advance through simulation.

Simulation Tool Overview

The Simulation tool allows you to execute a graphical model. You are able to verify the behavior of your design by examining the animation of the graphical elements in your design. You can also modify and examine the values of the textual element in your design. Using the Simulation tool, you are able to experiment with “What if?” scenarios and observe the effect on your design. This aids you in detecting faults.

Simulation capabilities include:

- The ability to simulate in batch mode or in interactive mode.
- Batch mode allows for the automation of simulations with little or no user involvement.
- Global simulations can be started from the main Rational Statemate menu or local simulations can be started from an Activity-chart or Statechart.

  Interactive mode simulation allows you to have complete control over the simulation and is very useful when debugging the model.
- With global simulations you can setup a Simulation Profile. This profile allows you to set the scope of your simulation and define simulation settings. The Simulation Profile can be saved and run later.

- Playback scripts can be recorded to automate repeated execution of the model in the same scenario.
- Trace files can be recorded so that simulation data can be examined following a simulation.
- Monitor windows can be used to examine and change the current value of elements of the system.
- Waveforms can be used for graphical representation of the execution history; they display the current and past values of element of the system.
- Graphical panels representing a realistic mock-up of the system's user interface can be used to change inputs and examine outputs of the system.

Opening a Project and Workarea

To open a project and workarea:

1. Select **File > Open Project** from the top menu bar of the Rational Statemate main menu.

   The Open Project dialog box opens.

2. Select the project name to open. The screen displays all the available workareas.

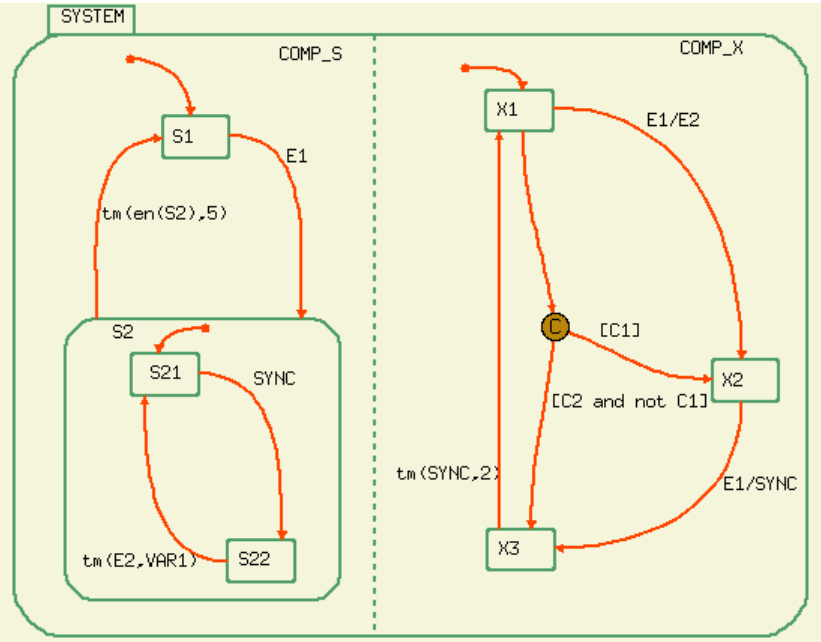
3. Select a workarea to open.

   After you make your selection, the workarea path name appears in the Workarea**:** selection box.

4. Select **OK**. The Rational Statemate main menu appears with the tool icons available for use.

Creating a Statechart to Simulate

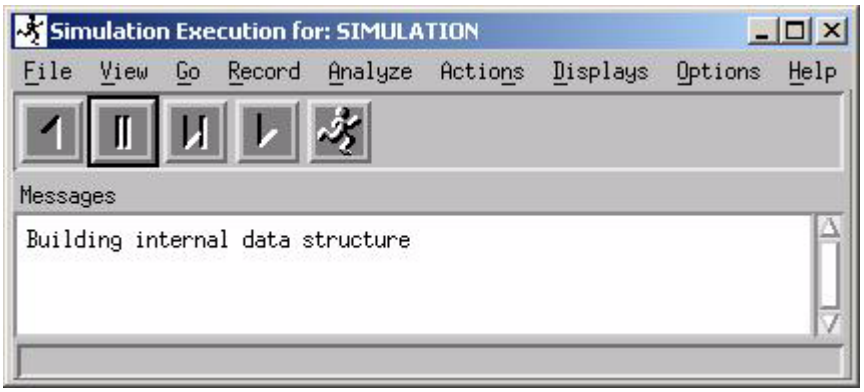
The example used in this section is of a single Statechart. Enter the statechart shown below into your workarea using the Rational Statemate Graphic Editor. Refer to the *Rational Statemate User Guide* for details on creation of Statecharts. After creating the statechart, save it and return to the Rational Statemate main window.



Opening the Simulation Tool

Simulation can be started from either the Rational Statemate Main window, an Activity-chart, or a Statechart. In this exercise, we are starting simulation from a Statechart.

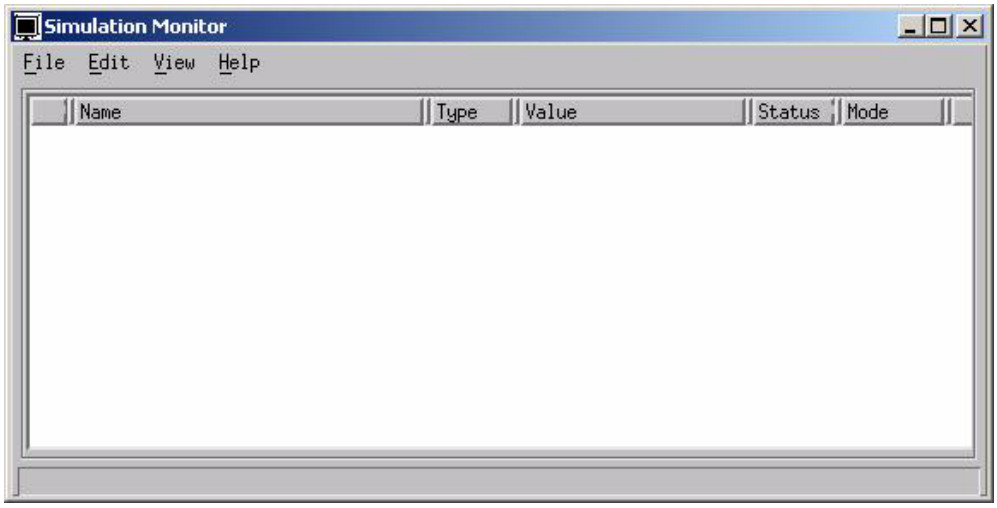
Select **Tools > Simulation** from the **Graphic Editor** menu bar. The Simulation Execution window opens. From this window, you are able to control simulation and define simulation parameters.



Opening a Monitor Window

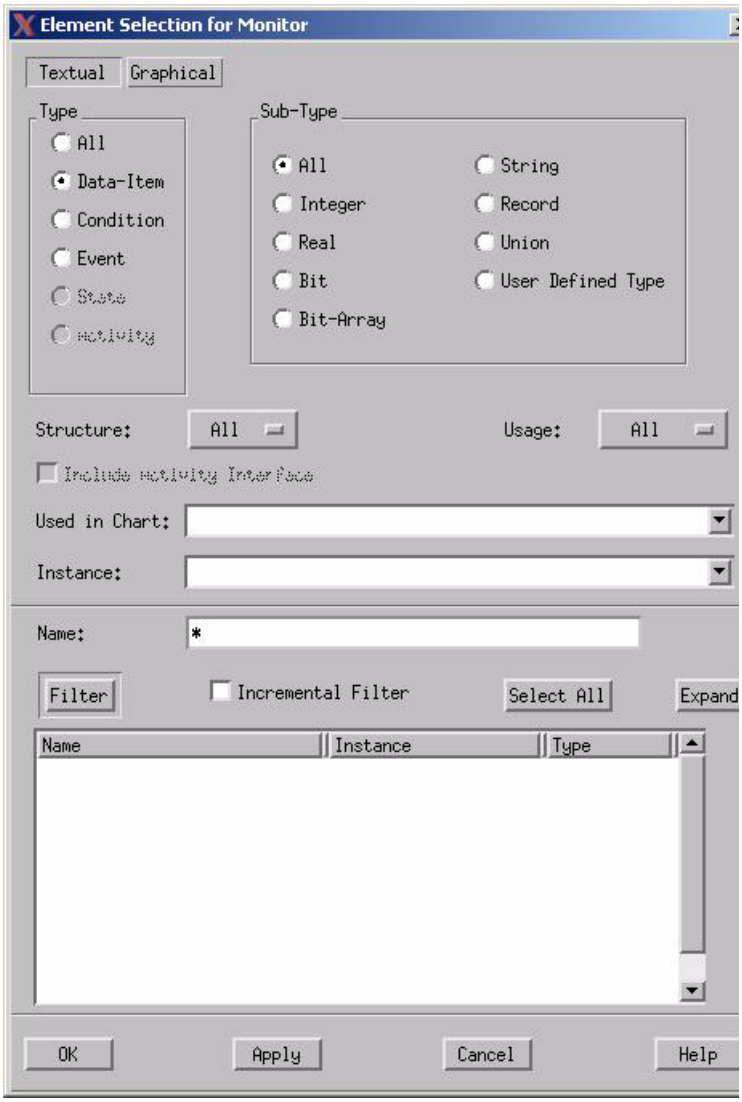
Monitors allow you to examine and change the status of elements within your model.

1. Select **Displays > Monitors**. The Simulation Monitor opens

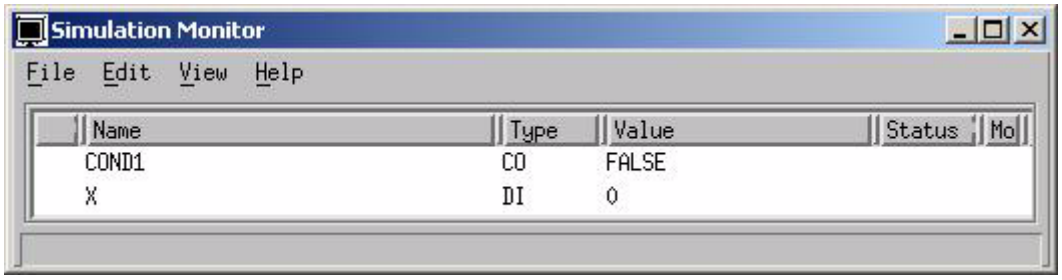


2. Select **Edit > Add** from the Simulation Monitor window. The Element Selection for Monitor browser opens on your screen.
3. Under **Type**, the default is **Data-Item**. Change this setting to **All**. This specifies that all the textual elements within the chart are added to the Statechart list.
4. Select **Filter**.

   C1, C2, E1, E2, SYNC and VAR1 are listed.
5. Click **Select All**.



6. Click **OK**. All selected elements are added to the Monitor window and their types; values and status are displayed.



Note

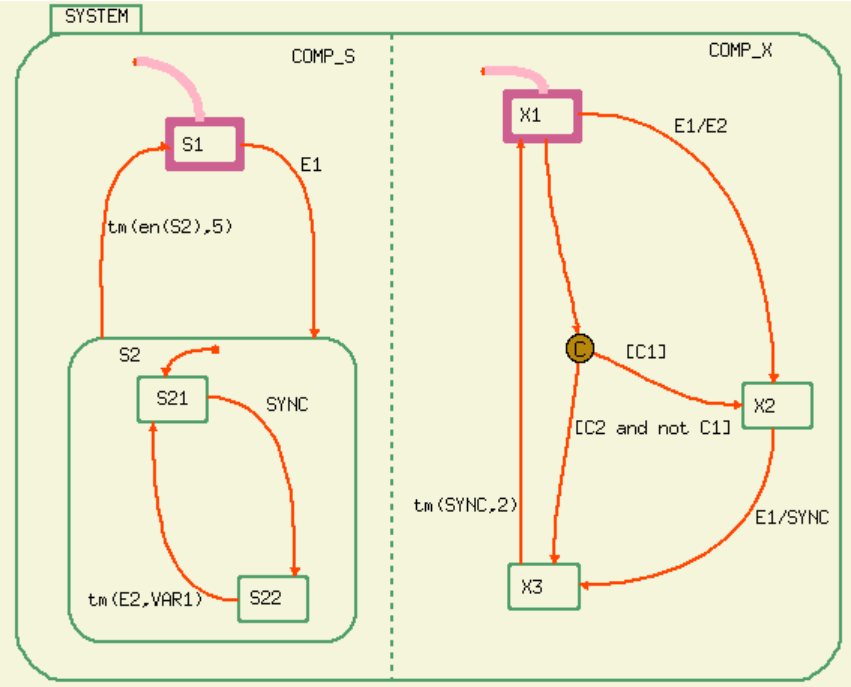
You can resize the **Simulation Monitor** dialog box so only the elements your are monitoring are displayed.

Advancing Through A Simulation

When simulating a Rational Statemate model, you advance through the simulation based on steps and time. This is accomplished by using various *Go* commands.

Simulation Stage 1 — The *GoStep*

The most basic **Go** command is the **GoStep**. The **GoStep** causes the simulation to attempt to advance one step



1. From the **Simulation Execution** menu, select **Go** > **GoStep** or click  .

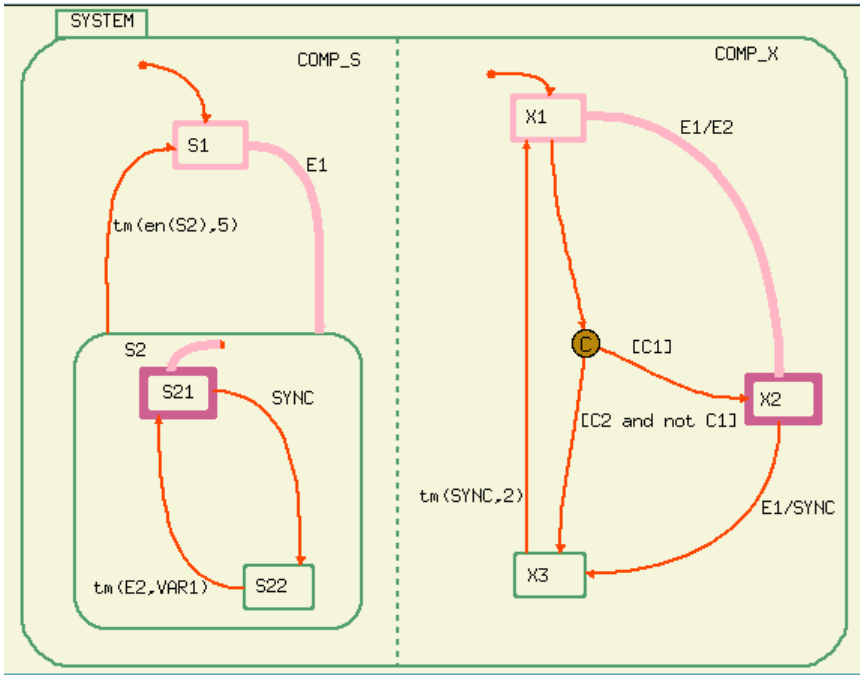
   The result is entrance of the statechart into its default states of S1 and X1.

2. From the **Simulation Monitor** generate E1 by selecting the value cell with the left mouse button.

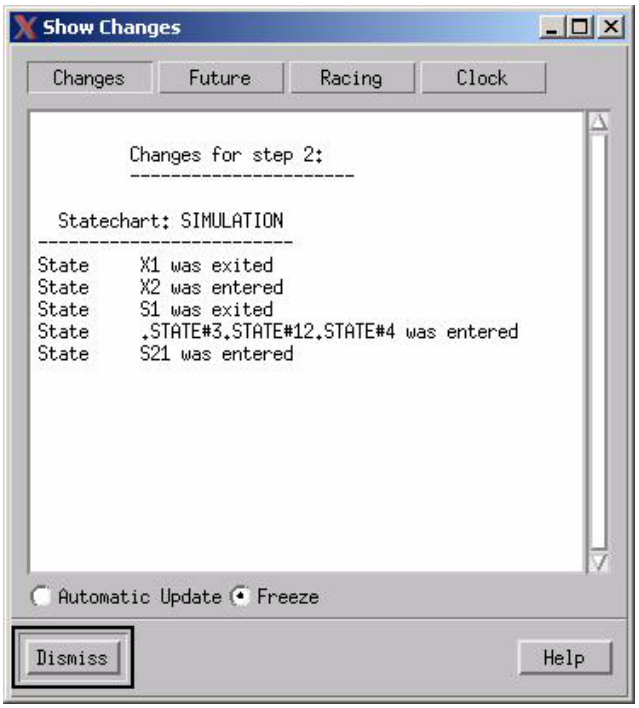
   An X appears in the value column indicating that the event is generated and is an input for the next step.

3. Select *another* **GoStep**.

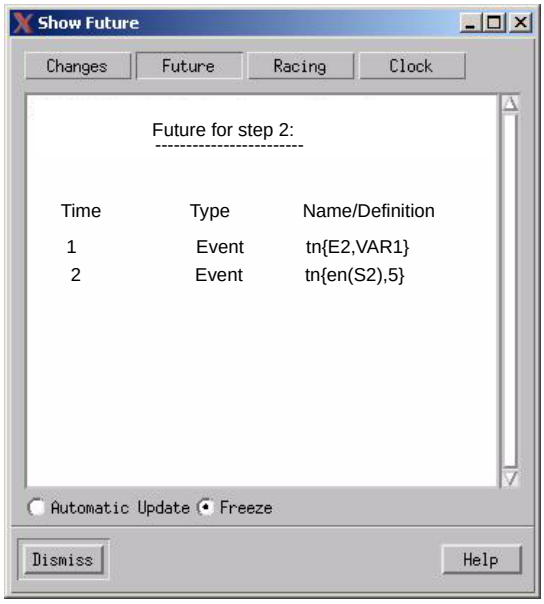
   The transition from state S1 to state S2 is taken because the trigger E1 was present during this step. The default transition into state S21 is also taken. The transition from state X1 to state X2 is taken because the trigger E1 was present during this step. The action of generating event E2 also occurred as a result of the transition from X1 to X2.



4. Select **Analyze > Show**. The Show Change dialog box opens on your screen. The **Show Change** dialog box lists all the changes that occurred in the model during the last step.



5. Select **Future**. The **Show Future** dialog box opens. The **Show Future** dialog box displays all events and actions scheduled to occur in the future.



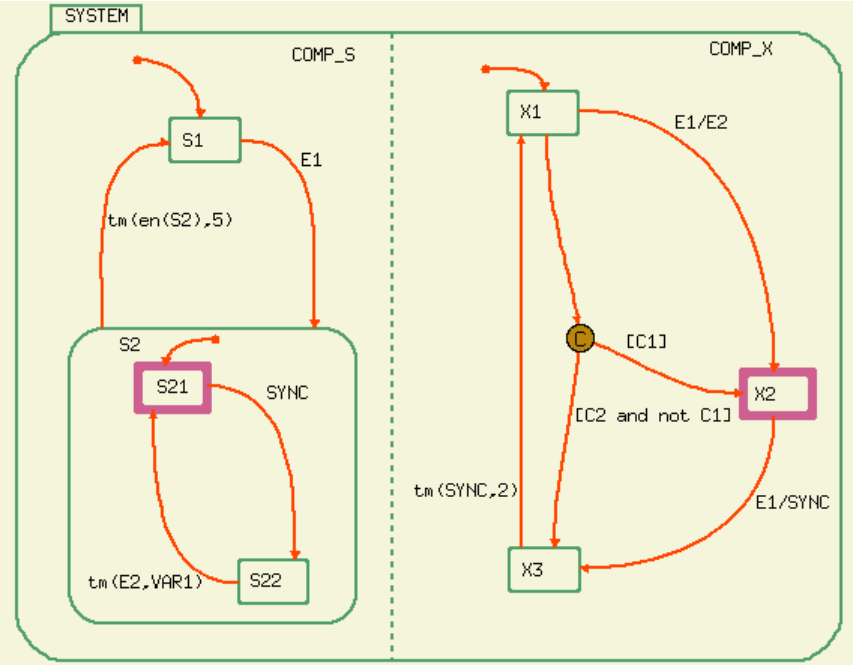
The transition from state X1 to X2 is made when E1 is generated, that, in turn, generates the event E2. The timeout event, tm(E2, VAR1), shown in the Show Future dialog box, was scheduled based on the generation of E2. E2 is still pending and requires another *GoStep* to allow the model to react to it. Also the timeout event, tm(en(s2), 5) was scheduled based on the fact that the system entered the state S2.

6. Select **Automatic Update**.

   This allows you to observe all changes related to scheduled timeouts as the changes occur.

7. From the **Simulation Execution** menu, select **Go > GoStep** or click **GoStep**.

   E2 has occurred but no transition is directly depending on it so the system remains in states S21 and X2.



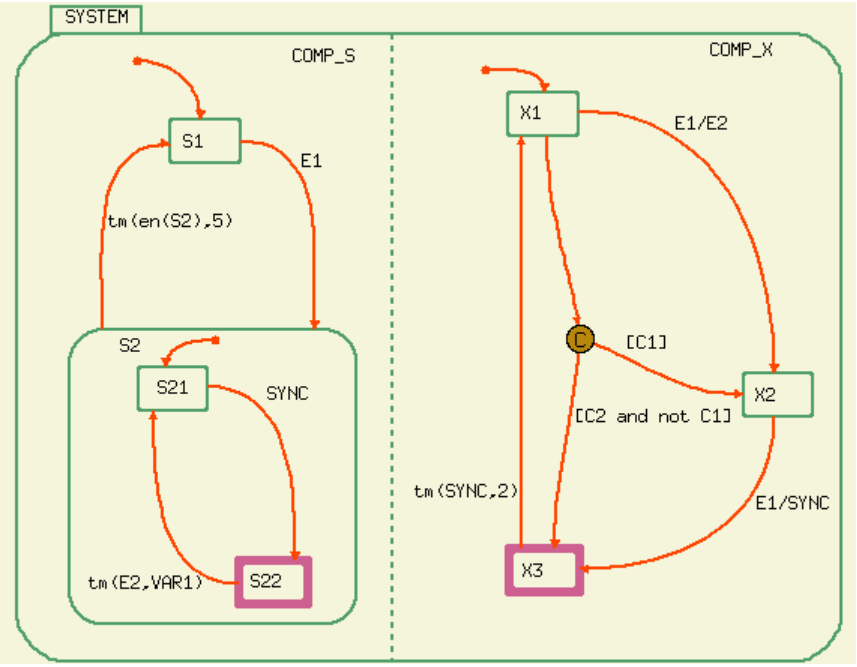
Simulation Stage 2 — *GoRepeat*

The **GoRepeat** advances simulation steps until the system reaches a stable configuration (this is referred to as a superstep). A stable condition occurs when no further steps can be taken without changing a system input value or incrementing time.

1. From the **Simulation Monitor**, generate E1 by selecting the Value cell with the left mouse button.

2. Select **Go** > **GoRepeat.** The transition is made from X2 to X3 and the event SYNC is generated. The SYNC event causes the transition from S21 to S22 to be taken.

3. The timeout events tm(en(s2), 5) and tm(E2,VAR1) have previously been scheduled. During the `GoRepeat` command, the timeout event tm(SYNC,2) was scheduled when the event SYNC was generated. Observe that the timeout is added in the Show Future dialog box.



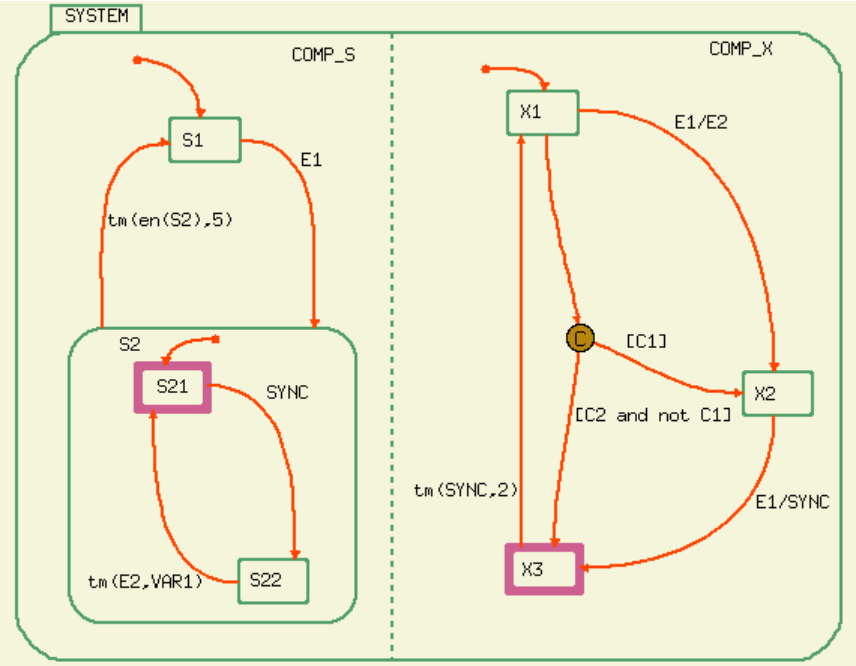
Simulation Stage 3 — *GoExtended*

The **GoExtended** either executes a **GoRepeat** or if no steps can be taken, it advances time to the nearest timeout or scheduled action. It then runs a **GoRepeat**.

1. Select **Go > GoExtended** from the **Simulation Execution** menu. This advances the time to 1.

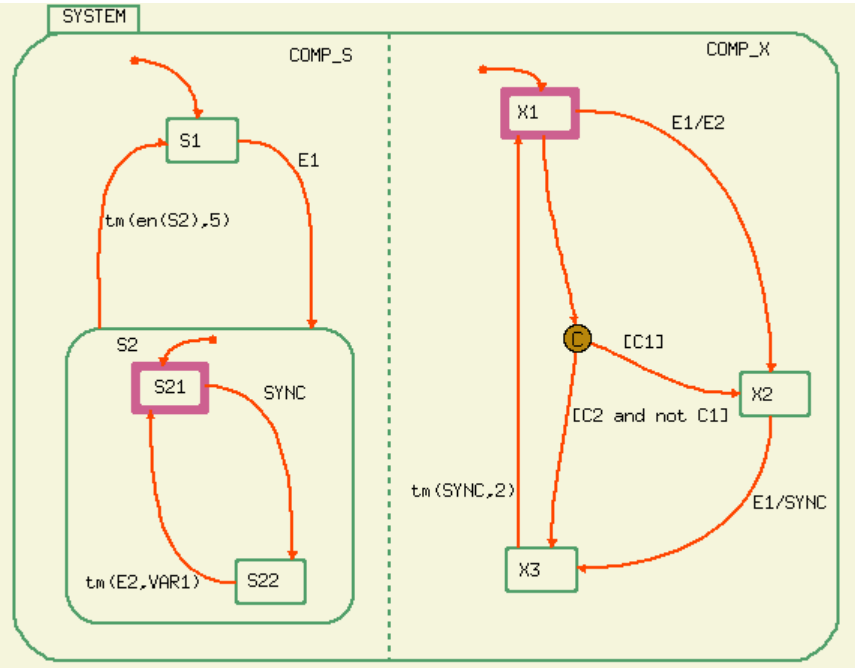
Note

Observe the time stamp in the status line of the Simulation Execution window. It shows the current time as 00:00:01. The timeout tm(E2,VAR1) occurs and a transition is made from S22 to S21.



2. Observe that the timeout tm(E2,VAR1) no appears in the **Show Future** dialog box.
3. Select **Go > GoExtended** From the **Simulation Execution** window.

   The time is now 2 and the transition from X3 to X1 is made because the timeout tm(SYNC,2) occurred.



4. Observe that the timeout tm(SYNC,2) is no longer in the list of scheduled timeouts in the **Show Futures** dialog.

Simulation Stage 4 — *GoAdvance*

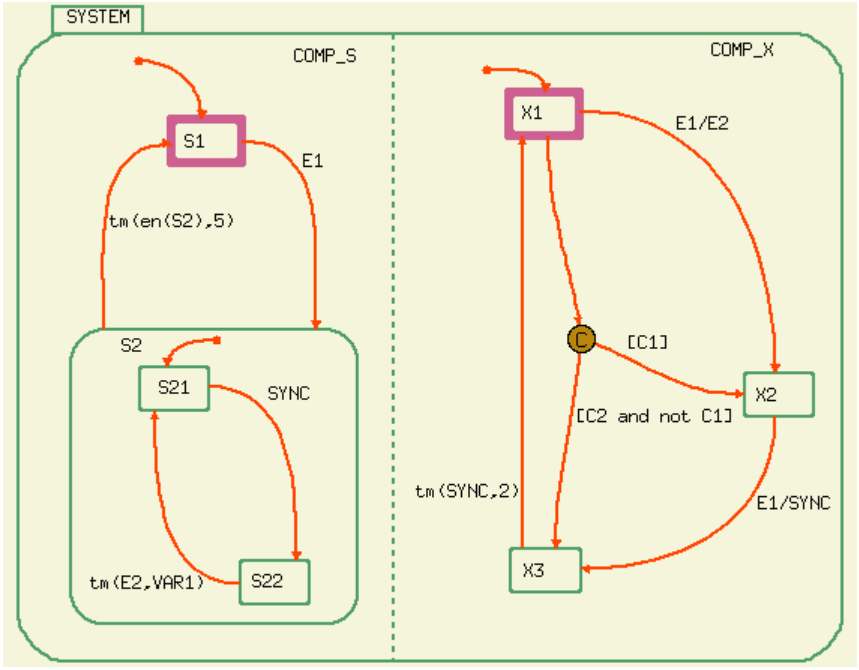
The GoAdvance advances the simulation time by a specified number (n) of units, then perform a superstep.

1. Select **Go > GoAdvance** from the **Simulation Execution** menu.

   The **Go Advance** dialog box appears on your screen.

2. Enter **3** into the Value field. Click **OK**.

   This advances the time by three units causing a transition between S2 and S1 because it is due in five time units since S2 was entered. This is the meaning of the timeout tm(en(S2),5).



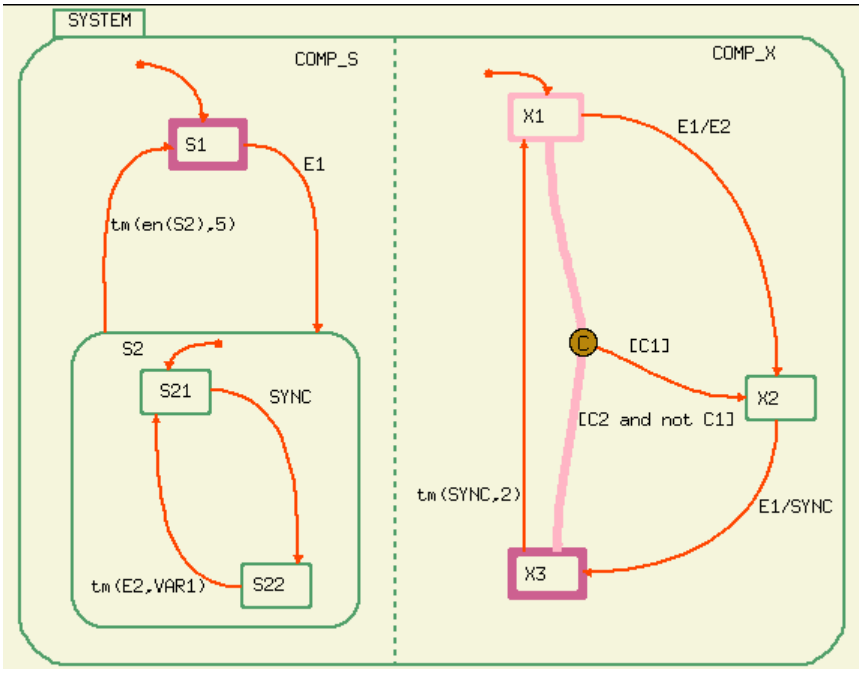
Simulation Stage 5 – Condition Connectors

1. Toggle condition C2 by selecting the value cell with the left mouse button in the **Simulation Monitor** dialog box.

   The value C2 should be true.

2. Select **Go** > **GoStep**.

   The transitions to state X3 from X1 is taken via the condition connector because the trigger [E1] evaluated the true during the step.



Exiting Simulation

1. Select **File > Exit.**

   A message appears asking if you want to save in profile your simulation environment (in this case the definition of the scope and of the monitor).

2. Click **Yes**.

   The Simulation tool is terminated for this session and simulation setup is saved for reuse.

Model Execution: Concepts and Terms

This section details the terminology and underlying concepts that make up the Rational Statemate Simulation Tool. Prior to reading this section, it is advised that you become familiar with the principles discussed in the *Rational Statemate User Guide*.

The Tool

The Rational Statemate Simulation Tool is used to examine the behavior of the specification modeled using the Statechart and Activity-chart graphic languages. During simulation, you can interactively simulate the model and view the results or you can write a program that runs in batch and portrays a test scenario. When simulating, you can examine the state of your system using graphical animation. Monitors and waveforms can be used to examine the value of elements during simulation. After the simulation is executed, you can analyze a trace report of that simulation. Refer to **Interactive Mode Simulation** for details on interactively simulating and **Batch Mode Simulation** for details about writing and executing a program in batch.

Simulation Scope

The Simulation Scope contains the set of components that are included in the simulation session. During the development of your specification, you may want to validate the behavior of only a portion of the model rather than study it in its entirety. Your choice of which aspect to examine is the simulation scope.

Determining a Simulation Scope

A simulation scope can combine any number of Statecharts and Activity-charts or portions of these. A portion a chart is a box (Activity and State) with its descendents. The simulation scope may include:

Single Statechart	This scope is used when analyzing a component of the system or when the behavior of the entire system is described by a single Statechart. The single Statechart need not be connected to a control activity.
Multiple Statecharts	This scope illustrates the interaction between components that are described by the different Statecharts. This technique is also useful when a Statechart represents the system and another represents the external environment.
Portion of a Statechart	This scope is useful when the specification is incomplete but you want to analyze the completed portion. For example, simulate one orthogonal component of a vast Statechart.
Activity-chart	This scope is used to analyze the interaction of various parts of the system each described in a different Statechart or mini-spec corresponding to the control activities.

Adding Testbenches to the Simulation Scope

To help analyze your model, it is often beneficial to add an auxiliary Statechart that monitors or drives your model during analysis. These Statecharts are called Testbenches. Testbenches have the unique ability to relate to all the elements in the simulation scope.

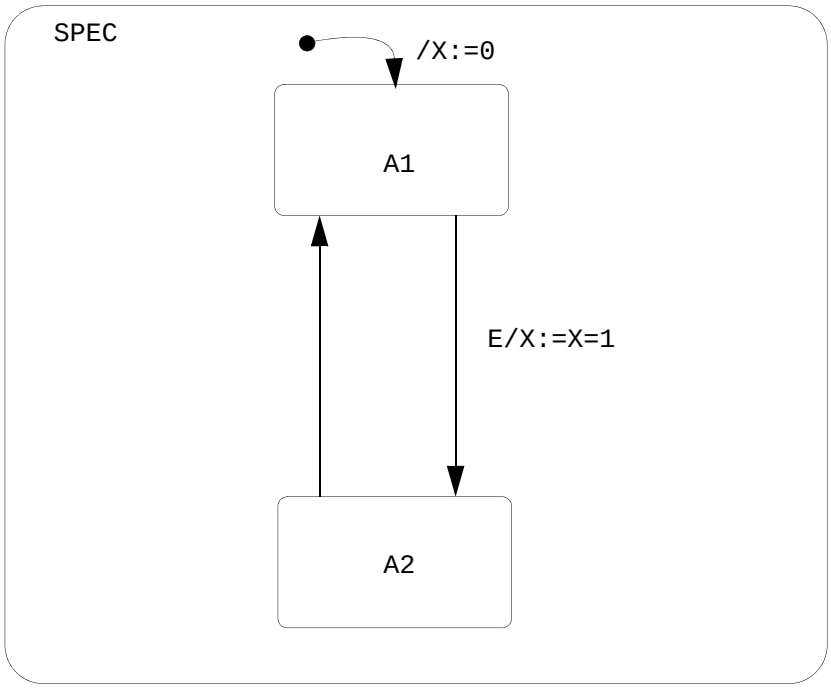
Syntactically, Testbench charts are no different than other Statecharts. Semantically, however, all elements in the scope are visible to the Testbench. Testbench charts can be used in the following ways:

Observers	Statecharts that help monitor, debug or check the performance of parts of the system. These charts do not influence the behavior of the system.
Drivers	Statecharts which represent the environment and feed the system with the needed input.
Components	Statecharts which are currently not part of the system but may be integrated later as part of the entire specification.

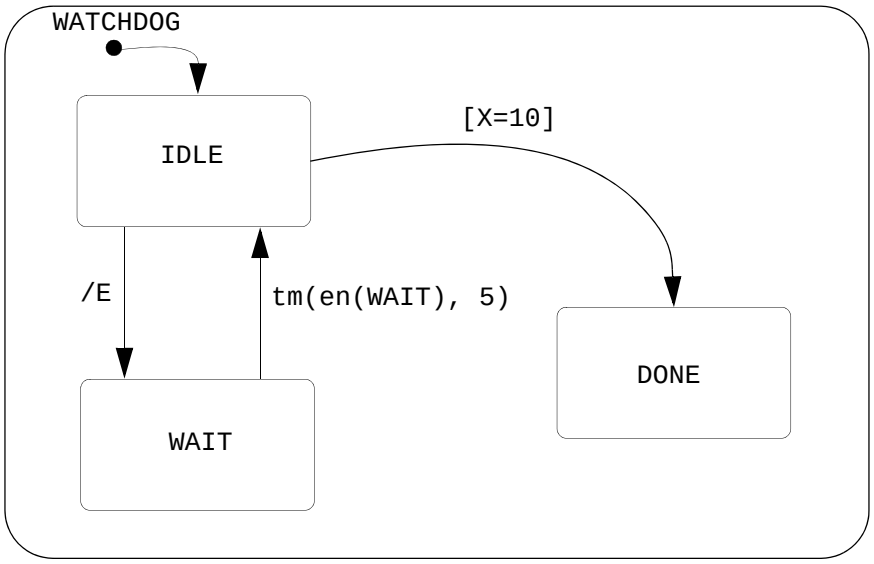
Note

Testbenches cannot be used to relate to elements in generic instances.

The following figure represents a simple Statechart where the event E is detected and causes a transition from state A1 to A2 during this transition the value of X is incremented. This Statechart is the specification of a system, where the event E is defined as coming from the environment and the system reacts to it.



Assume that the specification also states that if event E occurs, it must wait 5 time-units before reoccurring. This is expressed in the Testbench Statechart as shown in the following figure. Also assume that once X is equal to 10, event E is no longer generated.



When the Testbench chart is simulated with chart SPEC, the reference to E in the Testbench is resolved to event *E* defined in SPEC. Therefore, SPEC receives E every 5 time-units until X gets the value of 10.

Note

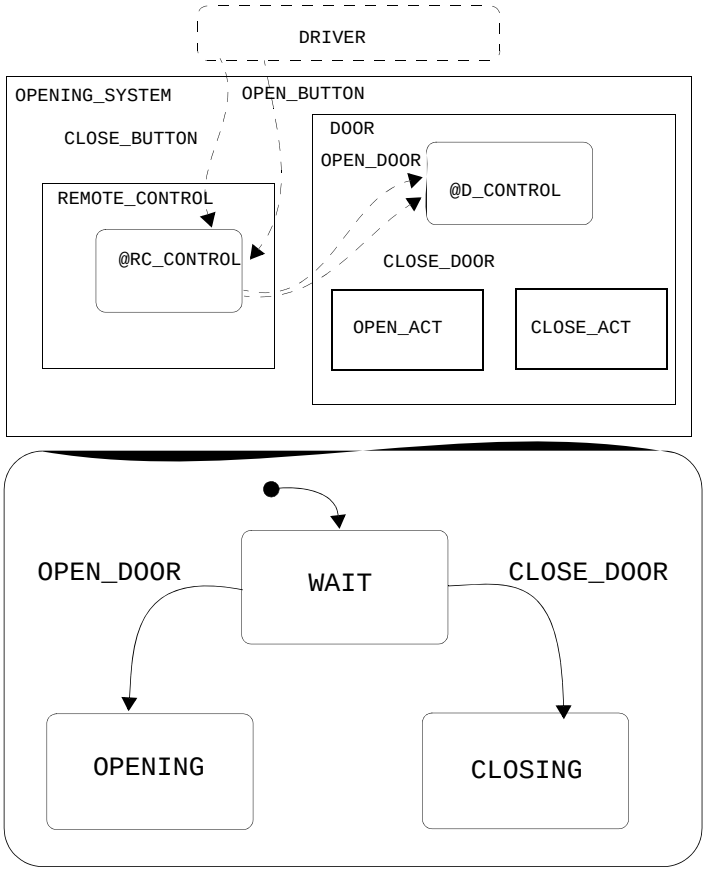
The elements X and E is defined to a chart in the model, but must be left unresolved in the testbench Statechart.

External Elements

External elements are inputs and outputs to the system. Some of the elements in the simulated scope are marked as external. These are the elements that according to the specification may change outside of the simulated scope. Elements that flow from activities outside the simulated scope into activities inside the simulated scope are external.

It is recommended that you modify only the external elements, since these appear to be environment-driven. Modifying non-external elements is allowed to provide corrections to the behavior of the model, or to complete under-specified portions of the model.

The following figure illustrates a portion of the Activity-chart for a garage door opener. Activities DOOR and REMOTE_CONTROL correspond to the two system components.



Assume that the primary interest is in the garage door subsystem. The scope is set to the activity DOOR and contains the Statechart D_CONTROL. With DOOR as the scope, the REMOTE_CONTROL becomes part of the environment and, therefore, event OPEN_DOOR and event CLOSE_DOOR are external in this simulation execution. You must generate the events OPEN_DOOR and CLOSE_DOOR.

If you later want to execute the model on the entire system, the scope is defined as the OPENING_SYSTEM activity. In this case, the events OPEN_DOOR and CLOSE_DOOR are no longer external, but the events OPEN_BUTTON and CLOSE_BUTTON are now external.

Status Of The System

Throughout the remainder of this section, the term status of the system is defined to include:

- Status of activities in the scope (active, hanging ornactive)
- The set of states the system is in (configuration)
- Values of all conditions and data-items in the scope
- Events generated in the previous simulation step
- Time delays until each scheduled action and timeout event occur
- History of the states

Context variables (their name begins with a dollar ($) sign), are not part of the status of the system. They do not retain their value from one step to another.

Simulation Step

A simulation step is a change in the system status in response to external stimuli or internal changes. A step can be triggered by an action (internal or external) or the trigger can be a timeout event occurring as a result of incrementing time.

A simulation step is a two stage process; it occurs as follows:

- Stimulus to the system occurs via actions or timeout events
- The system reacts by processing transitions, static reactions and mini-specs.

When the simulation execution begins, and before the first step is performed, the default initial status of the system is:

- When using software style activities, the activities in the top level hierarchy in the scope are active.
- When using hardware style activities, all activities in the scope are active.
- The system is not in any of its states.
- All primitive conditions are false. All primitive numeric data-items are zero and string data-items are blank.
- No events are generated.
- No timeout events or actions are scheduled.
- States have no history.

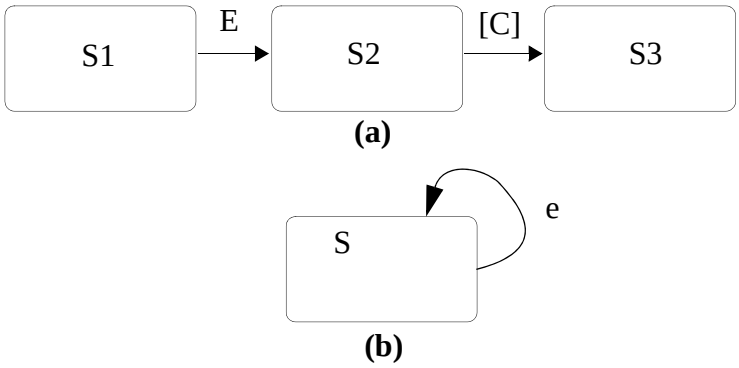
After the first simulation step is taken, the system status is:

- The state configuration includes the default states of the Statecharts connected to any active control activity, or defined to be a Testbench.
- All other elements of the system status are modified in accordance with actions performed on default connectors or by static reactions on entrances into these states.

Notes on Simulation Steps

A state cannot be entered and exited in the same step. Consider the following figure (a). Assume the system is in state S1 and the condition c is true. When event e is generated, a transition from S1 to S2 is taken. The transition from S2 to S3 is taken only in the next step.

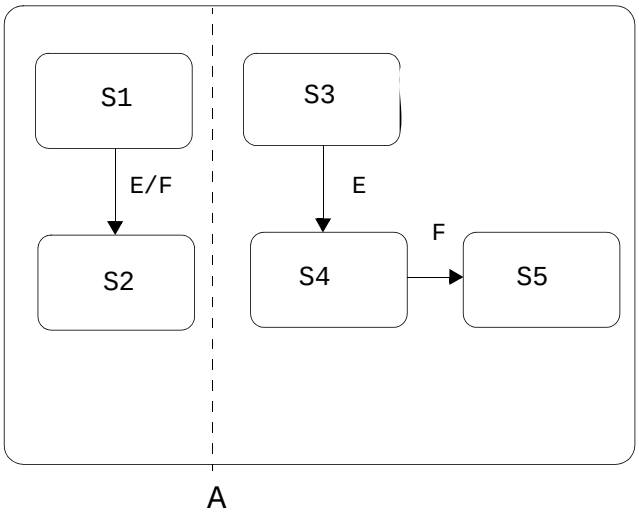
In the following figure, assume the system is in state S1 and condition C is true. When event E is generated, a transition from S1 to S2 is taken. This represents one simulation step. The transition from S2 to S3 is taken only in the next simulation step.



There is a special case in which a state may be exited and then entered in the same step. This happens when the target state and the source state are the same – as in (b) above.

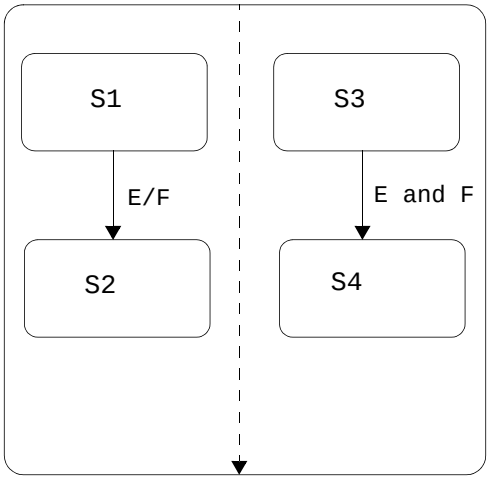
Events

An event is *alive* from the end of the step that generated it until the end of the following step. The following examples illustrate how events work.



In Chart A, make the following assumptions:

- The current active states are S1 and S3.
- Event E is generated, and the step is performed. When this happens, the following occurs:
  - The transition from S1 and S2 is taken. This causes the action of generating event F (which is present during the next step).
  - The transition from S3 to S4 occurs.
- Perform another *Step* triggered by F (event E is no longer present).
  - The transition from S4 to S5 occurs.



In Chart B, make the following assumptions:

- The current active states are S1 and S3.
- Event E is generated and the step is performed. When this happens, the following occurs:
  - The transition from S1 and S2 is taken. This causes the action of generating event F (which is present during the next step).
  - The transition from S3 to S4 does not occur because only event E is present during this step.
- Perform another *Step*.

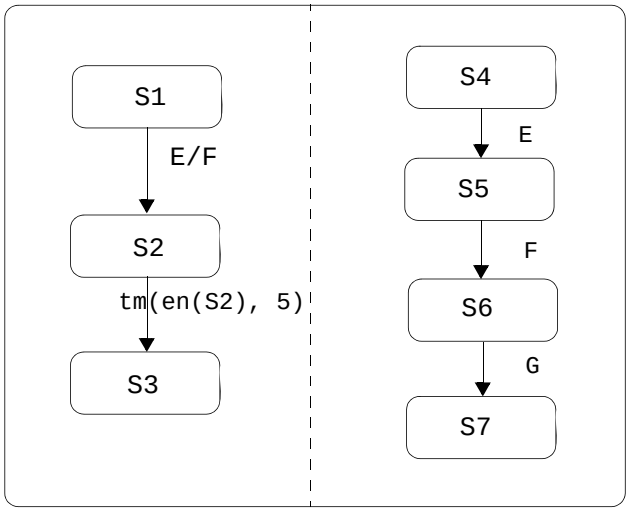
  The transition from S3 to S4 is not taken because only event F is present during the step.

Microstep

The execution of a single subroutine statement is termed a ‘microstep’.

Superstep

Sometimes, as a reaction to external changes, the system is able to perform more than one step without additional external stimuli. Each step in such a series of steps, except for the initial one, is triggered by changes the system itself produced in the previous step. This chain of steps continues until the system reaches a status from which it cannot advance without further external input and/or without advancing the clock. Such a status is called a *stable status*. The progression from one stable status to another is called a *Superstep*.



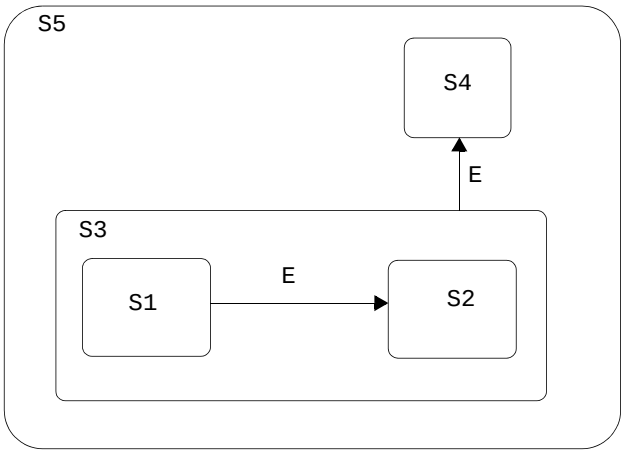
- States S1 and S4 define a stable status since no change in the system status occurs without an external stimuli being introduced.
- Event *E* is generated.
- When the step is performed, the new configuration becomes S2 and S5**.**
- Event *F* is generated internally. This allows the simulation to take the transition from S5 to S6 without additional external stimuli.
- The resulting configuration, S2 and S6, is the next stable status. Without generating event *G* or advancing the clock at least 5 units, no transition takes place. Therefore the sequence of steps from states S1 to S2 and S6 is a superstep.

Nondeterminism And Racing

Thus far all the Statechart examples have had unambiguous reactions. For each given system status only one set of reactions was enabled and the next status was clearly determined. Simulation progresses smoothly along the one *legal* path.

Transition Priority Rule

Conflict may occur when there are two or more enabled transitions which cause an exit from the same state. Some of these situations are resolved in the semantics of Statecharts by the transition priority rule.



When event *E* is generated, both the *S3* to *S4* transition and the *S1* to *S2* transition are enabled.

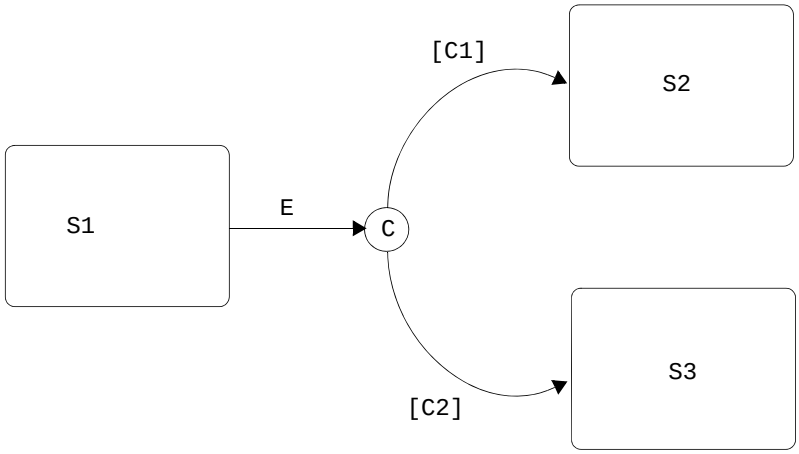
In such cases, priority is given to the transition for which the parent state common to both target states is of a higher hierarchical level.

In the above figure, the parent state of *S2* is *S3*. The parent state of *S4* is *S5*. Since *S5* is a higher level state than *S3*, the *S3* to *S4* transition is taken.

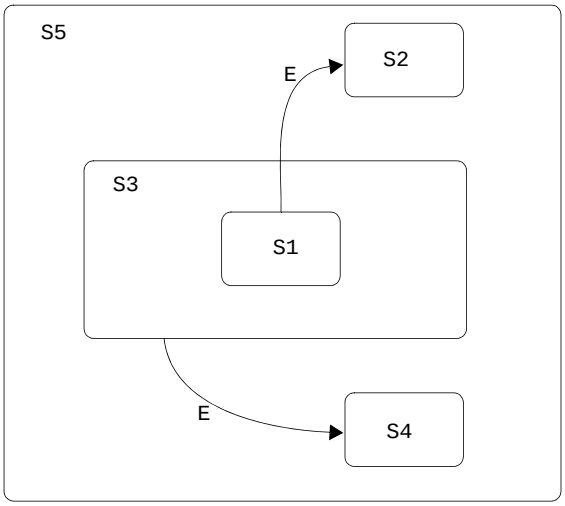
Another conflict situation in which the Transition Priority Rule applies is when some state in the current system's status contains an enabled static reaction simultaneously with an enabled transition exiting the state. The priority is given to the transition and the static reaction is not performed.

Non-determinism

Conflict situations can occur where the system's reaction is not deterministic and thus the next status can be defined in several different ways. These situations are known as non-determinism and racing.



In the above figure, if both conditions are true when event *E* occurs, there are two legal Statechart reactions. This is an example of non-determinism. During simulation execution, you must tell the tool which solution is appropriate.



Non-determinism – Example 2

When event *E* is generated, both the *S1* to *S2* transition and the *S3* to *S4* transition are enabled. Since in both cases the parent state of the target state is identical, no transition priority rule can determine the correct transition. A nondeterminism has been found.

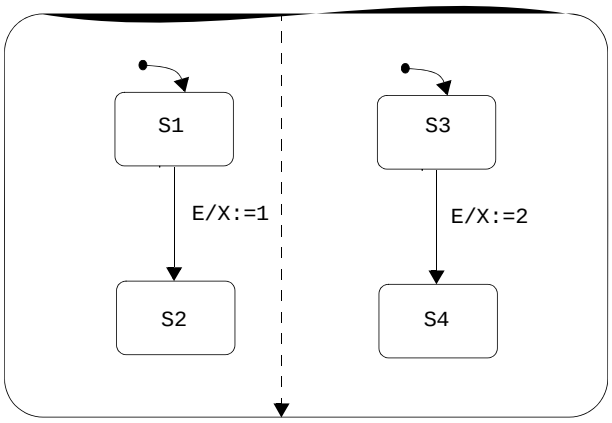
User-defined Transition Priorities

To handle situations where non-determinism may occur, the user can manually set explicit priorities on transitions directly in the Statechart, by selecting the relevant transitions and assigning them a priority.

- The priority value must be a positive integer number, where the lowest value is the highest priority.
- When priorities are not assigned to all transitions exiting a state, the transitions with no priority setting will have the lowest priority.
- 'else' triggered transitions should not be assigned a priority, since 'else' already denotes lowest priority.
- The user-defined priorities cannot violate Statechart semantics, i.e. transitions which according to the regular Transition Priority rule following the Statechart hierarchy, have higher priority than others, cannot be assigned a lower user-defined priority.

Racing

Another type of conflict, *racing*, occurs when (at the same point in time) a condition or data-item is modified more than once.



If *E* occurs when the system is in *S1* and *S3*, *X* may be assigned two different values. This is a racing condition. In this example, racing occurs when a condition is modified more than once in the same step. See **Show Racing** for information how to resolve a racing condition.

User-Case Diagnostics

The ability to record a sequence diagram during a simulation run is enhanced to allow the creation of multiple lifelines, following the selection of activities in the model.To record a sequence diagram:

1. Select **Simulation profile > Options > Sequence Diagram Generation.**
2. Select **Generate Sequence Diagram**.

This dialog box has an additional field named Activities Lifeline Selection.

Two recording modes are now supported:

- **Record Only Toplevel** - The tool generates a sequence diagram with two lifelines: one for the system and one for the user.
- **Lifeline Entities to Record** - A dialog box opens to enable you to select the activities in the model that the tool should generate a lifeline for. In this mode, the tool generates an external lifeline for each external activity that interacts with the defined lifelines (activities).

Note

The list of (internal) activities is based on semantic entities, those activities that have a separate internal clock, like a control-activity with a statechart, or a reactive mini-spec.

Click **Add** to open an easy to use tree-view of the model hierarchy. When an activity is selected and added to the list, a check is made so that only a single activity in the hierarchy is recorded at any time. For example, if both the activity and its descendant are listed in the Lifeline Entities list, only one of them has the flag that controls recording set ‘yes’. The top-most element is the default.

Time In The Simulation Execution

Until now, simulation execution has dealt with its progression in terms of steps. This section discusses time in the simulation execution.

Relationship Between Step and Time

The question is, “How does the progression of the simulation (steps) relate to the progression of time?” The Simulation Tool provides two time schemes. In both, transitions between states and static reactions within states take place in *zero* time, that is, no time passes during the step.

Step-Independent

In this scheme (called *asynchronous time scheme*), there is no relationship between the simulation step and incrementing time.

Several steps can be performed at the same time without advancing the time and time may be advanced without any steps occurring.

Therefore, the Simulation Tool differentiates between the role of step and time during the simulation. In the normal flow of simulation, time is advanced when the system is in a stable status.

In the step-independent time scheme, time is advanced after each *superstep* and not after each *step* as in the step-dependent scheme.

Step-Dependent

In this scheme (called *synchronous time scheme*), steps and time are related. Time is advanced one clock unit with each simulation step. Therefore, the Simulation Tool does not differentiate between the role of step and time during the simulation. In the normal flow of simulation, time is advanced based on stepping through the model regardless of external stimuli.

Synchronous and Asynchronous Time Scheme

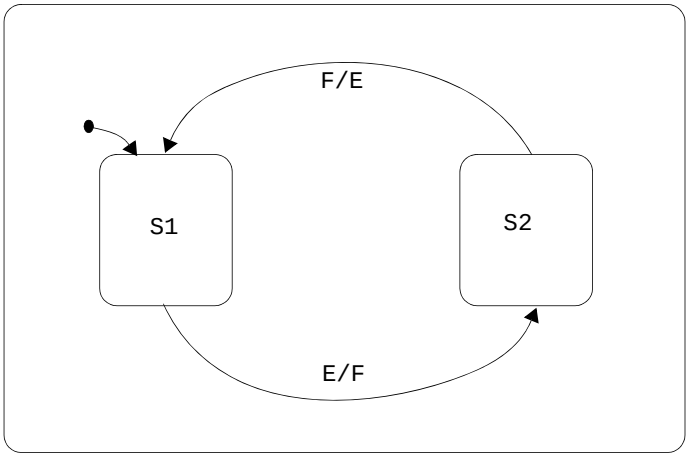
The division between synchronous and asynchronous time has been performed in the Simulation Tool.

The two schemes, synchronous and asynchronous are strictly separated. They cannot be mixed in the same simulation session. You must choose the scheme before the model execution starts.

Time in Asynchronous Simulation

Phase Limit

Since in the step-independent scheme more than one step may be taken at a time, there may be a situation in which the specified system is able to perform an infinite number of reactions without incrementing time. This is called an *infinite loop*.



If the system is in state *S1* and event *E* is generated or if in state *S2* and event *F* is generated, the simulation toggles infinitely between *S1* and *S2*.

To avoid infinite loops, a phase limit is defined which restricts the number of steps that can be taken without advancing time. Phase limit restricts the length of all supersteps, even those that would not result in an infinite loop. The phase limit is set using the *Steps per Go* parameter.

Time in Synchronous Simulation

Statechart Clocks

For each of the Statecharts in a simulation scope, the time increment per step (duration of the step in the chart) can be defined individually. A clock increment of a chart defines the point in time when the chart gets control for step execution. If a simulation session includes Statecharts which have different clocks, the following interactions occur:

Steps in Synchronous Time Scheme

At the very first step of the simulation, all top level Statecharts in the scope execute their default transitions; this step finishes at time 0.

At any other step, the clock is advanced to the nearest time when some chart can obtain control for step execution.

Empty Steps

If the execution of a step by a chart does not cause any changes in the model, then the step counter is not advanced. Note that the clock is always advanced as described above.

Buffering Events

Whenever an event is generated, either internally or externally, it is buffered by all charts waiting for their turn to execute a step.

A buffered event remains active for a chart until the chart gains control (i.e., until the chart senses the event and reacts to it). After the chart accomplishes its steps, the event becomes non-active in reference to that chart.

Scheduling Timeouts

When the same timeout TMO (defined as tm(E,T)) is used in Statecharts with different clock increments, it is scheduled differently for each of the charts. In chart S, TM0 is scheduled to occur at T0 +T, where T0 is the first moment of time after the event E was generated that chart S gets control. S reacts to TMO in the first step it runs after T0+T.

Toggling Events

When an event is generated, its status with respect to each of the charts is toggled. If the event is active for some charts that are waiting for their turn to step, it becomes non-active for these charts. For charts that the event is not active, it becomes active.

Go Commands

Go commands are used to advance the model execution. This section provides an initial definition of each `Go` command. The following table supplies a description of each step in asynchronous and synchronous simulation mode.

	Asynchronous Simulation	Synchronous Simulation
GoStep	Runs one step and consumes no time.	Runs one step and consumes one time unit.
GoStepN	Runs a specified number (N) of steps.	Runs a specified number (N) of steps.
GoRepeat	Performs a superstep and consumes no time.	Performs a superstep and consumes the amount of time units equal to the number of steps taken during the superstep.
GoNext	Advances time to the next scheduled action or timeout event. Then performs a GoRepeat.	Advances time to the next scheduled action or timeout event. The number of steps taken are equal to the number of time units.
GoExtend	Performs a GoRepeat. If a superstep can't be taken, then it performs a GoNext then a GoRepeat.	Performs a GoRepeat. If a superstep can't be taken, then it performs a GoNext then a GoRepeat.
GoAdvance	Allows you to advance the time by specifying the increment of time. A GoRepeat is taken.	Allows you to advance the time by specifying the increment of time. The number of steps are equal to the number of time units advanced.
AutoGo	The *AutoGo* command is used to perform a GoStep in an unstable status otherwise it performs a *GoNext*.	The `AutoGo` command is used to perform a GoStep in an unstable status otherwise it performs a `GoNext`.
GoBack	Undo last Go command	Undo last Go command

AutoRun Mode

In **Autorun** mode, simulation runs continuously with the entire interaction being performed through panels and monitors. Simulation behaves as if it was a real-time execution of the model. This is achieved by the following:

- Each time the simulation clock is advanced by T time units, a real-time delay is included with a duration proportional to T.
- By default, one unit of simulation time is represented by 1 second of real-time. When needed, a different mapping can be defined. This is done using the field Autorun Time Factor of the Time Setting dialog box in the Analysis profile.

AutoRun can be interrupted and the simulation clock advanced busing the *Go Advance* and *Go Next* commands. This does not involve any real-time delay.

Asynchronous Time Model

When there are external inputs, *AutoRun* performs a *GoRepeat c*ommand and the clock is not advanced.

In a stationary situation, *AutoRun* continues to advance the simulation clock by 1 time unit.

When distance to the nearest scheduled item is less than 1 time unit, the simulation clock is advanced to the time of this scheduled item, and not by 1 unit.

Synchronous Time Model

AutoRun performs an ongoing execution of the *Go Step* command. Correspondingly, the simulation clock is advanced between successive control points, in which at least one of the charts takes control according to the definitions of local clock.

Simulation Support of Flowcharts

The Rational Statemate Simulation was enhanced to support Flowchart as a control activity implementation. That means that a control activity may be now an instance of a Flowchart, in addition to a Statechart.

The controlling Flowchart may be the top-level of a hierarchy of flowcharts descending from it, through offpage and generic instances.

Flowchart Semantics

There is a major difference between Statechart and Flowchart behavior in the model: whereas the execution of a Statechart is time-consuming, with each transition in a Statechart is considered a simulation “step”, a flowchart is executed in “zero time” and single simulation step. Whenever the Flowchart is entered, it is executed from start to end in one simulation step (much like a Procedural-Statechart).

Since the Flowchart is executed in a single simulation step, the elements used in the flowchart are single-buffered during the Flowchart execution, except for events and event expressions. Events and event expressions are sensed in the next step. Timeout and Delay expressions are not allowed at all.

A Flowchart connected to a control-activity are executed (start to end) once in every simulation step, for as long as the controlled activity is active.

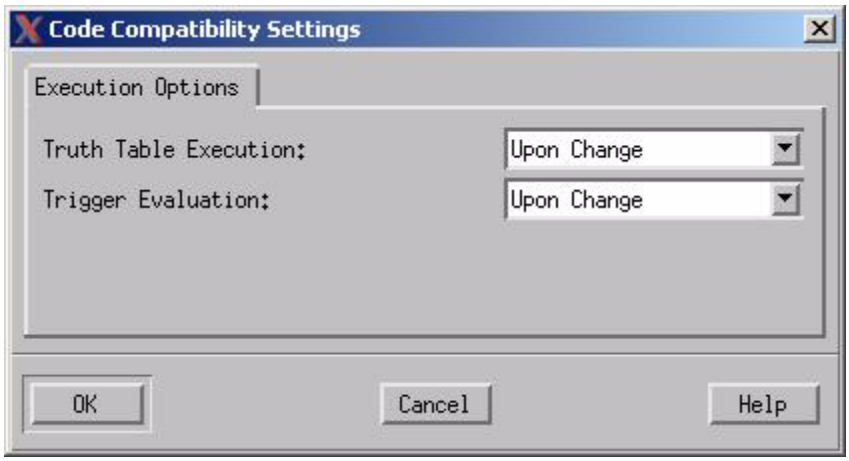
Examples:

- **Example A**: When setting a value in one action box of the Flowchart, that (new) value is sensed in the next action box, e.g.: The condition “[num==5]” evaluated immediately after the action box “num=5” is always “true”.
- **Example B**: When the action in the action-box is “st!(A1)”, with A1 being a sibling activity of the control activity, the event “st(A1)” is “true” only in the next step, i.e., the expression “st(A1)” is evaluated to “false” immediately after the action box, and to “true” in the next step, i.e., the next run of the Flowchart.

Code Compatibility Settings

Simulation behavior can be customized to be compatible with the Code Generator. The options are combined into **Code Compatibility Settings** dialog box.

To access the **Code Compatibility Settings** dialog box, from the **Simulation** menu, click **Options**, click **Code Compatibility,** and then click **Code Compatibility settings**. The **Code Compatibility Settings** dialog box displays.



This dialog box controls the options for the Simulation profile and are saved with it.

- **Truth Table Execution**—A Truth-Table is executed:
  - **Upon Change**—Only if one of the Truth-Table's inputs is changed.
  - **Every Step**—Every Simulation step.
- **Trigger Evaluation**—A Trigger is evaluated:
  - **Upon Change**—Only if one of the elements in the Trigger expression is changed.
  - **Every Step**—Every Simulation step.

Flowchart in Simulation

The Simulation Micro-Debugger was enhanced to allow debugging of Flowcharts. This is done by setting a breakpoint on a special subroutine, created by the Simulation Micro-Debugger for each Flowchart, through the “Simulation Breakpoint Editor”.

The special subroutine is named as follow:

```
“FLOW_<flowchart_name>_PROC”
```

with the “<flowchart_name>” being replaced with the actual Flowchart name.

When a Flowchart with a breakpoint set on it is executed, the Micro-Debugger pop-ups and highlights the executed Flowchart. The Micro-Debugger allows micro-stepping through the Flowchart while seeing it being animated as well as watching element changes in the Micro-Debugger monitor.

Flowchart in Simulation - Limitations

Queue operations in Flowcharts are not supported. A warning is generated for expressions including queue references that the operation is ignored.

Interactive Mode Simulation

The brief example in **Getting Started with the Simulation Tool** illustrated some of the techniques and concepts handled by the interactive simulation. This section details the menus, options and forms you need to properly analyze your design model. A comprehensive interactive example completes the section and ties together some of the issues discussed.

This section assumes an understanding of the Statechart and Activity-chart principles as well as the command input techniques discussed in the *Rational Statemate User Guide*.

The Three Phases Of Interactive Simulation

Performing an interactive simulation involves the following three primary phases:

- **Starting the Interactive Simulation Tool** – This phase involves starting the tool from either the Rational Statemate Main Menu or from the Statechart or Activity-Chart Graphic Editor.
- **Simulation setup** – This phase involves defining the simulation scope, naming the files for storing the recorded inputs and outputs to the simulation and setting simulation parameters.
- **Executing Commands and Observing the Results** – This phase involves using the various interactive command menus and describes how to interpret the graphical and textual results of the simulation.

Starting the Simulation Tool

The Simulation Tool gives you the capability to interactively analyze your design. It uses the power of graphical animation and interactive batch stimuli in combination with monitor windows, graphic panels and waveforms.

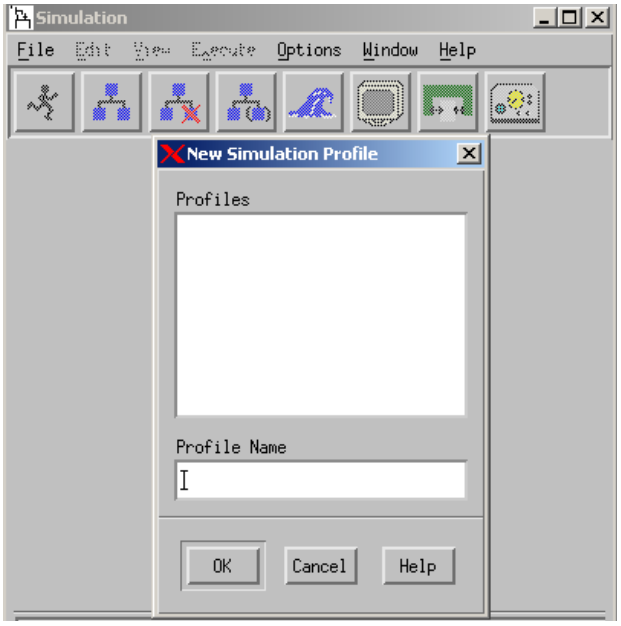
The Simulation Tool graphically depicts the behavior of your design by animating the statecharts and activity-charts. This tool can be started from either the Rational Statemate Main Menu or from one of the Statechart or Activity-chart Graphic Editors. The following discussion outlines how to start the tool from these sources and how to *connect* the charts to the Simulation Tool.

Starting the Simulation Tool from the Rational Statemate Main Menu

This section shows how to start the Simulation tool from the Main Menu. (**Starting Simulation from the Graphic Editor**, provides information on how to start simulation from the Graphic Editors.)

1. Select **Simulation** from the Rational Statemate main window. 

   The **Simulation Profile** editor appears along with another window that allows you to open an existing Simulation profile or if one is not available create one.



2. Select a **Profile** from the *Profiles* list and select **OK** or **create a new profile** and click **OK**. All the menu options and icon functionality are enabled.

Note

- Profiles can also be selected by double clicking on the profile name.
- If there are no profiles on your list, use the **File > New Profile** command to create a new one.

Starting Simulation from the Graphic Editor

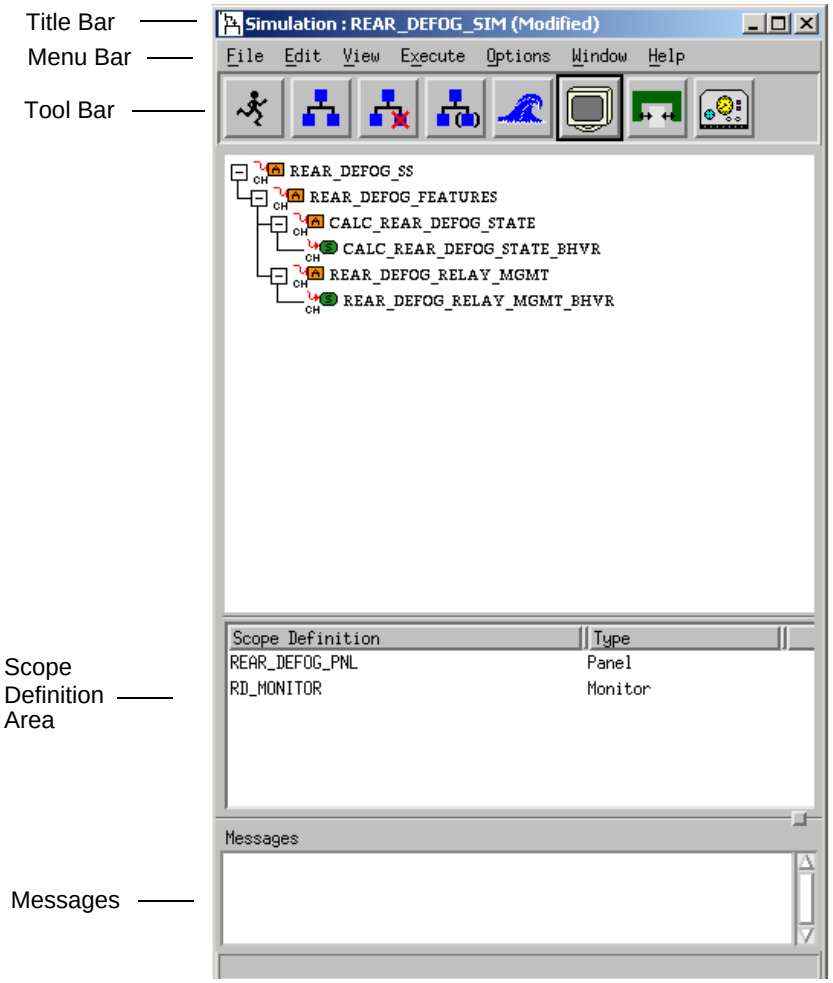
Starting simulation from the Graphic Editor tool can be useful for debugging one or several Statecharts or Activity-charts. With this option, the Simulation Scope is automatically set to the needed chart(s). Environment elements such as monitor windows, waveforms and panels must be set within the simulation.

1. With the Graphic Editor open, activate a Statechart or Activity-chart.
2. Select **Tools > Simulation** from the Graphic Editor menu. The **Simulation Execution** menu displays.

The Profile Editor

The purpose of the Profile Editor is to provide the user the ability to build a robust, reusable, simulation environment. The Profile Editor allows you to identify the scope of the simulation including design components (charts), recording mechanisms for input and output, and system parameters such as clock rates to be used throughout the simulation run. The Profile Editor allows you to build the framework for interactive or batch simulation.

This section introduces you to the Rational Statemate Simulation Profile Editor. A set of procedures for creating and customizing a profile is included in this section. This section assumes an understanding of the Statechart and Activity-chart principles as well as the command input techniques discussed in the *Rational Statemate User Guide*.



Profile Scope Definition

The Profile Editor allows you to specify the scope of the simulation as well as select the parameters that control the simulation. Use the Profile Editor to define a simulation's scope by identifying the components to be simulated such as charts, panels, and waveforms.

Once created, the profile is a record of what was included in its definition scope. Profiles also provide a convenient way to store settings that are used repeatedly in different simulation sessions.

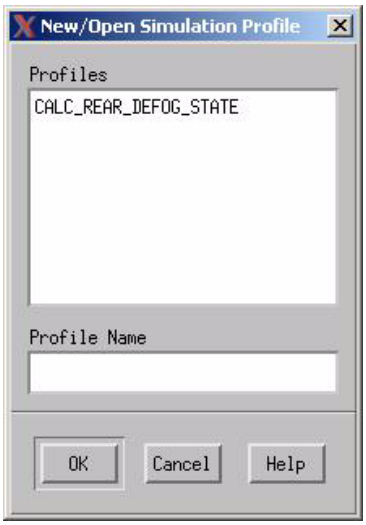
You can store the profile in your workarea where it can be retrieved, edited, and used over and over again for subsequent simulations.

The following procedures show you how to create a new Simulation Profile using the Simulation Profile Editor and how to customize it.

Creating a New Simulation Profile

This section shows you how to create a new Simulation Profile.

1. Select **File > New Profile**. The **New/Open Simulation Profile** dialog box opens.



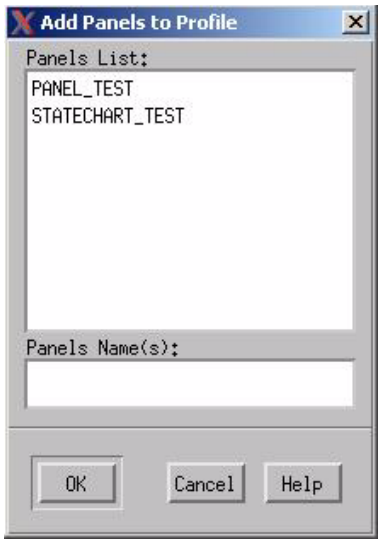
2. Enter the new profile name in the **Profile Name** text box and click **OK**. The **Profile Editor** appears now with all the options enabled and the name of the profile in the title bar.

Adding Components to the Profile

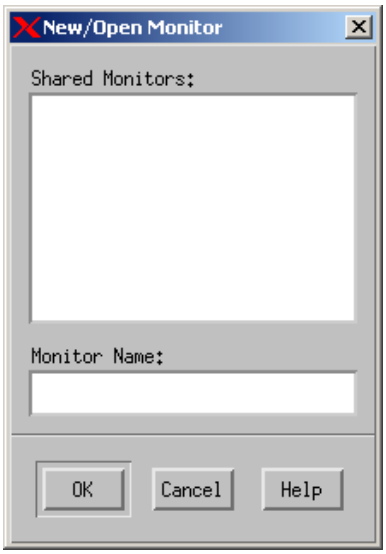
This section describes how to add components (i.e., Statecharts, Activity-charts, Panels, etc.) from the Chart Tree to the Scope Definition of the Simulation Profile.

1. Click **Add Chart(s) with Descendants to Profile**  or select **Edit > Add with Descendants** to bring up the Charts Tree and select charts to add to the profile.

2. Click **Add Panel to Profile**  or select **Edit > Add/Edit Panel** to bring up the Add Panels to Profile dialog box and select a panel(s) to add to the profile.



3. Click **Define New or Edit Existing Monitor Definition**  or select **Edit > Monitors** to add or create a monitor to add to the profile.



Note

You can view the scope definition in either Tree (graphical display) or List (textual display) format by selecting **View > Show Scope as Tree** or **View > Show Scope as List.**

Saving the Profile

To save the profile, select **File** > **Save**.

Starting Simulation from the Simulation Profile Editor

Starting simulation from the **Simulation Profile Editor** allows you to predetermine the environment for your simulation. Monitors, waveforms, and panels can be included in a profile. When a profile is executed, all components included in the profile participate in the simulation.

To start a simulation from the Simulation Profile Editor, click **Invoke Simulation**  or select *Execute* > *Execute Simulation*. The **Simulation Execution** window for the selected profile opens.

Entering Commands To The Simulator

Menus/Toolbars

Commands to the Simulator can be entered by the use of a number of menus that are pulled down from the menu bar or activated via the toolbar. A description of each command can be found in **Supplementing the Model with Handwritten Code**.

Command Line

One method of entering commands to the Simulator is by using the command line in the **Simulation Execution** dialog box.

To utilize the command line options, from the **View** menu, click **Command Line**. The command line appears below the toolbar.

Input Changes

Do Action Commands

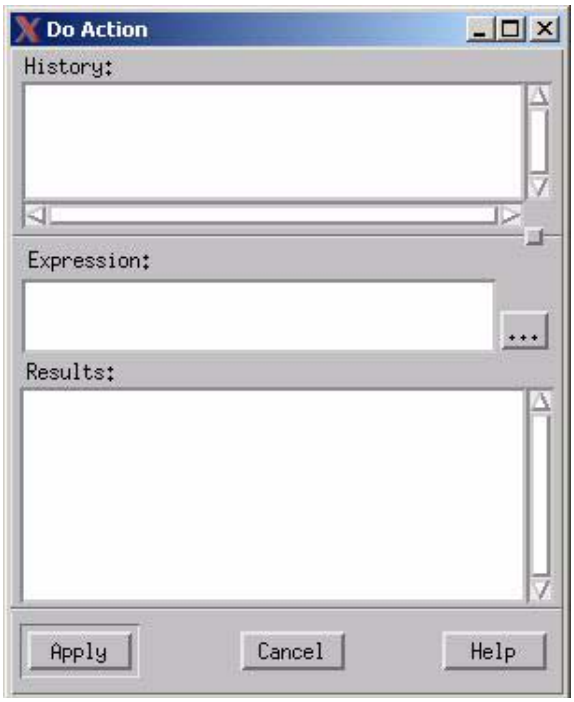
One method of entering information about the design's environment is the `Do Action` command. `Do Action` is directed for those users who know exactly the names of the actions to be taken.

Note

It is also possible to use the Monitor Window or the Panel to generate events, conditions, and actions.

Using DO Action

1. In the **Simulation Execution** dialog box, select **Actions > Do Action**. The **Do Action** dialog box opens.



2. Enter any valid Do Action expression into the Expression text box.

Note

You can select elements for your `Do Action` by clicking the ellipsis button  . This starts the Select Element browser.

3. Click **Apply**. The entered action is executed and echoed in the **History** field.

   Note: Clicking **Cancel** before **Apply** does not execute the *Do Action*. Clicking **Cancel** after **Apply** runs the *Do Action* and dismisses the **Do Action** dialog box.

Valid Input To Do Action

Input into the `Expression:` area may be any valid Rational Statemate action. The following are some examples:

- `clear_buf`

  where `clear_buf` is a defined action.

- `i := i+1;ax:j :=5; day::SUNDAY;`

  where `i` and `j` are data-items. In this example, `i` is unique in the WorkArea while `j` is not, therefore, is specified as belonging to a specific chart `AX`. The day is the data item of an enumerated type and SUNDAY is on of its values.

- `st!(A)`

  where `A` is an activity

- `sc!(e,delta_t)`

  where `e` is an event and `delta_t` is a data-item

- `scr:e`

  where `e` is an event belonging to the chart **SCR**

Invalid Input to Do Action

The input to Do Action cannot be any of the following:

- A syntactically invalid action expression.

  For example, the expression *"i=5"* and *"st(A)"*. These are a condition and an event, respectively.

- An action which is semantically incorrect since it uses an element contrary to its type.

  For example, `if d then A end if`, where `d` is defined as a data-item and not as a condition.

- An action that refers to an item defined in your WorkArea but not uniquely identified.

  For example, the input `j:=5` where `j` is defined as a data-item in a number of different charts. You must specify which chart is being referenced.

- An action that refers to an element not belonging to your WorkArea.

  For example, the input `i:=5` where `i` is neither defined nor referenced in the charts in your WorkArea.

- An action that changes the value of a compound item (an item defined in terms of other items).

Response to Invalid Do Action

The Do Action command responds to your command input in one of the following ways:

- If the input is in error, a message is issued and the input is discarded.
- If the action is syntactically correct but the action affects elements which are not within the simulation scope, then the element is included in the scope, a message is issued, and the action is performed.

For example, consider the action `i:=5` where *i* is a data-item defined in the WorkArea but not used anywhere in the simulation chart or in any textual element associated with it. The following message is issued:

> Data-item I inserted into the Simulation scope

Go Commands

Go commands are used to advance the model execution. This section provides an initial definition of each Go command. The following table supplies a description of each step in asynchronous and synchronous simulation mode.

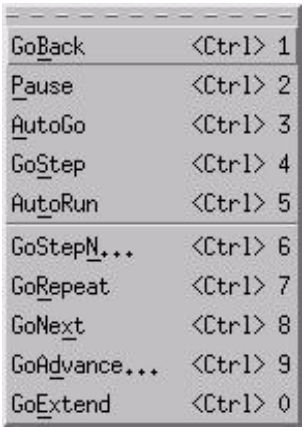
	Asynchronous Simulation	Synchronous Simulation
GoStep	Runs one step and consumes no time.	Runs one step and consumes one time unit.
GoStepN	Runs a specified number (N) of steps.	Runs a specified number (N) of steps.
GoRepeat	Performs a superstep and consumes no time.	Performs a superstep and consumes the amount of time equal to the number of steps taken during the superstep.
GoNext	Advances time to the next scheduled action or timeout event. Then performs a GoRepeat.	Advances time to the next scheduled action or timeout event. The number of steps taken are equal to the number of time units.
GoExtend	Performs a GoRepeat. If a superstep can't be taken, then it performs a GoNext then a GoRepeat.	Performs a GoRepeat. If a superstep cannot be taken, then it performs a GoNext then a GoRepeat.
GoAdvance	Allows you to advance the time by specifying the increment of time. A GoRepeat is taken.	Allows you to advance the time by specifying the increment of time. The number of steps are equal to the number of time units advanced.
AutoGo	The **AutoGo** command is used to perform a **GoStep** in an unstable status otherwise it performs a **GoNext**.	The **AutoGo** command is used to perform a **GoStep** in an unstable status otherwise it performs a **GoNext**.
GoBack	Undo last Go command	Undo last Go command

A Go command may be entered whenever the simulation is waiting for input. There are three exceptions:

- When the simulation is in the Autorun or in the Batch mode;
- When the simulation is in an unresolved nondeterministic situation. In this case, the user must choose a solution before proceeding with simulation.
- When the simulation has reached a termination connector. In this case, the only relevant commands are **Quit** and **Restart**.

The Go Menu

The following figure illustrates the **Go Menu**. Some important notes about using the **Go** commands are listed below.



- The commands **GoRepeat, GoExtend**, **GoNext** and **GoAdvance** may cause the simulation to enter an infinite loop. The simulation, of course, does not loop indefinitely. Instead it loops until it reaches the Steps per Go limit. When this maximum number of steps allowed per Go is reached, the Simulation Tool issues the message:

  ```
  Reached MAX NUMBER OF STEPS PER GO limit (see parameter under OPTIONS).
  ```

  Note: The Steps Per Go parameter is set by selecting **Options > Execution Options** from the **Simulation Execution** menu or the **Profile Editor.**

- The simulation then continues as if the Go command had finished normally.
- **GoAdvance** requires a time parameter. This parameter is entered in the **GoAdvance** dialog box that is displayed when the command is selected from the **Go** menu.
- **GoNext** is only relevant when timeout events and/or scheduled actions are due to occur. When no items are scheduled or when an item is due in zero time, **GoNext** has no effect on the simulation.
- **AutoGo** runs a **GoStep** in an unstable status otherwise it performs a **GoNext**.

Pausing Execution

To pause a running simulation, select **Go > Pause** or click  .

Observing The System's Behavior

When a simulation step is taken, the system status may change. This section discusses how to display the simulation output which illustrates the new status and recent changes made. Change in status is shown graphically through chart animation and textually through the **Show** commands.

Graphic Animation Display

When the simulation is connected to a graphic editor, the status information is displayed graphically on the workstation. The state configuration and recent transitions are highlighted in the Statechart Graphic Editor and the active activities are highlighted in the Activity-chart Graphic Editor. The following table summarizes the graphical changes made in the charts.

Color	Assigned to	Color
Purple (default)	Transition taken in the last step. Basic states the system is in or has entered in the last step.	Purple (default)
Violet (default)	Basic states left in the last step.	Violet (default)
Orange (default)	Transitions other than those listed above.	Orange (default)
Green (default)	States other than those listed above.	Green (default)

The Simulation Tool changes the color of the lowest level state visible. Also, viewing commands such as *dive* and *surface* have no effect on the simulation results.

For information on setting preferences, refer to the *Rational Statemate User Guide*.

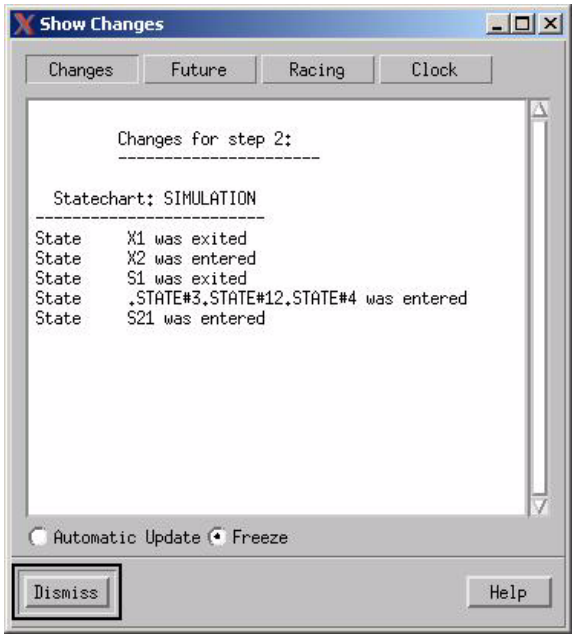
Show Command

The **Analyze > Show** command displays the **Show** dialog box that allows you to view Changes, Future, Racing and Clock. The Show commands are used to display a textual description of changes in the model. They display changes for both graphical and non-graphical elements.

Show Changes

The **Show Changes** command is used to display the system changes since the last simulation step. This includes all changes in the system status and manual changes that occurred during the last **Go** command.

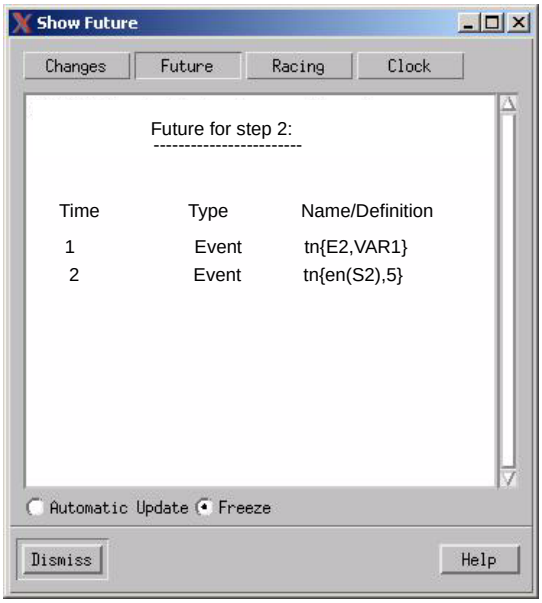
- **Automatic Update** changes in the model are automatically updated and can be viewed.
- **Freeze** keeps the current information in the display. Changes in the model are not automatically updated.



Show Future

The **Show Future** command displays all scheduled actions and timeout events due to occur. Also, Simulation Control Program breakpoints triggered by the EVERY clause are listed.

- **Time** is the amount of time (global clock units) until the scheduling of the item (event, action, EVERY clause). If the value is zero, the item is activated just prior to the next step.
- **Type** is the type of scheduled item (event, action, EVERY clause)
- **Name/Definition** is the name of the scheduled item or, for a nameless item, an expression that defines it.



Show Racing

The **Show Racing** dialog box displays racing problems. The Simulation Tool notifies about two types of racing situations:

- Read/Write Racing
- Write/Write Racing

The racing analysis is performed only when the appropriate options are set in the **Execution Parameter** dialog box. When a racing situation occurs, the tool resolves it by randomly choosing one of the possible outcomes. The following message is displayed:

Racing problems encountered

Show Clock

The **Show Clock** command displays information on:

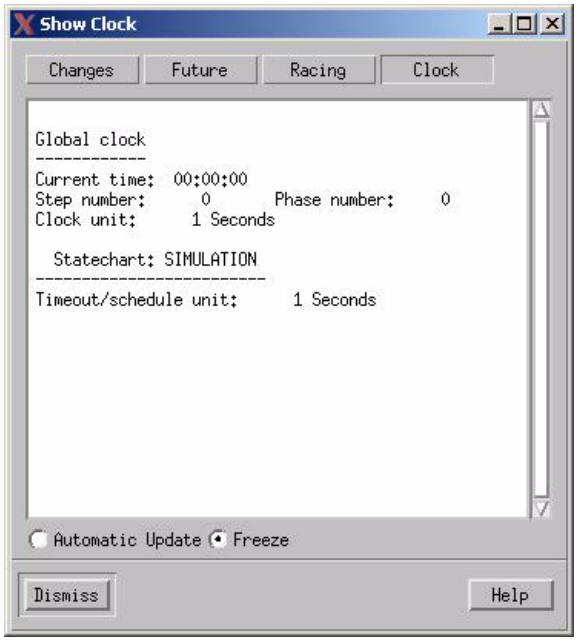
- Global Clocks that show the passage of time for the entire system.
- Clocks for each of the system's components, i.e., for each activity or for each Statechart that is not connected to the activity's control.

For Global Clocks, the following is displayed:

- **Current Time** shown in the standard format HH:MM:SS.
- **Step Number** shows the total number of steps taken from the start of the simulation.
- **Phase Number** is the number of steps taken at the current time.
- **Clock Unit** is the unit of global clock specified in Time Settings dialog box of the Profile Editor.

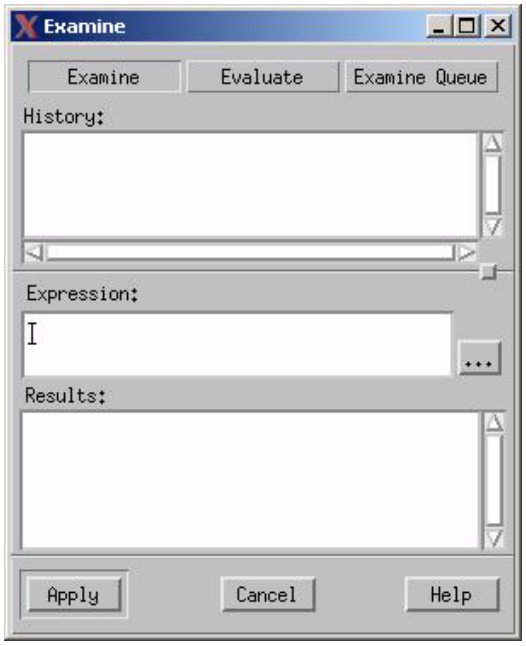
For each of the system components, these are shown:

- **Clock Unit** is relevant to synchronous time model only; different units can be specified for different components.
- **Timeout/Schedule Unit** is used to measure duration of timeouts and scheduled actions; relevant to both time models.



Examine

The **Examine** command (**Analyze > Examine**) allows you to examine Element Values, Queues and Expressions.



- **Examine** displays the value of an object.
- **Evaluate** displays the value of an expression.
- **Examine Queue** displays the content of the queue. This includes the queue's length and contents.
- **History field** saves a record of elements that were examined and evaluated. This can be used to retrieve elements for re-examination by selecting them using the left mouse button and **Apply**.
- **Ellipse button s**tarts the **Select Element** browser. 
- **Expression** displays the expression or name of the element that is being evaluated or examined.
- **Results** displays the results of Examine, Evaluate and Examine Queue.

Non-determinism

When the Simulation Tool encounters a nondeterministic situation, a Non-determinism dialog box opens.

The Non-determinism dialog box displays the name of the chart that the non-determinism occurred in along with a number of possible solutions. You are able to toggle all possible continuations by using the up and down arrow keys. There is also a field showing the name of a Statechart in which the non-determinism occurred. This helps the you to know which chart should be examined in order to select a desired continuation. Once an acceptable solution is chosen, the **OK** button confirms this choice and you can proceed with simulation.

Note

If the non-determinism is an undesirable behavior, you can choose to modify the chart(s) and rebuild simulation.

Panels in Simulation

Mock-up panels provide a clear and visual interface to the simulated mode and allow you to easily control the system's behavior. Panels can be built of

- Input and out interactors
- Graphical shapes drawn by you

Each of these graphical objects can be bound to an element in the model, for example, an event or state. These bindings allow you to drive the simulation by entering input values and to monitor the execution by observing the outputs.

Multiple panels can be attached to the same simulation, each presenting either:

- A group of logically related elements in the interface of the simulated system
- Its internal elements

You can display panels on different terminals and each single panel can be simultaneously displayed on several terminals. This provides a realistic effect and allows you to work with a system that includes multiple components.

You can combine panels with any other mechanisms supporting simulation input and output such as:

- Monitors
- Graphic editors for charts
- Do Actions and Examine commands
- Simulation Control Language programs

All these facilities provide a consistent picture of the current status of the model at any time during simulation.

The Panels tool allows you to attach panels to your simulation. The various characteristics of the panel can be saved in a Panel Profile. The saved profile can then be re-used in other simulation sessions.

Panels allow you to rapidly create a mock-up of the man-machine interface. This interface can be connected to simulation for testing purposes. Panels also can be used to aid in automating the simulation environment.

Defining and Editing Panel Profiles

This section describes how to add, edit or delete a panel profile from the Simulation Profile.

Adding a Panel to the Profile

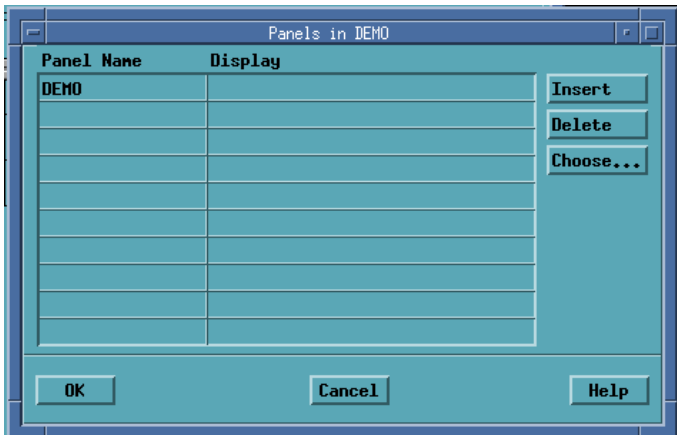
1. In the Workarea Browser select a panel.
2. Select **Edit > Add/Edit Panel** or select the **Add Selected Panel** icon from the **Profile Editor**. The panel is added to the profile and its name is displayed in the Profile's Scope Definition.

Editing a Panel in the Profile

For each panel in the profile, it is possible to specify on which display terminal it should be shown when simulating with this profile. By default, a panel is displayed on the same terminal on which all Rational Statemate windows appear. You may change this and cause the panel to be displayed an another terminal (or several terminals simultaneously).

To edit a panel in the Profile:

1. Select **Edit > Add/Edit Panels** from the **Profile Editor.** The **Panels in Profile** dialog box opens.



2. Specify names of terminals in which you want the panel to be displayed. Leave the field blank to display the panel on the default terminal.
3. Click **OK**.

Deleting a Panel from the Profile

1. Select a panel name in the Scope Definition area of the Profile Editor.
2. Select **Edit > Remove from Scope** or click the **Remove from Scope** icon.The selected panel is removed from the scope.

Font Appearances in Simulation Panels

Use the following procedure to correct erroneous behavior in the text-to-graphics ratio. This ratio is not kept when the same chart or panel is moved across different screen resolutions.

Set the environment variable:

```
STMM_ENABLE_FONTSIZE_CORRECTION
```

For example, on Windows systems, include the following line in the `run_stmm.bat` file:

```
SET STMM_ENABLE_FONTSIZE_CORRECTION-ON
```

Note

Exceed users see some differenced in the text-to-graphics ratio due to this change. However, XVision users may not notice any differences.

To move various charts and panels across various screen resolutions using XVision:

1. Select **Properties > Devices > Video** in the Monitor Resolution window of XVision.
2. Set the **DPI** to the appropriate number. The default value is 96. However, this is correct only for a very specific screen resolution.

Waveforms in Simulation

The *Waveform* tool has two modes; one that allows you to communicate with simulation and display changes as they occur and one that can be used for post-run analysis of traces produced by simulation.

The various characteristics of the Waveform window can be saved in a *Waveform Profile* during simulation or after simulation. The saved profile can then be re-used in other simulation sessions, or in the off-line mode.

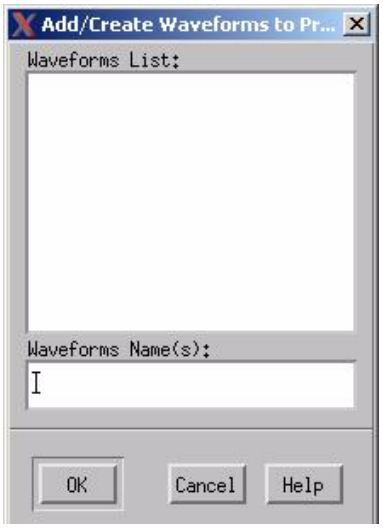
On-Line Mode of Waveforms

In the *on-line* mode, the Waveform tool runs as a process that communicates with simulation and displays the changes as they occur.

Setting Waveforms to be Displayed in Simulation

A waveform can be added to the Simulation Profile as follows:

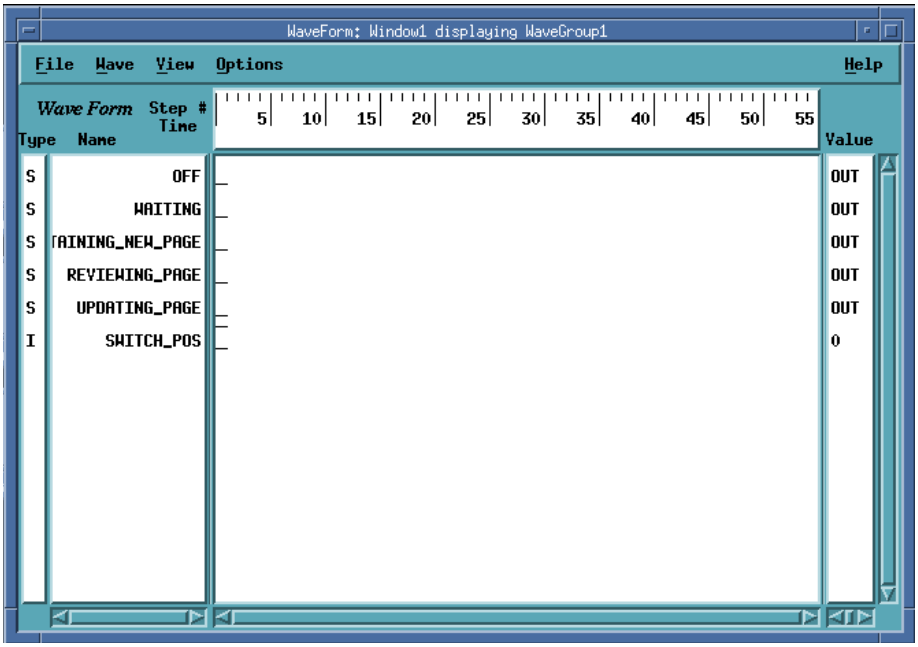
1. Click  or select **Edit > Add/Create Waveform**. The New **Waveform** dialog box opens.



2. Enter the name of your waveform into the **Waveform Name** text box, select **OK**. The new Waveform is added to your Scope Definition in the Profile Editor.

Activating Waveforms During a Simulation Session

A Waveform can also be activated directly during a simulation session by selecting **Displays > Waveforms**. This opens the **Waveforms in Workarea** dialog box. Select the needed names and click **OK** for activation. To start a new form, click **OK**; you are prompted to enter a new Waveform name. In this case, an empty Waveform window is created. Use the **Waveform** tool facilities to select the elements to be displayed.



Checking Waveform Elements

When a Waveform Profile is activated, the tool checks for the correctness of elements referenced in the profile. After performing this check, the tool displays a summary of the errors. Following are some of the recognized errors:

- The element is not unique in the scope
- There is a mismatch of element types
- An element does not exist in the scope
- The index of an array component is outside of the bounds of the array

Erroneous elements in the Waveform Profile are ignored and therefore are not displayed.

Unresolved Data-Items in the Scope

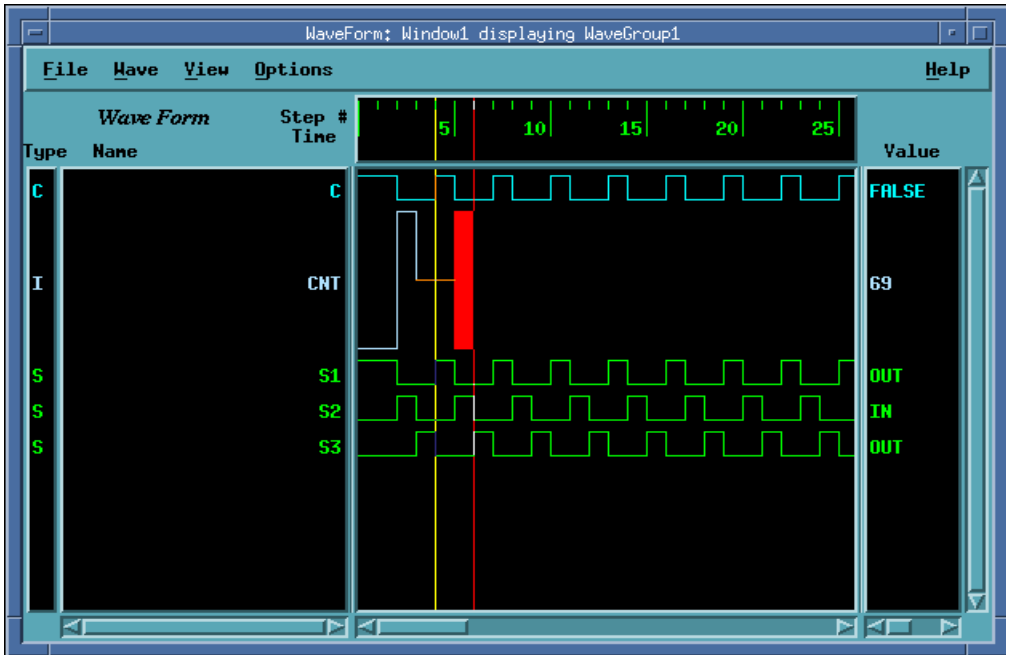
Unresolved data-items in the scope are treated by the Waveform tool as an integer. An appropriate message is provided when this situation is encountered.

If the data-item is actually used in the scope in a different way (for example, as a string) then the Simulation tool issues an error message the first time the contradiction is discovered during run-time.

Displaying Values in Waveform

When activated, the *Waveform* displays the current values of selected elements, the current time and current step number. When a new element is added to the Waveform, its current value is immediately displayed.

As simulation continues to run, the waveform displays a full history of element changes. To view the value of the elements, click at the desired point of the Step/Time scale. The elements at that point are displayed in the Value area of the waveform



Off-Line Mode of Waveforms

In off-line mode the tool is used for post-run analysis of traces produced by simulation.

Trace Files Menu

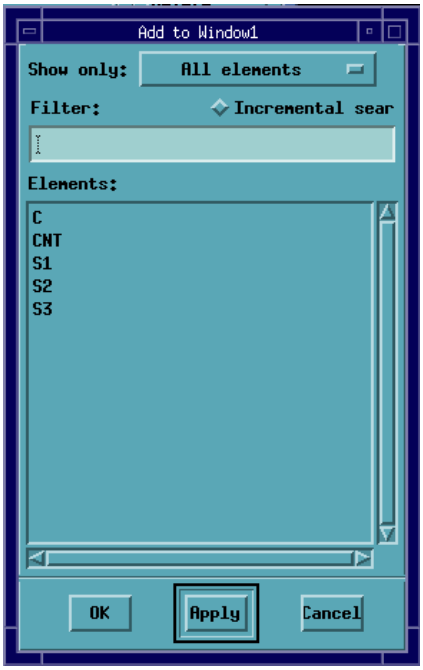
Select **File > Simulation File Management > Trace File Management** to name a trace to be analyzed.

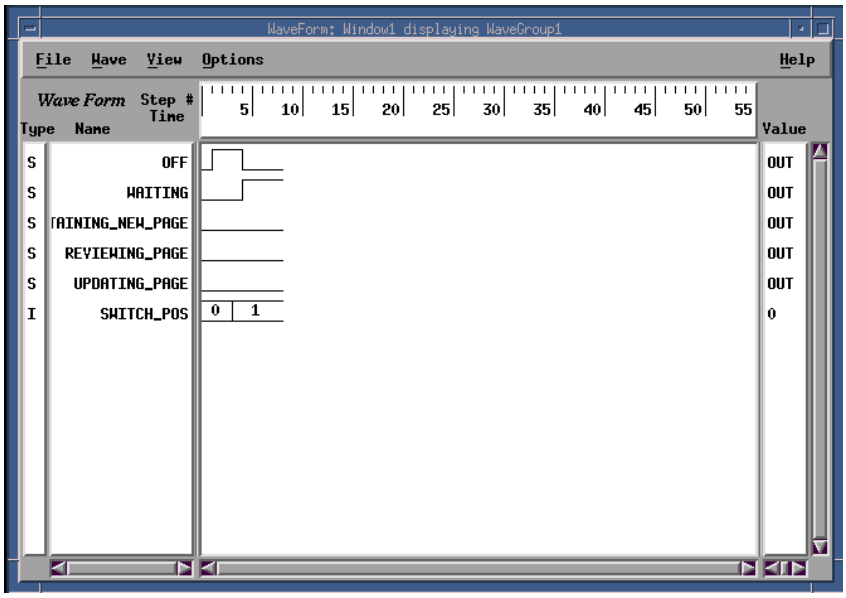
No Waveform in the Workarea

1. Select **Files > Simulation File Management > Trace Files**. The **Trace File Management** dialog box opens.

2. Select **Waveforms**. The **Select Waveform Profiles** dialog box opens.

3. Select **OK**. The **Add to Waveform** dialog and a **Waveform** window appears.





4. Select the elements to be viewed from the *Show* only selection. The elements appear in as a waveform in the Waveform window.

Waveform Profiles in the Workarea

1. Select **Files > Simulation File Management > Trace Files.** The Trace TFile Management dialog box opens.
2. Select **Waveforms**. The Select Waveform Profiles dialog box opens.
3. Highlight a trace file name for the **Files** list.
4. Select **Waveform**. The Select Waveform Profiles dialog box opens.

At this point you have two choices:

- Highlight a waveform profile from the **Profiles** list and click **OK.** The waveform from the selected waveform appears.
- Or, do not highlight a waveform profile and select **OK**. A new waveform window opens.

Waveform Profiles as Configuration Items

Waveform profiles can be saved as configuration items. These files are ASCII files with an extension.`wpf` that are stored in:

- **WorkArea**: `work_area_directory/ana`
- **DataBank**: `project_bank_directory/ana`

Use-Case Diagrams in Simulation

The Simulation supports a Use-Case Diagram (UCD) to the Simulation scope:

- When a UCD is added, all linked Sequence Diagrams (SDs) and statecharts are added with it.
- All the SDs and SD partition lines (scenarios) are listed in the Scenario Animation Control dialog box. The dialog box contains a table with the following fields:
  - **Animate (Yes/No)** - controls the painting of the specific SD partition lines and messages.
  - **Name**
  - **Activation Expression** - When you select **Yes** in the Animate field and the activation expression is not empty, the painting of the specific SD partition line and messages starts only after the expression defined in this field evaluates to TRUE.

Animation of Sequence Design

The Simulation animates the scenarios defined in the Scenario Animation Control table to show the scenarios propagation trace, as well as the statecharts.

Note

- Only a single instance of tan SD and an SD partition line can be animated simultaneously.
- The timing constraint, referenced SD, and SD scope constructs are ignored by the animation.

Recording a Sequence Diagram

To record a sequence diagram that includes the series of events that occurs during a simulation session:

1. Select **Options > Sequence Diagram Generation.**
2. Select the **Generated Sequence Diagram** check box.

The generated sequence diagram includes two lifelines, User and System, and the messages drawn between them. Internal message are displayed as messages-to-self.

Monitors in Simulation

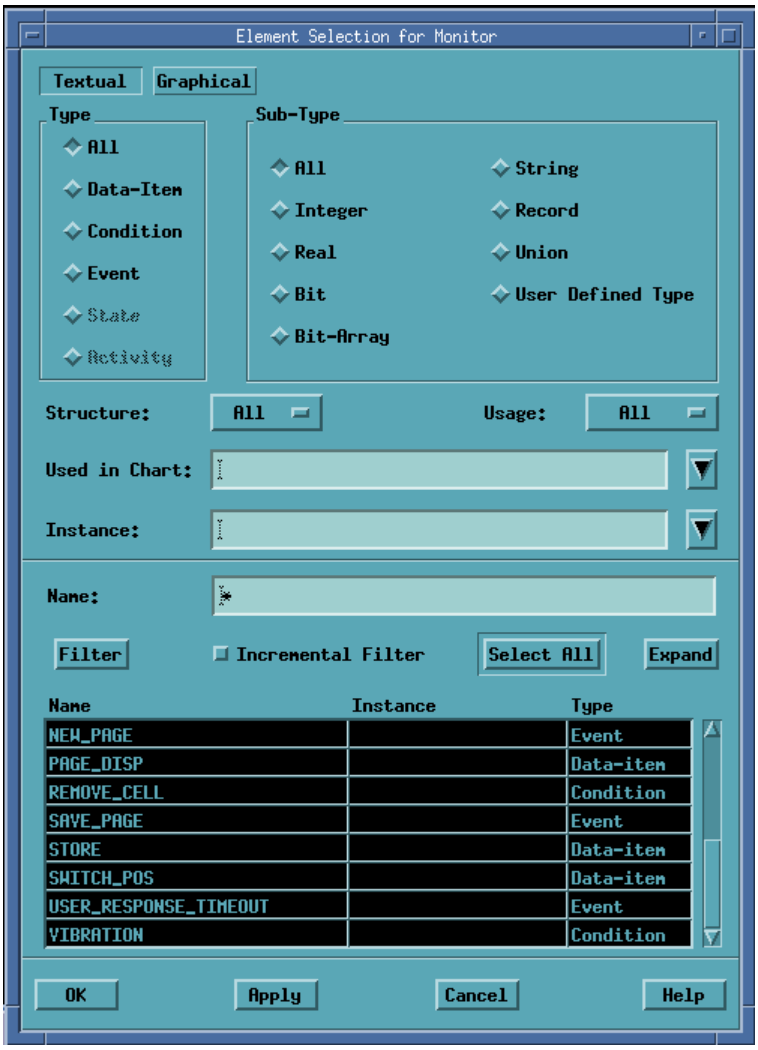
The Monitor Tool is a simulation debugging aid. It provides the user with a tabular display of textual and/or graphical element status during simulation. The Monitor can be used as an output device to display element status and /or an input device that accepts input stimuli during simulation. The various characteristics of a Monitor window can be saved in a Simulation Profile. This allows for re-usability of the Monitor in other simulation sessions.

Adding Monitors to the Profile

This section describes how to add a monitor to your Simulation Scope.

1. Click on the **New/Edit Monitors** icon  or select **Edit > Monitors.**.The **New Monitor** dialog box opens.

2. Enter a **name** for your monitor and select **OK**. The **Simulation Monitor** browser opens.
3. Select **Edit > Add**. The **Element Selection** dialog box opens.



The Element Selection for Monitor browser is used to select elements to show in the Monitor Window. This is done by:

- Creating a list of elements of the needed type and subtype.
- Selecting from the list the elements you want to view in the Monitor.

A description of each selection on this dialog box follows:

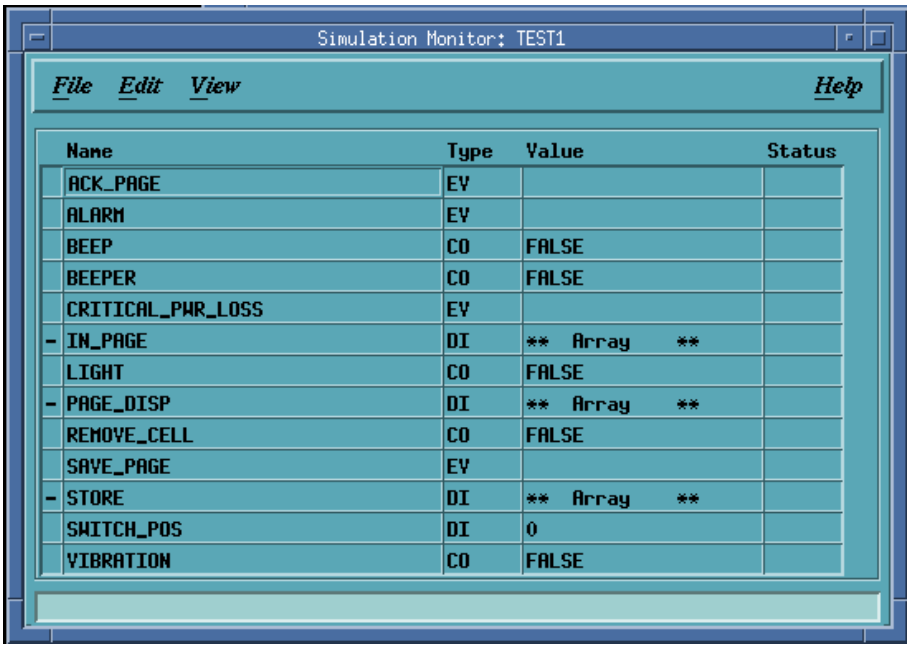
- **Primary Selection Area** – Select one of two element types in the top two buttons on this dialog box (the default is Textual).
- **Type** – Select the type of elements to be included in your Monitor. Textual types are All, Data-item, Condition, and Event. Graphical types are All, State, and Activity.

- **Sub-Type** – Used to further define a type. For example, a data-item can be defined as a real, integer, bit, etc.
- **Structure** – Allows you to select a structure type for textual elements (Single, Array, Queue or All).
- **Usage** – Allows you to select textual elements based on how they are used (All, Variable, Compound, Alias, Constant).
- **Used in Chart** – Used to select elements based on the charts in which they are used.
- **Instance** – Used to select elements based on the generic instances in which they are used.
- **Name** – Used to select elements based on their name. The asterisk (*) can be used as a wildcard character for part of the name or to select all element names.
- **Filter Button** – Used to generate a list of elements based on selected type, structure and usage.

  Note: The names that are being compared are the unique names of the elements that may be prefixed with Chart Names, Instance Names and Long Format Graphical Element Names. It is advisable to prefix any name search with the asterisk (*) wildcard character.

- **Incremental Filter** – Allows you to do an incremental search by entering a character string in the name text box.
- **Select All** – Selects all the elements in the pending list.
- **Expand** – Allows you to view structured type elements (arrays, records, unions) in more detail and to add to the Monitor only desired components

4. Make your selections within the **Type** listing. Appropriate selections are displayed for **Sub-Type**, **Structure** and **Usage** listings.
5. Make your selections within the **Sub-Type, Structure** and **Usage** listings (if necessary).
6. Specify a chart name in **Used in Chart**, if needed.
7. Provide a pattern for the name of elements to be filtered.
8. Select the **Filter** button. A list of elements appears.
9. Select the elements you want added to your monitor. Select **OK**. All elements selected from the **Element Selection Monitor** dialog box are added to the Monitor window. Click **Apply** to add elements to the Monitor list and retain the browser. Click **OK** to add the elements and dismisses the browser.



Simulation Monitor Fields

- **Name** – This field shows the element's name.
- **Type** – This field shows the element type:
  - **DI** – Data-item
  - **CO** – Condition
  - **EV** – Event
  - **ST** – State
  - **AC** – Activity
- **Value** – This field refers to the current value of the element. You can apply stimulus to the model by modifying the *Value* field. For primitive textual elements and for activities, a new value of element can also be entered into this field. Any error in the entered value (wrong type, etc.) causes an appropriate message and the current value remains intact.

  Note: For string Data-items, the value must be enclosed in quotes. Generation of an Event or changing a Condition can be done by clicking in the field Value with the left mouse button.

- **Status Field**
  - **For textual elements**: this field indicates if an element is read, written, or changed (rd, wr, ch).
  - **For States**: indicates whether a state is entered or exited (en, ex)
    - **For Activities**: indicates if an activity is started or stopped (st, sp).
- **Mode Field -** This field displays the mode for textual data-elements (data-items, conditions, events). The mode of a data-element is based on the graphic flow-lines in the top level Activity chart of the simulation scope. Therefore, the values in this column do not change throughout the execution of the Simulation.
  - To display this field, select **View > Show Mode**.
  - Values are: In, Out, In-Out, Local, Constant

Shared Monitor

A Simulation monitor can be tagged as a shared monitor. As a result, the monitor is saved separately from the Simulation profile. A shared Monitor can be shared between different Simulation profiles and workareas, and can be checked into and out from the Databank.

To turn a monitor (internal monitor) to a shared monitor, right-click on the monitor name in the Simulation main window and select **Make Shared Monitor**.

File Menu

Use the selections in this menu to manage the monitor files.

- **Save** – Saves the current monitor in your workarea for this project.
- **Exit** – Exits the Simulation Monitor and closes any associated windows that you may have left open.

Edit Menu

Use the selections in this menu to build a monitor.

- **Add** starts the Element Selection browser. The browser is used to select elements to show in the Monitor Window.
- **Remove** is used to remove unwanted elements from the Monitor window. Elements are removed by highlighting them in the Monitor window then selecting the Remove command.
- **Move** is used to move the position of an element in the Simulation Monitor. To move an element:

  a. Highlight the **element** to be moved.

  b. Select the **Move** command.

  c. Click on the location you want the element moved. The element is moved to the new location.

  Note: You can move either a single element or a group of selected elements.

View Menu

Use the selections in this menu to change how a monitor is displayed. The `Value Format` command allows you to change the format in the integer and bit-array data-items displayed in the Monitor. An element's value format can be changed as follows:

1. Select an **element** from the **Simulation Monitor** window.
2. Select **View > Value Format.** The **Formatted Value** dialog box opens.
3. Select the appropriate **Integer** and **Bit-Array** format.
4. Click **OK**. The value of the element is reflected in the Monitor window in the new format.
   - **Full Name** displays the full name of the selected element. This option is useful for elements with long names, for example elements in generic instances.
   - **Sort by Name** sorts all the elements in the current Monitor window alphabetically by name.
   - **Sort by Type** sorts all the elements in the current Monitor window by type, and within each type, in alphabetical order.
   - **Sort by Relevant** sorts all elements according their relevancy for the next simulation step to be taken. An element is relevant if it affects a trigger of a transition, static reaction or a mini-spec in the current model status.
5. Select **File > Save**. The **Simulation Monitor** is saved and added to the Profile.

The Microdebugger Tool

To run the microdebugger tool, you must first set breakpoints within the graphical and textual procedures on your model. Breakpoints can be set to occur upon entering a procedure. When the simulation tool reaches a breakpoint the microdebugger tool is executed. The Simulation microstep debugger allows microstepping through the implementation of activity mini-steps, state static-reactions, and action language actions, in addition to subroutine. The microstep debugger monitor is not available with these items because values are updated on a step boundary, not micro-step boundary. It is recommended that you use regular simulation monitors.

The Simulation microstep debugger also supports the inspection of context variables during debugging.

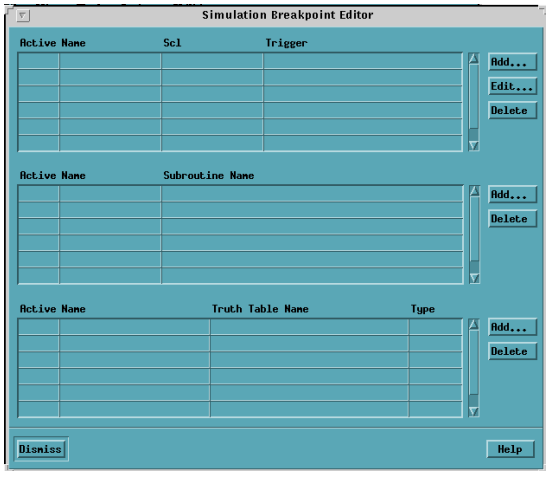
In the following steps, the process of setting breakpoints and debugging graphical/textual procedures is described.

Defining a Breakpoint in a Subroutine

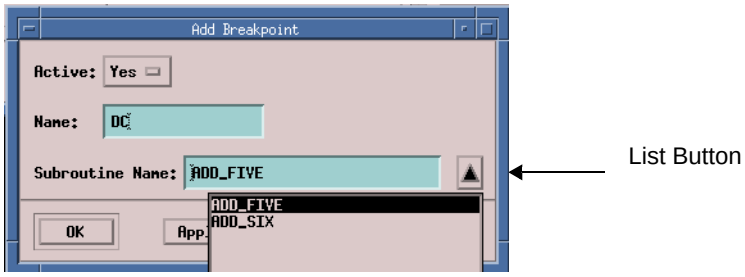
To add a breakpoint to a subroutine within a Rational Statemate model.

1. Select Actions > Breakpoints from the Profile Editor. The Breakpoint Editor appears.

Sub
rout



2. Click **Add** from the Subroutine list. The **Add Breakpoint** dialog box opens.



3. Enter the **name** for the breakpoint in the name field.

   Note: In models containing multiple breakpoints, each name must be unique.

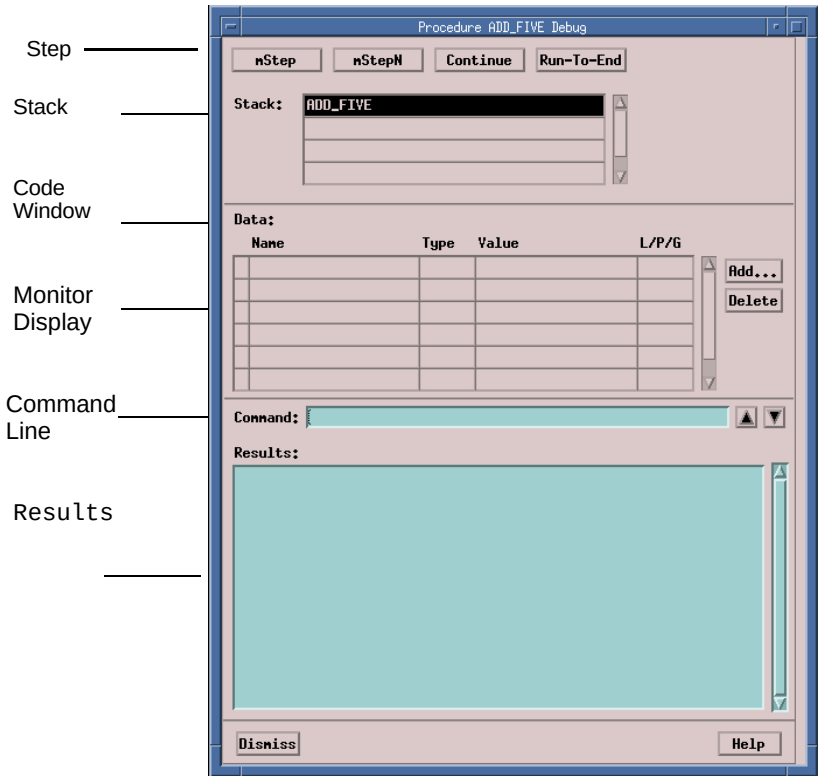
4. Click the **List** button. A selection menu listing all defined subroutines appears.

5. Highlight the subroutine that contains the breakpoint, then click **Apply**. A breakpoint is set to the selected subroutine.

Debugging a Textual and Graphical Procedure

This section provides information to setup a debugging session for a textual or graphical procedure. The microdebugger tool is start whenever a breakpoint is reached during the execution of a subroutine.

The debugger tool allows you to select elements within your model and monitor the execution of microsteps as you simulate. When monitoring textual procedures, code is viewable within the microdebugger tool. For graphical procedures, a read-only Graphic Editor opens, allowing you to view the execution of microsteps graphically.

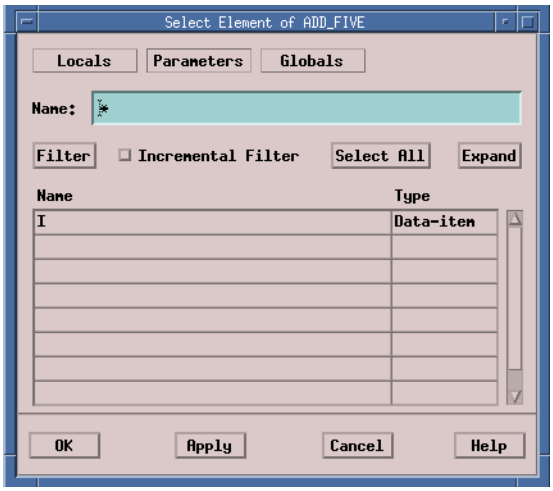


- **mStep** starts one microstep.
- **mStepN** runs a specified number of microsteps.
- **Continue** runs until the next breakpoint is reached or to the end of the current context.
- **Run to End** runs until the end of the current procedure.
- **Dismiss** closes the window.
- **Stack** shows the levels of procedures being executed. Procedure names are listed in tabular format. Double-clicking on any procedure name in the stack performs an up/down action.
- **Code window** displays the code within the current execution. The current execution line is highlighted.
- **Monitor Display** displays the status of selected elements.
- **Command Line** allows you to enter commands to examine data (or Do Actions). The results are displayed in the Results area.
- **Results** displays results for each action.
- **Dismiss** turns off the dialog.

Adding Elements

To aid you in debugging your model, you can add elements from your model to the Microdebugger Monitor. The Monitor provides a tabular display of elements status during simulation. You add elements by:

1. Click **Add**. The Select Element dialog box appears.



2. Select the **Parameters > Filters.** A list of elements appear in the **Name** and **Type** field.
3. Select the **element** you want to add to the monitor display of the Procedure dialog box.
4. Select **Apply > OK**. The selected elements are added to the monitor display of the Procedure dialog box.

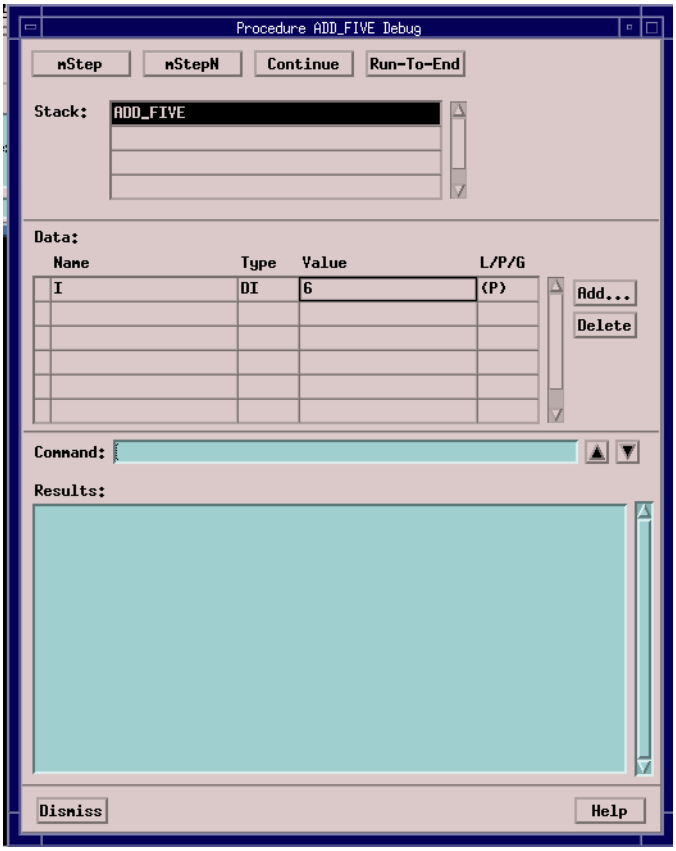
Simulating a Textual Procedure

After you have added elements to the Procedure dialog box, the procedure can be simulated. Each time a step is taken, changing values can be observed in the monitor. After the subroutine is completed, execution of the step continues.

Textual procedures started from subroutines that have been defined using code (i.e., C, Ada) cannot be simulated. This is only true for this instance. Textual procedures started from anywhere else can be simulated. The correct code is generated for Textual procedures started from anywhere.

Simulating a Graphical Procedure

After adding elements to the Graphical Procedure dialog box, you can simulate it. After the subroutine execution is completed, the execution of the step in the model continues. The simulation can be viewed via the Graphic Editor.



The step semantics for a Graphical Procedure are the same for a Textual procedure. The execution of the procedure appears atomic from the view of the model. From the view of the procedure being called, the execution acts like a GoRepeat.

When a procedure is called, it runs to completion before the model advances. At each occurrence of a procedure calling another procedure, the “caller” does not advance until the procedure being called returns.

If more than one procedure is called in the same compound action, they are treated concurrently. IN and INOUT parameters are read from the value at the beginning of the step. INOUT and OUT parameters should be written at the end of the step. Context variables can be passed as parameters in order to create sequentially.

For example, the following results in racing:

```
/init (MY_ARRAY);
  sort_incr (MY_ARRAY);
  sort_decr (MY_ARRAY)
```

In the next example, the final value of MY_ARRAY is sorted in decreasing order.

```
/init ($MY_ARRAY)
  sort_incr ($MY_ARRAY)
  sort_dcre ($MY_ARRAY)
  MY_ARRAY:=$MY_ARRAY
```

In the next example, the final value of I is incremented a single time from the value at the beginning of the step.

```
/my_incre(I);
 my_incre(I):
 my_incre(I)
```

In the next example, the final value of I is incremented three times from the value at the beginning of the step.

```
/$I:=I;
my_incre($I);
my_incre($I);
my_incre($I);
I:=$I
```

Multiple procedures started from the same compound action are debugged sequentially.

Graphical procedures must always run to completion when started, if the procedure reaches a stable situation (no more micro-steps can be taken with the current parameter, local, global values), a runtime error should be generated by the simulation. When this error condition is encountered, within simulation, the procedure immediately returns with the current global and parameter values and the simulation step is interrupted. At this point, you are prompted with an error message and is given the ability to continue the step and simulation.

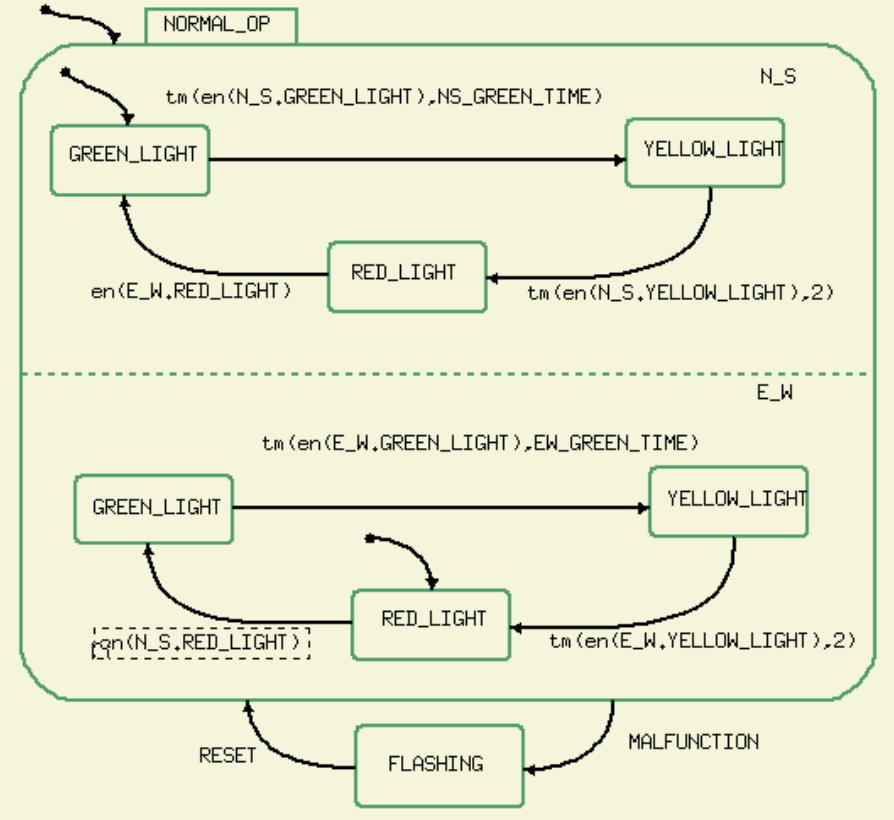
Interactive Simulation Example

The Traffic Light System

To illustrate some of the principles and commands discussed in this section and in **Model Execution: Concepts and Terms**, the following simple Traffic Light example has been devised. Enter the Statechart shown in the Traffic Light Example into your Rational Statemate system. After the Statechart is entered, you are instructed on how to interactively execute the commands shown in the scenarios. This provides you with both the simulation basics and command mechanics.

Description Of The Traffic Light System

- A traffic light system controls the intersection of two streets, one going north-south (N_S) and the other going east-west (E_W).
- The traffic lights remain green (in their respective directions) for a specified amount of time. The time east-west remains green may not be the same as that of north-south.
- The time the lights remain green can change according to traffic conditions.
- The lights can be disabled. In the event of electrical malfunction, the lights blink yellow in all four directions.



Simulating the Traffic Light in the Asynchronous Time Model

With the Statechart entered into your system, it's now time to observe the model's behavior using the Simulation Tool's interactive mode. Please execute each step on your workstation and observe the behavioral results.

Initiating the Simulation Tool

From the Statechart graphic editor, initiate the Simulation Execution tool by choosing **Tools** > **Simulation**. The **Simulation Execution** window appears.

Setting Some Time Parameters

As previously stated, the time each traffic light remains green is a variable. During this step, you set the east-west time for 15 seconds and the north-south time for 20.

1. Select **Action > Do Action**.
2. Enter the following In the Expression field:

```
ew_green_time:=15;ns_green_time:=20
```

3. Click **OK**.

Stage 1

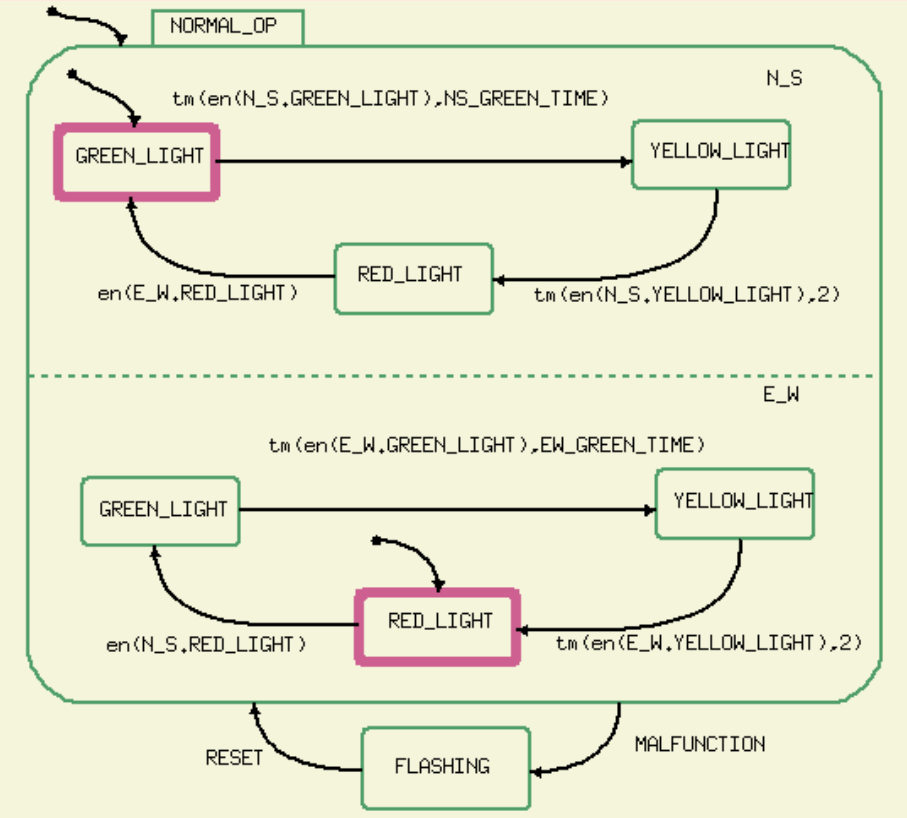
This stage begins the simulation of the Traffic Light statechart.

Select **Go > GoRepeat**.

The traffic light begins to operate. The north-south lights are green and the east-west lights are red.

Simulation Time: (see the status line of the **Simulation Execution** window) 0 seconds

The statechart enters its default states. **GoRepeat** advances the simulation to the next stable status, in this case the default entrances.

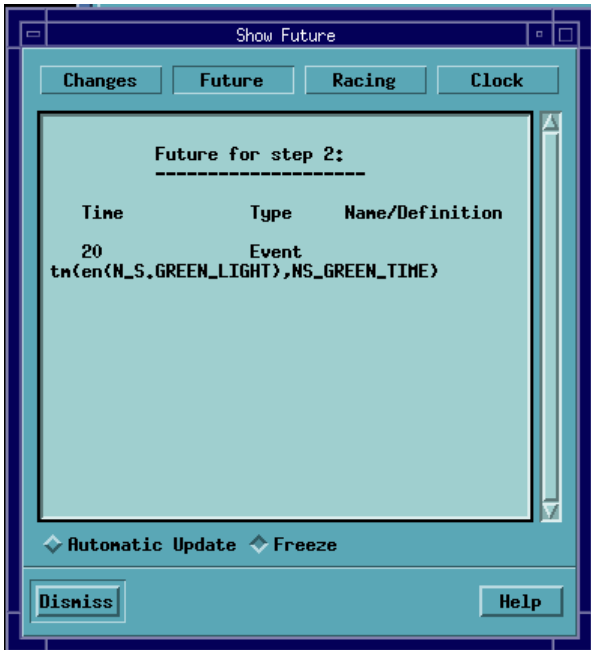


Interactive Simulation Stage 1

Stage 2

At this stage, you display the scheduled events and move to the point where the traffic lights first change.

1. Select **Analyze > Show**.
2. Click **Future** in the Display dialog box.



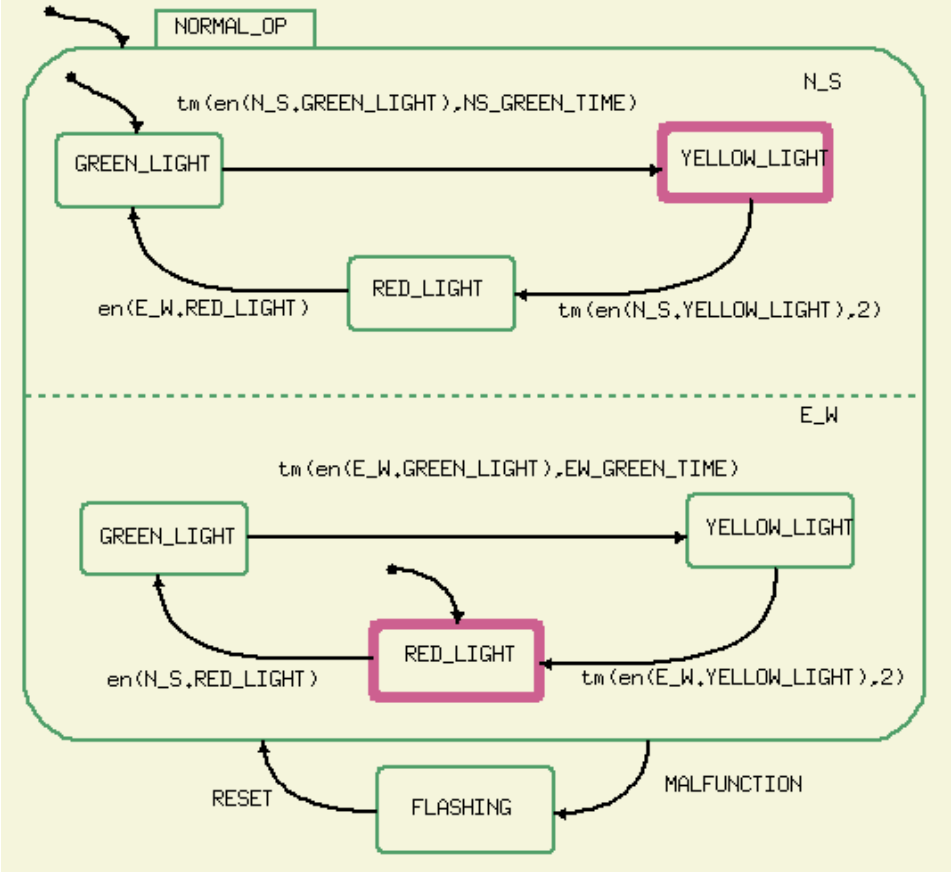
The north-south traffic lights turn yellow after 20 seconds. This indicates the effect of setting the **ns_green_time** parameter.

3. Select **Go > GoExtend.**

Simulation Time: 20 seconds.

GoExtend advances the simulation and increments the clock until the next change occurs in the system status - when the north-south lights turn yellow.

Observe that the system is now in states N_S.YELLOW_LIGHT and E_W.RED_LIGHT.



Interactive Simulation Stage 2

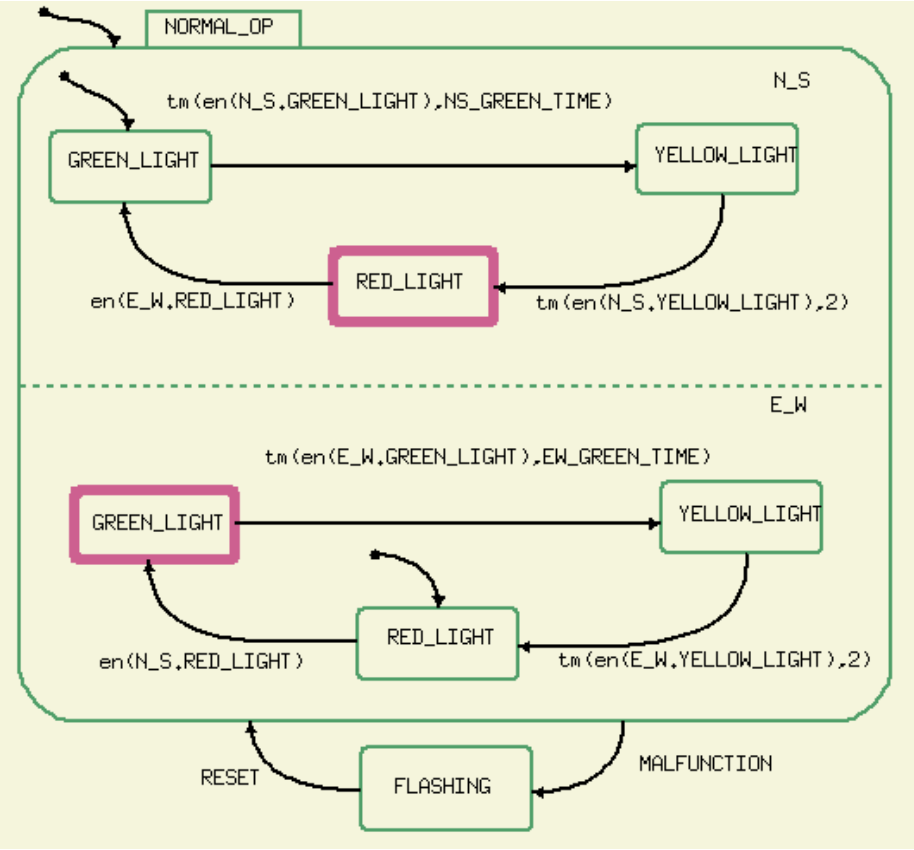
Stage 3

At this stage, **GoAdvance** is used to advance the time.

1. Select **Go > GoAdvance**.
2. Enter the value 2 and click **OK**.

   The system is now in states N_S.RED_LIGHT and E_W.GREEN_LIGHT.

   Observe that the simulation time is now 22 seconds.



Interactive Simulation Stage 3

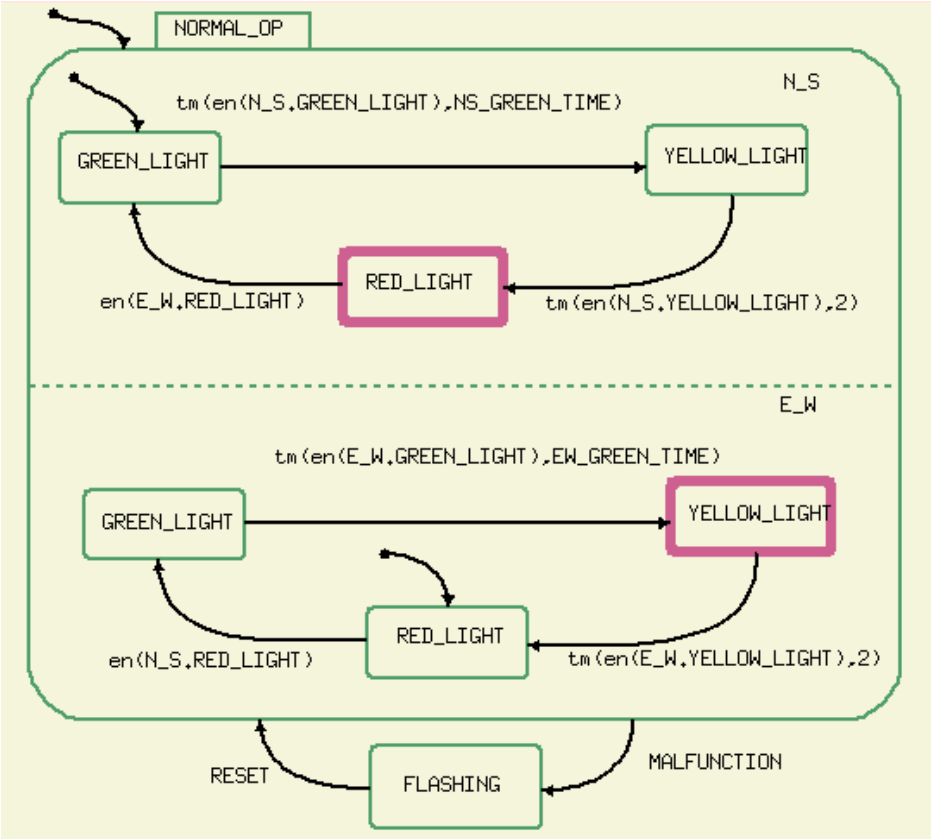
Stage 4

At this stage **GoAdvance** is used to increment time.

1. Select **Go > GoAdvance.**
2. Enter the value **16** and click **OK**.

   Simulation time: 38 seconds.

   Our design calls for the east-west light to remain green for 15 seconds. Since the clock has been advanced 16 seconds, the simulation informs you that there are reactions completed before the specified time. The east-west lights change from green to yellow and the clock is incremented the full 16 seconds.



Interactive Simulation Stage 4

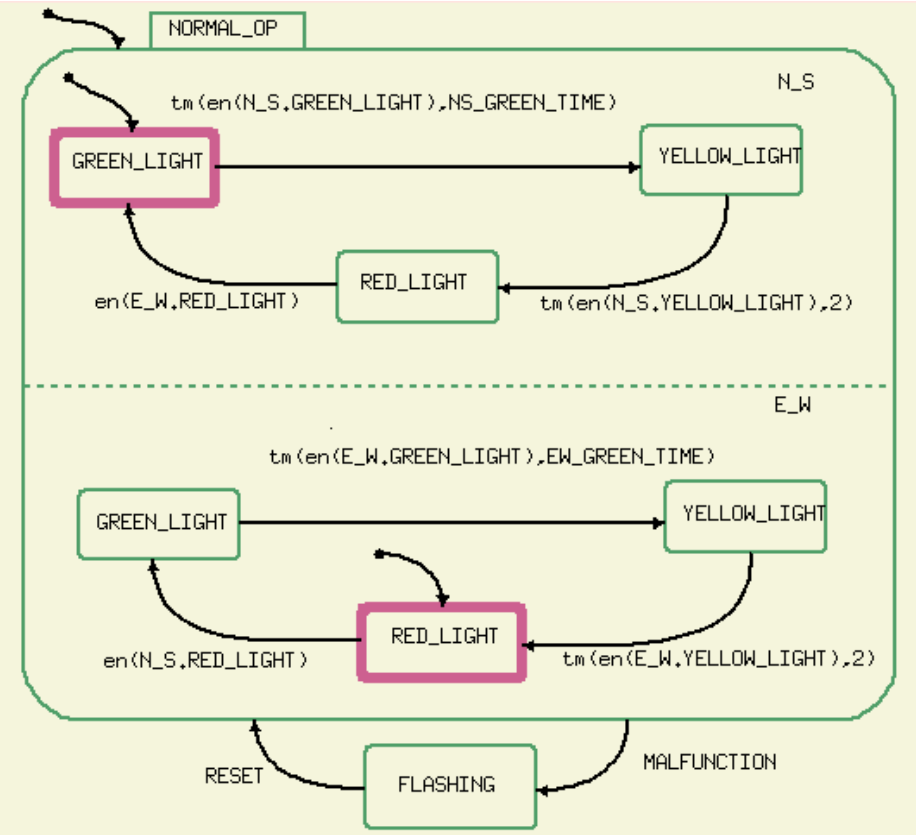
Stage 5

At this stage, the next change in the traffic light is reflected.

Select **Go > GoExtend.**

Simulation time: 39 seconds.

The traffic light system has completed one full cycle. The cycle takes 39 seconds (north-south stays green for 20 seconds, east-west stays green for 15 seconds and each yellow light stays for 2 seconds). The system appears to behave as expected,



Interactive Simulation Stage 5

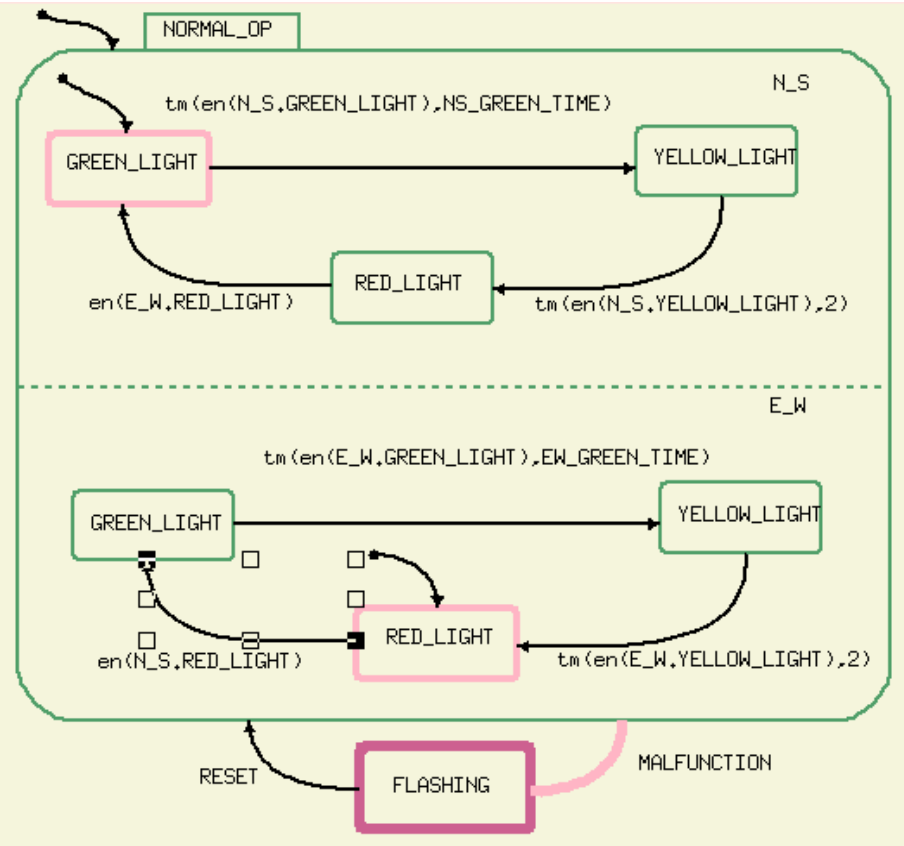
Stage 6

At this stage, an electrical malfunction is detected.

1. Select **Action > Do Action.** The **Do Action** dialog box opens.
2. Enter `malfunction` into the **Expression Field** and click **OK**.
3. Select **Go Step** from the Simulation window.

   Simulation time: 39 seconds.

   The traffic light stops its normal operation and moves to the **FLASHING** state. Note that no time has been incremented. When a malfunction occurs, the traffic light begins to flashing immediately.



Interactive Simulation Stage 6

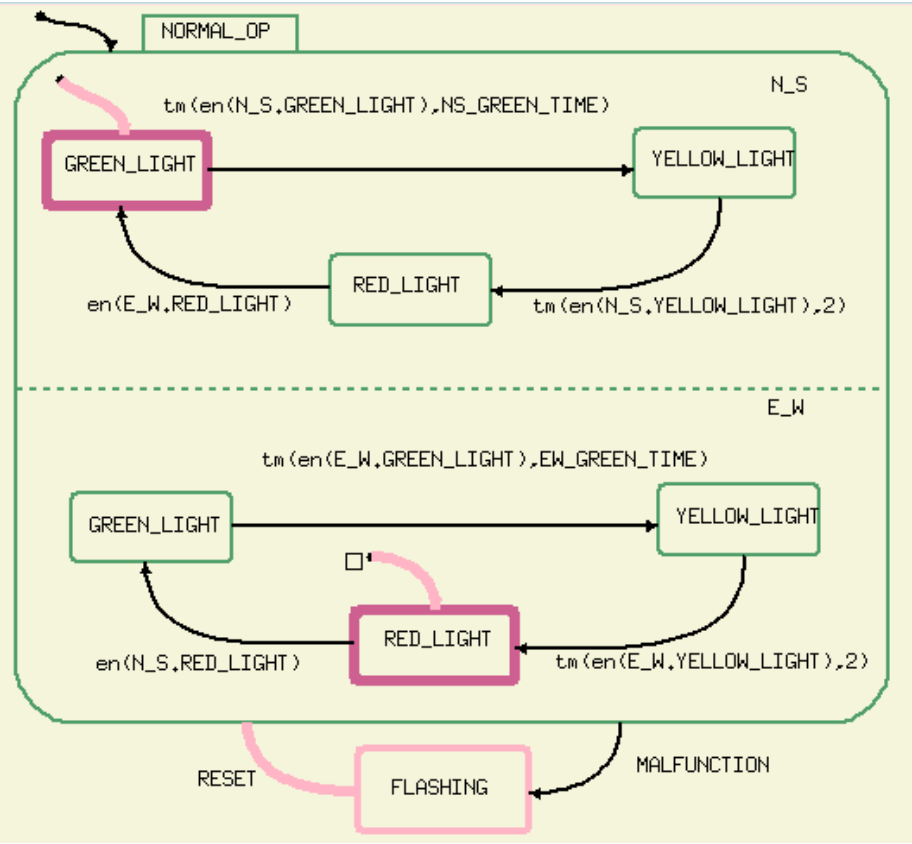
Stage 7

At this stage, the electrical malfunction is corrected.

1. Select **Action > Do Action**. The **Do Action** dialog box opens.
2. Enter **Reset** into the **Expression** field and click **OK**.
3. Select **Go > GoStep**.

   Simulation time: 39 seconds.

   Again, no time has elapsed. Note that generating a reset event returns our system to its original default states.



Interactive Simulation Stage 7

Stage 8

When the simulation scenario completes, exit Simulation Tool.

1. From the **File** menu, click **Exit**.
2. Click **OK** to confirm.

Some Variations to Consider

The example presented is quite simple and works properly. At this time, you may want to consider altering the model and its parameters to study the effects. The following suggestions are made:

- Add the capability for a policeman to manually change the traffic lights in time of heavy traffic or an accident. This may be accomplished by changing the label on the `N_S.GREEN_LIGHT to N_S.YELLOW_LIGHT` transition to:

  ```
  tm(en(N_S.GREEN_LIGHT),N_S_GREEN_TIME) or SWITCH
  ```

  where `SWITCH` is an event generated by the policeman.
- Note that during the malfunction, all lights begin flashing immediately. During this period, no time advances. And when the lights are repaired, the lights return to their default configuration. You may want to change the model to reflect more realistic traffic situations.
- These traffic lights perform the same cycle 24 hours a day. At night, it might be unrealistic to have the same light delays as appear during daylight hours.

Simulating the Traffic Light in the Synchronous Time Model

Using the same time setting as in the asynchronous example, follow the commands in the following table to execute the model with the synchronous time model. Please observe the time increments and stable and unstable configurations.

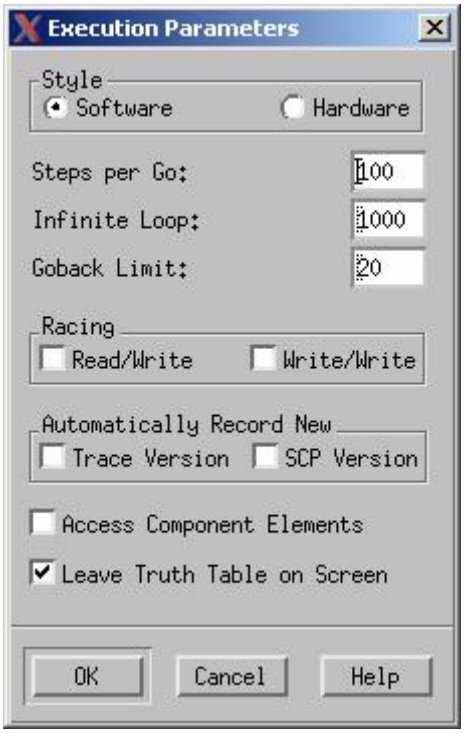
COMMAND	INITIAL CONFIGURATION	FINAL CONFIGURATION	TIME	COMMENTS
GoRepeat		n_s.green_light, e_w.red.light	1	A stable configuration is reached. Show Future displays next timeout in 20 clock units.
GoNext	n_s.green_light, e_w.red_light	n_s.green_light, e_w.red_light	20	Show Future indicates one unit until next timeout.
Go StepN(2)	n_s.green_light, e_w.red_light	n_s.yellow_light e_w.red.light	22	First Go Step makes transition. Second goes to stable configuration.
GoAdvance(3)	n_s.yellow_light e_w.red.light	n_s.red_light, e_w.green_light	25	Clock advanced 3 units and transitions are made based on timeout.
Go Step	n_s.red_light, e_w.green_light	n_s.red_light, e_w.green_light	26	Move to stable configuration.

Recording a Simulation Session

Setting the Simulation Parameters

The Simulation tool is controlled by a number of user-specified parameters. These parameters are set from the **Execution Parameters** dialog box. This dialog box is accessed as follows:

Select **Options > Execution Options** from the Simulation Profile window menu bar.. The **Execution Parameters** dialog box opens.



A description of each selection on the **Execution Parameters** dialog box follows.

- **Steps per Go** – Sets the maximum number of steps that can be performed when executing a `Go` command.

  When the phase limit is reached, the Simulation tool assumes an infinite loop and interrupts the execution of the `Go` command. At this point the `SIM>` prompt is displayed.

  If the `Go` was performed by a batch program (SCP), the predefined SCL variable `infinite_loop` is set to true and Set Interactive is automatically started if there is no defined breakpoint triggered by `infinite_loop`.

  ```
  Default value: 100

  SCL command: Set Infinite loop number
  ```

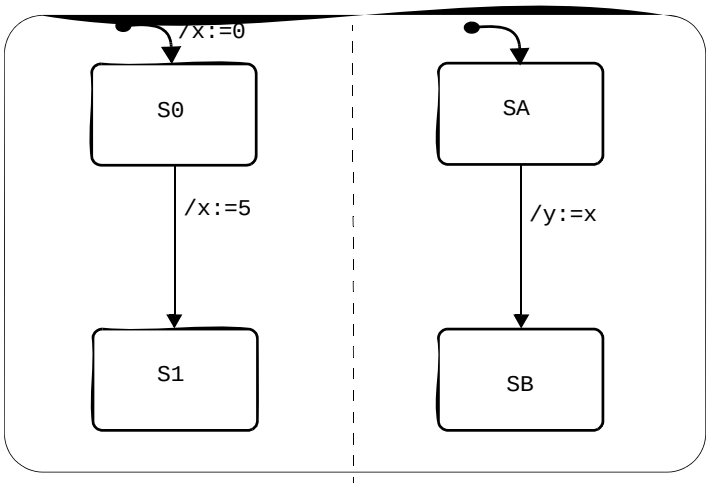
- **Infinite Loop** – For any `WHILE` loop executed in the simulation, Infinite Loop forces it to completion when the number of repetitions exceeds this parameter.Graphic procedures can have an infinite loop or a loop on a transition. This stops when the preset parameter is reached.
- **Goback Limit** – Determines the maximum number in succession the `GoBack` command may be used.

  ```
  Default value: 5

  SCL command: SET GO BACK number
  ```

- **Racing Read/Write** – Enables/disables the reporting of read/write racing conditions within/ between Statecharts. Messages appear on the workstation terminal.

  ```
  Default value: OFF

  SCL command: none
  ```

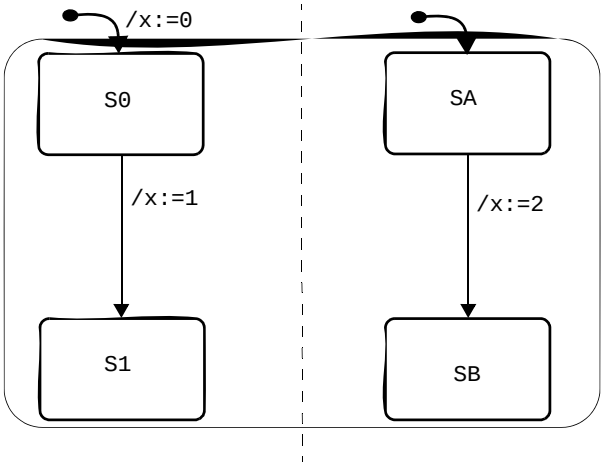
  In the following example, although a racing condition is reported, the language semantics would cause `y` to be updated before `x`.



- **Racing Write/Write** – Enables/disables the reporting of write/write racing conditions within/between Statecharts. The following messages appear on the workstation terminal.

  ```
  Default value: OFF

  SCL command: NONE
  ```

  The value of x is unknown because we cannot determine if x gets 1 before or after x gets 2.



- **Automatically Record New Trace Version** – This feature causes a trace file to be recorded every time the Simulation profile is executed. For more information, refer to **Tracing a Simulation** later in this section.
- **Automatically Record New SCP Version** – This feature causes an SCP file to be recorded every time the Simulation profile is executed.

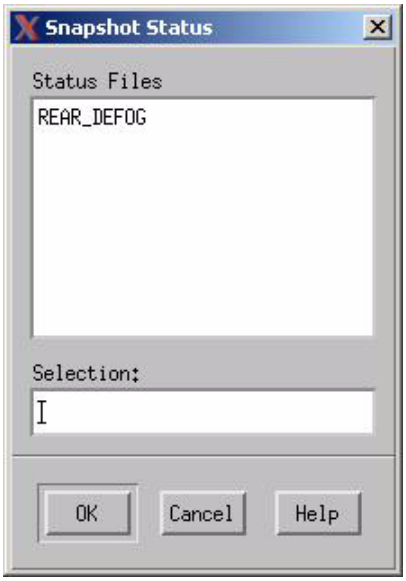
Saving and Restoring Status

The Simulation tool allows you to save the current system status for future reference. This is often useful when trying to backtrack to a certain point in the Simulation (e.g., nondeterministic solutions), to continue your work later or to use the current status in another Simulation scope.

Record > Snapshot Status – Saving the Status

The **Snapshot Status** command is used to save the current simulation status in a reloadable file.

1. Select **Record** > **Snapshot Status** from the **Simulation Execution** menu. The **Snapshot Status** dialog box opens. It displays a list of existing Status files.



2. Select the status file you want to over write from the **Status Files** list or enter a new name in the Selection text box.

3. Select **OK**. The current simulation status is save in a reloadable file.

Actions > Restore Status – Restoring the Status

1. Select **Actions** > **Restore Status**. The **Restore Status** dialog box opens.
2. Select the status file you want to restore from the Status Files list and select **OK**.

   The Simulation Status saved in the selected file is restored.

   When restoring a status, the Simulation tool checks the consistency between the current Simulation scope and the one in which the status was saved. When the two scopes are coincident, all saved values are restored. Attention must be paid to compound elements (their values are not saved, see below). Also, since you may wish to use different global/local clocks during the restoration, the `Show Future` command may show different times than when the status was saved.

   To effectively use the restore status facility when the stored status is a subset of the restored status or vice versa, the following points apply:

- Changes in the hierarchies of activities and/or states cause the saved status to become unrestoreable. This includes cases when a state/activity is added, removed or when it changes its place in the hierarchy.
- When a textual element is deleted, its saved value is ignored at the time of restoration. When a new textual element is added, its current value remains unchanged after the restoration.

The Status File

The status file is a non-ASCII file containing the following information:

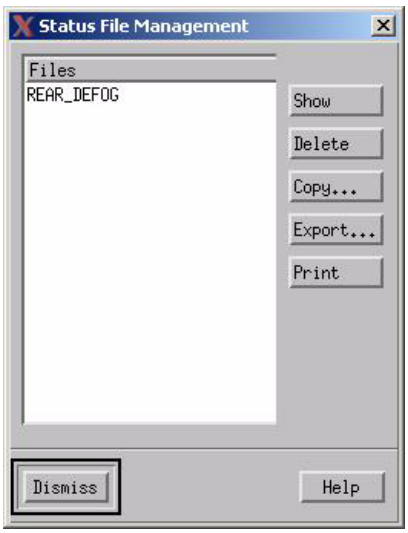
- Timing information (starting time and the current time)
- Status of activities
- Basic states configuration
- Values of primitive conditions
- Values of primitive data-items
- Generated primitive events, as well as generated events associated with other elements, such as en(S), st(A), tr(C), etc.
- Scheduled timeouts and actions with their respective times left till expiration.

Status File Management

The Status File Management command allows you to display, delete, copy, export and print selected Status files. This command is available in the **Profile Editor** and the **Simulation Execution** window.

1. Select **File > Simulation File Management > Status File Management.** The **Status File Management** dialog box opens.

2. Select the file from the **Files** list that you want to manipulate using the left mouse button. A description of each command contained in the **Status File Management** dialog box is provided in the following list.



- **Show** – Shows the selected Status file in ASCII format. In the viewer select your preferences.
- **Delete** – Deletes the selected Status file from the workarea.
- **Copy** – Copies the selected file after you re-name it. (Works the same way as **Save as.**).
- **Export** – Works the same way as *Copy* except you can save it to another workarea or any directory you want.
- **Print** – Used to print the selected file.
- **Dismiss** – Dismisses the **Status File Management** dialog box.

Tracing a Simulation

When executing a model, you may record all external changes and the system's reactions to these changes. This captured raw data is used as the basis for the creation of various spreadsheet trace reports, as well as graphical viewing of the simulation results using waveforms.

There are two ways to interactively enable the creation of the trace file:

- From the Execution Parameters dialog box.
- From the Record command.

Automatically Recording a New Trace File

1. Select **Options** > **Execution Parameters** from the Simulation Profile Editor menu. The **Execution Parameters** dialog box opens.
2. Select **Automatically Record New Trace Versio**n.
3. Select **OK**. The Simulation Trace is automatically saved.

Record > Start Trace – Creating a Trace File

A Trace can be also be started and stopped from the *Record* command from the **Simulation Execution** dialog box.

1. Select **Record** > **Start Trace** from the Simulation Profile Editor menu. The **Start Trace** dialog box opens. It displays a list of already existing trace files.
2. Select a **Trace** file name from the **Files** list to over write and existing trace or a new name in the **Selection** text box.
3. Select **OK**. All external changes and the system’s reactions to these changes are recorded.

In the batch mode, the Simulation uses the commands `set trace` and `cancel trace` to toggle the tracing facility. The trace file is closed when one of the following commands is entered: Exit, Restart Simulation or Rebuild Simulation.

Trace File Management

Trace files can be displayed, deleted, copied, exported and printed using the `Trace File Management` command. It can be accessed from the **Simulation Profile** window and **Simulation Execution** menu.

1. Select **File** > **Simulation File Management** > **Trace File Management**. The **Trace File Management** dialog box opens.

2. Select the **Trace file** from the *Files* list that you want to manipulate using your left mouse button. The **Trace File Management** dialog box opens.

- **Display** – Shows the selected Trace file.
- **Delete** – Deletes the selected Trace file from the workarea.
- **Copy** – Copies the selected Trace file after you re-name it. (Works the same way as *Save as*.).
- **Export** – Works the same way as **Copy** except you can save it to another workarea or any directory you want.
- **Print** – Used to print the selected Trace file.
- **Reports** – The Reports button is use to create reports. See **Creating Reports** later in this section.
- **Waveforms** – The **Waveform** button is used for analysis of traces produced during simulation. Refer to **Waveforms in Simulation** for addition information on Waveforms.
- **Dismiss** – Dismisses the **Trace File Management** dialog box.

Create Execution Log

Simulation can record all States,Transitions and Truth-Table lines that were visited during a simulation run, as well as executed mini-specs and static-reactions.

To create an execution log file:

1. Select **Options > Test Settings** from the Simulation Profile Editor menu. The **Test Settings** dialog box opens.
2. Check the **Create Execution Log** box.
3. Press **OK**, the log file will be created during Simulation execution.

The execution log file is named execution-<date-time>.trace and created under the ana/<profile name> directory in the workarea.

The logging continues into the same file until 'Quit' or 'Rebuild' operation. 'Rebuild' operation start a new execution log file.

The format of the execution log file is a list of:

- State Ids
- Transition Ids
- Truth-Table element Ids + executed row number
- Mini-spec Activity Ids + executed expression
- Static-reaction State Ids + executed expression

The separator '^' is used for Activity/States of generic instances.

Mini-spec and Static-reaction executed actions are logged in the following format:

<State/Activity Id> **Action{**<action part of the triggered expression>**}Action ExecutedAction{**<actually executed action>**}ExecutedAction**

A single **Action{**<expression>**}Action** may be followed by none, one or many **ExecutedAction{**<action>**}ExecutedAction**

For Example:

2533300560199681

2533300560199682

25663300660245681^2533300560199684

2566300660245681^2533300560199685

2566300660245681^2533300560199686

2566300660245681^2533300560199687

2533300560199677

2533343509872644 Action{for $I in MIN_IND to (MAX_IND-1) loop MY_INT_ARR_GOOD($I)=STRING_LENGTH(CONST_STR_ARR($I)); end loop;MY_INT_ARR_GOOD(MAX_IND)=1000-1}Action ExecutedAction{for $I in MIN_IND to (MAX_IND-1) loop MY_INT_ARR_GOOD($I)=STRING_LENGTH(CONST_STR_ARR($I)); end loop}ExecutedAction ExecutedAction{MY_INT_ARR_GOOD(MAX_IND)=1000-1}ExecutedAction

2533300560199678

2251825583489024 4

2533300560199680 2

2533300560199681^3940679738720256 1

2533300560199681^2251825583488024 6

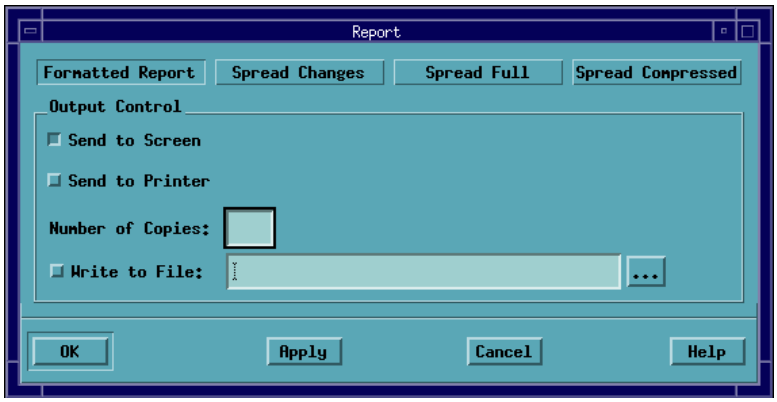
2533283380330502 Action{SUB_BY_TT1(IN_SUB_TT1,IN_SUB_TT2,OUT_SUB_TT1,OUT_SUB_TT2)}Action

2533347804839940 Action{SUB_BY_STM_ACT_LANG(B1,BA1,C1,E1,I1,R1,S1)}Action ExecutedAction{SUB_BY_STM_ACT_LANG(B1,BA1,C1,E1,I1,R1,S1)}ExecutedAction

Creating Reports

Reports can be generated, manipulated and printed through the `Trace File Management` command.

1. Select **File >Simulation File Management > Trace File Management** from the Simulation Execution dialog box. The **Trace File Management** dialog box opens.
2. Select the **Reports** button. The **Report** dialog box opens.



3. Select:
   - **Formatted Report**
   - **Spread Changes**
   - **Spread Full**
   - **Spread Compressed**

Additional information on each report follows.

Formatted Report

This report groups information on a stepwise basis:

- The st ep number (from the beginning of the Simulation), the time (in Global Clock Units) and, in the case of a superstep, the phase number (step number within that time).
- The changes caused by the environment: changes caused by the external actions that you enter prior to performing the step, either interactively or from the SCP (includes generated events, changes in conditions data-items, etc.).
- Changes caused by the system. These are the outcome of internal actions. A change can trigger a chain reaction producing other actions.
- The new configuration of states reached at the end of the step.

```
================================================================================
===  Step:      4 Phase:    4  Time:      0 clock units  ===
================================================================================

Changes caused by SUD
-----------------------------

  Generated events: ALARM

 Activated activities: MANIPULATE_PAGE


  States exited: OFF

  States entered: ALERT_MODE, ON, OPERATE, VIBRATION_MODE, WAITING

  Basic states configuration
  --------------------------

  IDLE, VIBRATION_MODE, WAITING
```

Spread Changes

To show, for each element in the Simulation scope, when and how its values/statuses changed during the Simulation.

- The first and last lines show the initial and final values/statuses of the elements in the session.
- The delta column shows the current step number.
- A row in which the *delta* value is marked by the letter `E` represents the external changes occurred in the step. In such cases, the next row corresponds to the same value of the `delta` and summarizes the system reaction.

```
Simulation name: PAGER
Time unit: 1 SECONDS



                        VIBR
   Time Delta   LIGHT ATION ALARM
------- -----   ----- ----- -----
      0     0   False False    ..
      0     1      ..    ..    ..
      0     2      ..    ..    ..
      0     3E     ..    ..    ..
      0     4      ..    ..     X
      0     5      ..    ..    ..
      0     6    True  True    ..
      0     7    True  True    ..
^L
Simulation name: PAGER
Time unit: 1 SECONDS



                  ALE
                 RT_CO ALERT
   Time Delta    NTROL _MODE  IDLE   OFF    ON
------- -----    ----- ----- ----- ----- -----
      0     0     Out   Out   Out   Out   Out
      0     1      ..    ..    ..    In    ..
      0     2      In    ..    In    ..    ..
      0     3E     ..    ..    ..    ..    ..
      0     4      ..    In    ..   Out    In
      0     5      ..    ..   Out    ..    ..
      0     6      ..    ..    ..    ..    ..
      0     7      In    In   Out   Out    In
```

Spread Full

The values/statuses of elements in the scope are shown at all moments, not only when changed.

```
Simulation name: PAGER                                      Page: 1 - 1
Time unit: 1 SECONDS


                                                             CR
                                      R                   ITICA
                      B       EMOVE  VIBR   ACK       L_PWR   NEW  SAVE
   Time Delta    BEEP EEPER LIGHT _CELL ATION _PAGE ALARM _LOSS _PAGE _PAGE
------- -----   ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
      0     0   False False False False False    ..    ..    ..    ..    ..
      0     1   False False False False False    ..    ..    ..    ..    ..
      0     2   False False False False False    ..    ..    ..    ..    ..
      0     3E  False False False False False    ..    ..    ..    ..    ..
      0     4   False False False False False    ..     X    ..    ..    ..
      0     5   False False False False False    ..    ..    ..    ..    ..
      0     6   False False  True False  True    ..    ..    ..    ..    ..
^L
Simulation name: PAGER                                      Page: 1 - 2
Time unit: 1 SECONDS

                    U
                SER_R               ALE                   MAINT
                ESPON   ALE       RT_CO     B             AININ
                SE_TI RT_CO ALERT NTROL EEPER         LOW G_NEW
   Time Delta   MEOUT NTROL _MODE :BEEP _MODE  IDLE _CELL _PAGE   OFF    ON
------- -----   ----- ----- ----- ----- ----- ----- ----- ----- ----- -----
      0     0      ..   Out   Out   Out   Out   Out   Out   Out   Out   Out
      0     1      ..   Out   Out   Out   Out   Out   Out   Out    In   Out
      0     2      ..    In   Out   Out   Out    In   Out   Out    In   Out
      0     3E     ..    In   Out   Out   Out    In   Out   Out    In   Out
      0     4      ..    In    In   Out   Out    In   Out   Out   Out    In
      0     5      ..    In    In   Out   Out   Out   Out   Out   Out    In
      0     6      ..    In    In   Out   Out   Out   Out   Out   Out    In
```

Spread Compressed

This report contains three parts:

- **Dictionary**– List of all elements in the scope, with the short names by which elements are referenced in the report. Elements of each type are enumerated and a short name is a combination of a letter indicating the element's type and a number.
- **Legend** – Shows correspondence between values of elements (except data-items) and numeric values representing these values in the report.
- **Spreadsheet** – Table summarizes the evolution of the elements' values in the Simulation.

```
Simulation name: PAGER
Time unit: 1 SECONDS

Dictionary:
===========

C3 Condition LIGHT
C5 Condition VIBRATION
E2 Event ALARM
S1 State ALERT_CONTROL
S2 State ALERT_MODE
S5 State IDLE
S8 State OFF
S9 State ON
S10 State OPERATE
S12 State PAGER_CONTROL
S15 State VIBRATE
S16 State VIBRATION_MODE
S17 State WAITING
A1 Activity ALERT_USER
A4 Activity LIGHT
A5 Activity MANIPULATE_PAGE
A7 Activity VIBRATION
D32 Data-item SWITCH_POS

Legend:
=======

   State     :   0 - Out           1 - In
   Condition :   0 - False         1 - True
   Event     :   0 - Not Occuring  1 - Occuring
   Activity  :   0 - Nonactive     1 - Hanging     2 - Active

 String data items:
  1 - ''
```

1. Select the **Output Control.**

   - **Send to Screen** – To copy the contents of the report to the terminal screen.
   - **Send to Printer** – To send the report to the printer.
   - **Number of Copies** – To specify the number of copies printed. The number of copies must be specified in the text box.
   - **Write to File** – To write the report to a file. The file name must be specified in the text box. You can select a file name by selecting the ellipse [...] button. This opens a File dialog box on your screen. From this dialog box, you can select a file name and directory.

2. Select **Apply** *or* **OK**. The selected options are started.

Interpreting Raw Data

The Simulation raw data may be interpreted as follows:

For Activities:

```
Activity SET_UP A Int
Activity SET_UP N Ext
```

where `A` and `N` are active and nonactive, while `Int` and `Ext` point to the source of the change (internal, external).

For States:

```
State OFF I Int
State ON O Ext
```

where `I` and `O` are in and out of the state and `Int` and `Ext` are relevant to external states only.

For Conditions:

```
Condition IN_CONNECTED T Int
Condition IN_CONNECTED F Ext
```

where `T` and `F` are true and false.

For Data-items:

Data-item FACT I 5 Ext

```
Data-item DELTA R 3.2 Ext
```

where `I` and `R` are integer and real.

For Events:

```
Event SET_UP X Int
```

where `X` indicates the occurrence of the event

Trace messages describing changes which occurred in a particular step are grouped together and preceded by a line showing the step and the phase number (step number within that time) and the current time.

Also, the moment the trace is enabled, the current values and statuses of all elements in the Simulation scope are placed into the trace file.

A trace file is a record of all the different statuses that occurred during the simulation. Whenever time changed or the state (value) of an element within the scope of the simulation changed, it was recorded. A trace file is often used to examine how a system behaved during a simulation session once the session is completed.

Record and Playback of Simulation

During the simulation execution, you may enable and disable the recording of commands to a Simulation Playback file.

The recording may be enabled and disabled as many times as necessary during a single Simulation. The result is a single playback file which contains the various scenarios. This playback file has the form of a Simulation Control Language program (refer to **Simulation Command Reference**) and can be run like a normal SCP.

Record For Playback

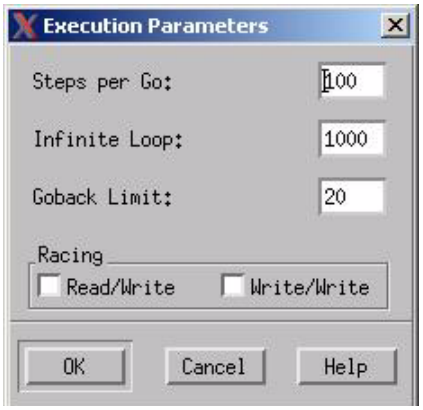
In general, the Simulation tool stores in an SCL playback file only information which affects the behavior of the simulated system, such as Simulation parameters, changes of specification elements, and *Go* commands. It does not record parameters which affect the various report facilities, information concerning only the viewing of changes and changes of internal variables of the running SCPs.

The detail is described below:

- Each time you enable the recording:
  - The current system status is saved (as a starting point of the recorded session fragment) and a corresponding `restore_status` statement is put into the SCL playback file.
  - The current setting of Simulation parameters is recorded. This includes for example, *Set Infinite Loop* and *Set Go Back*.
- During the session, the following information is recorded:
  - Each external change of a specification element is recorded as appropriate action on the element. These changes may come from a *Do Action, from a panel* or *monitor* or from the SCP.
  - Each entered `Go` command is recorded by its full name.
  - Each change of the simulation parameters.

The playback files are treated like SCP files and can be manipulated as such.

1. Select **Options > Execution Parameters** from the Simulation Profile Editor. The **Execution Parameters** dialog box opens.



2. Select the **Automatically Record New SCP Version** option.
3. Select **OK**.

Once Simulation execution is started, the recording is started. From the Simulation Execution window, you can start or stop the playback recording by selecting options: **Record > Record SCP** or **Record > Stop SCP Recording**.

Batch Mode Simulation

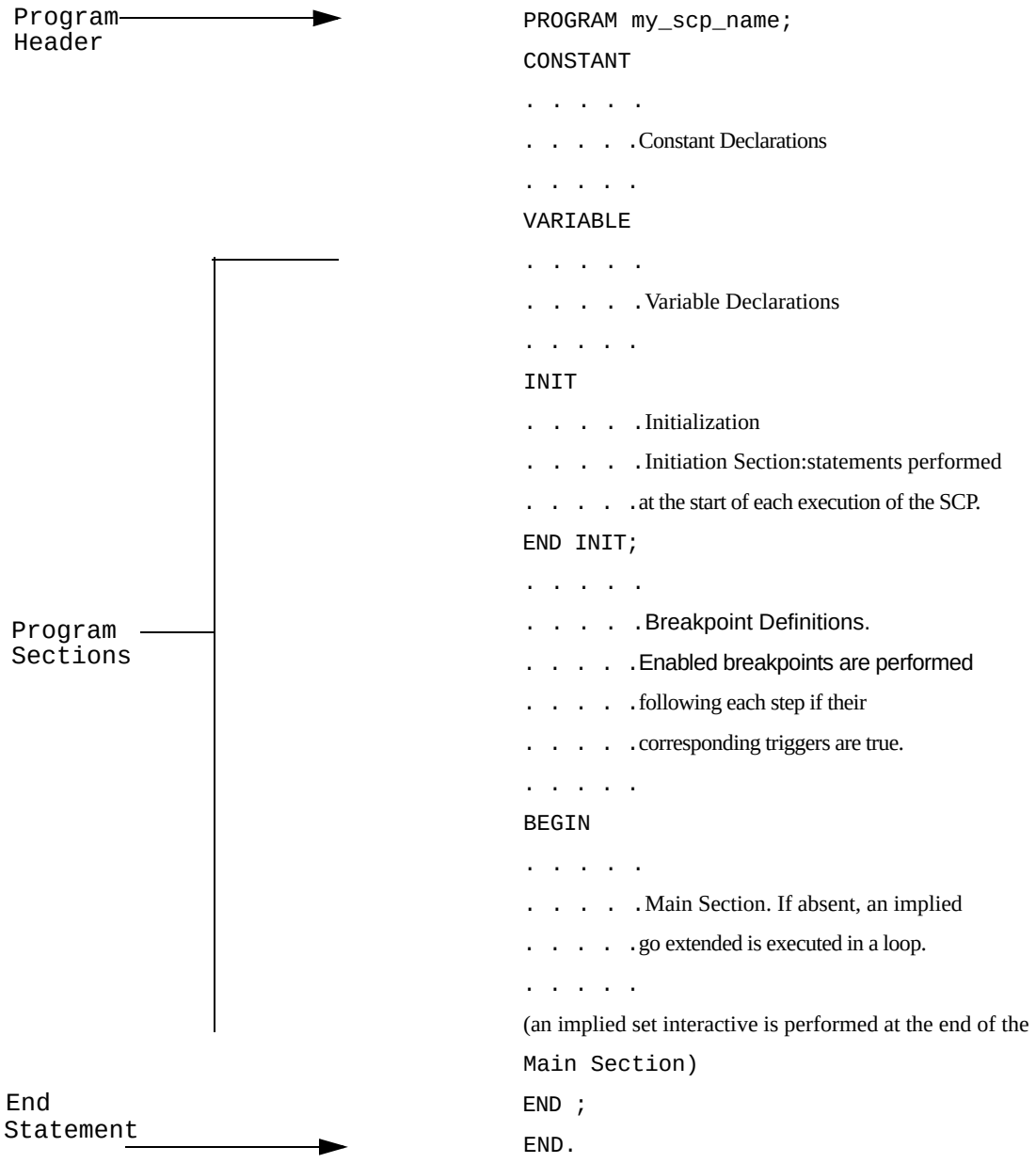
As your models grow in complexity, use of the interactive mode of Simulation, can become inefficient. To ease the entry of large amounts of data and to better describe a scenario-based execution, the Rational Statemate Simulation tool provides a batch mode of operation.

Simulation batch mode operates from a formatted text file of commands linked as a program language - *Simulation Control Language*. Batch Mode simulation is controlled by programs written in the Simulation Control Language. Executing a Simulation Control Program animates the Statecharts and activity-charts in the Simulation Scope in the same manner as Interactive Simulation.

This section details the use of Batch Mode Simulation and the SCL constructs.

The Simulation Control Program

The Simulation Control Program is a text file containing Simulation Control Language commands which drive the simulation of the Rational Statemate model. The Simulation Control Program has a very specific structure and execution order. A sample Simulation Control Program template is shown in the following diagram.



The Structure Of The Simulation Control Program

The Simulation Control Program has a specific structure. It contains the following Sections: program header, program sections, and end statement. There are five program sections:

- Constant section
- Variable section
- Initialization section
- Breakpoint section
- Main section

A valid Simulation Control Program may include any, all, or none of the program sections. All sections included in a Simulation Control Program must appear in the order described below.

The Program Header

The Program Header consists of the `PROGRAM` keyword followed by an identifier naming the program. The header has no impact on the execution of the program.

```
PROGRAM identifier;
```

Constant Program Section

This program section consists of the `CONSTANT` keyword followed by the constant declarations for the Simulation Control Program. Constants are always local to the Simulation Control Program.

A constant declaration contains a type (integer, string, float, etc.) followed by assignment statements. Multiple assignments are separated by commas and the last assignment within the type is followed by a semicolon.

```
CONSTANT
    INTEGER x:=5, y:=200, z:=1, z2:=10;
    STRING alpha:=‘unexpected loop’,
       beta:=‘enter action for yy’;
    FLOAT A:=2.5, B:=700.234;
    BIT B:=0;
    BITARRAY (1..16)A2:=0xab37;
    ARRAY (1..2) of STRING str:={‘Undefined action’,
    ‘Nonexistent’};
    ARRAY (1..6) of INTEGER int:={10,2,3,4,12,-72};
```

Variable Program Section

This section is used to declare the local and global variables used by the SCP. Variables may be local to the Simulation Control Program or declared globally for use in multiple Simulation Control Programs. Global variables must be declared in each Simulation Control Program where they are referenced.

Each variable declaration has a type (integer, string, float, file, bit, array, Boolean or bit array) followed by variable names. Multiple variable names are separated by commas with the last name followed by a semicolon. Variables can be initialized.

```
VARIABLE
      INTEGER xx, yy, u5:=5, za2;
      STRING gamma;
      GLOBAL STRING delta;
      FLOAT aa, bb, cc, dd:=10.1879;
      FILE f1, f2, f3;
      BOOLEAN valid, invalid;
      GLOBAL BOOLEAN correction;
      BITARRAY (1..10) BA1;
      BITARRAY (1..20) BA2;
      BITARRAY  (1..31) BA3;
```

Initialization Program Section

This program section contains the statements (except `Go` commands) to be executed upon running the Simulation Control Program. All statements are contained within the keywords `INIT` and `END INIT`.

```
INIT
      cc:= 23.6;
      yy:= 4000;
      gamma:= 'light standard';
      SET INFINITE LOOP 50;
END INIT;
```

Breakpoint Program Section

This program section contains the breakpoint definitions. Breakpoint definitions are contained within the keywords `SET BREAKPOINT` and `END BREAKPOINT`. Once defined, a breakpoint is automatically enabled.

Breakpoints are checked at the beginning of each `Go` command, and at the end of each execution step. Enabled breakpoints whose triggering expression is true have their corresponding statements (no **GO** commands permitted) executed. Below is an example of a Breakpoint.

```
SET BREAKPOINT [valid] DO
   WRITE ('VALID is True.');
END BREAKPOINT;
```

The breakpoint section may contain several breakpoint definitions, each delimited by the `SET BREAKPOINT` and `END BREAKPOINT` commands.

Main Program Section

This program section can contain the SCL statements that make up the main body of the program. These statements are contained within the keywords `Begin` and `End` and are executed sequentially.

The Main Section is only executed if the Simulation Control Program is started by the `Run` command. If started by `Exec`, this section is ignored and a warning message is issued.

After finishing the execution of the Main Section, execution is automatically switched to the interactive mode where a `Continue` command may be used to execute the *default main* (see below).

If the Main Section is omitted, the *default main* is executed. This default is a `Go Extended` in a continuous loop (until interrupted by the user):

```
BEGIN
WHILE TRUE LOOP
GO EXTENDED;
END LOOP;
END;
```

Basic Syntax Rules

The basic syntax of the Simulation Control Language is presented as follows:

- SCL is not case sensitive, except for text strings inside apostrophes.
- An identifier may be any string, beginning with a letter and consisting of any of the following characters: a-z, 0 -9, _. Identifiers have a maximum length of 16 characters.

  When a Rational Statemate element name coincides with an SCL reserved word or an SCL variable/constant identifier, an *underscore* is added as a prefix to the Rational Statemate element name. For example, your specification contains a state SET. However, set is also a reserved word in the SCL. To reference this name in the Simulation Control Program, precede it with an underscore (i.e., _SET).
- It is illegal for a Rational Statemate element name to be the same as a reserved word in the Rational Statemate action language (i.e., *WHILE*).Multiple SCL statements are permitted on the same line if they are separated by semicolons (;).
- A single SCL statement may span several lines.
- SCL has a set of reserved words and syntactical elements. Each of these has a special meaning and context. The SCL reserved words are listed in **SCL Reserved Words**.
- Comments are preceded by a double backslash (//). This symbol can appear at any point in the line, except within a literal string. The end of the line concludes the comment.

SCL Statements

SCL statements are made up of keywords, syntactical elements and identifiers. Statements are either *simple* or *structured*.

Simple statements consists of one or two keywords followed by identifiers and may span more than one line.

Structured statements contain other statements. They usually span several lines and use keywords to begin and end their structure and delimit their components. IF, THEN, ELSE are examples of a structured statement:

```
IF a > b
       THEN
              x := y;
       ELSE
              x := x + 1;
END IF;
```

Semicolons As Delimiters

Semicolons are used to separate Simulation Control Program statements. A semicolon optionally follows the last in a sequence of statements. This example uses semicolons only where required:

```
PROGRAM sample
VARIABLE FILE f1;
INIT
   OPEN (f1,‘my_file.doc’,OUTPUT)
END INIT
SET BREAKPOINT analysis => [STEP] DO
   WRITE (‘a will be greater than b \n’) ;
   alpha := 200 ;
   IF a<= b
      THEN
      alpha := 400 ;
      st!(transfer) ;
      a := 1 + b // optional semi-colon omitted
      ELSE
      sp!(counter)
   END IF;
   WRITE (f1,a,“\n”)
END BREAKPOINT
END.
```

Rational Statemate Expressions In the Simulation Control Program

In order to interact with the simulated system model, the Simulation Control Program must be able to detect the system status. It must also be able to change the system status by performing actions on the specification elements. Rational Statemate expressions are used in the SCL for this purpose.

When used in a Simulation Control Program, Rational Statemate expressions can reference both specification elements and SCL variables and constants.

Two types of Rational Statemate expressions are used in the SCL:

- **Rational Statemate Actions:** These are equivalent to the interactive mode input commands used to generate external changes. For example:

  ```
  if c then st!(A) else st!(B) end if

  a1 ; a2 ; a3
  ```

  where c is a condition, A and B are activities and a1, a2 and a3 are actions.

- **Rational Statemate Triggers:** In most executions, it is useful to trigger the execution of some actions either conditionally or as the direct result of some event. Such triggers are written as Rational Statemate expressions and are used as part of an SCL structured statement.

  Some examples:

  - `if c then . . .`

    when condition c is true, then take actions . . .

  - `when tr(c) then . . .`

    if the condition c becomes true during the last execution step, then . . .

  - `set breakpoint ch(i) do . . .`

    sets a breakpoint when a data-item i has changed value during the last execution step

  - `while c loop . . .`

    when condition c is true, trigger actions in a loop

When writing a Rational Statemate expression, remember to follow the rules outlined in the *Rational Statemate User Guide*. Some exceptions apply:

- Actions write(v) and read(v), where *v* is an SCL variable
- Events written(v), read(v) and changed(v), where *v* is an SCL variable
- Events true(v) and false(v), where *v* is an SCL Boolean variable

Your workarea must contain the specification elements referenced in expressions in the Simulation Control Program. Although these elements need not be in the current scope, the references must be unique.

Predefined Variables

The Simulation tool provides a set of predefined variables which are available to every Simulation Control Program without being explicitly declared. Some of these variables are numeric and contain data such as the step number and the current value of the execution clock. Others are Boolean and represent conditions which relate to the execution status.

The predefined variables are only alterable by the Simulation tool. You cannot directly manipulated their values. They are displayed by the Monitor SCP command together with variables explicitly declared in the running Simulation Control Programs.

List of Predefined Variables

- **STEP_NUMBER -** an integer variable whose value is equal to the number of the current execution step
- **CUR_CLOCK** – a float variable whose value is equal to the current execution time measured in global Clock Units. This is used to manage the timing of the specification.
- **NON-DETERMINISM** – a Boolean variable that becomes true when a step execution leads to a non-deterministic situation. It it usually used to trigger a breakpoint. Any meaningful sequence of SCL statements associated with the non-determinism breakpoint must include one of the statements below to resolve the situation:
- **CHOOSE** - resolves the situation by selecting a specific solution number.
- **RANDOM_SOLUTION** - randomly selects one of the possible solutions and continues the execution.

  If all breakpoints are processed and the non-determinism is still unresolved, the Simulation tool issues a message and automatically moves to interactive mode. The execution can only continue if the situation is resolved with either a `Restart` or `Rebuild` command.

  When used as a breakpoint trigger, the Non-determinism variable must be used by itself:

```
set breakpoint [nondeterminism] do
random_solution ;
end breakpoint;
```

- **TERMINATION** – a Boolean variable that becomes true when an execution step leads to a Termination Connector. If all breakpoints are processed and the termination situation is not handled, the Simulation tool automatically moves to interactive mode.

- **INFINITE_GO –** a Boolean variable that becomes true when the tool exceeds the maximum number of steps allowed without advancing the clock. If all breakpoints are processed and the infinite loop is not handled, the Simulation tool automatically moves to interactive mode to prevent an infinite loop.

  When used as a breakpoint trigger, the variable `Infinite_Go` must be used by itself:

  ```
  set breakpoint alpha =>[infinite_go] do
  i := 1
  end breakpoint
  ```

- **STATIONARY** - a Boolean variable that becomes true if no changes occur in the system status during an execution step. This condition is always true after a `go` REPEAT.
- **STEP** - a Boolean variable that becomes true when an execution step ends. It is usually used to trigger an operation to be done at every step.

Random Functions

The Simulation tool provides a number of random functions for use in your specification. They are useful for specifying a system that accepts input from an external system that is only described statistically.

List of Random Functions

- **RANDOM** - accepts an integer argument *i* and returns random real value distributed uniformly between 0 and 1. If the passed argument is not zero, then a new sequence of random values, whose seed is the parameter `i`, is initialized.

  Syntax: random(i)

  Since the Simulation tool always initiates a session with the same seed for random functions, two consecutive executions behave identically. The advantage is that you can reconstruct a particular execution scenario. New scenarios are produced by providing different seeds.

- **RAND_EXPONENTIAL** - accepts a real argument and returns random real values distributed exponentially by the value `t`. Using the syntax `x:=rand_exponential(t)` make `x` equal to a randomly generated number. The syntax `x:=random_exponential(t)` is accepted, but it makes x=the first value in an array called `random_exponential`.

  Function: X ~ exp(t)
  Syntax: random_exponential(t)

- **RAND_BINOMIAL** - accepts two arguments *n* and *p*, where `n>0` and `0<p<1`. The returned random values are real number distributed according to a binomial distribution.

  Function: X ~ B(n,p)
  Syntax: rand_binomial(n,p)

- **RAND_POISSON** - accepts a real argument *r*. The returned random values are integers distributed according to a poisson distribution.

  Function: X ~ P(r)
  Syntax: rand_poisson(r)

- **RAND_UNIFORM** - accepts two real arguments `a` and `b`. The returned random values are real values distributed according to a uniform distribution in the interval `[a,b]`.

  Function: X ~ U[a,b]
  Syntax: rand_uniform(a,b)

- **RAND_IUNIFORM** - same as rand_uniform except that `a` and `b` are integers and the value returned is an integer in the interval [a,b].

  Function: X ~ U[a,b]
  Syntax: rand_iuniform(a,b)

- **RAND_NORMAL** - accepts two real arguments `a` and `b`. The returned random values are real values distributed according to a normal distribution.

  Function: X ~ N[a,b]
  Syntax: rand_normal(a,b)

Random Functions In Simulation Control Program Statements

The random functions described, when used in the Simulation Control Program, are treated like any other numeric function in Rational Statemate - where the value returned may be used in the expression. For example, the random function output may be assigned to a variable:

```
i := rand_uniform(a,b)
```

A condition's value can be distributed equally:

```
random (0) < 0.5
```

Or an event can be randomly generated:

```
sc!(e, rand_uniform(x,y))
```

SCL Session Control Statements

Statements which facilitate program execution are detailed in **Simulation Command Reference.**

File Operation Statements

This section contains the following information:

- **OPEN Statement**
- **READ Statement**
- **WRITE Statement**
- **CLOSE Statement**

OPEN Statement

The `OPEN` statement opens a file for input or output.

```
OPEN (file_variable, 'file_name', INPUT | OUTPUT)
```

where the file_variable is assigned a file_name for purposes of input or output. For example:

```
OPEN (file1, '/csw/source/sample.data', INPUT);
```

The file being opened for input must already exist. If a file is opened for output, it cannot be opened again before it is closed. The interactive command `Monitor SCP` lists the currently opened files. The same file may not be opened for both input and output.

READ Statement

The `READ` Statement takes input from the keyboard or a file. The information read is assigned to any of the variable types except **FILE**.

```
READ ( [file_variable, ] x1 [, x2 . . . ] );
```

The first parameter is optionally a file identifier, indicating the source of the read is a file. If not provided, the read is done from standard input (keyboard).

WRITE Statement

The `WRITE` statement outputs data to either a file or the display. The output may be any combination of printable characters and numeric values.

```
WRITE ( [file_variable, ] exp1 [, exp2 . . . ]
   [ '\n' ] );
```

The first parameter is optionally a file identifier indicating that the target of the `WRITE` is a file. If not provided, the data is written to standard output (display). The optional argument '*\n*' outputs a carriage return between lines. For example:

```
WRITE ('The data value is ', d1, '\n');
```

To display or print integers in hexadecimal format:

1. Define a local variable of type bit array whose length is that of a standard integer. For example:

   ```
   Variables
   BITARRAY hexa_int(0..31);
   ```

2. hexa_int: = int;

3. Write ('Print int in hexadecimal format, hexa_int, '/n');

CLOSE Statement

The `Close` statement closes a file that was previously opened. If your system environment limits the number of open files, this command is used. The Simulation tool automatically closes files when the Simulation Control Program is stopped.

```
CLOSE (file_variable);
```

Structured SCL Statements

As in most programming languages, the user has the ability to control the program flow and to perform repetitive actions or to make decisions. The Simulation Control Program provides loop constructs and decision statements for these purposes.

IF/THEN/ELSE Statement

The `IF/THEN/ELSE` statement is used for conditional execution of SCL statements.

```
IF Boolean_expression
      THEN statement [ ; statement . . . ]
      ELSE statement [ ; statement . . . ]
END IF
```

Note

The `ELSE` statement is optional.

In this structured statement, the statements following the `THEN` and before `ELSE` are executed if the `Boolean_expression` is true. If false, the statements following the `ELSE` are executed. The `Boolean_expression` may include references to Rational Statemate elements as well as SCL variables and constants.

For example:

```
IF in(scanning) and level > 200 THEN
          x := 5 ;
          WRITE (’reset variable x \n’);
ELSE
          IF level <= 20 THEN
                WRITE (’no variable reset \n’);
          END IF;
END IF;
```

Note

The `ELSE` statement is optional.

WHEN/THEN/ELSE Statement

The WHEN/THEN/ELSE Statement is used for execution of SCL statements upon the occurrence of an event.

```
WHEN event_expression
   THEN statement [ ; statement . . . ]
   ELSE statement [ ; statement . . . ]
END WHEN
```

In this structured statement, the statements following the THEN and before the ELSE are executed if the event_expression is true. If false, the statements following the ELSE are executed. The event_expression may include references to Rational Statemate elements as well as SCL variables and constants.

For example:

```
WHEN entered(A) [level>20] THEN
       WRITE (’too high level when A entered \n’)
ELSE
       WHEN entered(A)
              WRITE (’running normally \n’)
       END WHEN;
END WHEN;
```

WHILE/LOOP and FOR/LOOP Statement

The WHILE/LOOP statement is used to execute SCL statements in a loop.

```
WHILE Boolean_expression
LOOP
       statement [ ; statement . . . ]
END LOOP
```

The statements in the LOOP clause are performed repeatedly while the Boolean_expression is true. The Boolean_expression is checked prior to each execution of the LOOP. The Boolean_expression may include Rational Statemate elements as well as SCL variables and constants. There is no limit to the depth of nested structured statements. For example, where cax, cb and cq are conditions and a1, a2 and a3 are actions:

```
WHILE cax
       LOOP
              a1 ;
              a2 ;
              IF x = 3 THEN tr!(cax);
              ELSE
              WRITE('not tripped \n');
              WHILE cb or cq LOOP
              Go Step a3;
              END LOOP;
              END IF;
       END LOOP;
```

FOR/LOOP example:

```
FOR i in int1 to init2 LOOP
   array(i):=0;
END LOOP;
          where i, Int 1 and int2 are intergers
```

Go Statements

The `Go` statements available in the interactive mode are also available in the batch mode. Note that `GoAdvance` and `GoStepN` are special cases since they require a parameter.

- `Go Step`

  Runs a single step. Time is advanced in Synchronous simulation.

- `Go Repeat`

  Runs several *GoSteps*, until a stable status is reached.

- `Go Next`

  Advances the time to the next scheduled action or timeout event.

- `Go Extend`

  Runs `GoRepeat` or `GoNext` and `GoRepeat`. (Available only in asynchronous simulation.)

- `Go StepN num_steps`

  Execute `num_steps GoSteps`.

- `Go Advance  num_time_units`

  Runs all reactions until, and including, the specified moment of time (relative and absolute time).

- `Go Back`

  Undoes the previous *GO* command.

- `Auto Go`

  Attempts to execute a `Go Step`. If a `Go Step` cannot be taken then a `Go Next` is performed.

Breakpoints

Breakpoints are useful when dealing with special situations arising during Simulation, such as non-determinism and infinite loops. Breakpoints are also useful when debugging your Simulation Control Program.

For each breakpoint, there are associated name (optional), event trigger and sequence of statements. You can enable or disable breakpoints in the course of the execution. At the end of each execution step, triggers of all enabled breakpoints are evaluated. When a breakpoint's trigger is true, the associated statements are executed.

Note

`Go` commands are allowed only in the main section, not in the breakpoint definitions. Also the square brackets around condition expression `cond_expr` are required.

Breakpoint Definition

Breakpoints are defined using the set breakpoint statement:

```
SET BREAKPOINT [ breakpoint_name => ] trigger
DO
       statement
       [ ; statement ]
       . . .
END BREAKPOINT ;
```

where:

`breakpoint_name` is any valid SCL identifier; `trigger` is an event expression or the keyword `every` followed by a numeric expression; `statement` is any legal SCL statement except **GO** statements.

Definition of a breakpoint automatically enables it. If a Simulation Control Program is assigned to an activity in your system, suspension of this activity disables all breakpoints in this Simulation Control Program. Resumption of the activity re-enables the breakpoints.

Breakpoints are checked at the beginning of each `Go` command, and after each execution step. The statements associated with this breakpoint are executed whenever the breakpoint is enabled and the trigger is true. The trigger evaluates to true if the event expression is true, or if the amount of time specified by the numeric expression following every has passed.

Every numeric_expression

The numeric expression is first evaluated at the end of the breakpoint definition and then, each time at the end of the breakpoint execution. The obtained value defines when the breakpoint is triggered the next time.

If the matching time has not passed, then the breakpoint's statements are executed according to the trigger's original evaluation. If the matching time has passed when the Simulation Control Program was suspended, then the numeric expression is re-evaluated at the time of resumption and the breakpoint triggering is scheduled relative to this time.

For example, if the breakpoint trigger is every `GO` and the execution clock units are in seconds, then the breakpoint statements are executed at one minute intervals. Assume the breakpoint is in a Simulation Control Program assigned to an activity which was suspended from time 01:30 to 01:55. Since the execution clock has not passed beyond the next breakpoint (02:00), this breakpoint's scheduled execution remains at the original interval. Assume the activity is later suspended from time 05:36 to 10:08. When resumed, the breakpoint trigger is re-evaluated because the current time has passed the next scheduled interval (06:00). The breakpoint is executed thereafter at new one minute intervals of 11:08, 12:08, etc.

Examples:

1. The associated SCL statements keep track of the execution time in minutes and display the minutes elapsed every minute. The assumed Global Clock Unit is seconds.

```
SET BREAKPOINT check => EVERY 60
DO
x := x +1
WRITE(x,'Simulation minutes passed \n')
END BREAKPOINT
```

2. The associated SCL statements automatically solve a nondeterministic situation and display a message.

```
SET BREAKPOINT [ NONDETERMINISM ]
       DO
          RANDOM_SOLUTION ;
          WRITE('Nondeterminism situation
          solved randomly. \n')
       END BREAKPOINT
```

3. The WHEN statements and assignment statement for error is associated with the breakpoint name `device_full`.

```
SET BREAKPOINT device_full =>
[ max_buf > 7 ]
DO
WRITE('Device full -
  taking recovery action \n')
WHEN tr(aux_buf_empty) THEN switch_bufs
ELSE
WHEN tr(aux_buf_ful) THEN st!(A)
ELSE
WRITE('Something missing
    in recovery procedure \n')
END WHEN
END WHEN
error := error +1
END BREAKPOINT
```

Cancelling Breakpoints

A breakpoint may be disabled using the `cancel breakpoint` statement. When at the end of an execution step, breakpoint triggers are evaluated, a cancelled breakpoint is simply ignored. Only named breakpoints can be disabled.

```
cancel breakpoint breakpoint_name
```

Setting Breakpoints

Breakpoints are enabled when defined. If a breakpoint has been disabled (`cancel breakpoint`), it may be re-enabled with `set breakpoint`. When re-enabling breakpoints, `set breakpoint` is written without any subsequent definition. Only named breakpoints can be re-enabled.

```
set breakpoint breakpoint_name
```

Other Set/Cancel Commands

```
SET DISPLAY ;
CANCEL DISPLAY ;
   SET GO BACK number ;

   SET INFINITE LOOP number ;
   SET INTERACTIVE ;
   SET TRACE ;
CANCEL TRACE ;
   SET REPORT RW_RACING ;
SET REPORT WW_RACING ;
CANCEL REPORT RW_RACING ;
CANCEL REPORT WW_RACING ;
```

Miscellaneous Commands

```
SAVE_STATUS 'status_name' ;
RESTORE_STATUS 'status_name' ;

CHOOSE number ;
RANDOM_SOLUTION ;
```

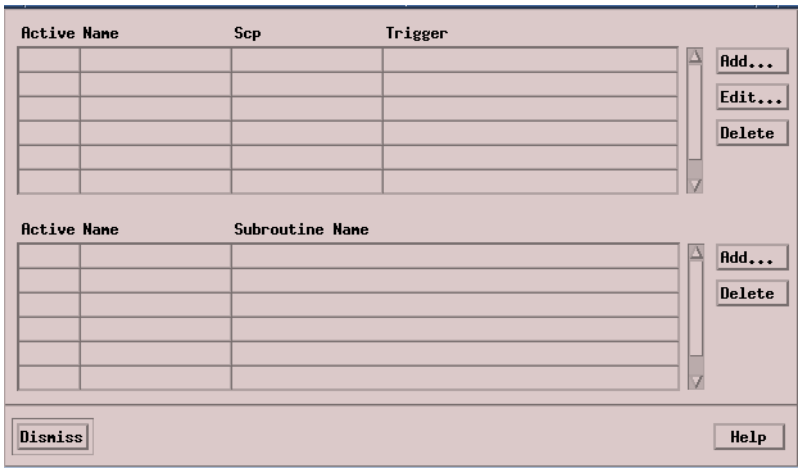
Manipulating Breakpoints with Menus

The **Breakpoints** command allows you to add, edit and delete breakpoints through the use of menus. In the following procedure we add, edit, and delete a breakpoint.

Select **Actions > Breakpoints** from the **Simulation Execution** menu. The **Simulation Breakpoint Editor** dialog box opens.

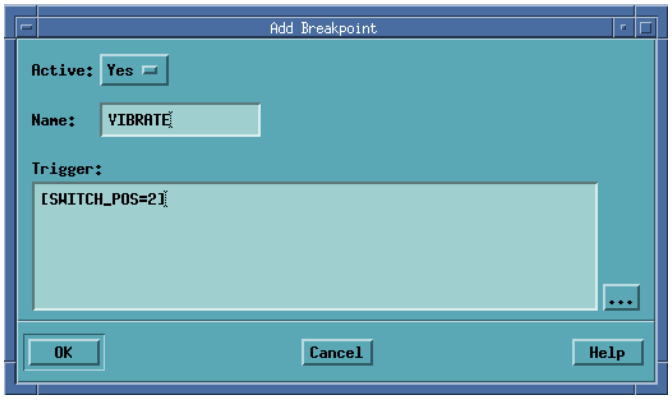
Note

See **Defining a Breakpoint in a Subroutine** for more information.



Breakpoint > Add – Adding a Breakpoint

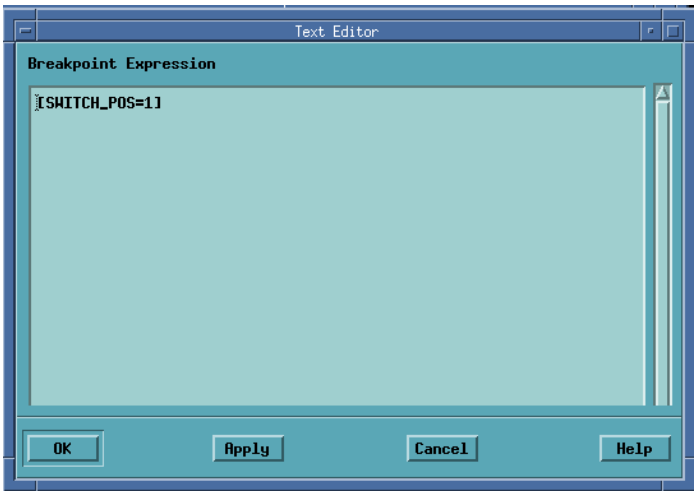
1. Click **Add**. The **Add Breakpoint** dialog box opens.



2. Enter the **Breakpoint Name** and **Trigger** into the appropriate text box areas and click **OK**.

Breakpoint > Edit – Editing a Breakpoint

1. Highlight the **Breakpoint** from the **Simulation Breakpoint Editor** dialog box.
2. Select **Edit**. The **Breakpoint Text Editor** dialog box opens. From the **Text Editor,** you can edit the Breakpoint Expression.



3. Click **Apply** *or* **OK** . The changes made from the Text Editor appear on the Breakpoint Text Editor.

Breakpoint > Deleting – Removing a Breakpoint

1. Highlight the **Breakpoint** to be deleted.
2. Select **Delete**.

Simulating a Truth Table

When the model is simulated and active breakpoints are inserted into the truth table, a read-only matrix of the truth table is started. From this table you can view the execution of each element in the table. When a step or microstep is started, depending on the truth table implementation, the “fired” row in the truth table is highlighted for one step.

Input and output logic from a truth table is included in the simulation when the model includes a truth table in its scope. Truth table inputs and outputs are Rational Statemate elements, therefore input values can be set and output values examined using Simulation debugging tools such as Monitors, the Examine command and the DoAction command.

The following sections describe how to insert a breakpoint into a truth table and simulate it.

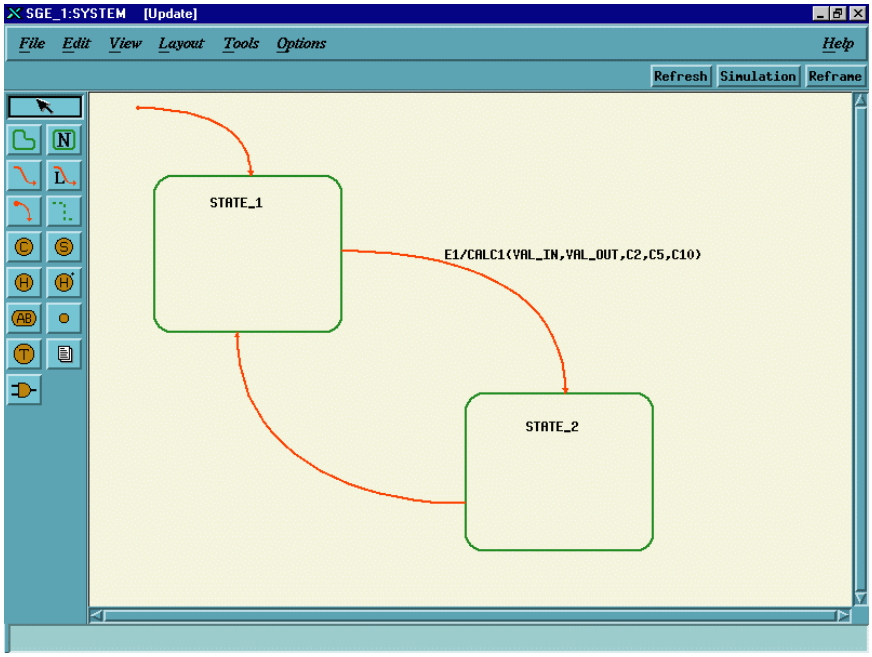
Note

For additional information on Truth Tables, refer to the *Rational Statemate User Guide*.

For the purpose of this discussion, we use the Statechart shown below as an example.This Statechart operates as follows:

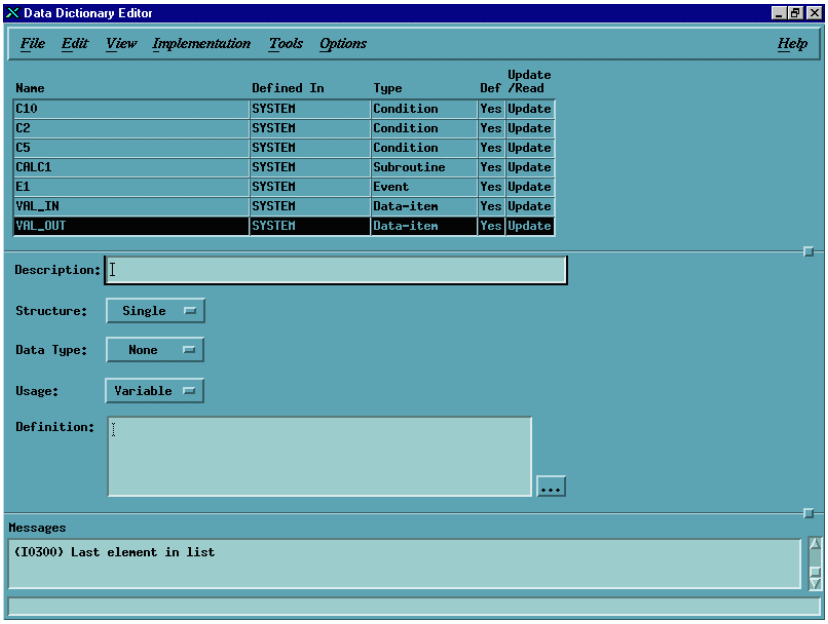
- When `C2` is true and `C5, C10` are false, `VAL_OUT` is assigned to two times `VAL_IN`.
- When `C5` is true and `C2, C10` are false, `VAL_OUT` is assigned to five times `VAL_OUT`.
- When `C10` is true and `C2, C5` are false, `VAL_OUT` is assigned to 10 times `VAL_IN`.

The actual logic that implements this functionality is contained in the truth table which implements the function `CALC1`.

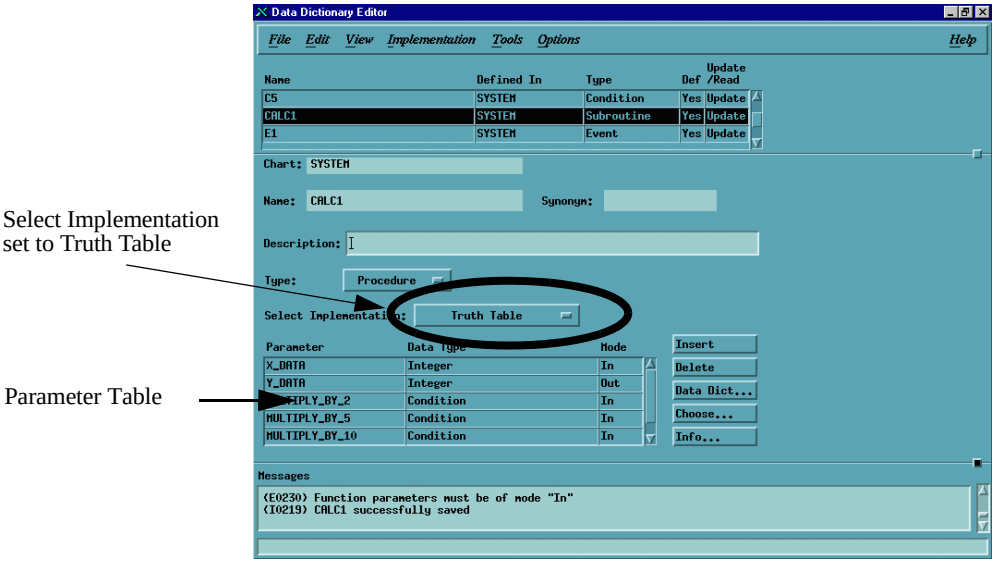


The elements for this Statechart need to be defined in the Data Dictionary as follows:

1. Define `C10, C2, C5` as a condition
2. Define `CALC1` as a subroutine
3. Define `E1` as an event
4. Define `VAL_IN` and `VAL_OUT` as data-items



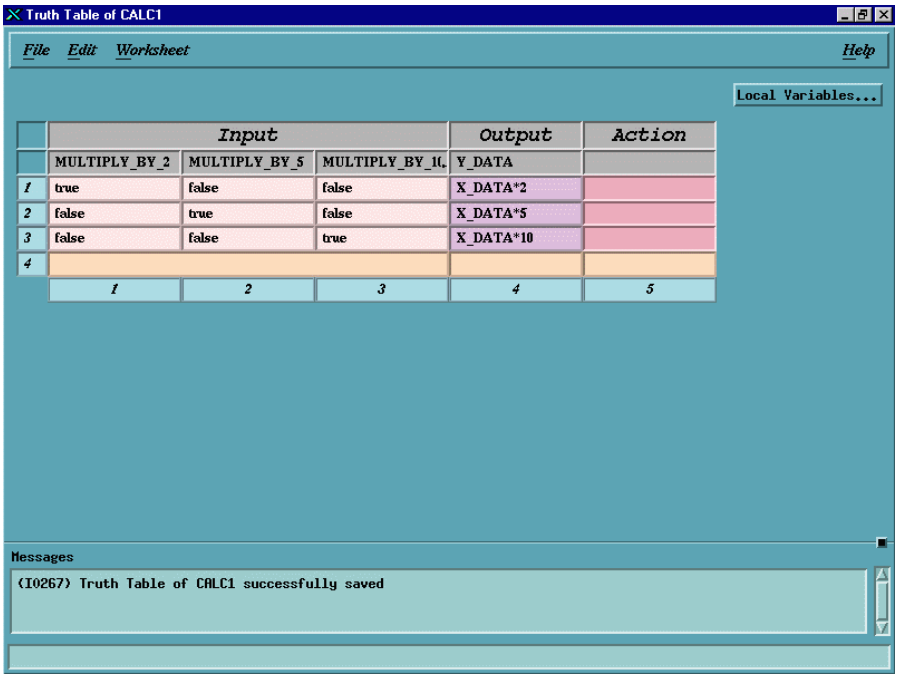
After defining the elements in the Data Dictionary for the sample Statechart, execute the Data Dictionary page for `CALC1`.



You can see in the Data Dictionary page above for the function `CALC1`, how the parameter table is filled. Note that the order of the parameters in the Statechart matches those in the parameter list. When these actual model parameters match with the formal parameters in the parameter list, `VAL_IN` is passed to `X_DATA`, `VAL_OUT` is passed to `Y_DATA` and the condition `C2` is passed to multiply-by-two, `C5` is passed to multiply-by-5, etc.

Note

The **Select Implementation** is set to Truth Table



To complete this exercise, construct the Truth Table shown above. For additional information on Truth Tables, refer to the *Rational Statemate User Guide*.

Setting Breakpoints in a Procedural Truth Table

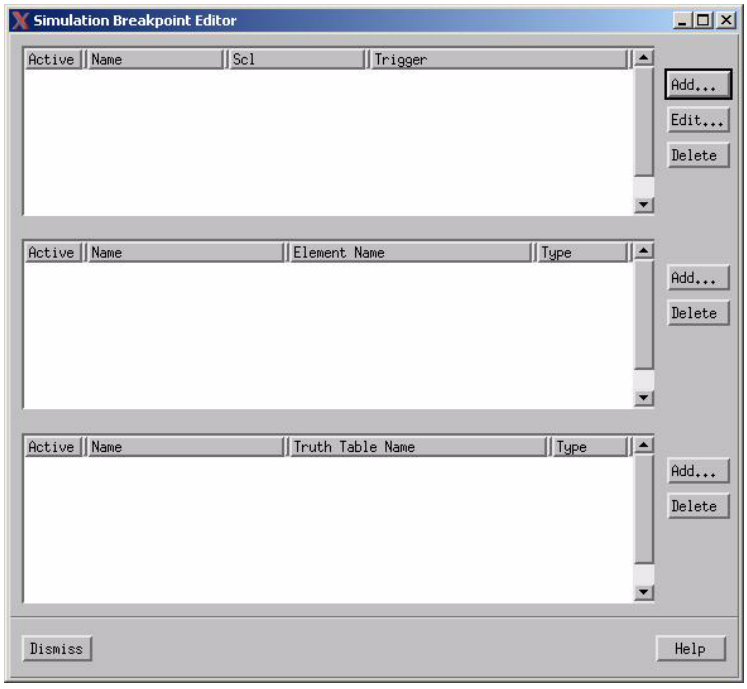
A breakpoint must be set in order to view the animation of a truth table during simulation. If a Procedural Truth Table is used, the breakpoint must be set in the corresponding subroutine. If an Action Truth Table is used, the breakpoint must be set in the activity which is implemented by the truth table.

The following procedure details how to set a breakpoint in a Truth Table.

1. Start **Simulation Execution** from either the Profile Editor or the Graphical Editor.

   Note: Open a monitor so you can view the variables.

2. Select **Actions > Breakpoints** from the **Simulation Execution** window.



The **Simulation Breakpoint Editor** dialog box opens.

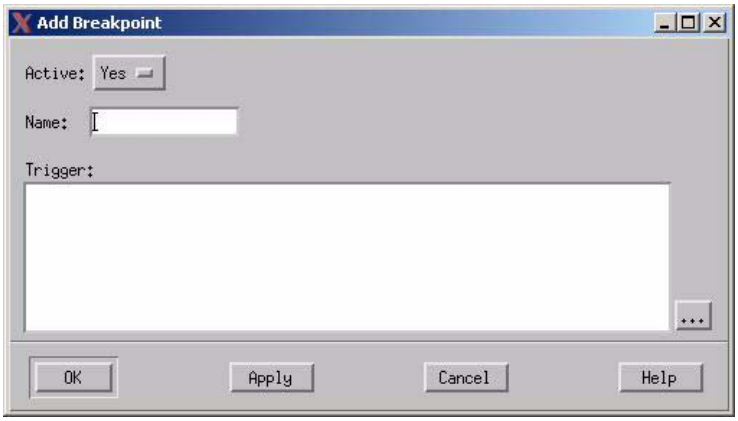
Note: A breakpoint can now be added in the subroutine `CALC1.`

Adding a Breakpoint to a Subroutine

If a Procedural Truth Table is used, a breakpoint must be set in a subroutine in order to start the debugger tool. When the breakpoint is reached during simulation, the debugger tool is started with the “Code” section empty. An additional form containing a read-only truth table is also opened.

In the following example, we set a breakpoint in the subroutine `CALC1`. This automatically sets a breakpoint in the truth table, because `CALC1` is implemented by the truth table.

1. Click **Add** from the Simulation Breakpoint Editor. The **Add Breakpoint** dialog box opens.



2. Enter the name of the breakpoint in the **Name:** field. In this example, the breakpoint name is `CALC1`.

3. Enter the name of the subroutine in which you wish to set a breakpoint in the **Subroutine Name:** field,. You can also use the pull-down menu to select the name.

   The **Simulation Breakpoint Editor** displays the subroutine breakpoint. Clicking **Apply > OK** if you want to close this window.

Note

In models containing multiple breakpoints, each name must be unique.

Subroutine Debug Tool

The Subroutine Debug tool is used to step through the truth table execution and monitor the execution of each step as the table is simulated. The behavior of the buttons located on the Subroutine Debug dialog box is described below.

- **mStep** starts one mStep (microstep). When an mStep is started, the evaluated rows are highlighted. As each mStep is started, the evaluated rows in the truth table is highlighted one by one until a row is fired. After a row is fired, the next mStep runs the output section (cell by cell). If an action section exists, the fired action is mapped into the code area and is debugged as an action language procedure. If an action is not specified, the next mStep dismisses the debugger. You can change or examine values of elements in the truth table. Changes take place immediately.
- **mStepN** runs a specified number of mSteps using the same rules as describe above.
- **Continue** highlight the fired row, runs the outputs and action section (if it exists), dismiss the debugger and continue to run.
- **Run to End** highlights the fired row, runs the outputs and action section (if it exists) and stays in the debugger.

Stepping through a Truth Table Simulation

In this section we complete a step-by-step simulation of the truth table.

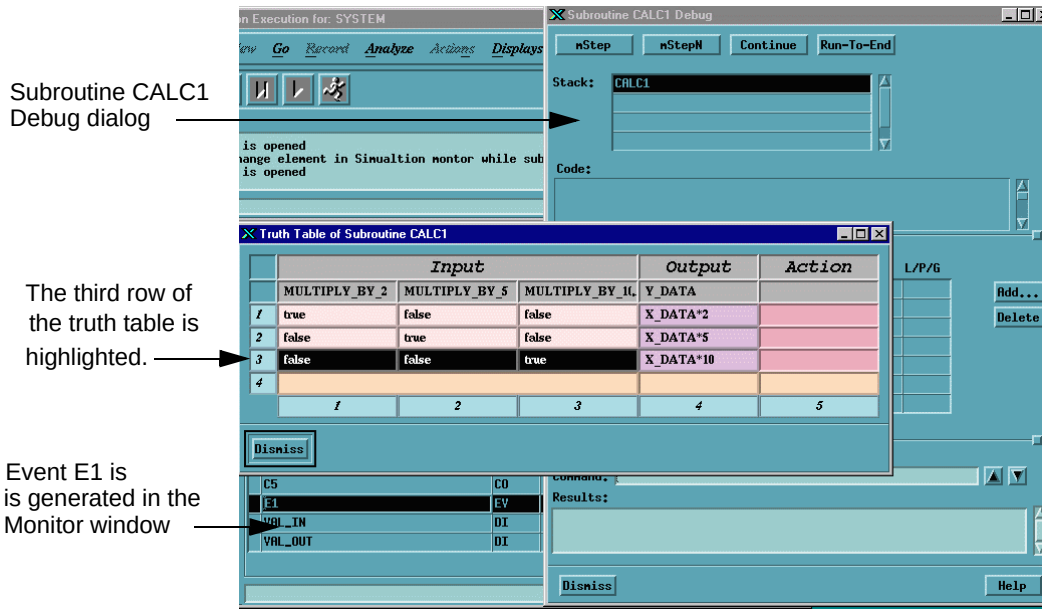
1. Execute a **Go Step**. A default transition is fired and we enter `STATE_1`.
2. Go to the **Monitor** and generate an event `E1.` Initialize `VAL_IN` to `1` and set the condition `C10` to `TRUE`.
3. Execute a **GO Step**.

   Note: The **Subroutine CALC1 Debug** window appears.

   Note: The first row of the truth table is now highlighted.
4. Clicking **mStep** to execute a microstep.

   The second row of the truth table is highlighted. The execution advanced to the second row of the truth table because the conditions `C2,C5,C10` in the model did not match the pattern in row 1 of the truth table. Remember, condition `C2` maps to the parameter multiplied-by-2, `C5` maps to the parameter multiplied-by-5, etc.
5. Execute another mStep.

   The simulation advances and row 3 is highlighted.



Note: In the output column of row 3, the output contains `X_DATA*10`. The pattern of row 3 matches the current pattern of the model. Since `X_DATA` corresponds to `VAL_IN` and `Y_DATA` corresponds to `VAL_OUT`, `VAL_OUT` is assigned to `X_DATA*10` (or, ten times one).

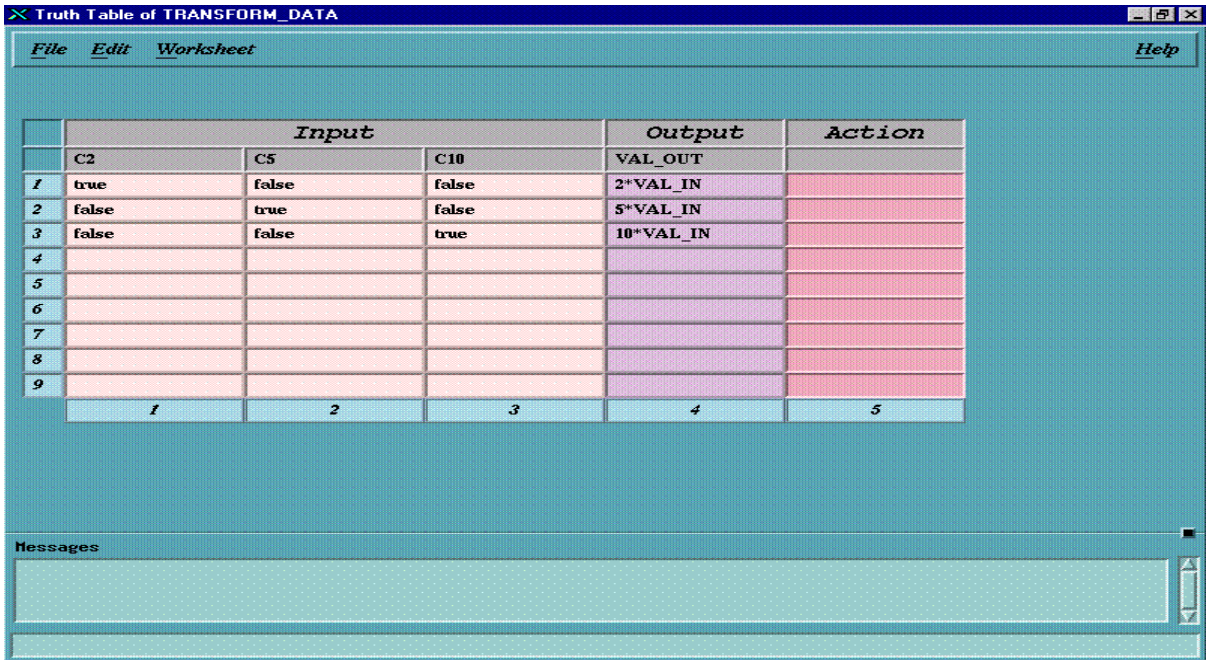
6. Execute the **Run-To-End** from the Subroutine `CALC1` Debug window. In the Monitor window, `VAL_OUT` should change to 10.

7. Execute a **Go Step**. You are returned to State 1. At this point you can experiment with different values for `C2`, `C5` and `C10` and observe their effect on the model.

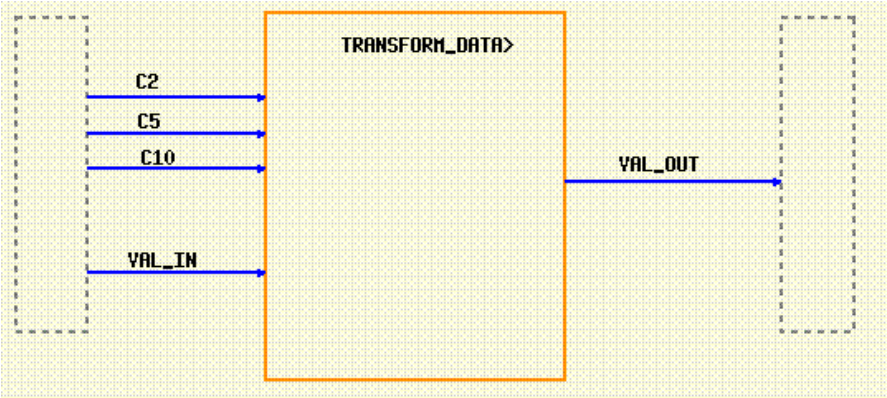
Simulating an Action Truth Table

Breakpoints are used to specify truth tables associated with activities and actions that are to be debugged. Truth tables are defined by a unique name (a unique instance name in the case of a generic) of the activity/action that they describe.

When a truth table is bound to an activity or action, then the assignments are made following the Rational Statemate step semantics. New values are only sensed at the next step. Writing to the same data-item twice flag a write/write racing condition. In the following example, `DATA_2` receives the previous value of `DATA_2`. If the truth table implements a subroutine such as in our previous example, then as soon as an assignment is made it is available to be used for this case. `DATA_2` and `DATA_3` receives the value of `5` when the row is fired regardless of the previous value of `DATA_2`.

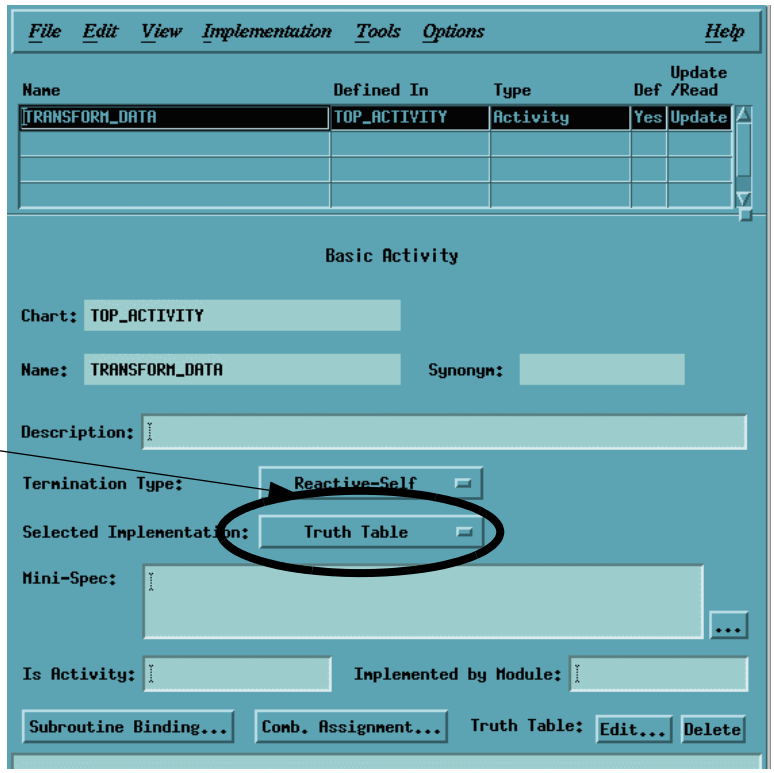


To illustrate the simulation of an Action truth table, we use the following Activity Chart and the above Truth Table.



Note

Before beginning you must first select implementation as Truth Table in the Data Dictionary.

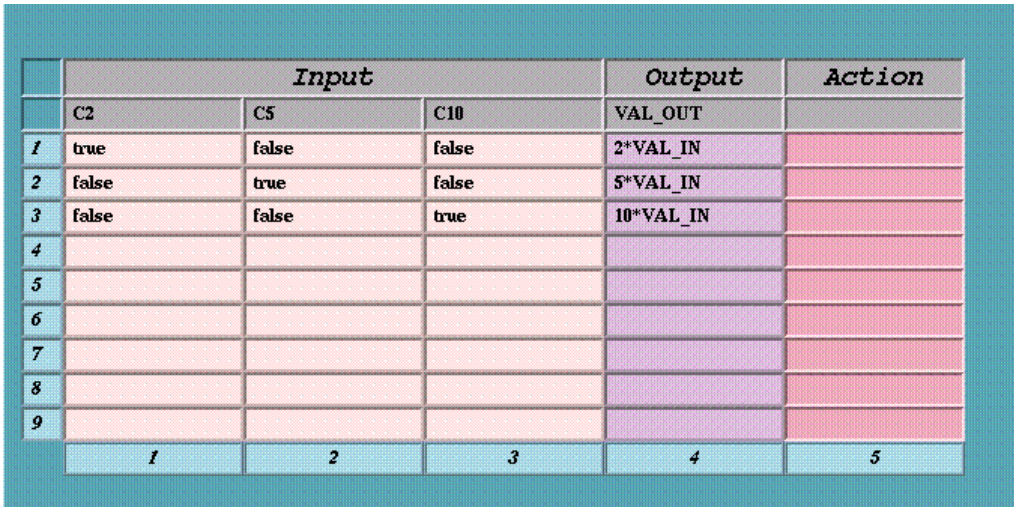


Selected Implementation set to Truth Table

Click **Edit**. This allows you to edit the truth table.

Note

This is an Action truth table so there are no parameters as in the previous example. The column headings for the truth table should match the actual parameter names in the model. The behavior implemented in the following example is the same as in the previous example.



	Input			Output	Action
	C2	C5	C10	VAL_OUT	
1	true	false	false	2*VAL_IN	
2	false	true	false	5*VAL_IN	
3	false	false	true	10*VAL_IN	
4					
5					
6					
7					
8					
9					
	1	2	3	4	5

Simulation of an Activity implemented by a Truth Table

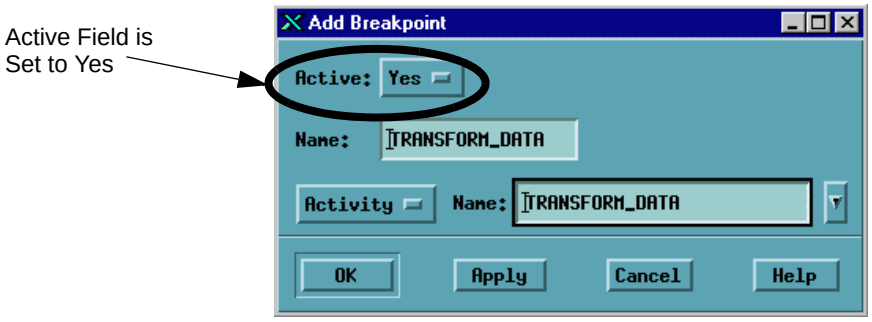
1. Execute a simulation from either the Graphic Editor or the Simulation Profile Editor. The **Simulation Execution** window appears.

   Note: Open a **Monitor** window and select all textual elements for display in the **Monitor** window.

   Note: As in the previous example, a breakpoint must be set to view the animation of a truth table.

2. Select **Actions > Breakpoints** from the **Simulation Execution** window. The **Simulation Breakpoint** editor appears.

3. Click **Add** located next to the **Truth Table Name** list. The **Add Breakpoint** dialog box opens.

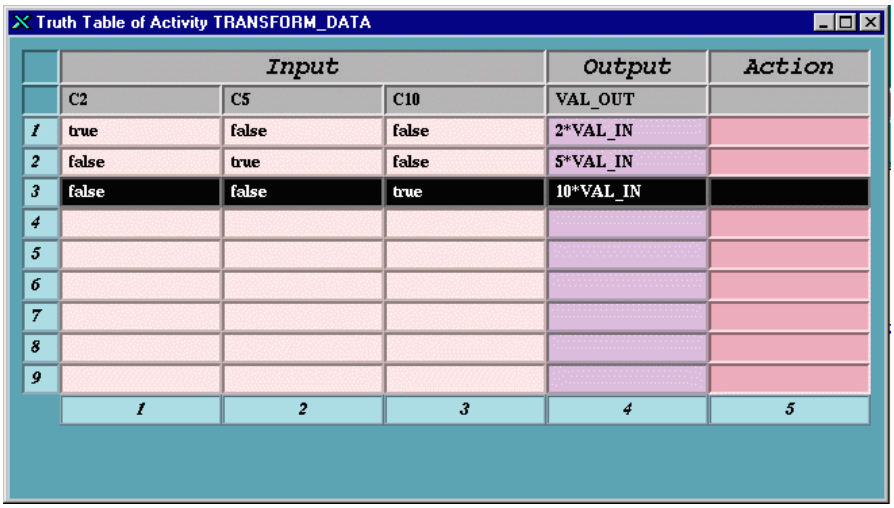


Note: The **Active** field is set to Yes allowing the simulation to pause when it reaches this breakpoint. If it is set to **No**, the breakpoint remains on the list but becomes inactive. Simulation does not pause at inactive breakpoints. Inactive breakpoints can be reactivated at anytime if the debugging tool for the subroutine is reinitiated.

4. Enter the **Breakpoint Name** in the **Name:** field.
5. Select **Activity** from the selection box.
6. Select **Transform_Data** from the pull-down list in the **Name:** field.
7. Select **OK**.
8. Click **Apply** > **OK** to close the Simulation Breakpoint Editor dialog box.

   Note: Before executing a `Go Step`, set `C10` to `True` and `VAL_IN` to `1` in the Monitor window.

9. Execute a **Go Step**. The animated truth table appears with row 3 highlighted. This corresponds to the status of `C2`, `C5` and `C10` in the model.



Note: Do not dismiss this truth table.

10. Go to the **Monitor** window and make `C5` true and `C2, C10` false.

11. Execute a **Go Step**.

Row 2 is now highlighted corresponding to the new status of `C5, C2` and `C10.` `VAL_OUT` is now 5 which is 5 times `VAL_IN`.

You can experiment with other values of `C5,C2` and `C10`.

Simultaneous SCP Execution

Multiple SCPs may be executed at the same time. Individual SCPs may be started and stopped during execution. The EXEC and STOP_SCP statements provide this function.

```
EXEC 'scp_name'
STOP_SCP [ scp_name ]
```

If the STOP_SCP statement is used without a scp_name, all SCPs are stopped.

Note that while the interactive RUN command stops any previously executing SCPs, the EXEC statement does not affect the execution of other SCPs. If an SCP was activated using RUN, its Main Section is executed, while that of any EXECed SCP is ignored.

Assign Files

The Simulation tool allows you to use programs to simulate the activities in your system. These programs may be written in SCL or a more conventional programming language (e.g., C). The SCL programs are ASSIGNed to either internal primitive activities or external activities.

External activities are part of the system environment and for purpose of the execution are considered permanently active.

Internal activities are part of the system. When an internal activity is started, the program ASSIGNed to the activity is started. Correspondingly, stopping, suspending or resuming of these activities causes appropriate changes in the status of the program.

The syntax to assign an activity is:

```
ASSIGN activity_name 'scp_name'
```

The Order of SCL Statements Execution

The structure of the SCP is fixed and determines the program's execution order. Below are some details.

Section Execution

If the Simulation Control Program does not contain breakpoints, the execution is sequential:

1. SCP constants and variables are defined.
2. Initialization Section's statements are executed.
3. Statements in the default Main Section or user-defined Main Section are executed.
4. When the end of the Main Section is reached, the Simulation tool switches to interactive mode. At this point, the command *STOP* terminates all SCPs and `CONTINUE` runs the default Main Section.

Breakpoint Processing

The execution of an SCP is altered by the breakpoints in effect.

- Breakpoints are defined after the Initialization Section.
- At the beginning and at the end of each execution step, the enabled breakpoints are checked and their associated statements executed.

Breakpoint processing is handled as:

- All the breakpoint triggers for all running SCPs are evaluated.
- The triggered breakpoints are executed.
- Repeat above until there are no triggered breakpoints.

Breakpoint processing is interrupted:

- When the `SKIP` statement is encountered (interactively or batch), the currently executing breakpoint is halted and any other breakpoints are skipped. That is, they are not processed for the current step.
- When a `GO` is issued interactively, a `SKIP` is implied. The last `GO` is terminated, any breakpoints are skipped and the interactive go is executed.
- The `STOP_SCP` statement can terminate a specific SCP or all SCPs, if no specific one is identified.

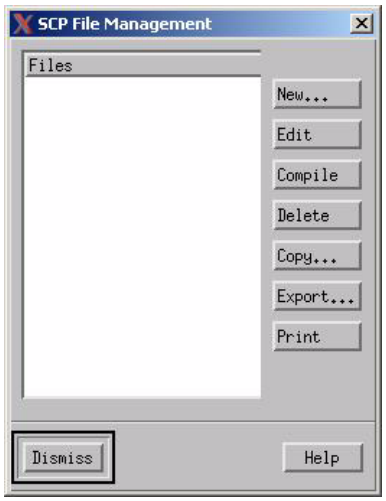
Working with a Simulation Control Program (SCP)

This section discusses the mechanics of using the Simulation tool to manipulate and run your Simulation Control Program. One way to create an SCP is to record the Simulation session for playback.

SCL files are manipulated through the use of the **SCL Files Management** dialog box. This dialog box can be accessed through the Simulation Profile Editor and the Simulation Execution menu.

To manipulate an SLP file, perform the following steps:

1. Select **File > Simulation File Management > SCP File Management**. The **SCL File Management** dialog box opens.



2. Select an SCP file from the **Files** list. The selected SCP file can be Edited, Compiled, Copied, Deleted, Exported, and Printed. Each command is described in detail below:

   - **New** – Used to create a new SCP file. You can create a new file from this dialog box.
   - **Edit** – Starts an editor so the selected SCP file can be edited.
   - **Compile** – Used to compile the SCP file and display any errors/warnings to the Simulation window.
   - **Copy** –Copies the selected profile after you re-name it. (Works the same way as *Save as*.).
   - **Export** – Works the same way as *Copy* except you can save it to another workarea or directory.
   - **Print** – Used to print the selected SCL file.
   - **Dismiss button** – The Dismiss button is used to dismiss the SCL Management dialog box.

Actions > Run SCP – Running an SCP File

Simulation Control Programs are run from the **Simulation Execution** menu.

1. Select **Actions** > **Run SCP.** The **Run SCP** file dialog box opens.
2. Select the SCP file you want to run from the **SCP Files** list using the left mouse button.
3. Select **OK**. The SCP file begins executing (running).

Switching Modes of Model Execution

Simulation control can be either Interactive or Batch, never a mix of both. Interactive mode means that Simulation control (`GO` commands, element value changes etc.) is done manually, either by typing commands at the command line or by selecting commands from the Simulation menus. Batch mode means that Simulation is controlled by the main section of an SCP program. When an SCP is started, its init section and breakpoint definitions are executed. If this is the only SCP running, its main section (if present) is also executed. The started SCP is now considered “active”, even if the execution of its main section is temporarily paused while interactive commands are performed.

Several SCPs can be active at the same time. Breakpoint definitions in active SCPs remain active during Interactive Simulation, until the SCP is stopped. Initially, when Simulation is started, you are in Interactive mode. When an SCP is started, you switch to Batch mode and the simulation is controlled by the main section in the SCP. If the SCP does not include a main section, the Simulator runs a `default main`, consisting of indefinitely repeated `GoExtend` command.

A running SCP can be temporarily paused while some interactive commands are performed, and then continued where it left off. During Batch mode (some SCP main section is executing) there are several ways to switch back to Interactive mode. The following automatically causes a return to Interactive mode:

- The SCL command `SET INTERACTIVE` is executed in the SCP. This command is often included as part of the command section for breakpoints.
- You select the `Pause` command from the **Simulation Execution** window to interrupt SCP execution.
- The main section of the SCP has executed to its end.
- The model has reached a stationary condition where no more changes are possible without intervention.
- A nondeterministic condition is encountered, that the SCP cannot resolve.
- The infinite loop limit is reached.

Switching from Interactive to Batch

From Interactive mode, Batch mode can be resumed by giving the Continue command, either typed at the Command line or by selecting **Continue SCP** from the **Actions** menu on the **Simulations Execution** menu. This continues execution of the SCP main section that was executing before entering Interactive mode.

Actions > Monitor SCP - Monitoring the SCP

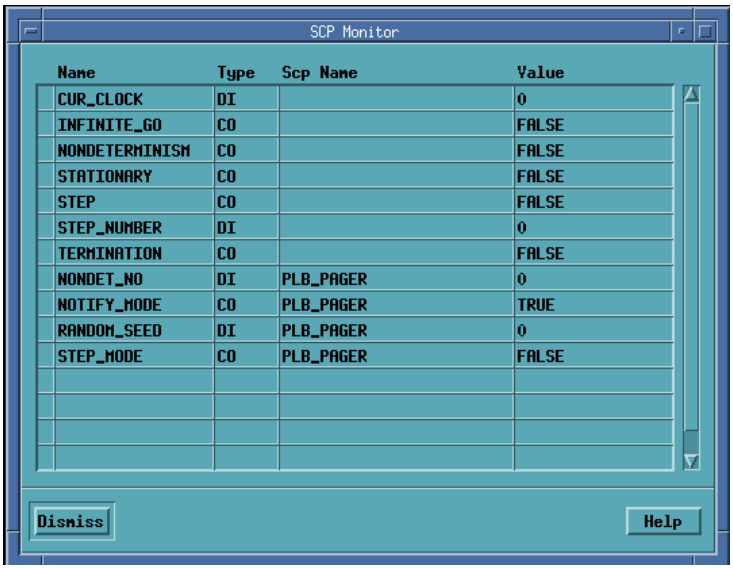
This command is used for displaying and changing information about active (running or temporarily paused) SCPs. The `Monitor SCP` command is used to start a dialog box containing all current:

- User defined SCP variables
- Rational Statemate defined SCP constants/variables defined in playback files: `NONDET_NO, NOTIFY_MODE, RANDOM_SEED, STEP_MODE`.
- Predefined variables: `CUR_CLOCK, INFINITE GO, NON-DETERMINISM, STATIONARY, STEP, STEP NUMBER, TERMINATION`.

To start this command:

Select **Actions** > **Monitor SCP**.

An **SCP Monitor** dialog box opens with the current values of elements with the SCP.



Actions > Stop SCP – Stopping an SCP

There are three ways to stop an executing SCP:

- Run another SCP. When an SCP is Running, all executing SCPs are halted.
- Halt all currently running SCPs by selecting **Actions > Stop SCP.**
- Halt a specific SCP with the `STOP_SCP` statement within the SCP.

To control simultaneous execution of several SCPs, the SCL commands `EXEC scp_name` and `STOP_SCP scp_name` are used. This means that multiple SCPs cannot be handled interactively, it must be done programmatically in another SCP.

Note

The main section of an SCP is not executed if the SCP is started with the *EXEC* command. The main section contains the *GO* commands. There can only be one main section controlling the Simulation. In this case the main section of the SCP containing the *EXEC* command.

The Auto-Run feature cannot be used with SCPs. If Auto-Run is active, it is stopped if you attempt to start an SCP. You are not allowed to start Auto-Run until all SCPs are stopped.

A special case is when an SCP is assigned to a basic activity. This connection is set up by the SCL command `ASSIGN`. This cannot be done interactively, only in an SCP. The assigned SCP is started, suspended, restarted and stopped along with the assigned activity.

Actions > Continue SCP - Restarting an Interrupted SCP

This command is used to resume the running of an interrupted SCP from the point of interruption.

Select **Actions** > **Continue SCP**.

The **interrupted SCP** resumes running.

A Sample Simulation Control Program

To illustrate some of the principles discussed in this section, a Simulation Control Program has been written against the Traffic Light system. Refer to **The Traffic Light System**. The remainder of this section discusses the Simulation Control Program in detail.

What the Traffic Light Simulation Control Program Accomplishes

Recall that the Statechart for the traffic light system has a state **NORMAL_OP** which is influenced by the values of two data-items which control the amount of time the traffic flows either in east-west or north-south directions. In this example, the **ns_green_time** is assigned randomly, while the **ew_green_time** is assigned from an external file **trial.dat.**

The external file **trial.dat** has the following records:

```
12
40
14
8
18
20
46
7
37
23
19
```

After initializing the two control values, **ns_green_time** and **ew_green_time**, the execution begins. Breakpoints are defined and the execution is driven by the **go step** operation. Since we use the Synchronous Time Model, **go step** increments the clock on each step.

Each time a **malfunction** occurs in the system, the Simulation Control Program resets the control values. The Simulation Control Program filters the **trial.dat** against a maximum allowed value of 20. The file **trial.out** contains a time-stamp of each **malfunction**.

The Program

```
PROGRAM  tlight;
VARIABLE
       FILE fin, fout;
       INTEGER delay;
       BOOLEAN run;
INIT
       OPEN (fin,'/mickey/home/dos/tmp/trial.dat', INPUT);
       OPEN (fout, '/mickey/home/dos/tmp/trial.out', OUTPUT);
       ns_green_time:=15 ;
       ew_green_time:=20 ;
       CANCEL BREAKPOINT gen_reset ;
END INIT;
SET BREAKPOINT
       [in(normal_op)] DO
              WRITE('current time = ', cur_clock,'\n') ;
              IF rand_iuniform(1,100) = 1
              THEN
              malfunction ;
              write(fout,'malfunction occurred
              at ',cur_clock,'\n')
              END IF
END BREAKPOINT;
SET BREAKPOINT gen_reset=> EVERY delay DO
              reset ;
              CANCEL BREAKPOINT gen_reset;
              ns_green_time:=rand_iuniform(30,50) ;
              READ(fin, ew_green_time) ;
              WHILE ew_green_time > 20
              LOOP
              WRITE('Data for ew_green_time exceeds limit.') ;
              WRITE('Enter a value for ew_green_time less than
```

```
                20: \n') ;
                READ(ew_green_time)
          END LOOP
    END BREAKPOINT ;
    SET BREAKPOINT
          en(flashing) DO
          delay:=RAND_IUNIFORM(1,10);
        SET BREAKPOINT gen_reset ;
    END BREAKPOINT;
    BEGIN
          tr!(run) ;
                WHILE run LOOP
                GO STEP
                end loop
    END;
    END.
```

Explaining the Program

In this portion, each line of the program is explained.

```
Program tlight
```

This line names the program. This line is for documentation purposes only.

```
variable
```

Begins the Variable declaration Section. Note that each type of declaration concludes with a semicolon.

```
file fin, fout ;
```

Defines two file variables, `fin` and `fout` that are used for I/O purposes.

```
integer delay;
Boolean run ;
```

Defines an integer used to delay the reset event and a Boolean variable that is used as a condition to create a continuous loop in the Main Section.

```
init
```

Begins the Initialization Section which contains statements executed once - each time the Simulation Control Program is started.

```
open (fin, 'trial.dat', input) ;
```

Opens an input data file and attaches it to the variable name *fin*. This data file contains the test values for the east-west traffic flow.

```
open (fout, 'trial.out', output) ;
```

Opens the file, *trial.out*, for output and attaches it to the variable fout. This file records all the malfunctions of the system.

```
ns_green_time := 15 ;
ew_green_time := 20 ;
```

Initializes the value of the data-item used to determine the duration of the green lights in both east-west and north-south directions. The Global Clock Unit is assumed to be seconds.

```
cancel breakpoint gen_reset ;
```

Initializes the gen_reset breakpoint so that it does not execute.

```
end init
```

End of Initialization Section

```
- - breakpoint Section
set breakpoint
```

Beginning of the Breakpoint Definition Section. The defined breakpoint is simultaneously enabled. The name of this breakpoint is not defined. Unnamed breakpoints cannot be cancelled and reset in batch mode. They are continuously enabled during the execution of the Simulation Control Program.

```
[in(normal_op)] do
```

Defines the breakpoint trigger as "being in the state of **NORMAL_OP"**. The breakpoint statements following the keyword *do* are executed when the trigger evaluates to true. That is, the statements are executed at each step while the system is operating normally.

```
write('current time = ', cur_clock, '\n') ;
```

Displays the current execution time in the Simulation Window after each `go` - in this case, after each step is concluded. The Simulation tool automatically updates the value of the predefined variable cur_clock at the end of each `go`.

```
if rand_iuniform(1,100) = 1
```

Randomly selects an integer uniformly distributed between 1 and 100 and tests this value to see if it is equal to 1. This simulates situations that arise 1% of the time during normal operation.

```
then
```

When the test is true, the statements following the keyword `then` are executed.

```
malfunction ;
```

Generates the event `malfunction`. This simulates an electrical malfunction of the system. When the malfunction event occurs, the traffic lights flash.

```
write(fout,'malfunction occurred at ', cur_clock, '\n')
```

The time of the malfunction is recorded in the output file.

```
end if
```

Ends the IF structured statement.

```
end breakpoint
```

Ends the definition of the breakpoint.

```
set breakpoint gen_reset => every delay do
```

Beginning of the Breakpoint Definition Section. The defined breakpoint is simultaneously enabled. The breakpoint is named `gen_reset` and runs `every delay` amount of time. The integer `delay` is set in the next breakpoint.

```
reset ;
```

Generate the event reset. This resets the malfunction of the traffic lights.

```
cancel breakpoint gen_reset ;
```

Cancels any further execution of the gen_reset breakpoint until needed after the next malfunction.

```
ns_green_time := rand_iuniform(30,50) ;
```

Generates a new random value for the north-south traffic flow. It generates a value between 30 and 50 seconds.

```
read(fin, ew_green_time) ;
```

Reads a new value from the data file for the east-west traffic flow.

```
while ew_green_time > 20
```

Tests that the value from the data file does not exceed a maximum value.

```
loop
```

The test is done in a `WHILE/LOOP` statement. This insures that the new input (from the user) cannot exceed the maximum value.

```
write('Data for ew_green_time exceeds limit. \n') ;
write('Enter a value for ew_green_time less than 20: \n') ;
```

When the input value fails the test, the user is prompted for a new value.

```
read(ew_green_time)
```

Note

A new value for the east-west traffic flow is read from the keyboard.

```
end loop
```

Note

Ends the WHILE/LOOP statement.

```
end if
```

Ends the IF structured statement.

```
end breakpoint
```

Ends this breakpoint definition.

```
set breakpoint
en(flashing) do
```

Starts a new breakpoint executed when the `flashing` state is entered.

```
delay:=rand_iuniform(1,10)
```

Sets the `delay` for the reset event to a number between 1 and 10.

```
set breakpoint gen_reset;
```

Enables the execution of the gen_reset breakpoint.

```
end breakpoint;
```

Note

Ends this breakpoint definition.

```
- - Define Main Section to use required Go Type
begin
```

Begins the Simulation Control Program Main Section whose statements are executed sequentially.

```
tr!(run) ;
```

Sets the Boolean variable *run* to true.

```
while run
```

Evaluates the trigger run and, if true, runs the WHILE/LOOP statements. Since this variable has just been set true, the loop runs continuously.

```
loop
```

Begins the WHILE loop.

```
go step
```

Note

The `go step` in the Synchronous Time Model advances the clock with each execution.

The Main Section is supplied here instead of using the default Main Section. The default main Section always advances the execution using `go extended`.

```
end loop
```

Ends the `WHILE` loop.

```
end
```

Ends the Main Section.

```
end.
```

Ends the Simulation Control Program.

Simulation Command Reference

In Simulation Execution, commands can be selected by pull down menus, hot keys, by typing the command into the command line, or by execution in batch mode. This section details the operation of each simulation command; both interactive and batch.

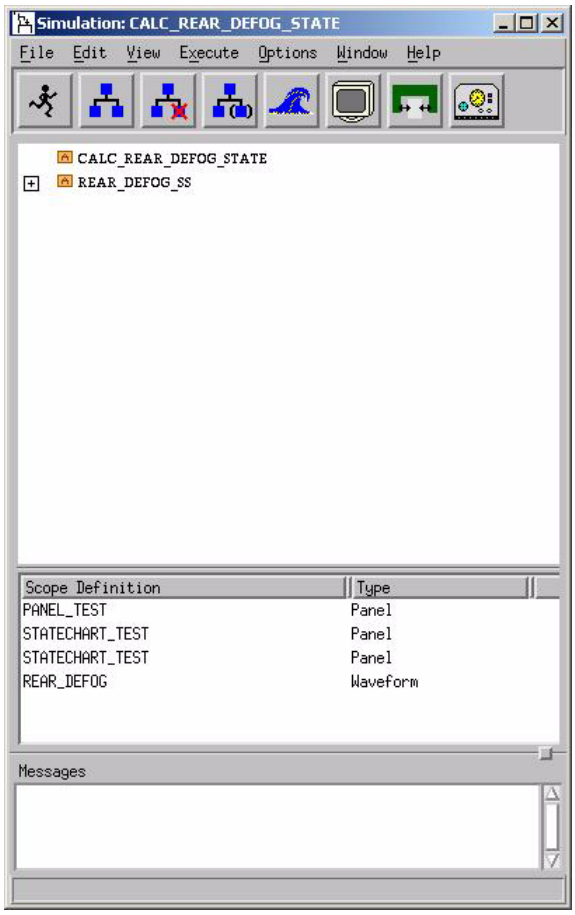
The commands are listed in two sections, Interactive mode and Batch mode, and are arranged in the order they appear on the interface. Each command contains a brief command description followed by an operation section that describes how to access that command.

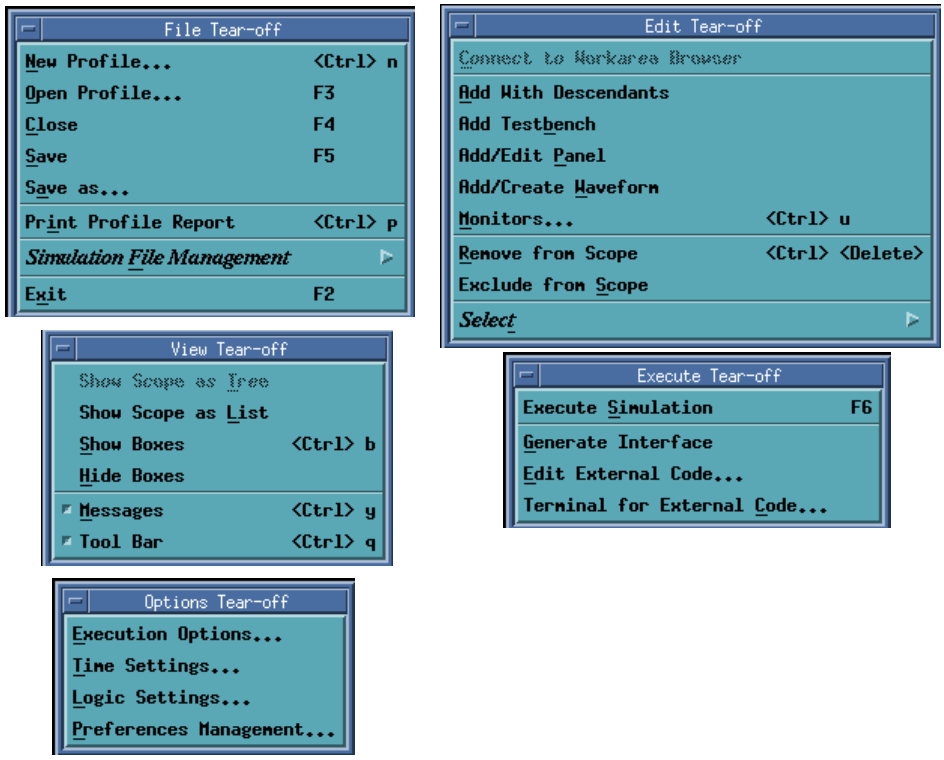
Interactive Commands

The following commands can be found in the menus of the **Simulation Profile Editor** and **Simulation Execution** menu.

The Simulation Profile Editor

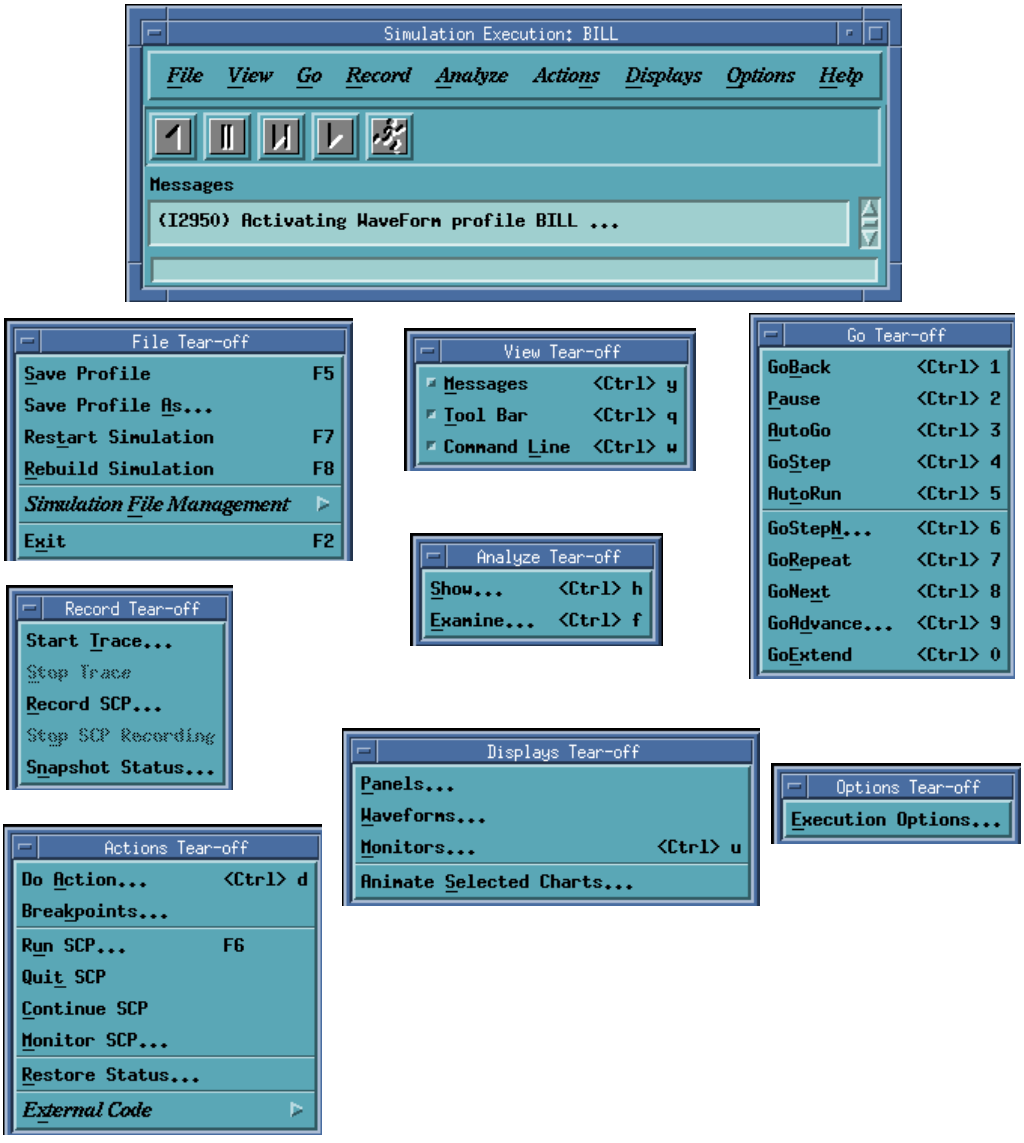
The following graphic of the **Simulation Profile Editor** illustrates the tool's five pull-down menus. Each menu selection is described in detail in this section.


Simulation: CALC_REAR_DEFOG_STATE
File Edit View Execute Options Window Help
CALC_REAR_DEFOG_STATE
REAR_DEFOG_SS
Scope Definition
Type
PANEL_TEST
Panel
STATECHART_TEST
Panel
STATECHART_TEST
Panel
REAR_DEFOG
Waveform
Messages


File Tear-off
New Profile... <Ctrl> n
Open Profile... F3
Close F4
Save F5
Save as...
Print Profile Report <Ctrl> p
Simulation File Management
Exit F2
Edit Tear-off
Connect to Workarea Browser
Add With Descendants
Add Testbench
Add/Edit Panel
Add/Create Waveform
Monitors... <Ctrl> u
Remove from Scope <Ctrl> <Delete>
Exclude from Scope
Select
View Tear-off
Show Scope as Tree
Show Scope as List
Show Boxes <Ctrl> b
Hide Boxes
Messages <Ctrl> y
Tool Bar <Ctrl> q
Execute Tear-off
Execute Simulation F6
Generate Interface
Edit External Code...
Terminal for External Code...
Options Tear-off
Execution Options...
Time Settings...
Logic Settings...
Preferences Management...

Simulation Execution Menu

The following graphic of the Simulation Execution menu illustrates the tool's various pull-down menus. Each menu selection is described in detail in this section.



Save Profile

The **Save Profile** command is used to save changes to an existing Simulation Profile.

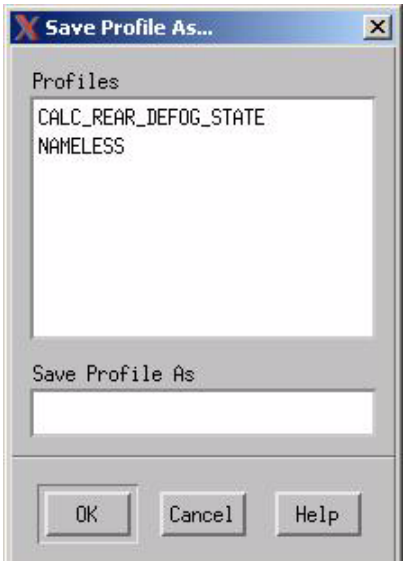
Select **File > Save Profile** from the **Simulation Execution** menu.

The **profile is saved if modified**.

Save Profile As

The **Save Profile As** command is used to name and save a new Simulation Profile or to save an existing Simulation Profile under a new name. For example, you might want to make changes to a profile, yet keep a copy of the profile as it existed before you modified it. By using the **Save Profile As** command, you can save modifications under a different name.

1. Select **File > Save Profile As** from the **Simulation Execution** window. The **Save Profile As** dialog box opens.



2. Enter a new profile name in the **Save Profile As** field.
3. Select **OK** to save.

Restart Simulation

The **Restart Simulation** command is used to restart the simulation at time zero.

Select **File > Restart Simulation** from the **Simulation Execution** window.

Simulation time and number of steps resets to zero

Note

If changes are made to a chart in this simulation scope, then the **Rebuild Simulation** command should be used. **Restart Simulation** does not load modifications of charts into the simulation.

Rebuild Simulation

The **Rebuild Simulation** command is used to reread any changes in the model and restart the simulation.

1. Select **File > Rebuild Simulation** from the **Simulation Execution** window. The following message displays:

   ```
   “Do You Want to Save the Simulation Environment for the Current
   Session?”
   ```

2. Select **Yes** to read in changes to the scope:

   Note: A **yes** or **no** response saves the changes to the scope. A **yes** response reads in modifications to the scope saving the simulation environment. A **no** response reads in the modifications to the scope and restores the original simulation environment losing any changes. The operation can be cancelled using the **Cancel** selection.

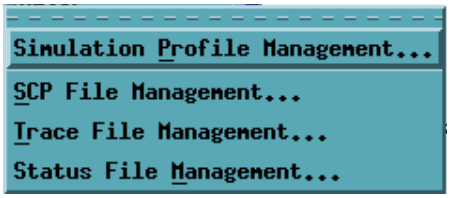
Simulation File Management

The **Simulation File Management** command is used to manage SCP, Trace, and Status Files.

1. Select **File > Simulation File Management** from the **Simulation Profile** menu.

   Note: The **Simulation File Management** command can also be accessed through the **Simulation Execution** dialog box by selecting **File > Simulation File Management**.

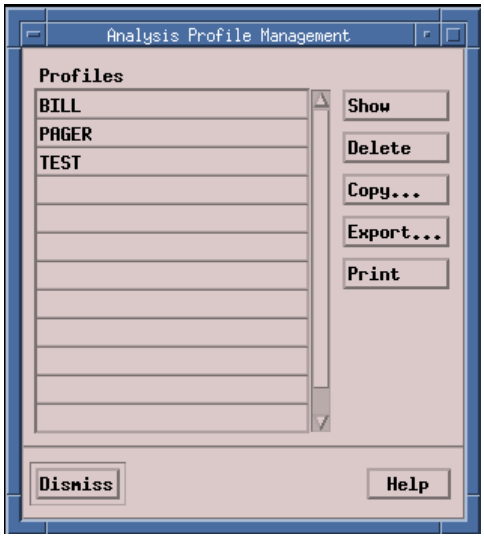
   The **Simulation File Management** dialog box **opens**.



Analysis Profile Management

The **Analysis Profile Management** command is used to display, delete, copy, export and print an Analysis Profile.

1. Select **File > Simulation File Management > Simulation Profile Management** from the **Simulation Profile Editor** or the **Simulation Execution** menu. The **Analysis Profile Management** dialog box displays.

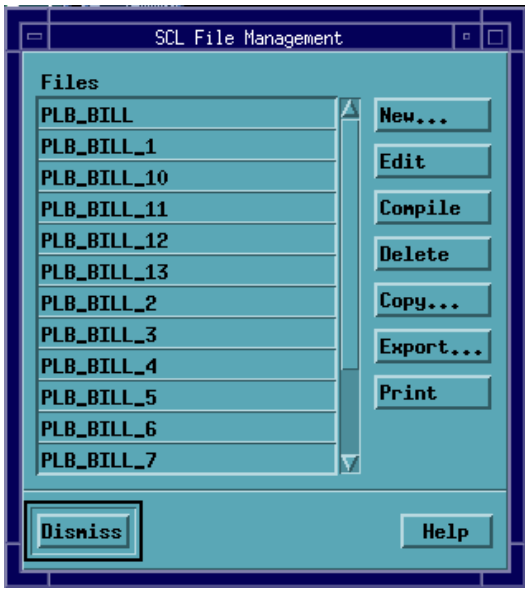


- **Show** – Displays the selected profile.
- **Delete** – Removes the selected profile.
- **Copy** – Copies the selected profile after you re-name it. It works the same way as **Save as**.
- **Export** – Works the same way as **Copy** except you can save it to another workarea or directory.
- **Print** – Used to print the selected profile.
- **Dismiss** button – The **Dismiss** button is used to close the dialog box.

SCP File Management

The **SCP File Management** command is used to display, delete, copy, export, and print an SCP File.

1. Select **File > Simulation File Management > SCP File Management** from the **Simulation Profile Editor** or the **Simulation Execution** menu. The **SCL File Management** dialog box appears.



Trace File Management

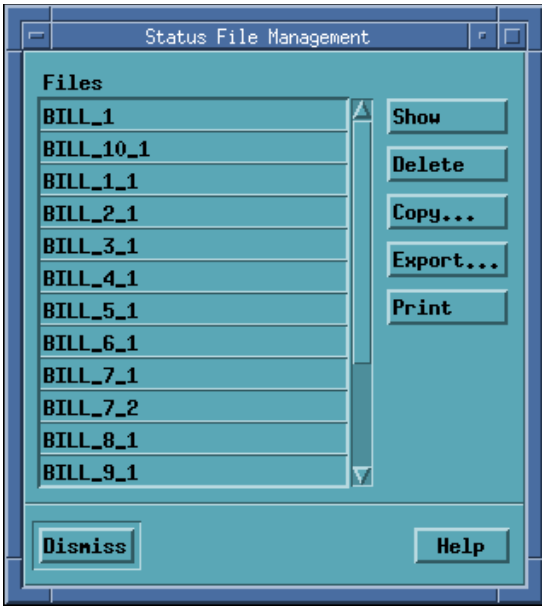
The **Trace File Management** command is used to display, delete, copy, export, and print a Trace File. It also can be used to print reports and view waveforms of a Trace File.

Select **File > Simulation File Management > Trace File Management** from the **Simulation Profile** Editor or the **Simulation Execution** menu. The **Trace File Management** dialog box displays.

Status File Management

The **Status File Management** command is used to display, delete, copy, export, and print your Status File.

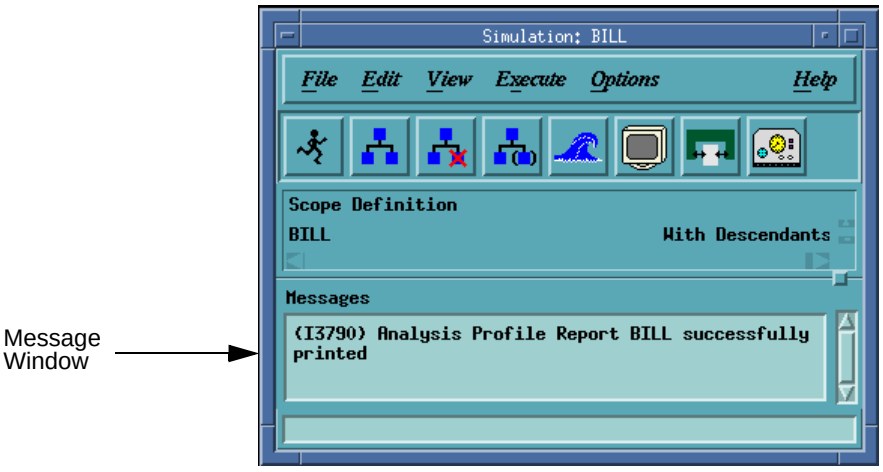
1. Select **Execute > Simulation File Management** from the **Simulation Profile Editor** of the **Simulation Execution** menu. The **Simulation Execution** dialog opens.
2. Select **File > Simulation File Management > Status File Management.** Another dialog box opens with the following commands:
   - Simulation Profile Management
   - SCP File Management
   - Trace File Management
   - Status File Management



Messages

The **Messages** command is used to open and close an area that displays messages about the status of the simulation.

Select **View > Messages** from the **Simulation Profile Editor** or **Simulation Execution** menu. The Message window opens on the Simulation Profile menu if not present. Otherwise, it disappears.



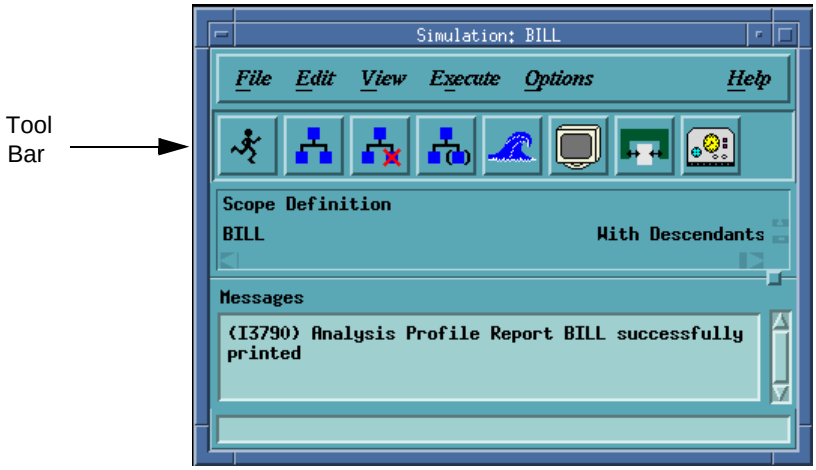
Tool Bar

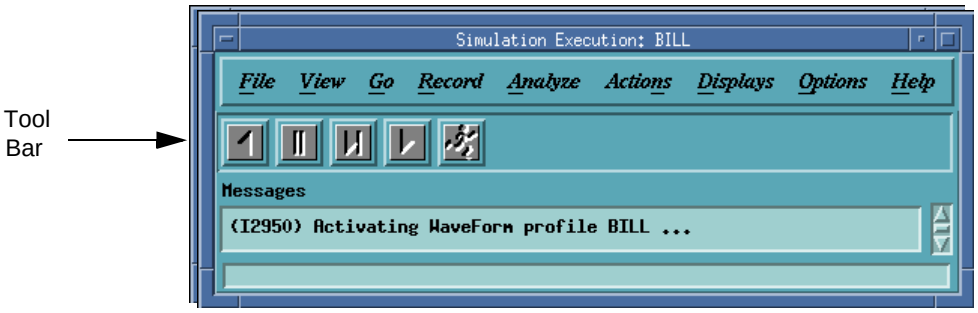
The Tool Bar command is used to open and close a tool bar containing icons that give you quick access to the most frequently used commands.

Select **View > Tool Bar** from the **Simulation Profile** menu. The Tool Bar displays on the Simulation Profile menu.

Note

The **Tool Bar** command can also be accessed through the **Simulation Execution** dialog box by selecting **View > Tool Bar.**





Command Line

The **Command Line** command is used to open and close the command line where the simulation commands can be typed.

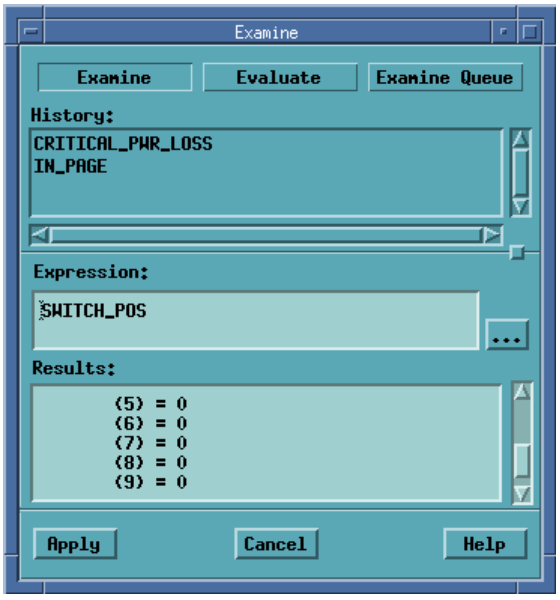
Select **View > Command Line** from the **Simulation Execution** menu.

The Command Line appears in the Simulation Execution dialog box. This command toggles the Command Line between view and hide mode.

Examine

The **Examine** command is used to display the status or value of a specified specification element or queue in the Simulation scope. It can also be used to evaluate a valid expression formed from elements in the Simulation Scope.

1. Select **Analyze > Examine** from the **Simulation Execution** menu. The **Examine** dialog box opens.



2. Select one of the following three operations: **Examine**, **Evaluate** or **Examine Queue**.

   - If **Examine** is selected, an element name can be entered it the Expression text box. The ellipse button can be used to browse valid elements (i.e., count).
   - If **Evaluate** is selected, then an expression can be entered consisting of elements in the Simulation Scope. (i.e., count, carry).
   - If **Examine Queue** is selected, then the length and the contents of each queue element is displayed.
   - Click **Apply**.

GoBack

The **GoBack** command is used to undo the previous *Go* command.

Select **Go** > **GoBack** from the **Simulation Execution** menu.

The status of all specification elements is returned to the value immediately before the most recent *Go* command. It does not effect the elements of a running SCP.

Note

- This command can be used repeatedly up to the default *GoBack* limit (the default value is 5). This can be change using the **Simulation Execution Options** command.
- Setting this option at too high a level may slow down your simulation since more simulation history must be saved.

Pause

The Pause command is used to temporarily stop a running simulation.

Select **Go** > **Pause** from the **Simulation Execution** menu.

The running SCP or AutoRun is interrupted.

AutoGo

The **AutoGo** command is used to perform a GoStep in an unstable status otherwise it performs a *GoNext*.

Select **Go** > **AutoGo** from the **Simulation Execution** menu.

A **GoStep** is performed if it causes a change; otherwise, a **GoNext** is performed.

GoStep

The **GoStep** command is used to perform a single simulation step. In the Asynchronous Time Model, the time is not advanced. In the Synchronous Time Model, the time is advanced one clock unit.

Select **Go** > **GoStep** from the **Simulation Execution** menu.  The next simulation step is executed.

AutoRun

The **AutoRun** command is used to start a mode of operating in which simulation runs continuously with the entire interaction being performed through panels and monitors.

Select **Go** > **AutoRun** from the **Simulation Execution** menu. 

In the autorun mode, simulation infinitely runs the **GoExtend** command. When a stationary situation occurs, simulation pauses. After entering the appropriate input from the panel, the simulation activates and it continues execution of the model.

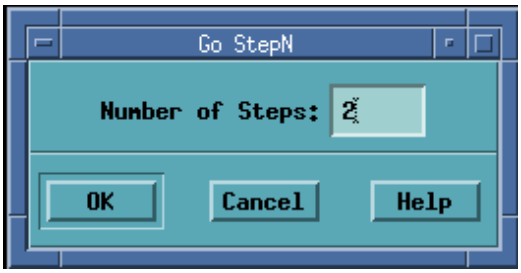
Note

The **Pause** command can be used to stop the **AutoRun** command.

GoStepN

The **GoStepN** command is used to perform N (specified number) of Simulation steps. The clock is advanced for each step in the Synchronous Time Model,.

1. Select **Go** > **GoStepN** from the **Simulation Execution** menu. The **GoStepN** dialog box opens.



2. Enter the **Number of Steps** you want to perform.

Note

GoSteps are performed until a stable configuration or the specified number of steps are reached.

GoRepeat

The **GoRepeat** command is used to execute a superstep.

Select **Go** > **GoRepeat** from the **Simulation Execution** menu.

All steps possible are executed until a stable status is reached. All steps are taken until an external event must be generated or a scheduled action or timeout event are sensed.

GoNext

The **GoNext** command is used to advance the clock to the time of the next scheduled action or timeout event and runs all reactions the system can perform before this time.

Select **Go** > **GoNext** from the **Simulation Execution** menu.

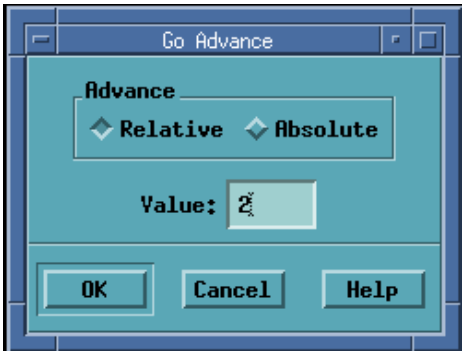
The clock is advanced to the time of the next scheduled action or timeout event.

GoAdvance

The **GoAdvance** command is used to advance the clock to the time of the next scheduled action or timeout event and runs all reactions the system can perform before this time. It advances the time to the time specified. Relative or Absolute time can be used.

Select **Go** > **GoAdvance** from the **Simulation Execution** menu.

The **GoAdvance** dialog box opens.



- **Advance Absolute** – Advances to the time unit specified.
- **Advance Relative** – Advances *N* time units from the current time.
- **Value** – This is the positive number representing the number of clock units advanced.

Go Extended

The **GoExtended** command is used to reach the next stable status without entering any external changes. It is available only in the asynchronous mode.

From the **Simulation Execution menu**, select **Go > GoExtended**.

Note

Attempts to execute a **GoRepeat**. If no steps are taken, the current time is advanced to the next scheduled action or timeout event and a **GoRepeat** is executed.

Simulation Execution Option

The **Simulation Execution Options** is used to set a number of user-specified parameters including: Number of Steps per Go, Infinite Loop Limit, GoBack Limit and Racing Options.

From the **Simulation Execution** menu, select **Options > Simulation Execution Options**.

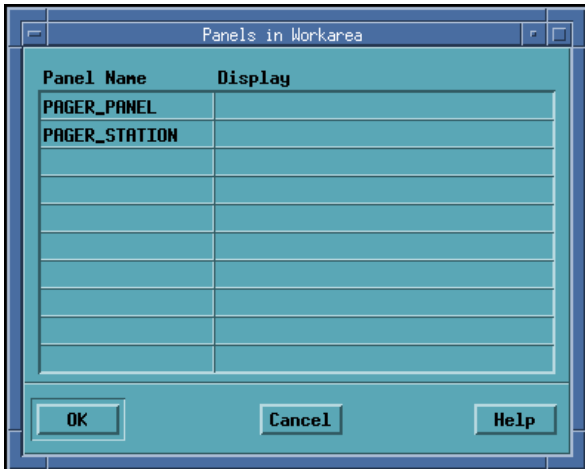
The **Execution Parameters** dialog box opens.

Panels

The **Panels** command is used to select panels to be included in the scope of simulation.

From the **Simulation Execution** menu, select **Display > Panels**.

The **Panels in Scope** dialog box opens.



From this dialog box, you can select the panels used in simulation and the terminal the panel is displayed on.

Waveforms

The **Waveform** command is used to connect a waveform to the current simulation session.

Operation

From the **Simulation Execution** menu, select **Display > Waveforms**.

The **Waveforms in Scope** dialog box opens.

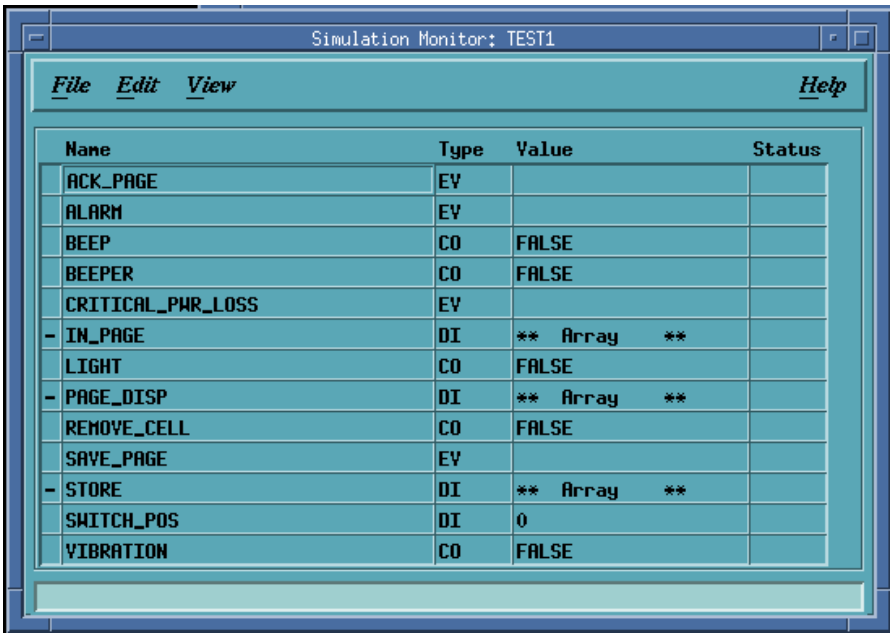
Note

For additional information on Waveforms, see **Recording a Simulation Session**.

Monitors

The **Monitors** command is used to define the monitors for the current simulation session.

From the **Simulation Execution** menu, select **Display > Monitors**. The **Simulation Monitor** dialog box opens.

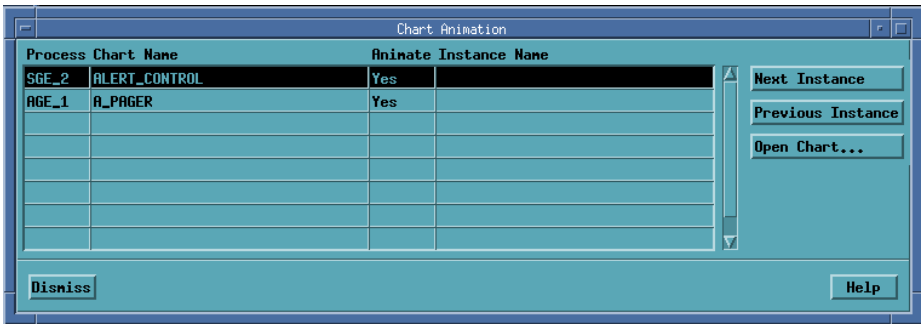


This dialog box allows you to display a simulation monitor.

Animate All Charts

The **Animate All Charts** command is used to enable and disable all chart animation.

1. From the **Simulation Execution** menu, select **Display > Animate All Charts**. The **Chart Animation** dialog box opens.



Note: The animation of a chart can be turned on or off by clicking on the Animation column of the chart.

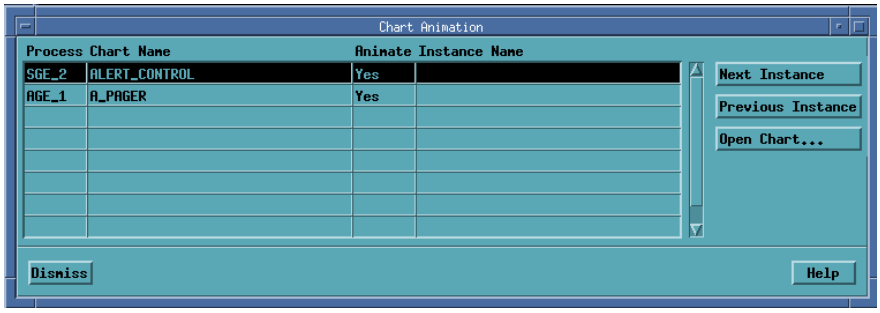
2. Charts in the Simulation Scope can be opened by selecting **Open Chart** from within the **Chart Animation** dialog box. The **Open Chart Editor** dialog box opens.

Animate Selected Charts

The **Animate Selected Charts** command is used to enable and disable individual chart animation.

From the **Simulation Execution** menu, select **Display > Animate Selected Charts.**

The **Chart Animation** dialog box opens.



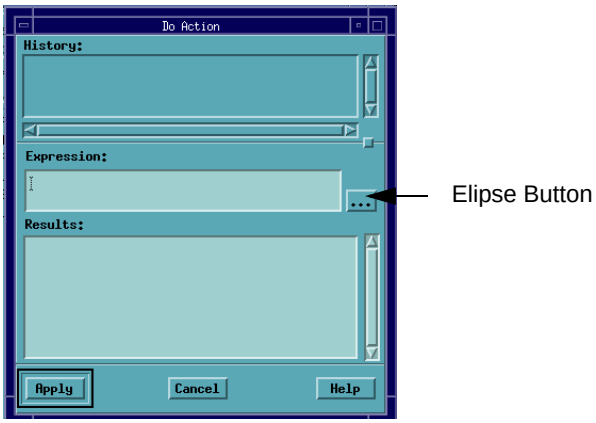
- **Process** – The Graphic Editor that is in the Simulation Scope and is being animated.
- **Chart Name** – The name of the animated Statechart.
- **Animate** – Select **Yes** or **No**.
- **Instance Name** – Animation can be turned off for each instance of a generic. To select different instances, use the *Next Instances* or *Previous Instances* selection. You can also hold the left mouse button in the Instance field of the generic chart to display a popup of the instance names.

DoAction

The **DoAction** command is used to change the value of simulation elements(s) using actions expression(s).

From the **Simulation Execution** menu, select **Actions > DoAction**.

The **DoAction** dialog box opens.



This dialog box allows you to change the value of an element based on an expression.

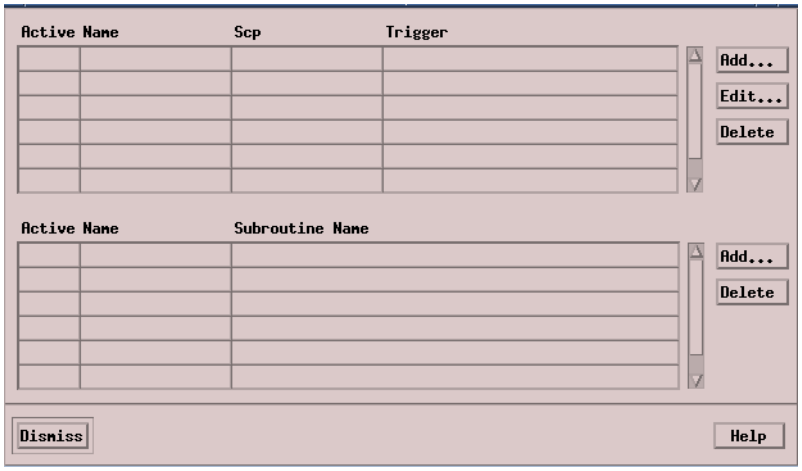
- **History** – Displays a history of previous expressions used in the *DoAction* command allowing you to select one.
- **Expression** – You can change the value of an element by entering a single action in the text box. After entering the action, apply it to perform the action.
- **Result** – The value of the element affected by the action with change.
- **Elipse Button** – Displays a browser used for selecting an element for the expression.

Breakpoints

The **Breakpoints** command is used to create, enable and disable breakpoints. For information on defining a breakpoint in a subroutine, see Defining a Breakpoint in a Subroutine.

From the **Simulation Execution** menu, select **Actions > Breakpoints**.

The **Simulation Breakpoints Editor** dialog box opens. This dialog box allows you to insert breakpoints.

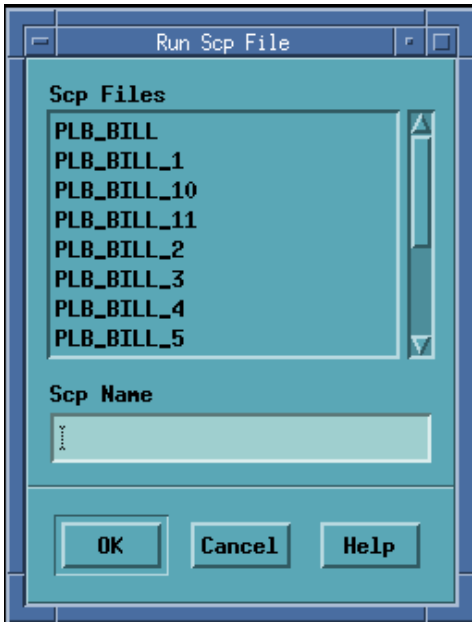


- **Active** enables and disables breakpoints.
- **Name** is the name of the breakpoint.
- **SCP** breakpoint is valid when specified SCP is running. A breakpoint associated with an interactive simulation has *USER* in the SCP field.
- **Trigger** mechanism that causes the break.
- Breakpoints can be added, edited or deleted.
- **Add** is used to add breakpoints. When selected, the **Add** dialog box opens.
- **Edit** starts a text editor so you can edit the trigger of selected breakpoints.
- **Delete** is used to remove a breakpoint.

Run SCP

The **Run SCP** command is used to start the execution of a selected SCP file(s).

1. From the **Simulation Execution** menu, select **Actions > Run SCP**. The **Run SCP File** dialog box opens.



2. From the SCP File list, select the SCP file you want to run.
3. Select **OK**. The select SCP file is executed.

Quit SCP

The **Quit SCP** command is used to stop the execution of selected executing SCP file(s).

From the **Simulation Execution** menu, select **Actions > Quit SCP**. The executed SCP File is stopped.

Continue SCP

The **Continue SCP** command is used to resume execution of an interrupted SCP file(s).

From the **Simulation Execution** menu, select **Actions > Continue SCP**.

The **SCP File** is resumed.

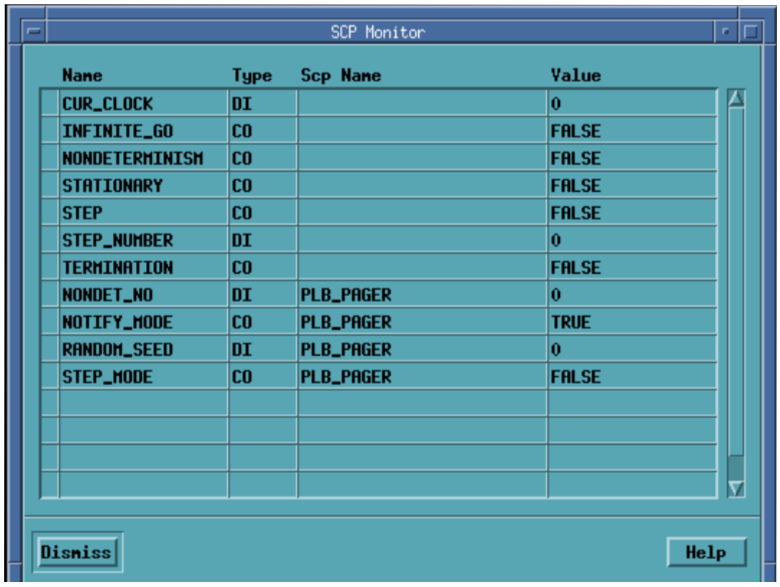
Monitor SCP

The **Monitor SCP** command is used to start a dialog box containing all current:

- User defined SCP variables
- Rational Statemate defined SCP constant/variables: NODET_NO, NOTIFY_MODE, RANDOM_SEED, STEP_MODE.
- Rational Statemate defined Simulation constant/variables: CUR_CLOCK, INFINITE GO, NONDETERMINISM, STATIONARY, STEP, STEP NUMBER, TERMINATION.

All the user defined SCP variables and Rational Statemate defined SCP constants/variables may be changed from the **SCP Monitor** dialog box. This dialog box gets its values from the SCP. It is not in continuous communication with the SCP and therefore cannot show updated values continuously.

From the **Simulation Execution** menu, select **Actions > Monitor SCP**. The **SCP Monitor** dialog box opens.



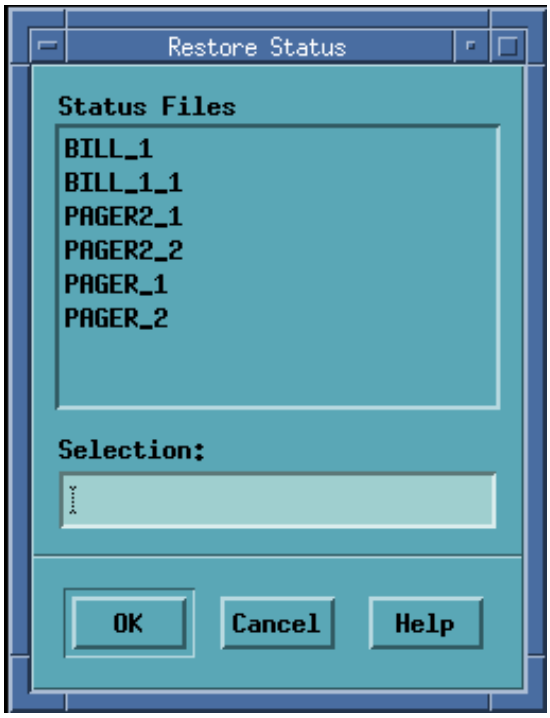
Note

The **SCP Monitor** dialog box is modal and must be closed before continuing.

Restore Status

The **Restore Status** command is used to load the saved Simulation Status file.

1. From the **Simulation Execution** menu, select **Actions > Restore Status**. The **Restore Status** dialog box opens.



2. Select a status file from the Status Files list.
3. Select **OK**. The selected status file is loaded.

Generate Interface

The **Generate Interface** command is used to generate the .c and .h files that are needed for user added code.

From the **Simulation Profile** menu, select **Execute > Generate Interface**.

Creates the header file for the C code. This is added to the prt directory of the workarea.

Note

The the **Generate Interface** command can also be accessed from the **Simulation Execution dialog box by selecting Actions > External Code > Generate Interface** from the **Simulation Profile** menu.

Start Trace

The **Start Trace** command is used to initialize the trace file and begin recording.

1. From the **Simulation Execution** menu, select **Record > Start Trace**. The **Start Trace** dialog box opens.

2. Select the desired file from the Trace Files list.
3. Click **OK**. The recording for the selected trace file is started.

Stop Trace

The **Stop Trace** command is used to close the current trace file.

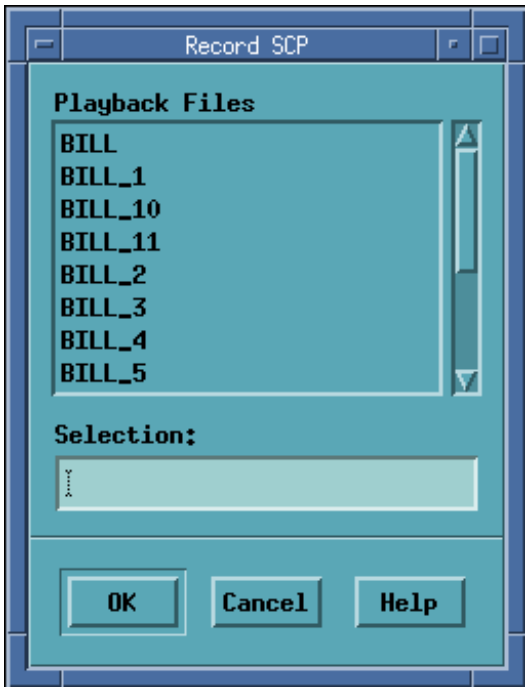
From the **Simulation Execution** menu, select **Record > Stop Trace**.

The executed trace file is closed.

Record SCP

The **Record SCP** command is used to initialize an SCP file and begin recording.

1. From the **Simulation Execution** menu, select **Record > Record SCP**. The **Record SCP** dialog box opens.



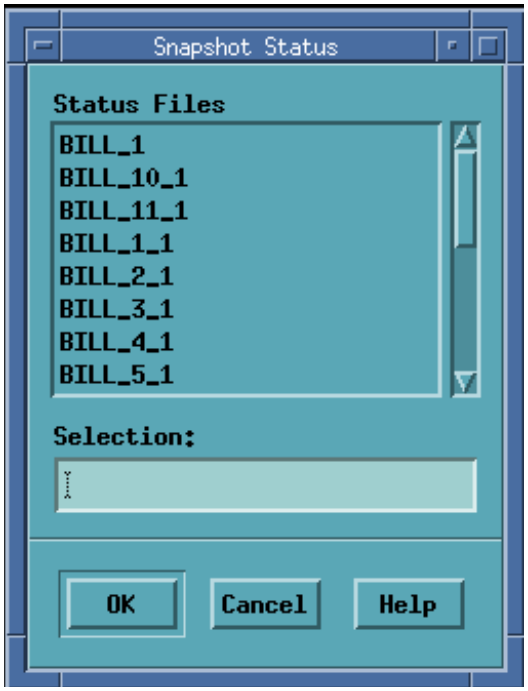
2. Select the SCP file you want to over write from the Playback Files list or enter a new name in the Selection text box.

3. Click **OK**. The SCP file is initialized and recorded.

Snapshot Status

The **Snapshot Status** command is used to save the current simulation status in a reloadable file.

1. From the **Simulation Execution** menu, select **Record > Snapshot Status**. The **Save Status** dialog box opens.



2. Select the status file you want to save from the Status Files list or enter a new name in the Selection text box.

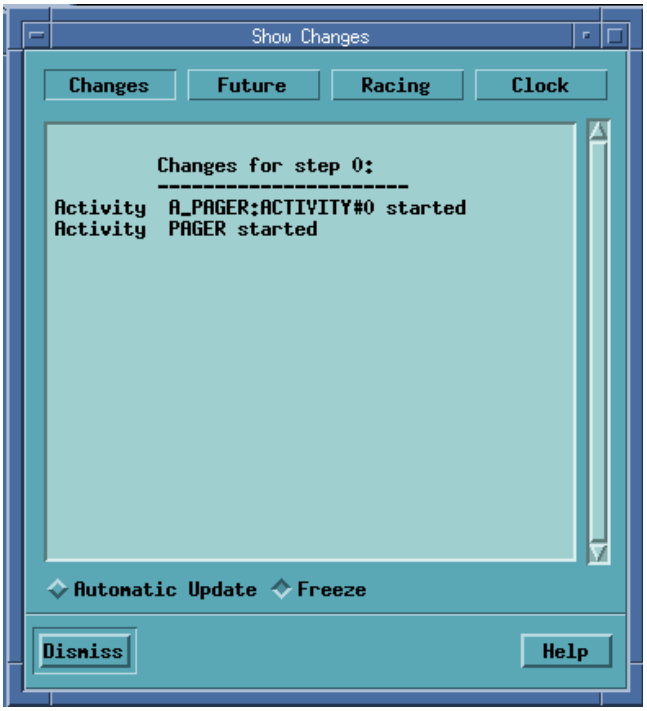
3. Click **OK**. The current simulation status is save in a reloadable file.

Show Changes

The **Show Changes** command is used to displays the changes in the system since the last **GO** command.

1. From the **Simulation Execution** menu, select **Analyze > Show**. The **Show Changes** dialog box opens. You can display Changes, Future, Racing and Clock. You can also choose between Automatic Update, or the Freeze option.

2. Select the **Change** button. The **Show** dialog box opens.



General Changes - Lists the changes in textual elements that occurred during the last go. Lists the changes that occurred in states (entered, exited) and activities (active, nonactive, hanging) during the last go.

Note: The changes are listed according to the order in which the modifications took place:

*** Conditions**: **"**became true" or "became false

*** Data-items**: "changed", "Changed from x to y", "was read" or "was written"

*** Events**: included in list if they were generated

*** Activity**: "started", "stopped", "suspended", "resumed"

*** State**: "entered," "exited".

If automatic update is selected, the dialog box updates as changes occur. Freeze retains the current output in the dialog box.

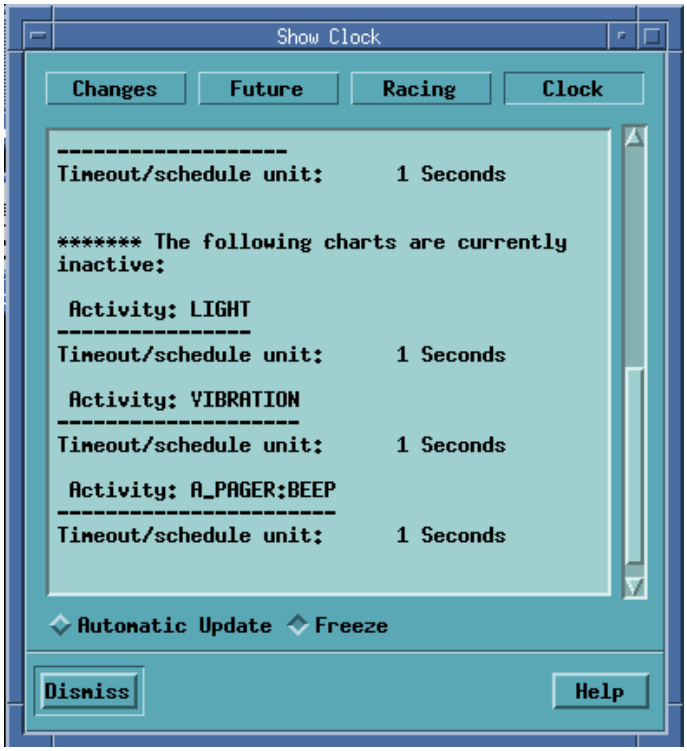
Show Clock

The **Show Clocks** command is used to display the time information for the global and local clocks.

1. From the **Simulation Execution** menu, select **Analyze > Show**. The **Show Changes** dialog box opens. You can display Changes, Future, Racing and Clock. You can also choose between Automatic Update, or the Freeze option.
2. Click **Clock**. The **Show Clock** dialog box opens.

   - **Clock Unit Name and Value** - Summary of information supplied during Simulation setup
   - **Elapsed Time** - Number of clock units passed since start of Simulation
   - **Absolute Time** - Starting time plus elapsed time
   - **Step Number** - Total number of steps taken from start of Simulation
   - **Phase Number** - Number of steps taken at the current time

If automatic update is selected, the dialog box updates as changes occur. Freeze retains the current output in the dialog box.

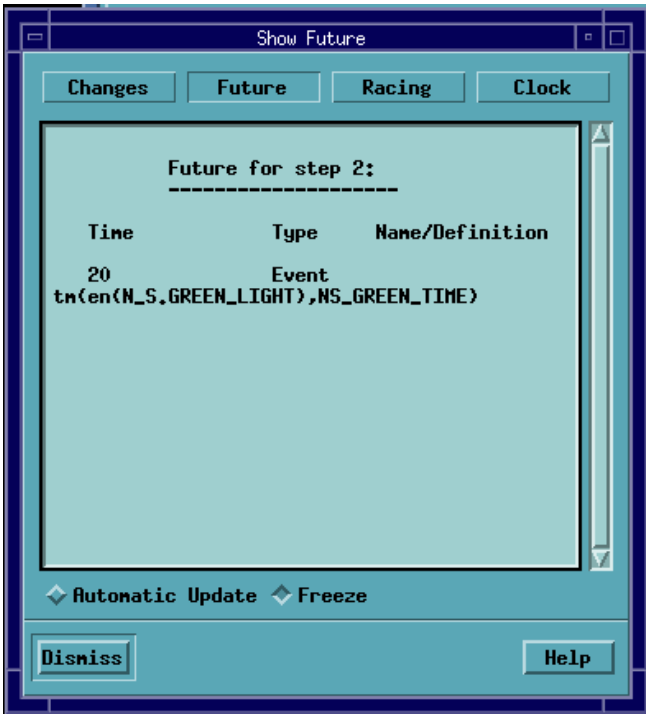


Show Future

The **Show Future** command is used to display a list of scheduled actions, timeout events and the SCL every clauses.

1. From the **Simulation Execution** menu, select **Analyze > Show**. The **Show Changes** dialog box opens. You can display Changes, Future, Racing, and Clock. You can also choose between Automatic Update or the Freeze option.
2. Select the **Future** button. The **Show Future** dialog box opens.
   - **Time** - The amount of time (global clock units) until the scheduling of the item (event, action, EVERY clause). If the value is zero, the item is activated just prior to the next step.
   - **Type** - The type of scheduled item (event, action, EVERY clause)
   - **Name/Definition** - The name of the scheduled item.

If automatic update is selected, the dialog box updates as changes occur. Freeze retains the current output in the dialog box.



Show Racing

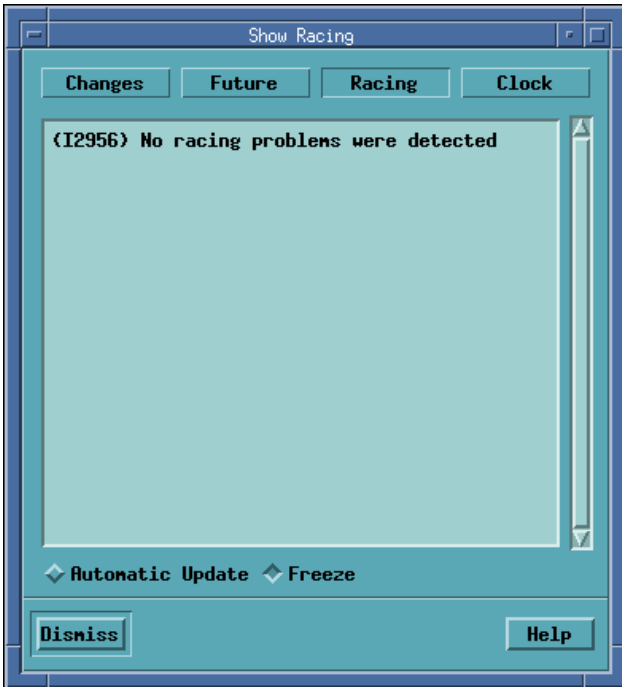
The **Show Racing** command is used to report on racing problems detected during the last *Go* command.

1. From the **Simulation Execution** menu, select **Analyze > Show**. The **Show Changes** dialog box opens. You can display Changes, Future, Racing, and Clock. You can also choose between Automatic Update or the Freeze option.
2. Click **Racing**. The **Show Racing** dialog box opens.

Note

The report is only available if the **Report Racing** parameter is set in **Set Parameters.**

If automatic update is selected, the dialog box updates as changes occur. Freeze retains the current output in the dialog box.



New Profile

The **New Profile** command is used to create a new Simulation Profile.

1. Select **File > New Profile** from the **Simulation Profile Editor**. The **New Simulation** dialog box opens.
2. Type the name of the new profile into the **Profile Name** text box or select a profile from the **Profiles** list.
3. Select **OK**. A new profile is created.

Open Profile

The **Open Profile** command is used to open an existing Simulation Profile.

1. From the **Simulation Profile Editor**, select **File > Open Profile**. The **Open Simulation Profile** dialog box opens.
2. Select a **profile** from the **Profiles** list.
3. Select **OK**. The selected profile is opened.

Close

The C**lose** command is used to close the current Simulation Profile.

From the **Simulation Profile** menu, select **File > Close Profile**.

The opened profile is closed.

Print Profile Report

The **Print Profile Report** command is used to print the current Simulation Profile.

From the **Simulation Profile Editor**, select **File > Print Profile Report**.

A report of the profile is printed.

Add With Descendants

The **Add With Descendants** command is used to add the selected chart from the **Workarea Browser** with decendents to the Simulation Profile.

From the **Simulation Profile** menu, select **Edit > Add With Decendants** or click **Add with Decendants** . 

The decendants of the selected chart in the Workarea Browser are added to the profile.

Add Testbench

The **Add Testbench** command is used to add the selected Statechart from the **Workarea Browser** to the Simulation Profile as a Testbench.

From the **Simulation Profile** menu, select **Edit > Add Testbench** or click **Add Testbench** . 

The selected Statechart is added to the profile as a Testbench.

Add/Edit Panel

The **Add/Edit Panel** command is used to add the selected Panel from the **Workarea Browser** to the Simulation Profile.

From the **Simulation Profile** menu, select **Edit > Add/Edit Panel** *or* click **Add/Edit Panel**.. 

The selected Panel is added to the profile.

Add/Create Waveform

The **Add/Create Waveform** command is used to add selected Waveform Profiles to the Simulation Profile.

 From the **Simulation Profile** menu, select **Edit > Add/Create Waveform** *or* click **Add/Create Waveform**.

If a Waveform Profile is selected in the connected Workarea Browser then it is added to the Simulation Scope. If a waveform is not selected, a dialog box opens that allows you to create a new waveform profile that is added to the scope.

Monitors

The **Monitors** command is used to define a new or edit an existing Monitor definition.

Select **Edit > Monitors** from the **Simulation Profile** menu, *or* click **Monitors**  .

The Monitors dialog box opens.

The **Monitors** dialog box is used for:

- Creating a new Monitor
- Editing a Monitor
- Deleting a Monitor

Remove From Scope

The **Remove From Scope** command is used to remove the select elements from the scope.

From the **Simulation Profile** menu, select **Edit > Remove From Scope** or click **Remove From Scope** The selected component is removed from the scope.

Exclude From Scope

The **Exclude From Scope** command is used to exclude the selected Activity from the Simulation Scope.

From the **Simulation Profile** menu, select **Edit > Exclude From Scope** or click E**xclude From Scope**. The selected component is removed from the scope.

Select

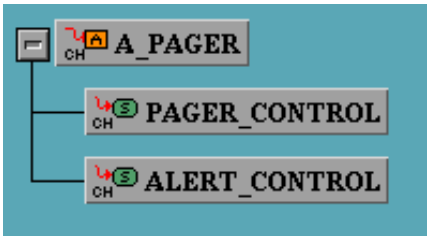
The **Select** command is used to select or deselect all elements in the current profile.

From the **Simulation Profile Editor**, select **Edit > Select**. The Select All and Deselect All commands appear.

Show Scope as Tree

The **Show Scope as Tree** is used to display the scope definition as a tree.

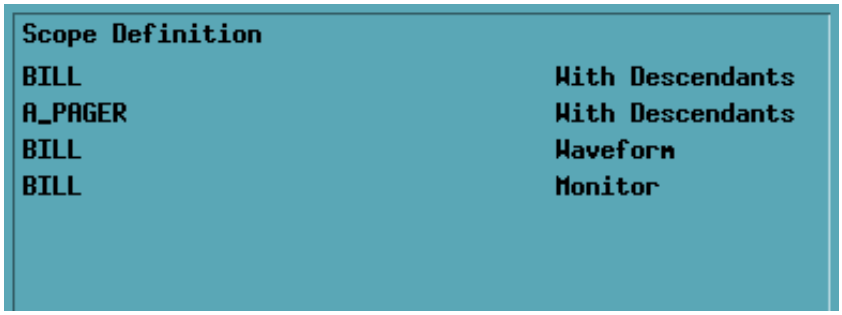
From the **Simulation Profile** menu, select **View > Show Scope as Tree.** It displays the selected chart as a tree.



Show Scope as List

The **Show Scope as List** command is used to display the scope definition as a list.

From the **Simulation Profile** menu, select **View > Show Scope as List.** It displays the selected chart as a list.



Show Boxes

The **Show Boxes** command is used to show the hierarchy of charts and boxes in the tree view.

From the **Simulation Profile** menu, select **View > Show Boxes**. It displays the hierarchy of charts and boxes in the tree view.

Hide Boxes

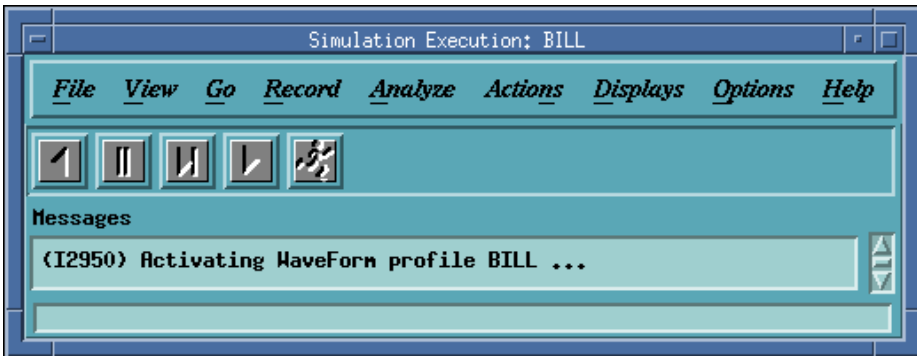
The **Hide Boxes** command is used to remove boxes and charts from the hierarchy in the tree view.

From the **Simulation Profile** menu, select **View > Hide Boxes**. Selected charts and boxes in the tree view are removed from view.

Execute Simulation

The **Execution Simulation** command is used to start a simulation based on the current profile.

From the **Simulation Profile** menu, select **Execution > Execution Simulation**. The **Execution Simulation** menu displays.



Note

The **Execution Simulation** menu can be executed by clicking **Execution Simulation**.

Simulation Execution Options

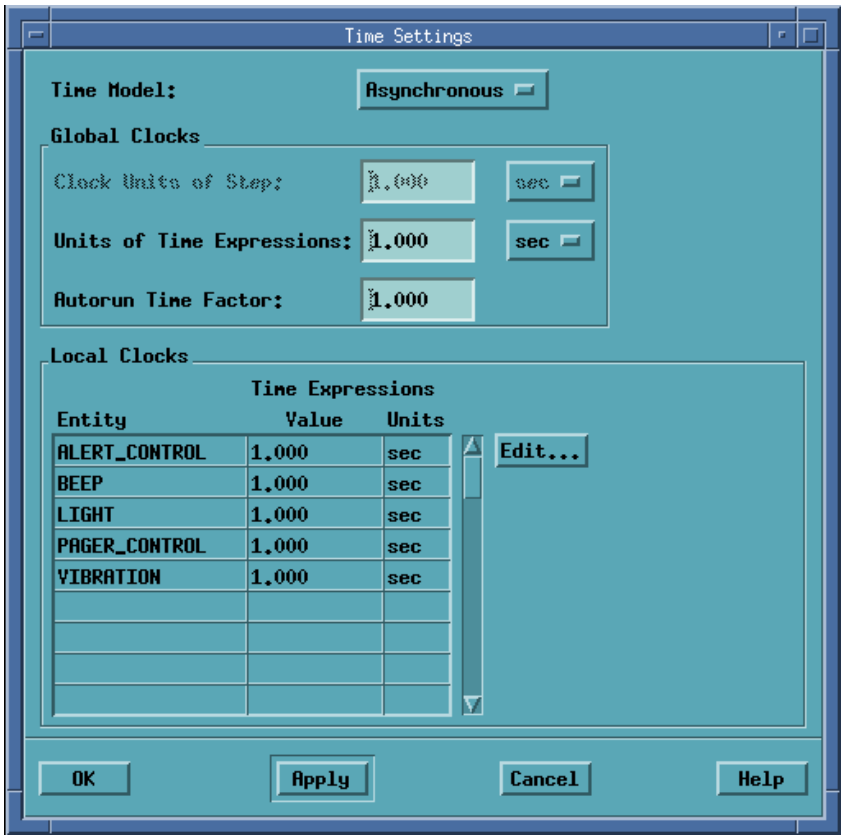
The **Simulation Execution Options** is used to define Activity Styles, Step Limits and Racing Notification.

From the **Simulation Profile** menu, select **Options > Simulation Execution Options**. The **Execution Parameters** dialog box opens.

Time Settings

The **Time Settings** command is used to select Time Settings and to define Clock Units.

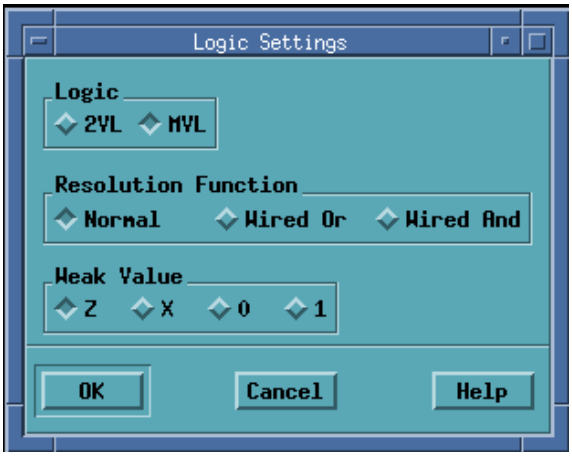
From the **Simulation Profile** menu, select **Options > Time Settings**. The **Time Settings** dialog box opens.


Time Settings
Time Model:
Asynchronous
Global Clocks
Clock Units of Step:
1.000
sec
Units of Time Expressions:
1.000
sec
Autorun Time Factor:
1.000
Local Clocks
Time Expressions
Entity
Value
Units
ALERT_CONTROL
1.000
sec
BEEP
1.000
sec
LIGHT
1.000
sec
PAGER_CONTROL
1.000
sec
VIBRATION
1.000
sec
Edit...
OK
Apply
Cancel
Help

Logic Settings

The **Logic Settings** command is used to select Multi-Value Logic, Resolution and Weak Values.

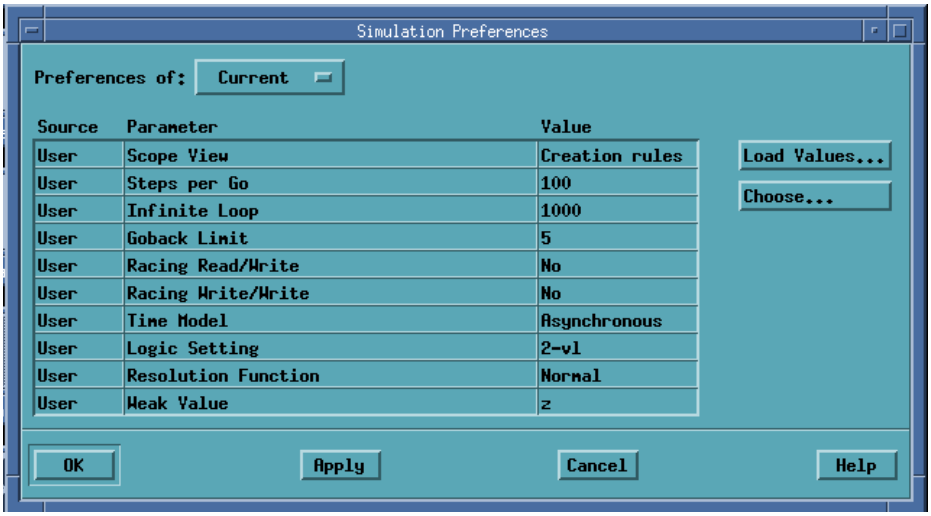
From the **Simulation Profile** menu, select **Options > Logic Settings**. The **Logic Settings** dialog box opens.



Preference Management

The **Preference Management** command is used to change the Simulation Profile Editor preferences.

From the **Simulation Profile Editor,** select **Options > Preference Management.** The **Simulation Preferences** dialog box opens.



Auto Batch Commands

Many of these commands have interactive equivalents of the same syntax. Additional details on how to use these commands can be found in **Batch Mode Simulation.**

ASSIGN

The **ASSIGN** statement is used to assign SCP files to represent primitive activities or external activities. The **ASSIGN** statement connects an SCP to an activity to represent the activity's behavior in the simulation model. When an internal activity is started, the program assigned to the activity is started. Correspondingly, stopping, suspending or resuming of these activities causes appropriate changes in the status of the program.

Syntax:

Assign activity_name "scp_name" where `activity_name` is the primitive or external activity name and `scp_name` is the SCP file name.

CANCEL

The **CANCEL** statement is used to cancel the setting of a simulation parameter or breakpoint.

Syntax:

Used in conjunction with `set breakpoint, set display, set trace`. See these commands.

CHOOSE

The **CHOOSE** statement is used in nondeterministic situations to choose a solution.

Syntax:

The **CHOOSE** statement is used to choose the integer where *integer* is the assigned number of the desired nondeterminism solution.

When nondeterminism is encountered and interactive mode enabled, the Simulation tool displays the first possible solution. Use **next** to inspect other solutions. Use **choose** to select the displayed solution. A **go resume** is performed after choose.

Example:

```
choose 3
```

CLOSE

The **CLOSE** statement is used to close an opened file.

Syntax:

CLOSE (file_variable) where `file_variable` is the variable of the opened file.

This statement closes an already opened file for the duration of the simulation run. It is primarily used to compensate for system limitations pertaining to the number of files open simultaneously.

Example:

```
close(file2)
```

COMMENT

The **COMMENT** statement specifies a comment line in an SCP.

Syntax:

```
// text
```

where *text* is any text string. Any text following "//" to the end of the line is a comment.

Note

Comments may be embedded within statements or on lines to themselves or you can use `/*<text>*/`. With this usage, text can be any string and can include new lines.

Example:

```
read(v,z) ; // This is a valid comment
// This is another valid comment line
```

CONSTANT

The **CONSTANT** statement is used to declare the program constants in the SCP file section.

Syntax:

constant

```
    [INTEGER id :=integer_val [, id := integer_val . . .] ; ]
where id is the name of the integer constant and integer_val is its value
    [STRING id := "text" [, id := "ext" . . . ] ; ]
where id is the name of the string constant and "text" is its value
    [FLOAT id := real_val [, id:=real_val . . . ] ; ]
where id is the name of the real constant and real_val is its value
    [BIT id:=bit_val]
where id is the name of the bit constant and bit_val is 0 or 1
    [ARRAY id(bound1..bound2):=array_val]
```

where id is the name of the bit-array, bound represents the bit boundaries and array_val is the value of the bit array.

The Constant Section if one of the five optional program sections of the SCP. The Section contains the keyword constant followed by the declarations of constants. One declaration may define several constants (separated by commas). Each declaration is concluded with a semicolon.

Example:

```
Constant
      integer x:=5, y:=200, z:=1, z2:=10 ;
      string alpha := "unexpected loop"
             beta:= "enter an action for yy" ;
      float  a:= 2.5, b:=700.234 ;
      array a(1..16):=0xAE07 ;
```

DO

The **DO** statement is a component of *Set Breakpoint* which defines a sequence of statements executed when the breakpoint is triggered.

Syntax:

See **SET BREAKPOINTS**.

ELSE

The **ELSE** statement is a component of the `IF/THEN/ELSE` statement used for the conditional execution of SCL statements.

Syntax:

See the **IF** statement.

END

The **END** statement is use to end a SCP section, structured statements and the SCP itself.

Syntax:

```
END BREAKPOINT

END IF

END INIT

END LOOP

END WHEN

END

END.
```

The **end** keyword concludes various SCP structured statements: Set Breakpoint, INIT, BEGIN, WHILE/LOOP, WHEN/THEN/ELSE and IF/THEN/ELSE.

The **End** statement concluding the entire SCP is followed by a period.

EVERY

The **EVERY** statement is the component of the Set Breakpoint statement used for setting a breakpoint at specific time intervals.

Syntax:

See **SET BREAKPOINTS**.

EXEC

The **EXEC** statement runs a specified SCP.

Syntax:

```
exec "name"
```

where `name` is the name of the SCP file to be executed.

Multiple SCPs may be executed simultaneously. An SCP may be **executed** and **stopped** from any other SCP. When an SCP is started by an **exec** statement, its Main Section is ignored.

Note

You should not exec SCPs that are assigned. exec does not affect the execution of already running SCPs.

Rational Statemate Actions

The **STATEMATE ACTIONS** statement performs Rational Statemate actions.

Syntax:

```
exv_action
```

where `exv_action` is any legal Rational Statemate action

Actions may refer to both Rational Statemate elements and SCL objects.

Note

If the scope contains several elements with an identical name, precede its name with the name of the chart.

Example:

```
card_id := 12345 ; start(verification) ;
if a>b then tr!(c) end if
```

AUTOGO

The **AUTOGO** command is used to perform a `GoStep` in an unstable status otherwise it performs a `GoNext`.

Syntax:

```
Auto Go
```

GO ADVANCE

The **GO ADVANCE** statement advances the clock to a specified moment and runs all reactions the system can perform before this moment.

Syntax:

```
go advance num
```

where `num` is a positive number representing the number of clock units.

GO BACK

The **GO BACK** statement undoes the previous GO command.

Syntax:

```
go back
```

GO EXTENDED

The **GO EXTENDED** statement attempts to execute a `go repeat`. If no steps are taken, the current time is advanced to the next scheduled action or timeout event and a `go repeat` is executed. Available only in the Asynchronous Time Model.

Syntax:

```
go extended
```

GO NEXT

The **GO NEXT** statement advances the clock to the time of the next scheduled action or timeout event and runs all reactions the system can perform before this time.

Syntax:

```
go next
```

GO REPEAT

The **GO REPEAT** statement runs all steps possible to the next stable status. It performs a superstep.

Syntax:

```
go repeat
```

GO STEP

The **GO STEP** statement is used to performs a single step. Time is advanced in the Synchronous Time Model.

Syntax:

```
go step
```

GO STEPn

The **GO STEP**n statement is used to perform a specified number of steps to advances the clock. Available only in the Synchronous Time Model.

Syntax:

```
go stepn [n]
```

IF

The **IF** statement is used to perform a conditional execution of SCL statements.

Syntax:

```
if condition then
       statement [ ; statement . . . ]
[else
       statement [ ; statement . . . ]
]
end if
```

where `condition` is any expression returning a Boolean value and `statement` is any SCL statement

The `IF/THEN/ELSE` structured statement is used to execute SCL statements conditionally. The statements following `THEN` and before `ELSE` are executed if `condition` is true. If `condition` is false, the statements between `ELSE` and `END` are executed.

Example:

```
IF a > b THEN
       err := err + 1 ;
ELSE
       WRITE (‘a is less than b’)
END IF
```

INIT

The **INIT** statement is used to define statements that are executed at the beginning of each execution of the SCP.

Syntax:

```
init
statement
. . .
[; statement ]
end init
```

Contains any SCL statements except **GO** statements.

Example:

```
INIT
      zb := 23.6 ;
      gen := 4000 ;
      rname := “light standard” ;
      SET INFINITE LOOP 50 ;
END INIT
```

LOOP

The **LOOP** statement is a component of the **WHILE** statement used to execute SCL statements in a loop.

Syntax:

See **WHILE** statement.

MAIN SECTION

The **MAIN SECTION** is used as a Sequence of SCL statements used to define the simulation scenario.

Syntax:

```
begin
statement
[; statement ]
. . .
end
```

The Main Section contains any SCL statements which are executed sequentially. When more than one SCP is running, only one Main Section is executed - the one belonging to the SCP activated with the interactive `RUN` command.

If the primary SCP has no explicit Main Section, a default main section is assumed: a `Go Extended` is performed in an infinite loop.

OPEN

The **OPEN** statement is used to open a file for input or output.

Syntax:

```
open ( file_variable, “file_name”, INPUT | OUTPUT)
```

where `file_variable` is assigned to a `file_name` for `INPUT` or `OUTPUT`.

The following rules apply:

- The file being opened must already exist
- If a file is opened for output, it cannot be reopened before it is closed
- The same file cannot be opened for both input and output

Example:

```
open (file1, “/csw/source/sample.data”, INPUT)
```

PROGRAM

The **PROGRAM** statement is used to name the SCP.

Syntax:

program `program_name` where `program_name` is the name assigned to the SCP.

This statement is required as the first statement in the SCP.

Example:

```
PROGRAM ATM_control
```

RANDOM SOLUTION

The **RANDOM SOLUTION** statement is used to select a random solution when nondeterminism occurs.

Syntax:

```
random_solution
```

This statement is used in conjunction with the nondeterminism breakpoint. It allows the model simulation to continue without user intervention when a nondeterministic situation is encountered. One of the solutions is chosen randomly and the simulation proceeds without the need for a `Go` statement.

Example:

```
set breakpoint [ nondeterminism ]
do
       random_solution ;
       write (“nondeterministic situation solved randomly. \n”)
end breakpoint
```

READ

The **READ** statement is used to read input from an external file or from standard input (keyboard) and assign it to specified variables and data-items.

Syntax:

```
read ( [ file_variable, ] x1 [, x2 . . . ])
```

where `file_variable,` if present, is the name of the external file; if absent, input is read from the keyboard to `x1`.

The information read is assigned to SCP variables of all types (except file) and primitive data-items.

Note: The file from which the read is performed must be opened before this statement is used. When multiple variables are read from the keyboard or a file, the values must be separated by a return.

Example:

```
read (file_name, a,b,c) ;
read (v,z) ;
```

RESTORE STATUS

The **RESTORE STATUS** statement is used to restore the status information saved in a save status operation.

Syntax:

```
restore_status “status_name”
```

where *status_name* is the name of the status to be restored.

SAVE STATUS

The **SAVE STATUS** statement is used to save the current status of the system at a requested point in the simulation.

Syntax:

```
save_status “status_name”
```

where *status_name* is the name under which the status is saved.

SET BREAKPOINTS

The **SET BREAKPOINTS** statement is used to define and enable a specified breakpoint.

Syntax:

```
set breakpoint [ breakpoint => ] trigger do
statement
[; statement ]
. . .
end breakpoint
```

where *trigger* is defined as: **event_expression | every num_expression**; and *statement* is any SCL statement except go

```
set breakpoint breakpoint_name
cancel breakpoint breakpoint_name
```

See **Breakpoints** for additional information on breakpoints.

SET DISPLAY

The **SET DISPLAY** statement is used to enable or disable the display of changes in graphic editors.

Syntax:

```
set display
cancel display
```

Enables or disables the animation of graphic charts which are connected to the Simulation tool.

SET GO BACK

The **SET GO BACK** statement is used to determine the maximum number of times a go back can be executed in succession.

Syntax:

```
set go back number
```

where *number* is a positive integer

Example:

```
set go back 5
```

SET INFINITE GO

The **SET INFINITE GO** is used to set a limit to the number of steps taken during a long `Go` command (i.e., `GoExtended, GoRepeat`). After a preset number of steps is reached (if not finished before), simulation stops and control is returned to the user. This parameter is also used to limit the number of iterations within a step used to calculate the combinational element value. For example, in the conditional expression X:=X+1, a stable value for X can never be reached because each time X changes the CE is recalculated resulting in a new value for X. When calculating a combinational element, if a stable state is not reached after 100 mini-steps (default), the calculating stops and the value remains at 100 mini-steps.

Syntax:

```
set infinite GO 100
```

where *n* is a positive integer

This statement permits the resetting of the maximum number of steps which can be executed during a long Go command.

Example:

```
set infinite go 5
```

SET INFINITE LOOP

The **SET INFINITE LOOP** is used to set a limit for the number of interactions of FOR loops or WHILE loops in the simulation. After reaching the preset limit, the simulation is terminated and reported.

Syntax:

```
set infinite loop phase_limit
```

where `phase_limit` is a positive integer

This statement permits the resetting of the maximum number of iterations that can be executed in a loop before the infinite loop condition becomes true.

Example:

```
set infinite loop 1000
```

SET INTERACTIVE

The **SET INTERACTIVE** statement is used to switch from batch to interactive mode.

Syntax:

```
set interactive
```

All running SCPs are suspended. To resume the SCPs and return to batch mode, enter the `Continue SCP` command.

SET TRACE

The **SET TRACE** statement is used to enable or disable the recording of the simulation results into a trace file.

Syntax:

```
set trace
cancel trace
```

This statement determines whether the results of the subsequent simulation steps are recorded in a trace file. If the simulation session is given a name, that name is given to the trace file, otherwise, the trace file name is nameless.

SKIP

The **SKIP** statement is used to skip the remainder of breakpoint processing stage in the current step.

Syntax:

```
SKIP
```

STOP SCP

The **STOP SCP** statement is used to stop the execution of selected executing SCPs.

Syntax:

```
stop_scp [ “name”]
```

where *name* is the name of the SCP to be stopped.

If name if provided, this SCP is stopped. This statement without an SCP name stops all running SCPs.

Example:

```
stop_scp “autolight”
```

THEN

The **THEN** statement is a component of the *IF/THEN/ELSE* and *WHEN/THEN/ELSE* statements used for conditional execution of statements depending, respectively, on a condition value or an event occurrence.

Syntax:

```
see IF and WHEN
```

VARIABLE

The **VARIABLE** statement is the SCP file section used to define variables.

Syntax:

```
variable
   [ [ global] integer id [:=integer_val] [, id := integer_val . . .] ; ]
```

where `id` is the name of the integer constant and `integer_val` is its initial value

```
   [[global] string id [:= “ext” [, id := “text”. . . ] ; ]
```

where `id` is the name of the string constant and *“text”* is its initial value

```
   [[global] float id [:= real_val] [, id:=real_val . . . ] ; ]
```

where `id` is the name of the real constant and `real_val` is its initial value

```
   [[global] file id [ ,id . . . ] ; ]
```

where `id` is the name of the file variable

```
   [[global] BOOLEAN id [, id . . . ] ; ]
```

where `id` is the name of the Boolean variable

WHEN

The **WHEN** statement is used for conditional execution of SCL statements depending on event occurrence.

Syntax:

```
when trigger then
       statement [ ; statement . . . ]
[else
       statement [ ; statement . . . ]
]
end when
```

where `trigger` is any event expression and `statement` is any SCL statement

The `WHEN/THEN/ELSE` structured statement is used to execute SCL statements when a particular event occurs. The statements following `THEN` and before `ELSE` are executed if the *trigger* is true. If the *trigger* is false, the statements between `ELSE` and `END` are executed.

Example:

```
WHEN tr(c) THEN
       err := err + 1;
ELSE
       WRITE (“Error Encountered”)
END WHEN
```

WHILE

The **WHILE** statement is used to execute SCL statements in a loop.

Syntax:

```
while condition
loop
   statement [ ; statement . . . ]
end loop
```

where `condition` is any Boolean expression and Statement is any SCL statement.

The `WHILE/LOOP` statement is used to execute SCL statements in a loop. The condition is any Boolean expression. While the condition is true, the statements in the loop are performed repeatedly. The condition is rechecked prior to each execution of the loop.

There is no limit to the depth of structured statements within the loop.

Example:

```
WHILE cax
      LOOP
            a1; a2
            if x = 3, then fs!(cax)
            else
            write(“not tripped”);
            while Cb or cq
            loop
            a3
            end loop
            end if
      end loop
```

WRITE

The **WRITE** statement is used to output the simulation data to either the display or a file.

Syntax:

```
write ( [file_variable], exp1 [, exp2 . . . ])
```

where `file_variable`, if present, is the name of the output file. If absent, output of `exp1` is sent to the display.

The output may be a combination of printable strings and numeric values.

Example:

```
write ( 'The data value is', d1,'\n')
write (file2, ax, by, '\n')
```

Supplementing the Model with Handwritten Code

This section explains how to supplement Rational Statemate simulation with handwritten code. Not only does this code become part of the simulation, but it is also included as part of generated code.

Rational Statemate enables you to extend the Rational Statemate model by supplementing the model with handwritten code. This means that you can implement those elements and aspects of the system's behavior that have not been explicitly defined by the controlling Statecharts and mini-specs.

You may want to use this feature to:

- Describe a particular function programmatically.
- Interface to your own or a third party's library.
- Use code that already exists.

There are several ways to supplement the generated code:

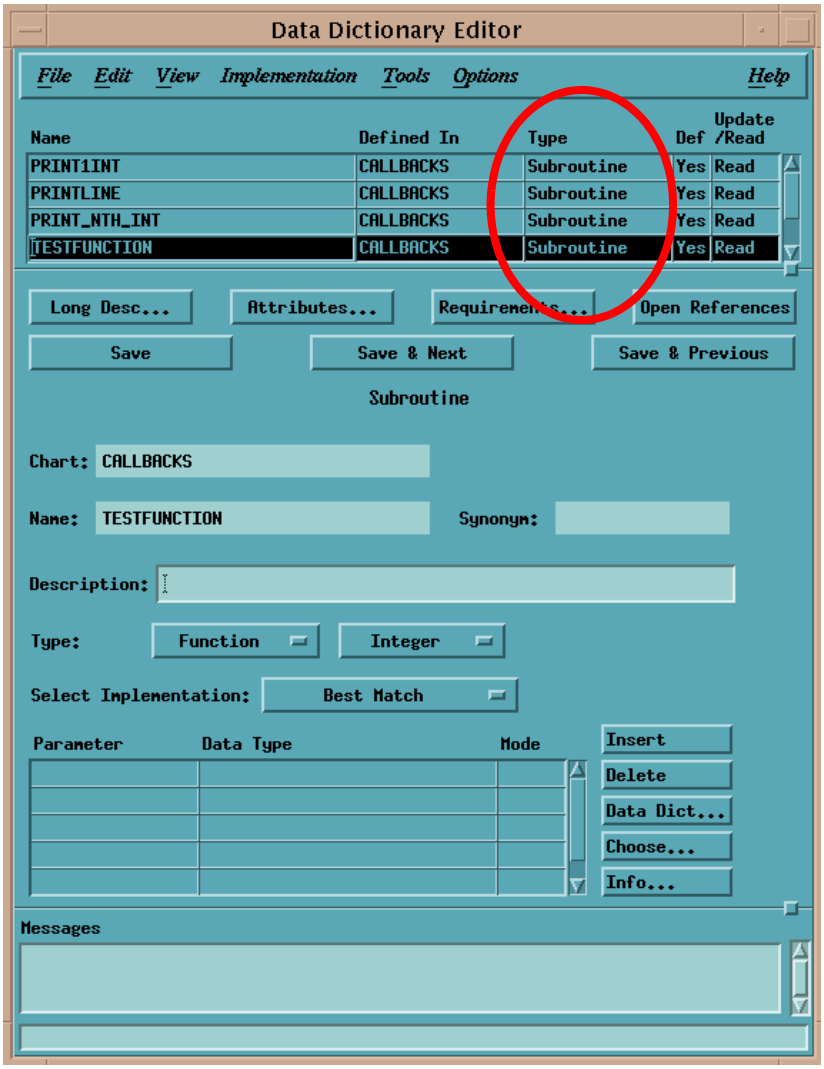
- Attach existing code to the model through the Data Dictionary Editor and select one or more languages in which to implement it (K&R C, ANSI C, or Ada).
- Write new code directly in Rational Statemate using the Rational Statemate Action Language.
- Use a graphic to define a function or procedure in a Procedural Statechart.
- Create a Truth Table to implement a subroutine, define a "named action," or describe an activity's behavior.

These methods enable you to add code that is used by both the Simulator and the Code Generator. Rational Statemate stores the code in the model's database and automatically includes it when you run simulation or code generation.

Supplementing the Model with Subroutines

The following subsections explain how to add handwritten subroutines (functions, procedures, or tasks) to your Rational Statemate model.

The method for adding all three subroutines in the Data Dictionary Editor (DDE) is similar. The major difference is that functions require a Return Type.



Note

In addition to storing subroutines in the Data Dictionary Editor, you can also store their formal parameters.

Entering Handwritten Code

Rational Statemate does not check your handwritten code. It is your responsibility to ensure that the code is legal and compilable. You can use **with**, **use**, **include** statements or any other mechanism supported by the language to reference packages or include files. Rational Statemate makes no attempt to interpret the code; it merely passes it on to the appropriate compiler.

To add your handwritten code to the template correctly, make sure you abide by the rules in the following sections:

1. Referencing model elements in the code.
2. Mapping Rational Statemate types (primitive or user-defined) into C types for variables and subroutine parameters.
3. Using synchronization services in tasks.

Using Subroutines

After you define a subroutine in the Data Dictionary, it becomes part of Rational Statemate and is stored as part of the model. Then you can use the subroutine in the following ways:

- Called in Rational Statemate actions and expressions.
- Bound to a primitive activity of the modeled system, thus providing their implementation.
- Bound to an external activity to describe behavior of the environment.
- Bound as a callback to a textual or graphical element in the model, and called when the element changes its value or status.

Disabling Subroutines

To disable a subroutine, open the Data Dictionary Editor and under **Select Implementation**, select **None**.

Rational Statemate does not implement the subroutine, and only generates a template (empty stub).

Supplementing the Model with a Procedure

This section explains how to add a handwritten procedure to your Rational Statemate model by showing the

- Dialog boxes and how to complete them
- Template that Rational Statemate produces
- Template filled in with an example of handwritten code

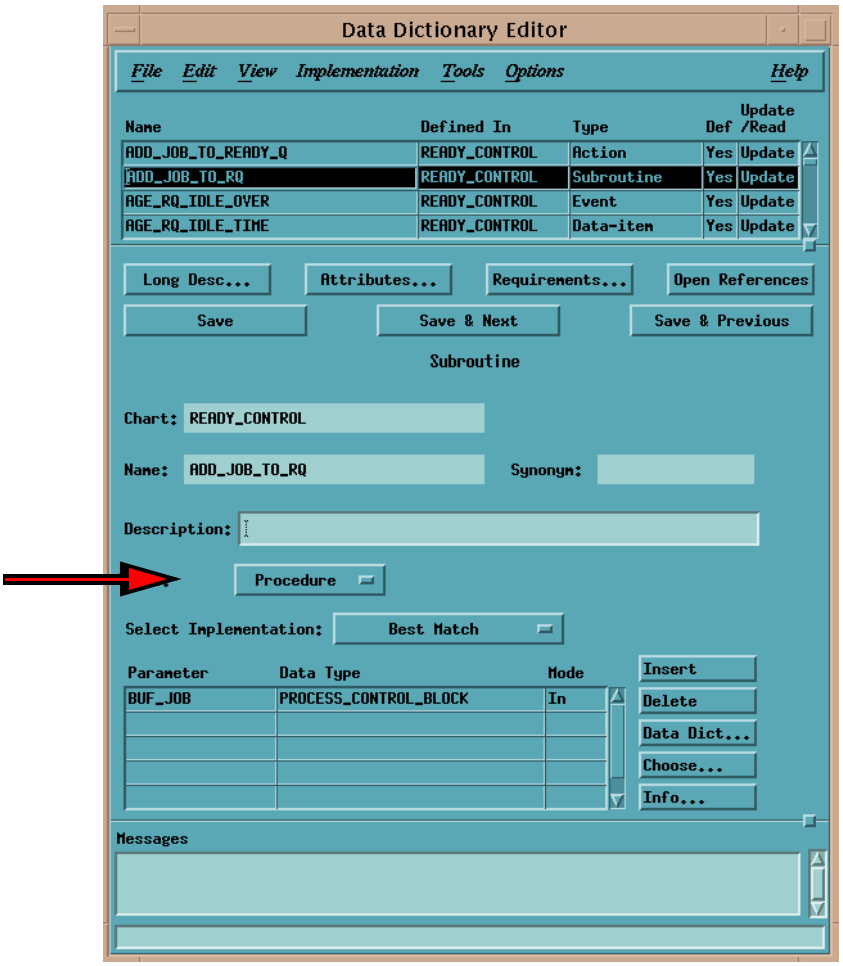
Note

Rational Statemate also provides templates for functions and tasks. The subroutine's template is a result of mapping the declarations into its C representation. This includes mapping the parameter types and, in the case of functions, the returned value.

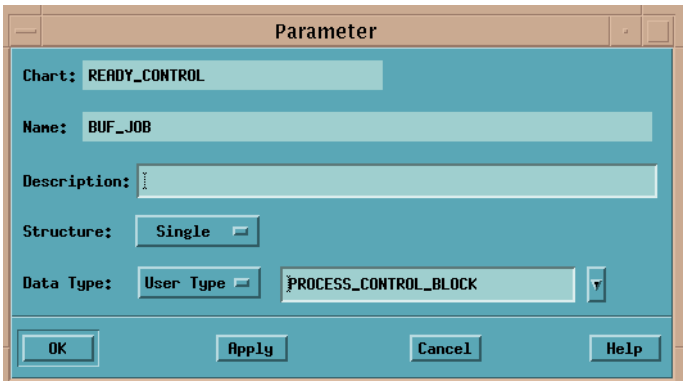
To add a handwritten procedure:

1. Select **File > New** in the Data Dictionary Editor.
2. Name the new element (in this example **ADD_JOB_TO_RQ**), then select its Chart Name.
3. Select **Subroutine** as the Element Type. The Data Dictionary Editor dialog box opens with the name of the new subroutine.

4. Define the subroutine Type as a **Procedure**

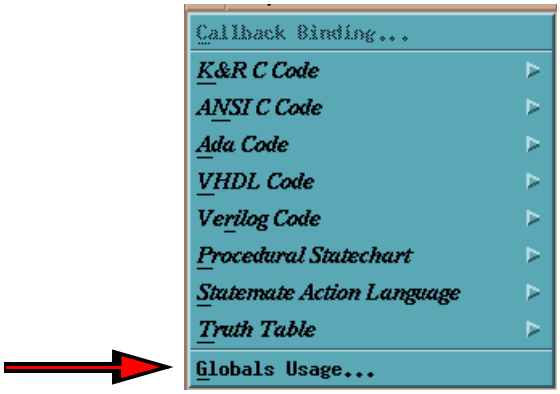


5. Enter the procedure's parameters if you want to store them in the DDE. Select a parameter and click Data Dict to display the following dialog box:



Using Globals

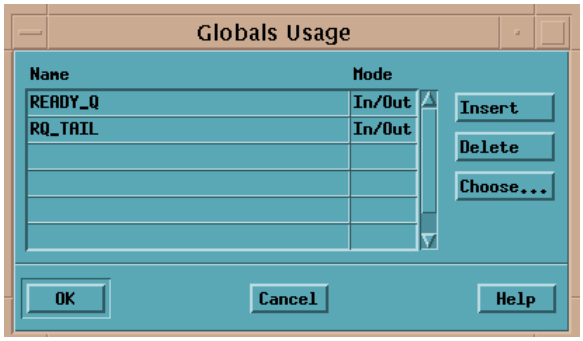
If you use the same parameters for multiple activities, you may want to define them as globals. If so, use the **Implementation** menu (shown below) to select **Globals Usage**.



For example, the **A**DD_JOB_TO_RQ procedure uses the globals shown in the following dialog box.

Note

Globals are elements that are external to the subroutine, but are not listed as parameters. The reading or writing of global data is called a side effect.



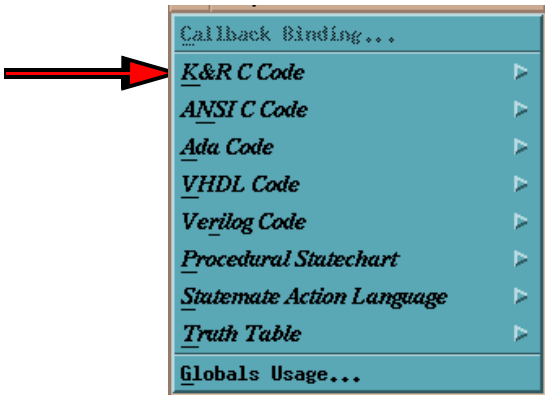
Writing more than once to a global element is considered *racing*. However, this racing differs from general racing where you have no way of determining which value will be assigned. In this case, the final value will be the resulting value of the global element. Therefore, it is your responsibility to ensure that the subroutine writes to global elements only a single time during its execution.

Note

It is strongly recommended that you do not write global data in a function called in a trigger expression. Side effects written as part of a trigger will behave differently between simulation and code.

Producing a Template for a Procedure

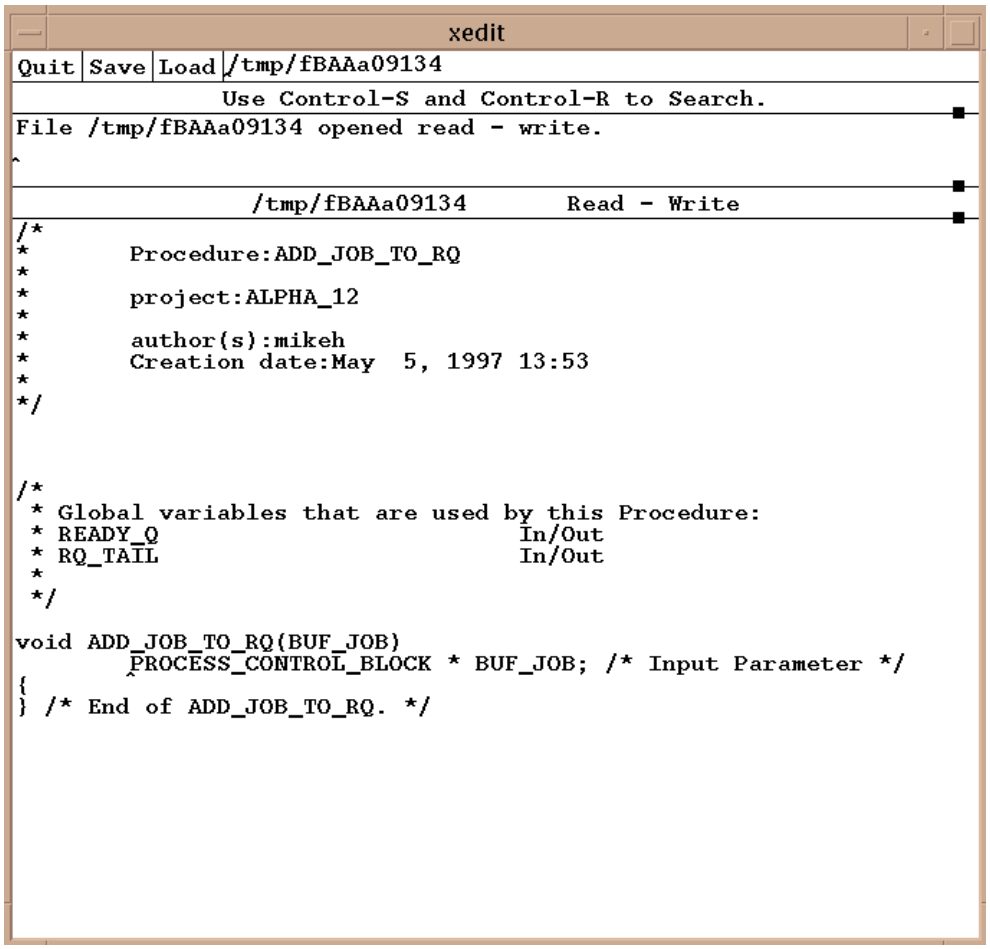
Tto produce a template for a procedure, open the **Implementation** menu to select a language for the code. This example uses K&R C.



Rational Statemate opens an editor and provides a template for you to attach your handwritten code

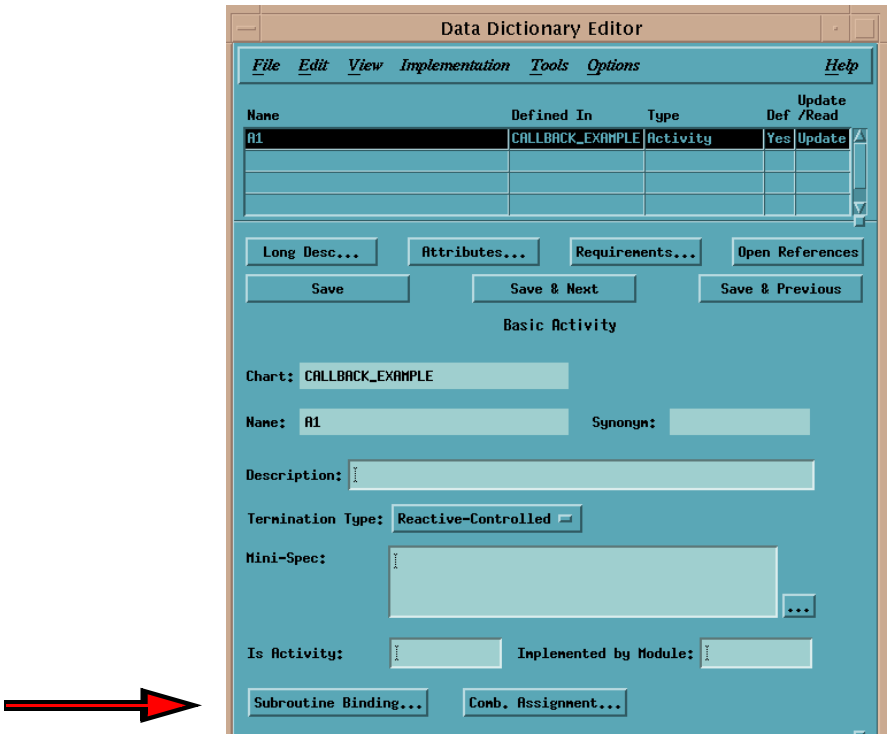
Filling in the Procedure's Template

The following example shows the template filled in with handwritten code for a complete procedure.

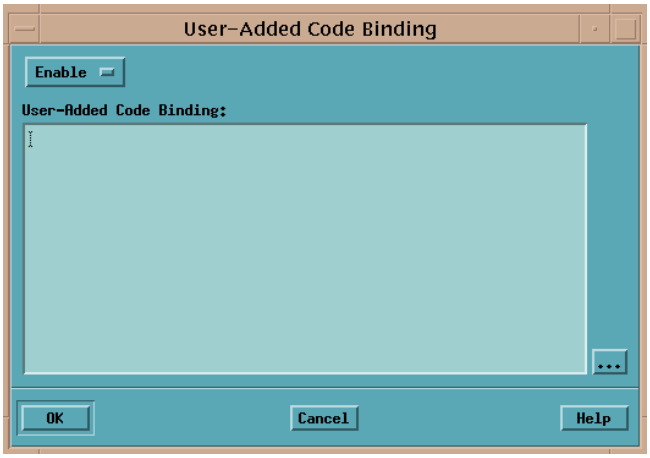


Subroutine Binding

Open the Data Dictionary Editor for an activity and select **Subroutine Binding** to connect subroutines.



The **User-Added Code Binding** dialog appears where you enter the name of the subroutine, which is to be bound to the activity.



Supplementing the Model with a Task

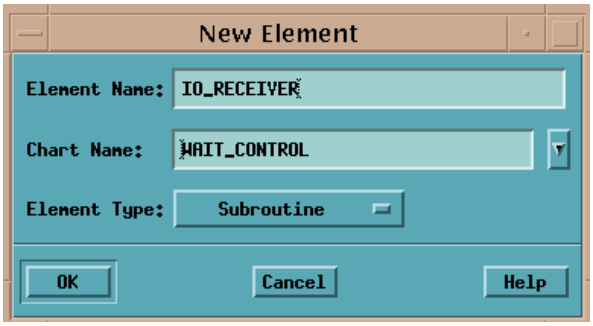
This subsection explains how to add a handwritten task to your Rational Statemate model by showing the

- Dialog boxes and how to complete them
- Template that the Code Generator produces
- Template filled in with an example of handwritten code

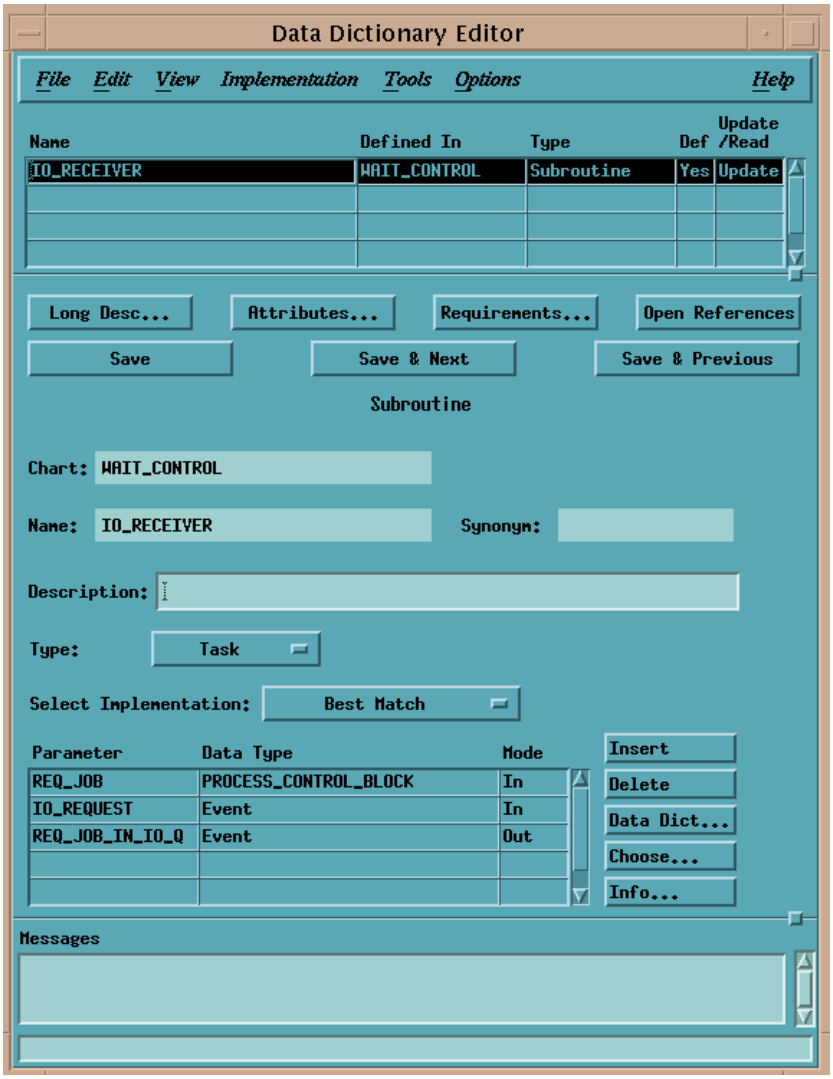
Rational Statemate also provides templates for functions and procedures. The subroutine's template is a result of mapping the declarations into its C representation. This includes mapping the parameter types and, in the case of functions, the returned value.

To add a handwritten task:

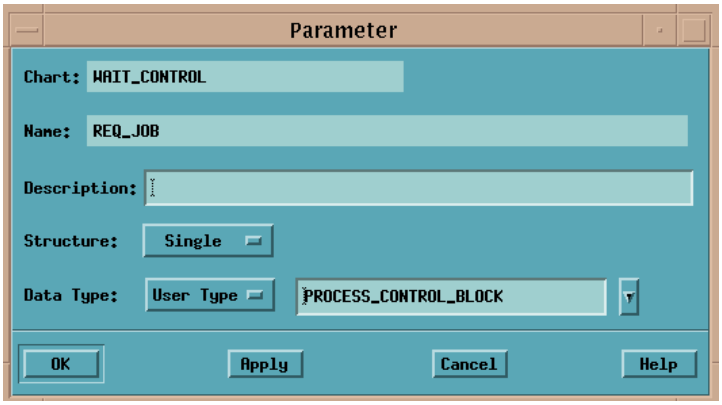
1. Select **File > New** in the Data Dictionary Editor. The New Element dialog box opens.



2. Enter the name of the new element (in this example `IO_RECEIVER`).
3. Select its Chart Name.
4. Select **Subroutine** as the Element Type.
5. Click **OK**. The Data Dictionary Editor appears with the name of the new subroutine.

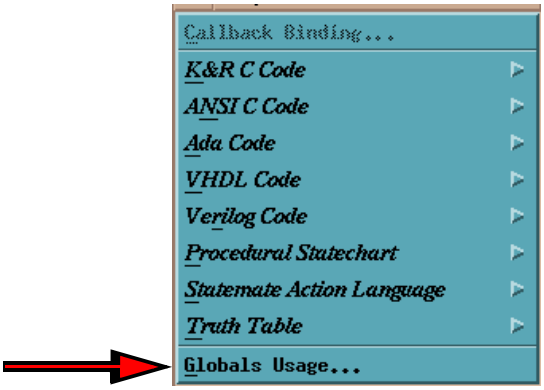


6. Define the subroutine Type as a **Task.**
7. Enter the task's parameters if you want to store them in the DDE.
8. Select a parameter and click **Data Dict.** The Parameter dialog box opens.



Using Globals

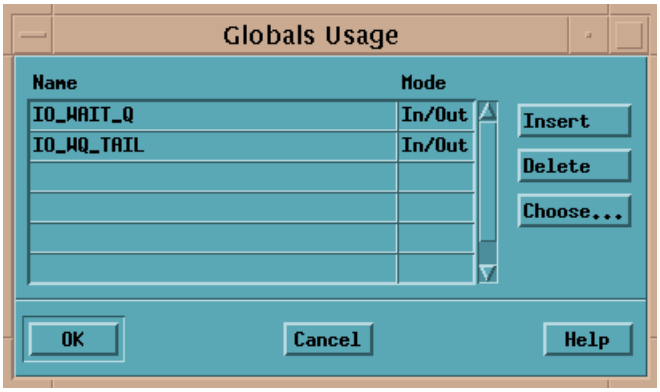
If you use the same parameters for multiple activities, you may want to define them as globals. If so, use the **Implementation** menu (shown in the following dialog box) to select **Globals Usage**.



For example, the `IO_RECEIVER` task uses the globals shown in the following dialog box.

Note

Globals are elements that are external to the subroutine, but are not listed as parameters. The reading or writing of global data is called a **side effect**.



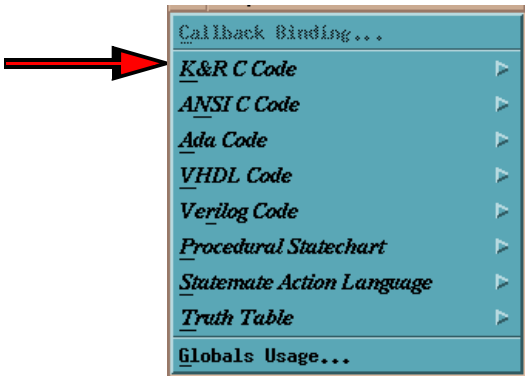
Writing more than once to a global element is considered racing. However, this racing differs from general racing where you have no way of determining which value will be assigned. In this case, the final value will be the resulting value of the global element. Therefore, it is your responsibility to ensure that the subroutine writes to global elements only a single time during its execution.

Note

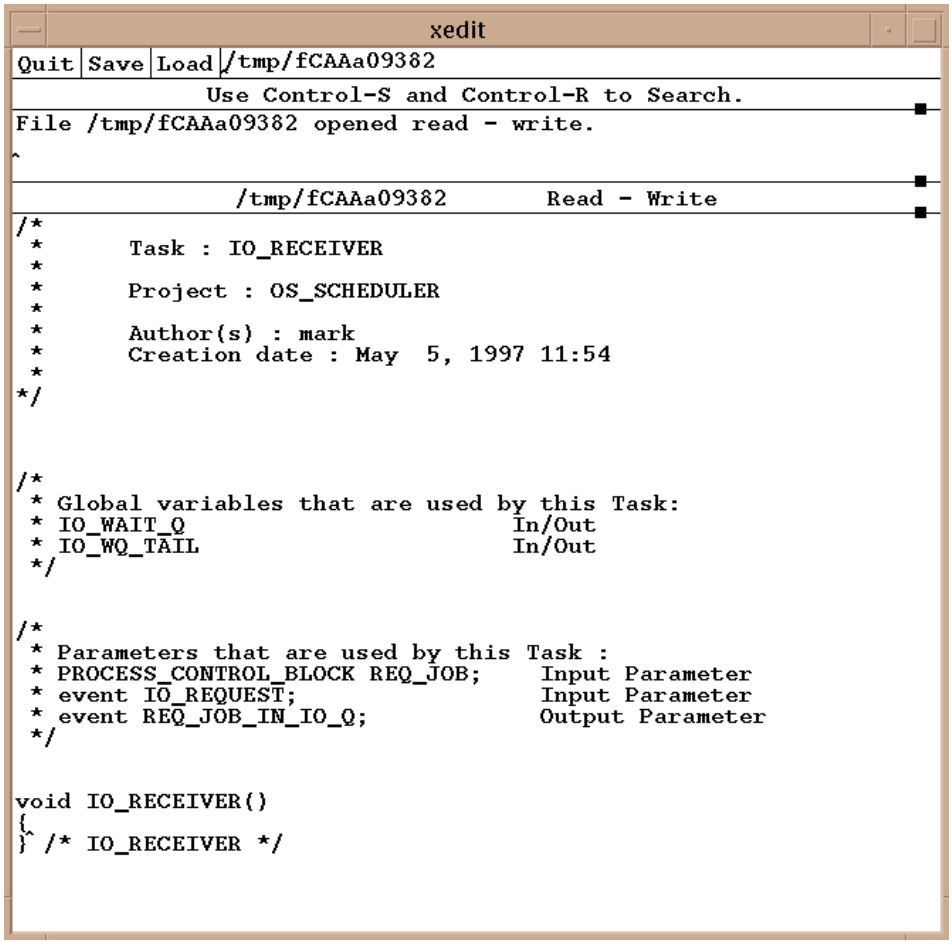
It is strongly recommended that you do not write global data in a function called in a trigger expression. Side effects written as part of a trigger will behave differently between simulation and code.

Using the Template for a Task

To produce a template for a task, open the **Implementation** menu to select a language for the code. This example uses K&R C.

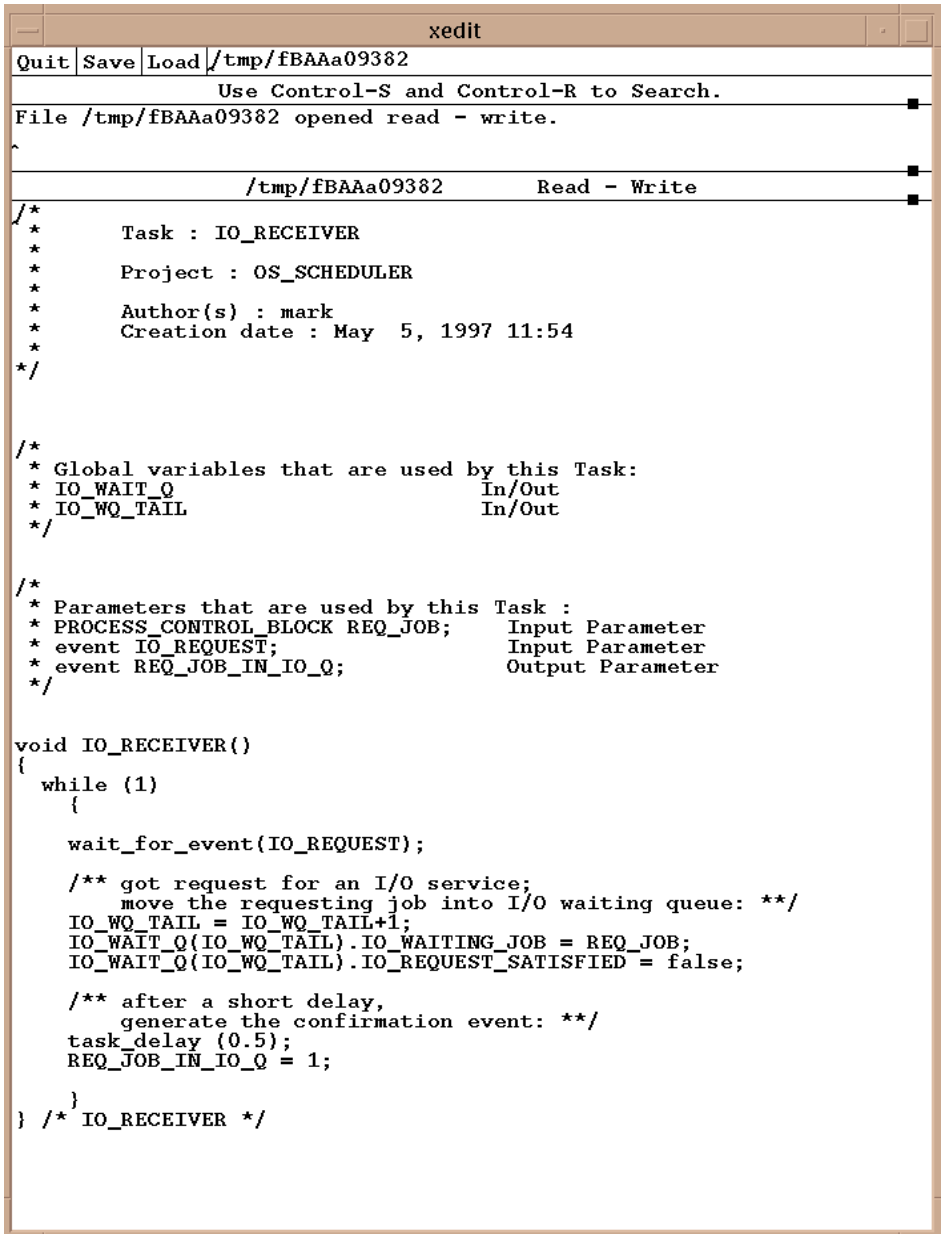


Rational Statemate opens an editor and provides a template for you to attach your handwritten code.


xedit
Quit
Save
Load
/tmp/fCAAa09382
Use Control-S and Control-R to Search.
File /tmp/fCAAa09382 opened read - write.
/tmp/fCAAa09382 Read - Write
/*
* Task : IO_RECEIVER
*
* Project : OS_SCHEDULER
*
* Author(s) : mark
* Creation date : May 5, 1997 11:54
*
*/
/*
* Global variables that are used by this Task:
* IO_WAIT_Q In/Out
* IO_WQ_TAIL In/Out
*/
/*
* Parameters that are used by this Task :
* PROCESS_CONTROL_BLOCK REQ_JOB; Input Parameter
* event IO_REQUEST; Input Parameter
* event REQ_JOB_IN_IO_Q; Output Parameter
*/
void IO_RECEIVER()
{
} /* IO_RECEIVER */

Filling in the Task's Template

The following example shows the template filled in with handwritten code for a complete task.



Synchronizing Tasks

User-written procedures are called when the system starts the corresponding activity (i.e., `st!(<activity>)`). In general, the user code and the simulation share the CPU time. That is, when the user code is executed, the Rational Statemate simulation (or other user activities) are suspended.

Tasks

The task mechanism allows you to integrate continuous or synchronized code into the primitive activity. For this purpose, Rational Statemate provides a special library that extends the C language to support tasking or multi-threading. (See the Scheduler section below, for details). Tasks can also be bound to either a primitive or an external activity.

The scheduler package allows you to define C functions as concurrent routines or co-routines. An activity that you choose to implement as a task is started by the control code as a co-routine, which is executed concurrently with the rest of the prototype. Since we are dealing with serial machines, concurrency means that the control is switched between these co-routines without interrupting their thread of control. That is, when the co-routine gets the control back, it resumes executing with the exact context it was before.

This mechanism allows the activity to use delay statements, wait for events, and perform continuous calculations without blocking the rest of the code from continuing execution. When a task is executed, however, the rest of the code is frozen. Thus, synchronization points are introduced. They allow the rescheduling of other tasks (or the control code) to proceed and actions (stop, suspend) to take effect.

Synchronization

There are three types of synchronization calls:

- `wait_for_event(event)`
- `task_delay(delay_time)`
- `scheduler()`

Each of these calls will suspend the calling task and reschedule another task or the `main_task` (statechart) on a round-robin basis.

The `wait_for_event` call suspends the activity until the specified event is generated. It is a way to synchronize the activity with other activities either user-implemented or statechart-controlled. When the event is generated, the code resumes execution after the wait call.

Example:

```
void sense_start()
{
  while (1) {
             wait_for_event(SENSE);
    /* here you are supposed to check status.*/
   printf("Time generated\n");
  }
} /* end sense_start */
```

The `task_delay` statement delays the activity for the time specified in the call. It is useful to implement polling processes that periodically perform checks on a time basis.

Example:

```
void poll_input()
{
       while (1) {
              mouse_input = read_input_from_mouse();
              if (mouse_input) {
              . . Do Something . . .
              }
              task_delay(0.1); /* delay 0.1 seconds */
       }
}
```

The `scheduler()` call is used when you have a calculation which is too long to be executed non-preemptively. For example, if you have to multiply two 10000x10000 matrices, you do not want the rest of the system to be blocked all that time.

The `Scheduler()` call will allow other activities to proceed and the calling activity will resume execution in the next available time slot unless a stop or suspend command was issued. The call should be placed in a loop in which one cycle can be executed without preemption, but an outer loop may take too long.

Note

No synchronization call should be used by a procedure-implemented activity.

Example:

```
void multiply()
{
  for (i = 1; i<=10000; i++) {
    for (j = 1; j<=10000; j++) {
      /* internal loop is short
          enough to complete */
    }
    scheduler();
  }
}
```

Scheduler Package

The user can specify that some of the primitive activities are to be implemented as tasks in the Profile Editor. The tasks are actually C functions started as co-routines. The Rational Statemate simulation itself is a task, which runs concurrently with the other started tasks.

Controlling all those tasks is the responsibility of statecharts, which issue different actions to the different activities (i.e., start, stop, suspend, resume). All this is handled by a scheduler package, which is supplied with the simulator and is available on Rational Statemate platforms only. This package supports multi-tasking programming within the context of a single process.

Below we describe how the user may add his own tasks, apart from those created for each task-like primitive activity, and how to use the scheduler for controlling them.

Status of a Task

Each task may be in one of four states:

- **Current** - The task is executing
- **Ready** - The task is ready for execution
- **Delayed** - The task is waiting for some event to occur
- **Stopped** - The task is not active

The calls that change the status of a task are described below.

Scheduling Policy

The context switch between tasks is done only in the following synchronization points:

- When a task explicitly calls the scheduler. This is done by calling the following routine:

  ```
  scheduler()
  ```

  If there are other ready tasks - one of them (chosen in a round-robin manner) becomes current, while the calling task becomes ready. If there is no other task ready, the calling task continues its execution.

- When a task issues a delay request by calling `task_delay`. The calling task then becomes delayed.
- When a task calls a `wait_for_event` service. The calling task then becomes delayed.

  ```
  wait_for_event(EVENT)
  event *EVENT;
  ```

- After the task function performs a return, it stops.

Restrictions

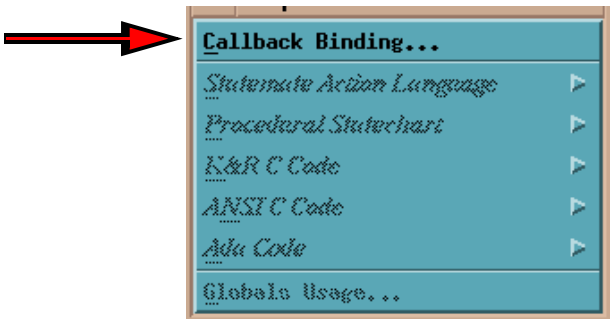
Any call to process blocking functions (e.g., sleep, scanf) of the operating system from a task will hibernate not only the calling task, but the whole process. Using `fork()` and signals is also not allowed, since it might confuse the scheduler.

Binding Callbacks

Callbacks are a powerful mechanism that enable you to connect user-actions or procedures to any change in a Rational Statemate element during execution. This mechanism is very useful when you wish to tie your external environment to the behavior represented by the simulation.

Callback Binding

To connect elements such as events, conditions, data items, and user-defined types, select the element in the Data Dictionary Editor and then **Implementation** > **Callback Binding**.



Callback Statement

The connection and binding statement for callbacks consists of:

```
proc_name(<"element_identifier">,param_1,param_2)
```

The <element_identifier> is required when and only when the callback is connected to an aggregate element. An aggregate element is an array, record, union, user-defined type, or any element referenced in a generic or instance. The <element_identifier> specifies what part of the aggregate element the callback is to be connected.

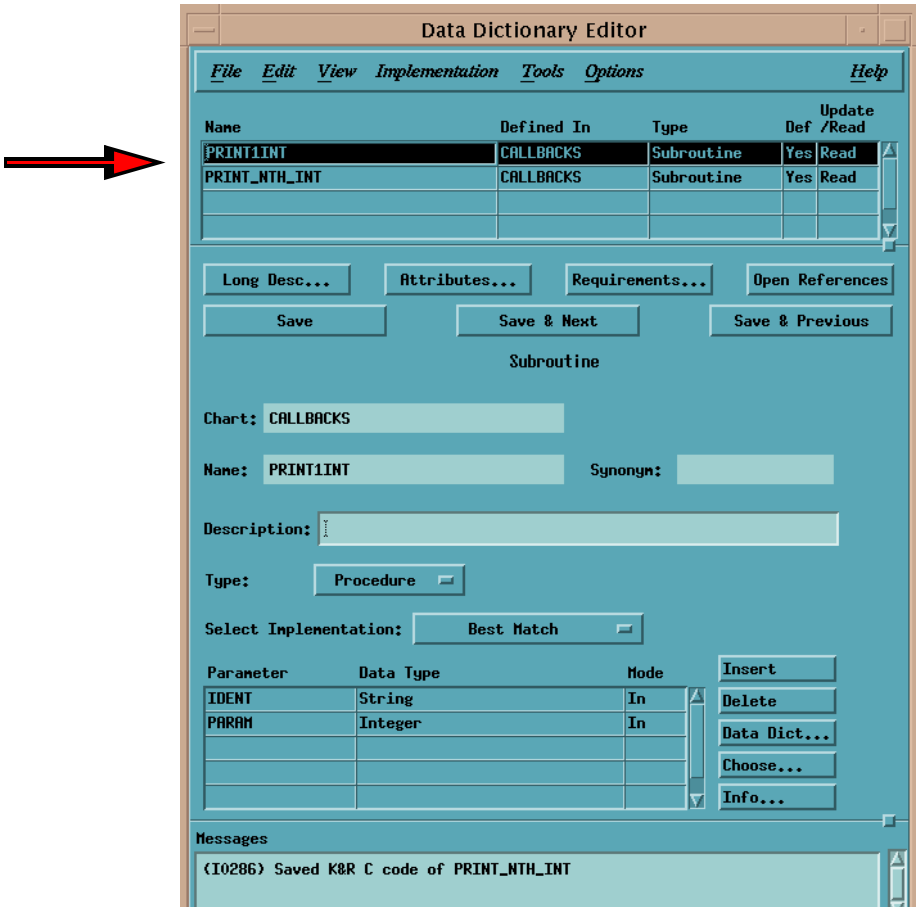
Disabling Callbacks

To disable a callback, change the **Enable** option in the **Callback Binding** dialog to **Disable**. This causes the simulator to generate code, but it “breaks” the code’s connection with the element.

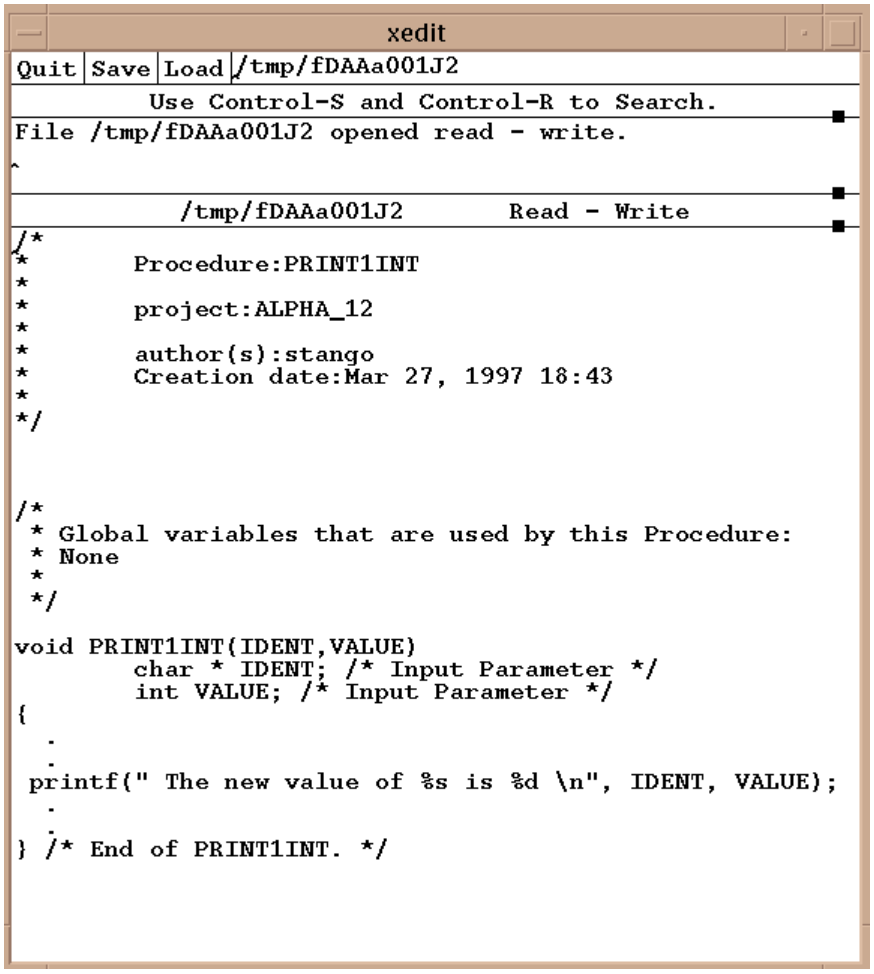
Callback Example

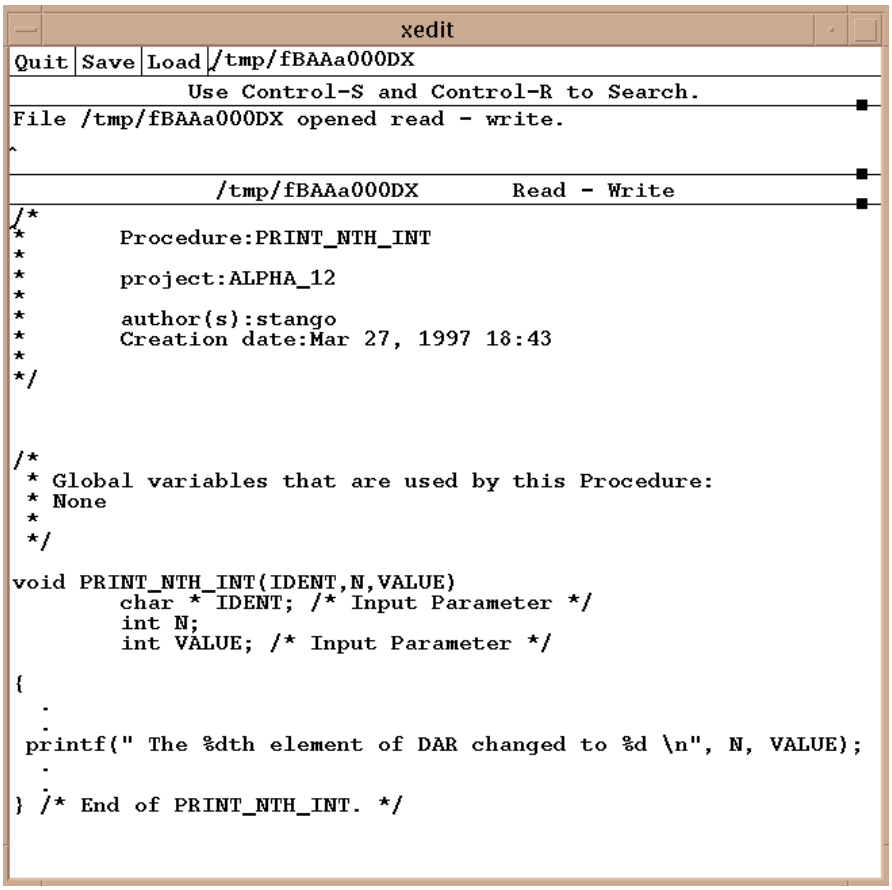
The following example illustrates the Rational Statemate **callback** utility by showing two subroutines that are bound to the callback `DAR`. Every time the `DAR` element changes, Rational Statemate runs both of these subroutines.

To create a subroutine, refer to the steps documented in **Supplementing the Model with Subroutines**



The next two figures show the code for the subroutines. The first one is the `PRINT1INT` procedure; the second one is the `PRINT_NTH_INT` procedure.


xedit
Quit
Save
Load
/tmp/fDAAa001J2
Use Control-S and Control-R to Search.
File /tmp/fDAAa001J2 opened read - write.
/tmp/fDAAa001J2 Read - Write
/*
* Procedure:PRINT1INT
*
* project:ALPHA_12
*
* author(s):stango
* Creation date:Mar 27, 1997 18:43
*
*/
/*
* Global variables that are used by this Procedure:
* None
*
*/
void PRINT1INT(IDENT,VALUE)
char * IDENT; /* Input Parameter */
int VALUE; /* Input Parameter */
{
.
.
printf(" The new value of %s is %d \n", IDENT, VALUE);
.
.
} /* End of PRINT1INT. */


xedit
Quit
Save
Load
/tmp/fBAAa000DX
Use Control-S and Control-R to Search.
File /tmp/fBAAa000DX opened read - write.
/tmp/fBAAa000DX Read - Write
/*
* Procedure:PRINT_NTH_INT
*
* project:ALPHA_12
*
* author(s):stango
* Creation date:Mar 27, 1997 18:43
*
*/
/*
* Global variables that are used by this Procedure:
* None
*
*/
void PRINT_NTH_INT(IDENT,N,VALUE)
char * IDENT; /* Input Parameter */
int N;
int VALUE; /* Input Parameter */
{
.
.
printf(" The %dth element of DAR changed to %d \n", N, VALUE);
.
.
} /* End of PRINT_NTH_INT. */

Referencing Model Elements

Communication between the handwritten code and the generated code is accomplished through the semantics of the following information elements:

- Events
- Conditions
- Data-items
- User-defined types

It is important to understand how to access the values of these elements and how to modify them. Each element has the following representation in the C target language:

- Conditions are represented as bytes
- Data-items are represented as integers, reals, strings or unsigned
- User-defined types are derived from primitive data-types

When you wish to pass structured elements (such as records and unions) from Rational Statemate to your handwritten code, you must define these elements as user-defined types.

When you write code in the template, refer to all elements by the names you assigned in the model. This applies to parameters of the subroutine, its local and global variables, to names of types, constants, and any other subroutines that you may use for the implementation.

Note

Write all element names in uppercase.

Referencing Events

Events are primitive elements and are special in the sense that software languages do not support them directly.

Note

Events are not allowed in subroutines as inputs, outputs, local variables, or accessible as global elements.

Events, in relation to handwritten code, are used in the following manner:

- **Callbacks**—You can associate a callback with a Rational Statemate event.
- **Tasks**—You can use the `wait_for_event` command to react to a Rational Statemate event.

Where Elements are Defined

An element can be local to a module or global to a profile. The element is globally defined when it is referenced by more than one module, i.e., defined in the top-level module. Each module ‘exports’ all its local elements as externals in its header file. This allows other user modules to access them. If you want to reference an element you must refer to its scope by including the appropriate header file. An example is shown below.

Example:

If you want to reference an element `BAUD_RATE` in module display, you should include the header file “display.h” to make the element visible.

```
/* my module */
#include “display.h”
.
.
br = BAUD_RATE ;
.
.
```

Accessing an Element Value

Since the element is a simple language element, it can be easily accessed by referring to its name.

Example:

```
my_data = XXX + YYY ;
```

Mapping Rational Statemate Types into C

The table below shows how Rational Statemate maps primitive types into corresponding C types:

Rational Statemate Types	C Type
Conditions	char (byte 0-false, 1-true)
Integer	int
Real	double
Bit	bit_array[1]
Bit array	unsigned int
User Type	struct
Record	struct
Union	struct
Enumerated Types	typedef

Note

All Rational Statemate elements of type *string* are translated into allocated C elements.

Records

Records become C constructs. For example, a record `INVOICE_TYPE` might become a structure defined as:

```
typedef struct INVOICE_TYPE {
      char NAME[80+1];
      char ITEM[80+1];
      real AMOUNT;
} INVOICE_TYPE;
```

Note that the name `INVOICE_TYPE` is normally named the same as the User-Defined Type name. If, however, the Rational Statemate model contains multiple textual elements with the same name, the C code names will be modified to make all the names unique. This name mapping information is listed in the *.info* file.

Unions

Unions become C unions with a declaration that is similar to the construct definition for records.

Arrays

Elements of all arrays in C are enumerated starting from 0. In Rational Statemate, there is no such restriction.

Enumerated Types

An Enumerated Type is a user-defined type with a finite number of values. The Simulation monitor allows you to select an enumerated-value from a list of possible values. Enumerated types with a large number of possible values are supported.

Enumerated values and other textual items cannot have the same name within the same scope. For example, data-item SUN cannot be declared in the same chart where an enumerated value SUN is declared.

Note

Enumerated range and indices of arrays are not supported in C. The C code generator shall approximate this capability in the generated code.

There are two constant operators and five general operators for enumerated types:

Constant Operators

`en_first(T)`	First enumerated value of T
`en_last(T)`	Last enumerated value of T

Parameters to these constant operators are user-defined types that were defined as enumerated types.

General Operators

`en_succ([T’]VAL)`	Successor enumerated value of T
`en_pred([T’]VAL)`	Predecessor enumerated value of T
`en_ordinal([T’]VAL)`	Ordinal position of VAL in T
`en_value(T,I)`	Value of the i’th element in T
`en_image([T’]VAL)`	String representation of VAL in T

Parameters to these operators are either enumerated values (literals) or variables. The T’VAL notation is used for non-unique literals.

Bit Arrays

Bit-arrays are stored in unsigned ints. Since unsigned ints can hold a maximum of 32 bits, bit-arrays larger than 32 bits are stored in arrays of unsigned ints. Arrays of bit-arrays are stored in two dimensional arrays of unsigned ints. Notice that multiple bit-arrays smaller than 32 bits are NOT packed into the unsigned int.

Data-Items	Results in these structures
BA1 is array 1 to 10 of Bit-array 31 to 0	bit_array BA1[10][1]
BA2 is array 1 to 10 of Bit-array 48 to 0	bit_array BA2[10][2]
BA3 is array 1 to 10 of Bit-array 3 to 0	bit_array BA3[10][1]

Note

In `$STM_ROOT/etc/prt/c/types.h` you will find the statement: `typedef unsigned int bit_array`.

Bit Array Functions

```
bit_array *AND(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;
  bit_array *ba2;
  int l_ba2;
  int from2;
  int to2;


bit_array  *NOT (ba1, l_ba1, from1, to1)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;


bit_array  *OR(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;
  bit_array *ba2;
  int l_ba2;
  int from2;
  int to2;


bit_array *XOR(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;
  bit_array *ba2;
  int l_ba2;
  int from2;
  int to2;
```

The following bit array function names are mapped through macros to their internal names, because these names are used by Ada runtime libraries, therefore they cannot be defined as functions in the intrinsics. (These same intrinsics are used by C and Ada environment.) It is important to include the *types.h* header containing these macros.

```
#define ASHR ashr
#define LSHL lshl
#define LSHR lshr
#define BITS_OF bits_of
#define CONCAT_BA concat_ba
#define EXPAND_BIT expand_bit
#define SIGNED signed_b
#define MINUS minus_b
#define NAND nand_b
#define NOR  nor_b
#define NXOR nxor
```

The functions are:

```
bit_array *concat_ba
(ba1,l_ba1, from1, to1, ba2, l_ba2, from2,to2)
       bit_array *ba1;

       int l_ba1;
       int from1;
       int to1;
       bit_array *ba2;
       int l_ba2;
       int from2;
       int to2;

bit_array *lshr(ba, len_ba, from, to, shift)
       bit_array *ba;
       int len_ba;
       int from;

       int to;
       int shift;


bit_array *lshl(ba, len_ba, from, to, shift)
       bit_array *ba;
       int len_ba;
       int from;
       int to;
       int shift;


int signed_b(ba_val, len, from, to)
       bit_array *ba_val;
       int len;
       int from;
       int to;
```

```
bit_array *ashr(ba, len_ba, from, to, shift)
       bit_array *ba;
       int len_ba;
       int from;
       int to;
       int shift;


bit_array  *nand_b(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
       bit_array *ba1;
       int l_ba1;
       int from1;
       int to1;
       bit_array *ba2;
       int l_ba2;
       int from2;
       int to2;


bit_array  *nor_b(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
       bit_array *ba1;
       int l_ba1;
       int from1;
       int to1;
       bit_array *ba2;
       int l_ba2;
       int from2;
       int to2;


bit_array *nxor(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
       bit_array *ba1;
       int l_ba1;
       int from1;
       int to1;
       bit_array *ba2;

       int l_ba2;
       int from2;
       int to2;
```

Use the following functions to convert between integer and bit-array types:

```
bit_array *int2ba(int_val)
       int int_val;


int ba2int(ba, len, from, to)
       bit_array *ba;
       int len;
       int from;
       int to;
```

Rules for Mapping into C

The following table summarizes the rules of mapping into C for:

- Types of parameters for procedures and functions
- Returned type of functions

Note

- The first level of all arrays should be defined as User-defined type in order to restrict the ‘second’ dimension.
- Unrestricted strings and bit-arrays are not allowed as returned type of a function.
- Numeric Input parameters can be mixed up i.e., integer, real and bit-arrays can be mixed when used as actual and formal parameters.

Type	Function Type	In Param	Out/InOut Param
Primitive (*)	int f();	int P;	int *P;
UDT defined as Primitive	UDT f();	UDT P;	UDT *P;
Record/Union	rec *f();	REC *P;	REC *P;
String	char *f();	char *P;	char *P;
UDT defined as String	char *f();	UDT P;	UDT P;
Bit	BIT_ARRAY *f();	BIT_ARRAY *P;	BIT_ARRAY *P;
Bit-array	BIT_ARRAY *f();	BIT_ARRAY *P;	BIT_ARRAY *P;
UDT defined as Bit-array	BIT_ARRAY *f();	BIT_ARRAY *P;	UDT *P;
UDT Array of Primitive	int *f();	UDT P;	UDT P;
UDT Array of String	-- Illegal --	UDT P;	UDT P;
UDT Array of Bit-array	-- Illegal --	UDT P;	UDT P;
UDT array of direct R/U	-- Illegal --	UDT P;	UDT P;
UDT array of UDT2	UDT2 *f();	UDT P;	UDT P;
Array of Primitive	-- Illegal --	int *P;	int *P;
Array of Record/Union	-- Illegal --	-- Illegal --	-- Illegal --
Array of String	-- Illegal --	char *P;	char *P;
Array of Bit-array	-- Illegal --	BIT_ARRAY *P;	BIT_ARRAY *P;

(*) Primitive type is one of: integer, real, condition, or enumerated type.
In the above matrix, integers are taken as example.

BNF Syntax, Structure and Conventions

Described in this section are the conventions for BNF (Bakus-Naur Form), a widely used notational scheme for formal languages. BNF was introduced in 1963 as a technique for defining programming languages.

BNF Structure And Conventions

BNF grammar follows the following general structure:

```
nonterminal_symbol => terminal_and or_nonterminal_symbols
```

Example:

```
action        =>action_name
              | primitive_event_name
              | start (activity_name)
              | stop (activity_name)
```

Symbols are delimited by spaces and thus the underscore is frequently used for longer names.

Symbol Types

Terminal symbols are basic symbols which are not parsed further to derive their meaning. Non-terminal symbols are those which may be further broken down by parsing. Examples of terminal symbols may be integer numbers which are intrinsically recognized as a numeric value, or language keywords recognized by the system as representing some particular operation or function.

In **BNF Syntax, Structure and Conventions** terminal symbols that are written exactly as they appear (i.e., keywords of the SCL) are show in capital letters. Non-alphabetic characters not belonging to the BNF notation below, are also part of the syntax (e.g., ;). Non-terminal symbols are written in lowercase or mixed case letters. Non-terminal symbols which are self evident are not further broken down.

BNF Notations

The | indicates a mutually exclusive choice between symbols in a non-terminal symbol definition.

Example:

```
variable_name | numeric_constant
| integer | function_name
```

The => separates the non-terminal symbols on the left from its definition on the right. Can be read as is “defined as ...”

Example:

```
relational_operator=>= | / = | < | <= | >=
```

Square brackets [ ] indicate that the symbols within the brackets are optional. This is a BNF convention. Recall that square brackets themselves may appear in the SCP as part of the Rational Statemate expression.

Example:

```
timeout (event[condition],3)
```

Curled brackets { } indicate that the symbols which they enclose are optional and can be repeated.

Example:

```
depend_on (state_name{, state_name})
```

BNF for SCL Statements Syntax

This section presents the Simulation Control Language statements expressed formally in the BNF syntax.

Syntax of actions, events, conditions, expressions is the same as in the specification itself. One may use SCL variables defined in the SCP in any Rational Statemate expression except for event expressions. For example, while make_true(c), where c is a local boolean variable, is legal, TRUE(C) is not.

```
scp_program=>PROGRAM program_name ;
       [declaration_section]
       [init_part]
       [breakpoint_part]
       [main_part]
       END.
init_part=>INIT
       sequence_of_statements[;]
       END INIT[;]
breakpoint_part=>breakpoint_definition
       {breakpoint_definition}
breakpoint_definition=>SET BREAKPOINT
       [br_name=>] br_trigger DO
       sequence of statements[;]
       END BREAKPOINT[;]
br_trigger=>event
       | EVERY expression
main_part=>BEGIN
       sequence_of_statements[;]
       END[;]
declaration_section=>[CONSTANT const_decl_list]
       [VARIABLE var_decl_list]
const_decl_list=>const_decl; {const_decl;}
```

```
const_decl=>const_type id_val_list
const_type=>INTEGER
       | FLOAT
       | STRING
       | BIT
       | ARRAY
id_val_list=>id := expression {,id := expression}
var_decl_list=>var_dec; {var_dec}
var_dec=>[GLOBAL] type id_opt_val_list
simple_type=>INTEGER
       | FLOAT
       | STRING
       | FILE
       | BOOLEAN
       | BIT
id_opt_val_list=>id [:= expression] {,id [:= expression]}
type   =>simple_type
       | array_type
array_type=>(constant..constant) of simple_type
bit_arrau type=>bit-array name (1..6)
sequence_of_statements=>scl_statement {; scl_statement}
scl_statement=>simple_statement
       | structured_statement
       | io_statement
simple_statement=>assign_statement
       | set_statement
       | go_statement
       | random_solution_statement
       | skip_statement
       | undo_statement
       | restore_statement
       | save_statement
       | choose_statement
       | exec_statement
       | stop_statement
       | simple_action_statement
```

```
structured_statement=>if_statement
       | for_loop
       | when_statement
       | while_loop
io_statement=>read_statement
       | write_statement
       | open_statement
       | close_statement
assign_statement=>ASSIGN activity_id scp_name
scp_name=>string_constant
set_statement=>set_operation BREAKPOINT br_name
       | set_operation TRACE
       | set_operation DISPLAY
       | SET INFINITE LOOP
        expression
       |SET INTERACTIVE
       | SET GO BACK  expression
       | set_operation REPORT RACING

set_operation=>SET
       | CANCEL
simple_action_statement=>action
go_statement=>GO [go_type]
go_type=>STEP
       | REPEAT
       | NEXT
       | ADVANCE
       | EXTENDED
       | STEPN
random_solution_statement=>RANDOM_SOLUTION
skip_statement=>SKIP
undo_statement=>GO BACK
restore_statement=>RESTORE_STATUS
       status_name
save_statement=>SAVE_STATUS status_name
choose_statement=>CHOOSE expression
exec_statement=>EXEC scp_name
stop_statement=>STOP_SCP [scp_name]
```

```
scp_name=>string_constant

status_name=>string_constant

file_name=>string_constant

if_statement=>IF condition THEN
       sequence_of_statements[;]
       [ELSE
       sequence_of_statements[;]]
       END IF;

when_statement=>WHEN event THEN
       sequence_of_statements[;]
       [ELSE
       sequence_of_statements[;]]
       END WHEN;

while_loop=>WHILE condition LOOP
       sequence_of_statements[;]
       END WHILE;

for_loop=>FOR condition LOOP
       sequence_of_statements[;]
       END FOR;

read_statement=>READ([file_var,] id_list)

write_statement=>WRITE([file_var,]
       write_expression_list)

open_statement=>OPEN(file_var, file_name,
       INPUT)
       | OPEN(file_var, file_name, OUTPUT)

close_statement=>CLOSE(file_var)

file_var=>id

id_list=>id {,id}

write_expression_list=>write_expression
       {,write_expression}

write_expression=>expression [, expression]
       | event [; expression]
       | condition [;expression]
```

SCL Reserved Words

Simulation Control Language (SCL) statements are built using the various keywords. These keywords are considered as reserved words - their unintended use will cause, in most cases, syntax errors in your SCP. This section presents these reserved words.

You should avoid using the keywords as names of SCL variables. There are three groups of keywords in SCL:

- Keywords and predefined function names used in Rational Statemate expressions. They all can be used in SCL statements.
- Names of predefined SCL variables:

CUR_CLOCK	INFINITE_LOOP
NONDETERMINISM	STATIONARY
STEP	STEP_NUMBER
TERMINATION	

- Keywords of SCL statements:

ADVANCE	ASSIGN
BACK	BEGIN
BOOLEAN	BREAKPOINT
CANCEL	CHOOSE
CLOCK	CLOSE
CONSTANT	DISPLAY
DO	END
EVERY	EXEC
EXTENDED	FILE
FLOAT	GLOBAL
GO	INFINITE
INIT	INPUT
INTEGER	INTERACTIVE

LOOP	NEXT	
OPEN	OUTPUT	
PROGRAM	RANDOM_SOLUTION	READ
REPEAT	RESTORE_STATUS	SAVE_STATUS
SET	SKIP	
STEP	STOP_SCP	
STRING	TRACE	
VARIABLE	WHILE	
WRITE		

Index

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W