PathFinder

A Design Exploration Tool

User's Manual

Version 0.1
September 20, 2001

IBM Research Laboratory
Haifa, Israel

contact: baruch@il.ibm.com

This product or portions thereof is manufactured under license from Carnegie Mellon University.

CHAPTER 1 *Introduction*

PathFinder is a design exploration tool. It allows you to explore important paths of your designs. Traditional methodology of path exploration requires a tedious step by step specification of all inputs to the design, to enable the simulator to produce an execution trace which is an example of the desired hardware path. In contrast, PathFinder allows you to specify only the important conditions that should hold along the required path, and the events that cause the hardware to move from one phase to another. PathFinder works to find a setting of the inputs for which the corresponding execution trace is consistent with the given path specification.

This document is constructed as follows:

- **CHAPTER 2: Tool Overview** - brief review on the system components, and system interface (inputs and outputs).
- **CHAPTER 3: Tutorial** - introduction to Pathfinder in the form of a tutorial. The tutorial presents a small design and shows how to explore paths in that design.

- **CHAPTER 4: Getting Started** - description of the initial settings (e.g. shell, working directory, design compilation) that should be done before starting to explore paths of the design.
- **CHAPTER 5: Inputs and Environment Specification** - settings to the design signals (mostly inputs) and other restrictions on them to be done before exploring the path.
- **CHAPTER 6: Path Specification** - description of a path specification.
- **CHAPTER 7: Search and results Analysis -** description of the search process and analysis of the results.

CHAPTER 2 *Tool Overview*

PathFinder has three main inputs:

- Design under test (*DUT*)
- Path specification
- Environment and input restrictions

The output of Pathfinder is a trace/s of the DUT. The trace is consistent with the path specification and the environment restrictions (see Figure 1).

2.1 Design Under Test

The primary input to PathFinder is the design under test (DUT). Currently, the only hardware specification language for DUTs that PathFinder supports is VHDL and verilog.

2.2 Path specification

A path is defined by a (ordered) list of phases. A phase is a sequence of cycles along the path that are specified by:

- *Satisfying:*

A condition that should hold during the phase. The condition is a boolean expression over the signals names.

- *Terminating*:

  A condition to terminate the phase. This condition can be either a boolean expression over the signal names or a time bound (given in cycles).

2.3 Inputs and environment restrictions

By default, Pathfinder sets all inputs of the DUT to be random every cycle. This setting may result in an invalid trace (trace that does not exist in the 'real world'). Pathfinder enables you to restrict the behavior of the inputs in the following ways:

- *Signal values*

  Pathfinder displays all signals of the design in the Signals List pane. In particular it shows the list of inputs to the design. Pathfinder allows you to attach a constant value to each input (bit or vector).

- *Restrictions*

  Pathfinder allows you to specify a list of boolean conditions that should hold in the resulting trace. For example the restriction: '*read -> !write*' means that at each cycle of the path if read is '1' then write is '0'. These restrictions are specified seperately for each path.

- *Environment*

  Pathfinder allows you to define an abstract behavior for the design inputs (or internal signals) in a separate file. The behavior is specified either in EDL (environment description language of Rulebase) or in VHDL. For example the following lines define a reset of 1 cycle:

  assign init(reset) := 1

  assign next(reset) := 0

2.4 Output - a trace of the DUT

The output of Pathfinder is a trace of the DUT that is consistent with its phases specification and its environment restrictions. Pathfinder displays this trace in the scope pane of the GUI and enables you to bring any other signal to the scope. If possible, Pathfinder finds more than one trace for the specified path.

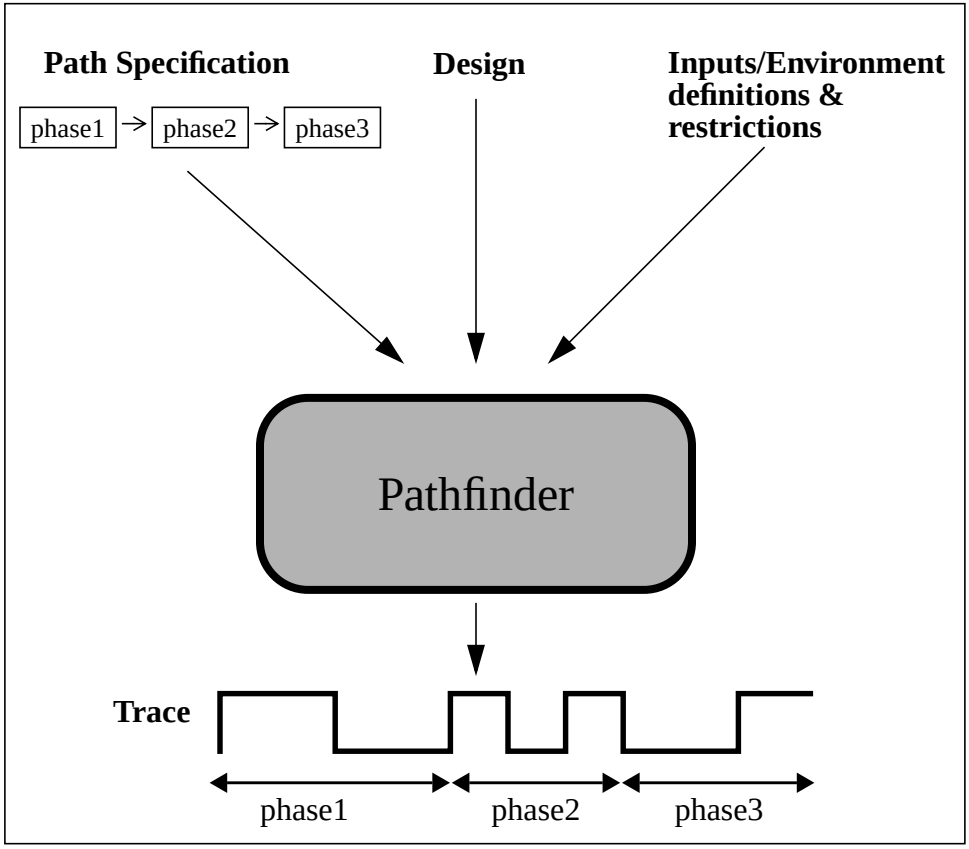


FIGURE 1. **System overview**

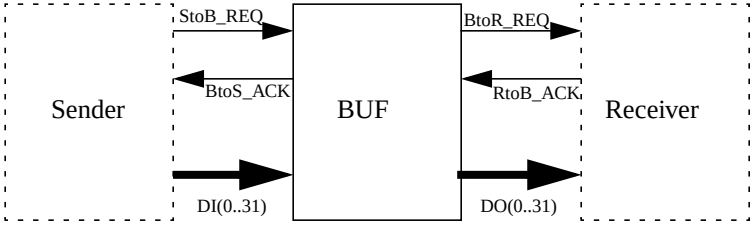
CHAPTER 3

Tutorial

This chapter introduces Pathfinder's path exploration in the form of a tutorial. The tutorial presents a small design named BUF and shows how to explore a path of this design.

3.1Design description

BUF is a design block that buffers a word of data (32 bits) sent by a sender to a receiver. It has two control inputs, two control outputs, and a data bus on each side, as shown by the block diagram:



Communication (on both sides) takes place by means of a 4-phase handshaking as follows:

When the sender has data to send to the receiver, it initiates a transfer by putting the data on the data bus and asserting StoB_REQ (server to buffer request). If BUF is free, it reads the data and asserts BtoS_ACK (buffer to server acknowledge). Otherwise, the sender waits. After seeing BtoS_ACK, the sender may release the data bus and deassert StoB_REQ. To conclude the transaction, BUF deasserts BtoS_ACK.

When BUF has data, it initiates a transfer to the receiver by putting the data on the data bus and asserting BtoR_REQ (buffer to receiver request). If the receiver is ready, it reads the data and asserts RtoB_ACK (receiver to buffer acknowledge). Otherwise, BUF waits. After seeing RtoB_ACK, BUF may release the data bus and deassert BtoR_REQ. To conclude the transaction the receiver deasserts RtoB_ACK.

In the following sections, we show how to make PathFinder provide a trace which demonstrates this four phase hand shake.

3.2 Initial setup for a working environment

- If you run Pathfinder for the first time add the following setting in your.cshrc file (or the shell you are working with)

  setenv RBROOT location_of_rulebase_executable

  setenv PFROOT location_of_pathfinder_executable

  alias pf $PFROOT/bin/pf
- Create a directory pf_example
- Copy $PFROOT/tutorial/* to your pf_example directory
  The following files will be copied to your directory:

- *buf.vhd* - your design
- *envs* - the inputs to the design written in EDL.
- *makefile* - points to names of your vhdl files.

and some other files and folders that are generated automaticly by pathfinder.

Invoke Pathfinder by typing: pf

3.3DUT compilation

To compile your design click the '*Compile*' button at the button menu bar. The status bar reflects the compilation status as follows:

- '***Compiling***' (red) message - the design is being coupled now.
- '***Compilation OK***' - the design has been successfully compiled.
- '***Compilation Errors***' - compilation process encountered some problems. To see the compilation log messages click the 'Compilation' button at the log pane (the '*view log'* button enables you to see the log message in a larger popup window).

3.4Inputs restrictions and behaviors

Once the design is compiled, a list of the design signals is displayed in the '*Signals*' pane. This list includes four sections:

- ***inputs*** - this list shows five signals inputs of the design which are: CLK, DI(0:31), RST, RTOB_ACK and STOB_REQ.
- ***outputs*** - this list shows three single output of the design which are: BTOR_REQ, BTOS_ACK and DO(0:31).
- ***internals*** - this list shows the internal signal of the design which are: OCCUPIED, R_STATE, and S_STATE.

- ***User Defined*** - this list shows the signals that you defined in your environment file.

The inputs list has an attribute attached to each of its members. This attribute denotes the type of the input. The type can be random (R), zero (0), one (1) or defined by the user in the environment file (E).

3.5Path definition

The path to explore is already defined in the '*Path Specification'* pane**.** The path is composed of three phases.

- **Phase 1:** satisfying condition is *'DI(0:31)=1'* and terminating condition is '*STOB_REQ*':meaning that the phase will start at cycle '0' when DI(0:31)=1 and STOB_REQ=0 and will finish when STOB_REQ=1. during the whole phase DI(0:31)=1 should hold.
- **Phase 2**: terminating condition is '*BTOR_REQ*' meaning that the phase will start at the end of phase 1 and will finish when BTOR_REQ is '1'. during the whole phase BTOR_REQ=0 should hold.
- **Phase 3**: terminating condition is '*fell(RTOB_ACK)*' meaning that the phase will start at the end of phase 2 and will finish when RTOB_ACK goes from '1' to '0'.

To view or change the definition of a phase click the '*edit phase*' button of the phase.

3.6Find the trace

To instruct Pathfinder to search for the path click the 'Find Trace' button. The status bar reflects the status of the search as follows:

- '***Searching***' - Pathfinder is searching for the specified path.
- '***Search OK***' - Pathfinder completed searching for the path.
- '***Search errors***' - Search failed.

3.7 View the trace

Pathfinder displays the trace at the '*Scope*' pane after it finishes searching for it. The cycles and mapping to phases are shown at the top area of the scope. The names of the signals and their values at cycle '0' are shown in the '*names*' and '*values*' columns.

As shown, phase 1 is of three cycles, phase 2 is of two cycles and phase 3 is of three cycle.

To see the signals value at a given point of time, left click at the wave area at the desired point. A red vertical line will be displayed and the 'values' column will show the signals values at this point of time. If you grab this vertical line the 'values' column will change to reflect the signals value at each given point of time.

3.8 Add a fourth phase to the trace

To add a fourth phase to our trace click at the right most arrow of the path (the last arrow). To edit this fourth phase click at its 'edit phase' button. Set the terminating condition to be 'fell(BTOR_REQ)'. Click 'OK' to exit the phase.

Click the 'Find trace' button to search for the new path.

CHAPTER 4

Getting Started

To explore paths of your design you should do the following initial setting:

- Add setting to your shell
- Create a working directory
- Customize Pathfinder to your working directory
- Compile your design
- Define behaviors for the DUT clocks and reset signals

4.1 Setting your shell

If you run Pathfinder for the first time make sure that you have the following settings in your shell files (e.g. .cshrc file)

setenv RBROOT location_of_rulebase_executable

setenv PFROOT location_of_pathfinder_executable

alias pf $PFROOT/bin/pf

4.2 Create a working directory

To investigate paths of your design create the following working environment:

- Create a main working directory
- Copy the VHDL files or Verilog file of your design to this directory.
- Create a file with a list of names of your VHDL files (each name in a separate line.
- Optional - create a file which includes the abstract definitions of your inputs. This step is optional and is needed only if you wish to give an abstract definition (behavior) for the design inputs.

4.3 Create a New Workspace

Invoke pathfinder (by typing 'pf') open a window, enter a name for your new workspace and press *'ok'*.

You can have several workspace on your pathfinder window, you can move from one to another, by clicking the *'file'* button and select the *'workspace'* option, select the workspace name you want to work with from the list and push *'ok'*,or you can add a new workspace name.

Each workspace open a new pathfinder window. each workspace can have different configuration and must be customized separately.

4.4 Customize Pathfinder

Click the '*Configure*' button and select the '*Setup*' option. Fill the setup options as follows:

- ***Compilation Path*** - the name of the script that compiles your design. it has several options for compilers: *koala,hiasynth,rb_tex1,rb_partial,vim.* If you wish to use your own specific compilation path then follow the instructions given in Section 9.1 .

For each compiler some of the following options must be defined.

- ***DUT "makefile"*** - a pointer to the file that has the name list of VHDL or Verilog sources.
- ***DUT entity name*** - the name of the main (top level) entity of your design.
- ***DUT architecture name*** - the architecture of your main entity.
- ***environment file*** (optional) - a pointer to the file that gives an abstract behavior to the design input signals.

Click 'OK' to commit your setting.

4.5 Compile your design

To compile your design click the '*Compile*' button at the button menu bar. The status bar reflects the compilation status as follows:

- '***Compiling*' (red)** message - the design is being coupled now.
- '***Compilation OK***' - the design successfully compiled.
- '**Compilation *errors***' - compilation process encountered some problems. open large popup window with compilation log messages, select '*exit*' to close this widow.
  To see the compilation log messages click the 'Compilation' button at the log pane. To view these log messages in a larger popup window click the '*View log*' button.

4.6 Define behavior for the clocks and reset signals

The last stage in the initial setting is to define a behavior to the clock (clocks) of your DUT and to the reset signal (if exists).(if you dont have environment file).

To invoke the Clocks/reset dialog click the '*Configuration*' button at the menu bar and select the '*Clocks/reset*' option. The Clocks/reset dialog is divided into two sections: Clocks setting and reset setting.

Clock setting

When searching for a trace, PathFinder's engines have their own 'ticking' facility, thus results are given in resolution of *cycles*. (No timing issues within a cycle can be checked). If only one clock rate exists in the DUT, clocks should be put to constant *1* (if active high), telling PathFinder to 'tick' every cycle.

You can either put a '1' for each clock in the input values window (see Chapter 5), or enter the clock names into the clocks dialog as described below.

Clocks dialog.

The clocks dialog allocates several lines for your DUT clocks. Each clock should be defined in a separate line. A line includes the following information:

- ***Clock signal name*** - the name of the clock signal as is appears in the DUT.
- ***Low active*** - design is active when clock value is '0'.
- ***High active*** - design in active when clock value is '1'.
- ***Environment*** - the clock has a special behavior that is defined in the external environment file.

Reset setting

The reset dialog defines the following entries for your DUT reset signal:

- ***Reset signal name*** - the name of the reset signal as appears in the DUT.
- ***Reset length*** - the length of the reset: The number of active cycles before turning inactive.
- ***Reset type*** - defines if the rest is a falling one (starts as '1' for X cycles and then falls or a raising reset (starts as '0' and then raises).

Click 'OK' to commit your setting.

Note that the signals list column reflects your clocks/reset setting (after committing them). *Low* active, *High* active and *Environment* will be reflected as

‘0’, ‘1’ and ‘E’ in the value entry associated with the clock. Any reset setting will be reflected as ‘*’ in the value entry associated with the reset signal.

CHAPTER 5

Inputs and Environment Specification

By default, Pathfinder sets all inputs of the DUT to be random every cycle. This setting may result in an invalid trace (trace that does not exist in the 'real world'). Pathfinder enables you to restrict the behavior of the inputs in the following ways:

- Assign constant values to inputs.
- Give *restrictions* to inputs behaviors.
- Give an abstract behavior in an external *environment* file.
- Define Templates for input signals.

5.1 Constant values to inputs

Pathfinder displays all signals of your DUT in the '*Signal List*' pane. In particular it shows the list of inputs to the design. Pathfinder allows you to attach a value to each input (signal or vector). This input setting is defined separately for each path. The column attached to the inputs signal list provides a type attribute to each input. The following attributes can be assigned to each input:

- **1** - the (one bit) input signal is constantly ‘1’
- **0** - the (one bit) input signal is constantly ‘0’
- **R** - the input signal will have a random value every cycle
- **I** - the input signal will pick a random value, and stick to it
- **V** - the input is a vector that has some value.
- **‘*’** - the inputs is a reset signal and its behaviour is defined in the Clock/reset dialog (click the ‘*Configuration*’ option at the menu bar)
- **E** - behaviour for this input is given by the external environment file (see Section 5.3). This attribute is set to ‘E’ only by Pathfinder and only when an abstract behaviour for this input is given in the environment file.
- **T -** the input signal is defined by a template.

To set the value of a one bit input:

- Left click on the signal’s type attribute toggles it over ‘1’, ‘0’ , ‘R’ and ‘I’.
- Right click on a signal (after being selected by a left click) opens a popup menu with the options: ‘*set*’, ‘*reset*’, ‘*random*’ and ‘*define behavior,* that set the signal value to ‘1’, ‘0’ ,‘R’ (random) and ,’T’ (template) respectively.

To set the value of a vector:

- Left click on the vector’s type attribute toggles it over ‘0’, ‘R’,’I, and ‘V.
- To set a value to a vector right click on the signal. A popup menu with the options ‘*reset*’, *randomize*’ , ‘*set value*’ and ‘*define behavior’* invokes. When ‘set value’ is selected a popup entry is opened in which you may enter an hexadecimal value. You may replace any hexadecimal character with the following:
  - A string of four binary digits enclosed with parenthesis. For example 5(1010)C is equal to 5AC.

- A string as described above where any binary digits may be replaced by the character ‘R’, denoting a random value, or ‘I’, denoting a random constant value. For example 5(0RRI)C meaning that bit 4 is set to ‘0’ bits 5 and 6 are random, and bit 7 may pick a random value, and stick to it for the entire run.
- The character ‘R’ means a random hexa value, and ‘I’ means an initially random hexa value. For example 5RC will give bits 4 to 7 a random value every cycle. IDF will give bits 0 to 3 to pick randomly their constant value.

Note, the inputs setting is defined separately for each path. The input values column reflects the inputs setting of the current (highlighted) path. If you wish to copy inputs setting from one path to the other use the ‘*copy inputs setting*’ and ‘*paste inputs setting*’ menu options that are provided for each path (see Section 6.2).

5.2Restrictions

Pathfinder allows you to specify a list of boolean conditions (*restrictions*) that should hold in the resulting trace. In particular you may restrict the inputs behavior of a given path. For example, the following restriction on the *request* inputs says: if reset is high then the next cycle reset should be low.

request -> !next(request)

This list is specified separately for each path. The syntax for the restriction statements is described in Section 6.3 .

5.3Environment

Pathfinder allows you to define an abstract behavior for the design inputs (or internal signals) in a separate file. The behavior is specified either in EDL (environment description language of Rulebase) or in VHDL. For example the following lines define a reset of 1 cycle:

assign init(reset) := 1

assign next(reset) := 0

If you give an abstract behavior in a separate environment file you should customize Pathfinder to point to this file. To point to the environment file, click the '*configure'* button, select the '*setup*' option and point to this file from the '*environment file*' entry and then '*compile'*. For more details see Section 4.4 .

Detailed description on the environment language is given in Appendix A .

5.4Templates

PathFinder allows you to define behavior for the design input signals from several permanent templates.

Right click on the input signal opens a popup menu, select *'define behavior'*, choose one of the templates from the following options:

- **Stair up -** the signal is raised after a number of cycle. (chosen by the user)
- **Stair down -** the signal is lowered after number of cycle. (chosen by the user)
- **Random pulse.**
- **Clock -** the signal behave like a clock, one signal up and the next down and so on. (with inisialization of 0 or 1, chosen by the user).
- **Request -** the signal behaves like requesting signal, in a request acknowledge protocol, the following must be defined:
  - when can a request be initialed.
  - after signaling a request, when can it be released.
- **Acknowledge -** the signal behave like acknowledge signal, in a request acknowledge protocol, the following must be defined:
  - when can an acknowledge be signalled.
  - after signaling an acknowledge, when can it be released.
- **Random data -** the signal behaves like data signal, with some bits set as 0, and some random, the following must be defined:

- when can data be changed.
- how many bits are active.

CHAPTER 6

Path Specification

A path is defined by its

- Path name
- Path restrictions
- Path phases definitions
- Path inputs setting

6.1 Path name

A path is identified by its unique name. The '*Path List*' pane shows the list of paths that you defined. If you invoke Pathfinder from an empty working directory (no path was defined by now), an empty path '*noname*' is presented.

Right click at a given path name in the Path List area invokes a popup window with the following options:

- ***rename*** **-** enables you to rename your path.
- ***delete*** **-** delete the path

- ***create copy*** **-** creates a copy of the current path with the name '*current_name_i*' where *current_name* is the name of the current pointed path, and *'i'* is the next available index to make the new name *current_name_i* unique. The new path will have the same phase-definition, restrictions list and inputs-setting as the source one (see Section 6.2 , Section 6.3 and Section 6.4).
- **add new path -** adds an empty new path at the end of the path list. This path has no restrictions and no phase definitions. In addition all inputs of the path are set to 'R' (random value).
- **Copy input setting -** creates a copy of the inputs setting of your current paht (to be used later -using the '*paste*' option - for a new path)
- **Paste input setting -** Gives the current path the inputs setting that you copied before of this current path.
- **copy scope visable signals -** create a copy of the signals shown on the scope, of this current path
- **paste scope visable signals -** show the signals on the scope that you copy before of this current path.
- **help -**

6.2 Path inputs definition

Pathfinder displays all inputs of your DUT in the '*Signal List*' pane and it allows you to define a value for each input (bit or vector) of the design (see Section 5.1). The input values column reflects the inputs setting of the current (highlighted) path. By default every new path gets a random setting for each input and every path that is a result of a 'copy path' operation gets the same inputs setting as the source path. The options '*copy input setting*' and '*paste input setting*' (see above) enable you to use the same settings for several paths.

6.3 Path restrictions

Pathfinder allows you to attach a list of restrictions to a given path. A restriction is a boolean expression (condition) that should hold along the path. The condition can be any boolean expression over

- DUT signals names.
- User defined signals names defined in the environment file.
- '*next(signal_name)*' - value of a signal in the next cycle

For example

- *write -> next(flush)* - means in our path *write* is always followed (in the next cycle) by a *flush.*
- *request -> (!flush & !busy)* - means if *request* is on then *flush* an *busy* should be off (at the same cycle).

The '*Path Restrictions*' window displays the list of restrictions associated with the current selected path. This window allows you to add/remove restrictions for the path. Right click at a given restriction opens a popup window with the following options:

- ***insert line*** - insert a new line (after the current pointed one) for a new restriction.
- ***delete line*** - delete the current pointed restriction
- ***cut, copy, paste*** - text operation on the restriction text, enabling you to copy (or move) text from one restriction to another.

Detailed description on the syntax of the boolean expressions is given in Appendix A .

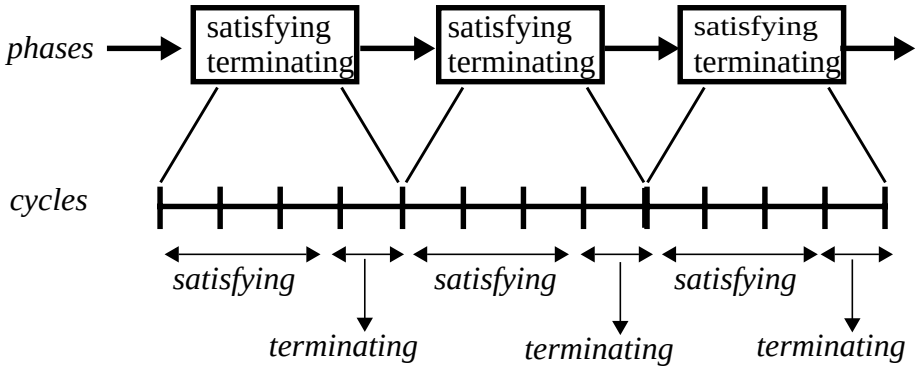
6.4Path phases definitions

A path is defined by a list of phases. A phase is a sequence of cycles along the path that are specified by:

- ***Satisfying****:*

  A condition that should hold during the phase. The condition is a boolean expression over the signals names.
- ***Terminating***:

A condition to terminate the phase. This condition can be either a boolean expression over the signals names or a time bound (given in cycles).



The '*Path Description'* pane enables you to define your path as a list of phases. The path is illustrated as a list of '*wagons*' where each wagon represents a phase. Click at a path name brings its wagon list to the 'Path description' area.

The following manipulations on wagons are provided:

- Left click at a given arrow adds a wagon right after this arrow.
- Right click at a given arrow opens a popup menu with the following options:
  - ***insert*** - insert a wagon right after the pointed arrow.
  - ***delete*** - delete the wagon right after the pointed arrow.
  - ***paste wagon*** - paste the last wagon that you copied after the current arrow.
  - ***delete to end*** - delete all the wagons until the pointed arrow.
- Right click at a given wagon opens a popup menu with the following option:
  - ***delete*** - delete the pointed wagon.

- ***copy*** - create a copy of the current pointed wagon (to be ‘pasted’ later).
- ***paste wagon*** - paste the last wagon that you copied after the current wagon.

To define a phase click the ‘***edit phase***’ button (of the required wagon). A popup dialog is opened enabling you to define the satisfying and terminating condition for the phase.

Satisfying condition

A satisfying condition is a boolean condition that should hold during the phase. The condition is a boolean expression over the signals names. For example the satisfying condition *‘req & urgent_priority*’ means that a request with an urgent priority is ‘on’ (true) along the phase.

If no condition is given, Pathfinder sets it to ‘True’, meaning that no special condition should hold during the phase. The phase will end when the terminating condition or bounds are satisfied.

Note, the satisfying condition may hold for zero cycles meaning that the phase is of a single cycle in which the terminating condition is true (see Section and Section).

Terminating condition

The condition to terminate the phase is either a boolean expression or a time bound (can not be both). Click at the ‘***terminating condition***’ option turns off the ‘*Terminating time bound*’ option and vise versa.

A terminating condition is a boolean condition over the signal names. The phase terminates when this condition comes true (meaning that the terminating condition is false during the phase). For example the terminating condition ‘*rose(ACK)*’ means that the phase should end when the acknowledge (ACK signal) goes from ‘0’ to ‘1’.If the ‘terminating condition’ option is selected but

no condition is given, the phase will be of X cycles where X is greater or equal to zero.

Terminating time bounds

Time bounds limit the length (in cycles) of the phase. Time bounds (lower, upper) should be integers. To give time bound to a phase click the '*time bound condition*' option (this will turn off the 'terminating condition' option).

The following settings are possible:

- If both lower and upper are given the phase will be of X cycles where

  lower <= X <= upper
- If only lower is given, the phase will be of X cycles where

  lower <= X
- If only upper is given, the phase will be of X cycles where

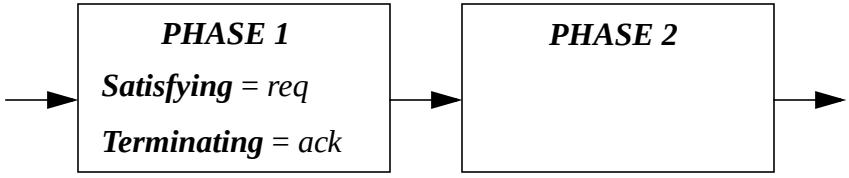
  0 <= X <= upper
- If neither upper nor lower are given, the phase will be of an arbitrary number of cycles (may be '0').

For example:

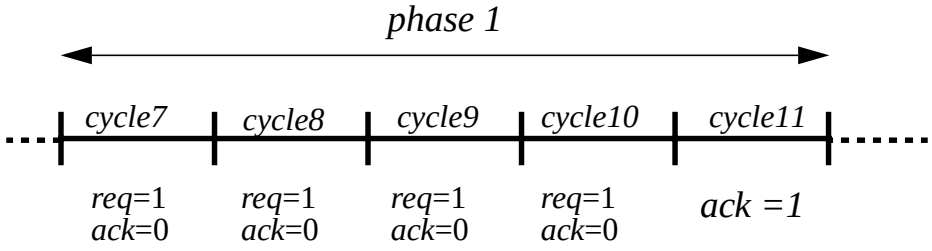
- lower=3 and upper=7 - means that the phase will be of at least 3 cycles and at most 7 cycles.
- lower=3, upper not defined - means that the phase will be of at least 3 cycles.
- upper=7, lower undefined - means that the phase will be of at most 7 cycles and this could be an empty phase.

Examples

1. Given :

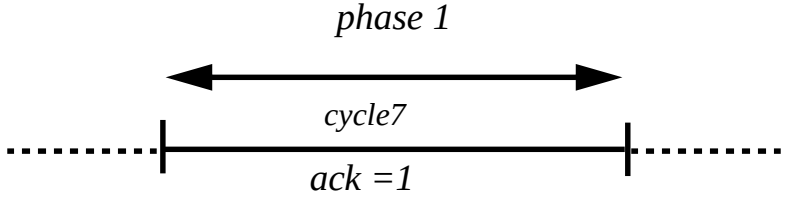


Phase 1 can be of several cycles where all cycles but the last one satisfy: satisfy *'req = 1'* and '*ack=0*', and the last cycles satisfies *ack=1* (*req* an be either '1' or '0').

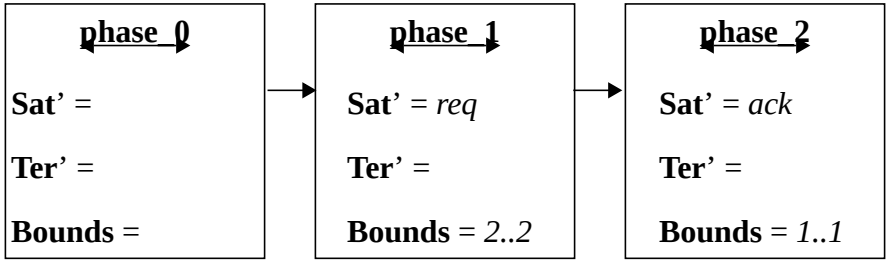


On the other hand phase 1 can be of a single cycle that satisfies *ack*=1.

This means that the satisfying condition holds for zero cycles and then we have the terminating condition.



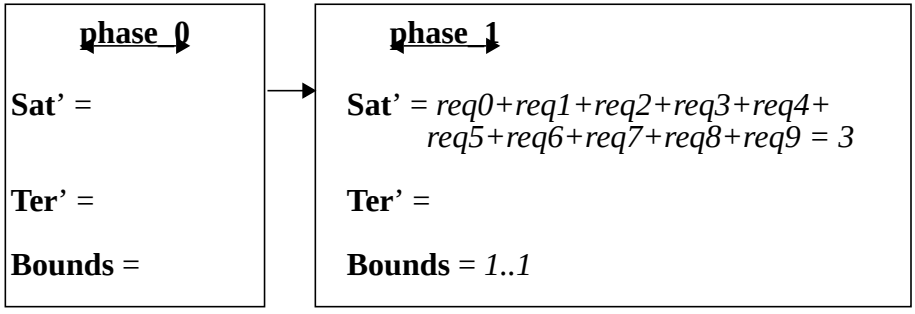
2. Suppose you wish to find a path in which a request (*req*) of 2 cycles is followed by an acknowledge (*ack*). The phase definition should be the following:



Note, the first phase should be empty in order to get paths where the two subsequent requests do not necessarily start in cycle 0.

3. Suppose you wish to find a path where in some cycle there are exactly **3** requests out of **10** possible ones (req0, req1, req2, ..., req9).

The phase definition should be the following:



The satisfying condition makes sure that exactly 4 of the request are active (get the value ‘1’) and the rest of the request are inactive (get the value ‘0’).

4. In a similar way, ***orthgonality*** of signals can be requested. Suppose you have a vector VEC(0..7), and you want to see exactly one bit of that vector active at a time. In the satisfying condition you can write:

$$VEC(0) + VEC(1) + \ldots + VEC(6) + VEC(7) = 1$$

CHAPTER 7 *Search and results Analysis*

To search for a given path select this path (click at its name in the pathlist area) and then choose if you want to use extra engines: ‘*interactive SMV’* or *‘beelze-bub’* or both, apart from the SMV engine.

if you want to add extra engines click ‘*configure’*,and choose ‘*engines’* option.

click the ‘***Find Trace***’ button. This step should be done only after the design is successfully compiled and all required input setting or path restrictions have been defined. Currently Pathfinder allows a single search process to run.

The status bar reflects the status of the search process as follows:

- ‘***Searching***’ - Pathfinder is searching for the specified path.
- ‘***Search OK***’ - Pathfinder completed searching for the path.
- ‘***Search errors***’ - search failed.

In case of a search failure a popup window will show, telling you that the search failed.

To kill a search click the '***Kill search***' button.

By default, Pathfinder looks for several paths that follow the path specification. The number of paths to look for is (currently) hardcoded in Pathfinder. After the search successfully completed, pathfinder loads the first trace to the '*Scope*' pane. In addition Pathfinder marks each signal in the Signals-List with the following signs:

- '-' : The signal is not in the cone of influence of the path.
- '**+**' : The signals is in the cone of influence of the path.

7.1 Scope window

The Scope window is divided into three columns:

- Signals values
- Signals names
- Signals wave

The cycles and mapping to phases are shown at top of the wave area. The trace defines a line for each signal that appears in the path specification (terminating condition, satisfying condition, or restrictions). Other signals can be brought to the scope as described in the following sections.

Signal name

This column defines a line for each signal that appears in the scope. Originally it includes all signals that appear in the path specification. Any click at a signal marks it (with a green color). Another click at the signal un-mark it.

Right click at a signal name opens a popup menu with the following options:

- ***insert line*** - insert an empty line after the signal.
- ***delete signal*** - delete the signal from the scope.
- ***delete marked signals*** - delete all signals that are marked.
- ***Align names to the left*** -

- ***Aligh names to the rigth -***
- ***slice vector***(only to data signal) - display the wave of a given slice of the vector. Selection of this option opens a popup menu that enables you to define the range of the slice. The wave of the slice will show right after the given vector.
- ***slice to bits***(only to data signal) - slice the vector into bits and display the wave of each bit separately. The slices will show right after the given vector.
- ***Create list of Global visible signals*** - create a file with the (names) list of all signals that currently appear in the scope. These signals will appear in each trace that will be brought to the scope.
- ***Remove list of Global visible signals*** - remove the file with the visible signals.
- ***Trace signal backward (sources)*** - display a list of all signals (of the design) that are sources of the selected signal. Pathfinder allows you to select a signal or a group of signals from this list and move it/them to the scope. To select a signal click at the signal. To select a group of signals hold the CTRL key while selecting the other signals. To select a range of signals click the first item of the group and then hold the SHIFT key while selecting the last item.
- ***Trace signal foreword (sinks)*** - display a list of all signals (of the design) that are driven by the selected signal. Pathfinder allows you to select a signal or a group of signals from this list and move it/them to the scope.
- ***Show cone signals*** - display a list of all in-cone signals. Pathfinder allows you to select a signal or a group of signals from this list and move it/them to the scope.
- ***Print Scope*** - Create a pdf file of the scope, a printable file of the scope.

Signal value

The ‘*values’* column reflects the value of each signal at the cycle pointed by the red vertical wave pointer. By default this vertical pointer resides in cycle 0. Any left click at some point in the wave area moves the vertical red pointer to that point. The values given in the values column will change to reflect the sig-

nals values at the new point of time. If you grab this vertical line the ‘values’ column also changes to reflect the signals value at each given point of time.

Signal wave

The wave area shows the behavior of each signal along the phases and cycles. Right click at the wave of a given signal opens a popup menu with the following options:

- ***Add/remove a marker*** - add a vertical line (marker) to the wave if a marker does not exist at this point. Remove a marker if a marker already exists at this point
- ***Remove all markers*** - remove all markers that exist in the wave.
- ***Zoom in*** - horizontal zoom in
- ***Zoom out*** - horizontal zoom out
- ***Zoom to fit*** - adjust the wave’s zooming to fit the scope frame size
- ***Vertical zoom in*** -
- ***Vertical zoom out*** -
- ***Change base*** - see description for signal names.
- ***Create list of Global visible signals*** - see description for signal names.
- ***Remove list of Global visible signals*** - see description for signal names.
- ***Print Scope*** - see description for signal names.
- ***set as force*** - restrict the chosen signal to behave the same way as in this trace, when searching for another trace.

Any ‘cone’ signal found in the ‘Signal List’ column (inputs, outputs, internals and user defined) can be brought to the scope as follows:

- To bring a signal to the scope select this signal (left click) and then right click. A popup menu with the option ‘*move to scope*’ is invoked. The ‘*append*’ option brings the signal to the bottom of the scope. The ‘*insert*’ option brings the signal after the first marked signal in the scope.

- To bring a group of signals to the scope, you should first select the group in one of the following ways:
  - Click the first signal of the group, then press the *CTRL* key while selecting the other signals.
  - Click the **first** signal of the group and then press the *Shift* key while selecting the **last** signal of the group. All signals in between will be selected. This option is applicable only for groups whose members are consecutive in the list.

  Right click to invoke the dialog with the '*move to scope*' option.

Pathfinder saves the state of each trace (signals in the scope, empty lines, order between signals). The state is used for future invocations of the trace.

7.2 View multiple traces

If possible, Pathfinder looks for several traces that satisfy the path specification. If more than one trace is available, a click at the '*Load trace*' button opens a popup menu that enables you to select the trace to be loaded. The index of the current loaded trace appears at the title of the scope pane. The button '*Next trace*', cyclically, loads the next trace to the scope.

Multiple traces are made as different from each other as possible. Nevertheless, it is sometimes difficult to detect the signals on which traces are different. For that, PathFinder provides a '***compare-traces***' feature. To activate it, press on the ***path*** menu in the top menu bar, and select the ***Compare traces*** option. You will be provided with a dialog, in which you have to select the traces to compare. Once selected, press **OK,** and PathFinder will show a list of all different signals. You will then be able to select signals of interest, and move them to the scope, just as it is done in the Show cone signals option.

7.3 Interactive Mode

Interactive mode is our name for a new feature of PathFinder's Search Engine *SMV.* In this mode of operation, SMV searches for the wanted path as usual. When a trace is found, SMV does not exit, but saves all information in memory, and waits for new commands from user, concerning the current path.

In order to operate this mode, choose **Configure->Engines->SMV is interactive.** The next search will run in *Interactive mode*. When reasch is finished and result trace is displayed on the scope, a pop-up menu will appear on the screen, asking for new requests/command from *SMV.* To hide the interactive menu, press '**DONE**'. You can resume this menu by pressing on the **INTERACTIVE** button on the top bar. *SMV* will stay in *interactive mode*, ready for your new requests until you deliberately tells it to exit (see below). While *SMV* is active, the red '**Engine Active**' sign will appear at the bottom bar.

All requests from SMV in this mode, will relate to the original path. This path will be signed 'Active!' in the path list. The user may switch to another path, load traces, simulate, and operate any feature of PathFinder, except for the **find trace** feature.

The features currently supported are:

for the current path:

- **Add cycles**: Enter number of cycles you want to add to the current traces, and press **OK**. PathFinder (through the *SMV* engine) will prolong the traces by the indicated number of cycles. Note the added cycles have nothing to do with the specified path, but are complient with the model under test.
- **Add Traces**: Enter the number of additional traces you want to be produced for the current path, and press **OK**. PathFinder will try to produce the additional traces as different as possible from the 5 existing traces already produced. You can view all traces through the 'load trace' or 'next trace' buttons as usual.

- **Find a longer Trace:** find a different and longer trace from the current one, on the same path.

for a new path:

- **Find Trace**: When selecting this option, PathFinder will search for a trace for the current path, using a special algorithm, called the *Merging* Algorithm.This algorithm uses the information gathered in the previous run, to speed up the new search. When the current path has a lot in common with the original path, there is a big chance that using the *merging algorithm* will be much quicker than starting the traditional search from scretch.
  Note that the only way to use this new algorithm is by changing the definition of the *Active* path.
  There are some limitations when using the *merging* algorithm:
  - Both the original path and the refined path must begin with an empty phase.
  - Signals mentioned in the new path must have the same cone-of-influence (or a subset of) as the cone-of-influence of the original formula.

CHAPTER 8 *Simulation*

Pathfinder enables you to modify the inputs of a given trace and to run simulation on these new values of inputs.

To edit inputs of a given trace bring the trace to the scope window (load trace), and click the '*Simulator*' option (at the right upper corner of your Pathfinder application). When switching to this mode, Pathfinder marks all signals other than inputs with a different color denoting that only inputs can be edited.

Middle click at a given bit-input (in the wave area), at a given cycle, toggles its value (at that specific cycle) from '0' to '1' and vise versa.

Middle click at a given vector-input (in the wave area), invokes a popup window that enables you to give a different value to the vector in a given range of cycles.

To run simulation on the new input values click the '*Simulate*' button.

The status bar reflects the status of the simulation as follows:

- '***Simulating***' - Pathfinder runs simulation on the new inputs.

- '***Simulation OK***' - Pathfinder completed simulation.

The resulted trace will appear in the scope area when simulation terminates.

Note!

The resulted trace is no longer connected to the path specification and to its phases definition.

8.1 Save/Load simulation traces

When in *Simulator* mode, you can save the current simulation trace, or load a previously saved simulation trace.

Click on the '*Simulate*' menu in the top menu bar, to get a popup menu with the following options:

- ***Simulate*** **-** The same as pressing the '*Simulation*' button.
- ***Save current simulation*** **-** Saves the trace currently appearing on scope. a popup window will require you to provide a name for the saved trace.
- ***Load simulations*** - Gives a list of all previously saved simulation traces. After choosing a trace name, you can display it on scope or remove it.

CHAPTER 9

Advanced

9.1 Setting your own compilation path

By default, PathFinder uses the TexVHDL compiler to compile your design. If you wish to use your own specific compilation path do the following.

1. **Add your script to the compilation configuration file:**

   Edit the file $PFROOT/etc/comp.pfcnfg

   This file should hold information about your new compilation script.

   The file includes lines of the format:

   <script_identifier> <script_path> <output_format>

   For example

   rb_texvhdl /afs/haifa/..../PFROOT/TexVHDL vim

   Each line should hold the following information:

   - ***script_identifier*** - a keyword to identify your script. This keyword will appear as one of the compilation path options in the 'Setup' dialog of Pathfinder (click the '**Configure**' option then select '**Setup**').
   - ***script_path*** - a pointer (full path) to your compilation script

- ***output_format*** **-** the format of compilation outputs. Can be either '**vim**' or '**proto**'

To use your compilation path just add a line that represents your own path.

2. **Adjust your script to the required format**

The compilation script (pointed by $PFROOT/etc/comp.pfcnfg) should be of a special format and should get a very specific list of parameters. If your own compilation script has a different format, then you may call your own script from this (top level) script.

The parameters sent by Pathfinder to the script are the following:

- ***make_file_name***:

  A path to the file containing the names of the VHDL files to be compiled (the contents of the '*DUT makefile*' option in the '*Setup*' dialog).

- ***entity***:

  Name of the top level entity (the contents of the '*DUT entity name*' option in the '*Setup*' dialog).

- ***architecture***:

  Architecture of your top level entity (the contents of the '*DUT architecture name*' option in the '*Setup*' dialog).

- ***working_directory***:

  A path to your working directory. The compilation output should reside under this directory.

- ***log_file_name***:

  Name of log file to where Pathfinder should redirect its log messages.

Script outputs:

- If your compilation output format is '**proto**', then the output of your script should be a file:

  <entity_name>.<architecture_name>.proto

  in the directory:

<working_directory>/dbout

- If your compilation output format is '**vim**', then the output of your script should reside under the vimdbase directory as illustrated in Figure 2 .

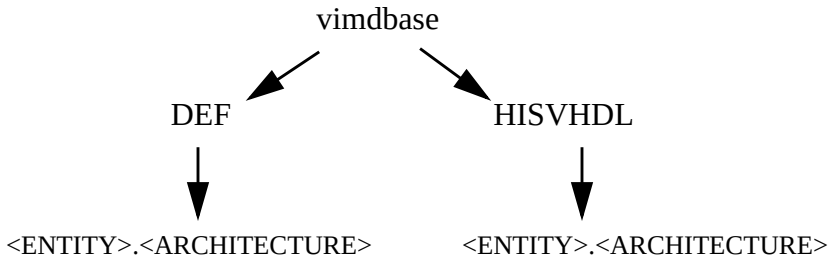


FIGURE 2. **vim format**

3. **Set Pathfinder to point to your script**

   Click the '**Configure**' option at the menu bar and select the '**Setup**' option. The name of your script should appear as one of the '**compilation path**' options. Select it to be your compilation path.

Appendix A

Specification Language

A.1 Expressions

A.1.1 Variables and constants:

The basic expressions are numbers, enumerated constants, or variable references.

A **number** is :

- A **decimal** if it has only decimal digits and no suffix (e.g. 1276).
- A **binary** number consists of binary digits and ends with ‘B’ (e.g. 1011B).
- A **hexadecimal** number begins with a decimal digit, has hexadecimal digits and ends with ‘H’ (e.g. 7FFFH, 0FFH).

An **enumerated** constant is one of the symbolic values which a variable can take on. For instance, if we declare the following:

var state: {idle, st1, st2, st3, waiting};

then each of the 5 symbolic values “idle”, “st1”, “st2”, “st3”, and “waiting” are enumerated constants.

A variable reference has one of the following formats:

name-- simple variable
name (number)-- one bit of array
name (number..number)-- a range of bits

A.1.2 Operators

An expression can be a combination of sub-expressions, connected by operators:

Boolean connectives:

! exprnot
expr & exprand
expr | expror
expr ^ expr (or: expr xor expr)xor
expr -> exprimplies
expr <-> expriff (xnor)
(Boolean operations can be applied only to boolean expressions.)

Relational operators:

expr = exprequals
expr != exprnot equals
expr > exprgreater than
expr >= expr greater than or equals
expr < exprless than
expr <= exprless than or equals
(>, >=, < and <= can be applied only to integer or boolean expressions.)

Arithmetic operators:

expr - exprminus

expr + exprplus

expr * exprmultiplication

expr / expr division

expr **mod** exprmodulo

(Arithmetic operators can be applied only to integer and boolean expressions.)

A.1.3 Operator precedence and associativity

The following operators are listed in decreasing order of precedence (the first ones are the strongest):

++ (concatenation)

! (not)

+ -

* / mod

= != < <= > >=

Temporal operators (will be introduced in CHAPTER 5)

& (and)

| (or)

xor ^

<-> (iff)

-> (implies)

All the operators, except ->, have left to right associativity.

Use parentheses in any case that you don’t know or don’t remember the precedence. Even if you know, others may find explicit parenthesizing easier to read and understand.

A.1.4 Case and If expressions

EDL provides two constructs which express a choice between two or more expressions. They are the **case** and **if** expressions, described below.

The **case** expression has the following format:

case
  $condition_1$: $expr_1$;
  $condition_2$: $expr_2$;
  ...
  else : $expr_n$;
esac

A **case** expression is evaluated as follows: $condition_1$ is evaluated first. If it is true, $expr_1$ is returned. Otherwise, $condition_2$ is evaluated. If it is true, $expr_2$ is returned, and so forth. Although the **else** part is not essential, it is advisable to use it as the default entry if you are not certain that the other conditions cover all the cases. Falling through the end of a case statement may have unpredictable results. Notice that from the description of the case expression above, it follows that an earlier condition takes precedence over a later one. That is, if two conditions are true, the first takes precedence.

The **if** expression is shorthand for a case with two entries. If has the following format:

if condition then exprA else exprB endif

In the above **if** expression, *exprA* is returned if *condition* is true, and *exprB* is returned if *condition* is false.

A.1.5 Built-in functions

The built-in functions **fell**() and **rose**() have the following functionality:

- **fell**(expr) is true if expr is 0, and was 1 on the previous cycle
- **rose**(expr) is true if expr is 1, and was 0 on the previous cycle

The usage of **fell** and **rose** results in additional state variables, one for each expression to which they refer. However, multiple references to the same variable will add only one extra variable.

A.1.6 The var statement

A **var** statement declares state variables. It has the following format:

var name, name, ... : type; name, name, ... : type; ...

The type can be one of the following:

- boolean
- { enum1, enum2, ... }
- number1 .. number2

(Arrays will be described in Section A.2)

For instance, the following are legal **var** statements:

var request, acknowledge: **boolean**;
var state: {idle, reading, writing, hold};
var counter: {0, 1, 2, 3};
var length: 3 .. 15;

The first statement declares two variables, “request” and “acknowledge”, to be of type boolean. The second statement declares a variable called “state” which can take on one of four enumerated values: “idle”, “reading”, “writing” or “hold”. The third statement declares a variable called “counter” which can take on the values 0, 1, 2 and 3. The fourth statement declares a variable called “length” which can take on any of the values between 3 and 15, inclusive.

A **var** statement only declares state variables. The **assign** statement, described below, defines the behavior of these variables.

A.1.7 The assign statement

An **assign** statement assigns a value to a state variable declared with a **var** statement. It has one of the following formats:

assign init(name) := expression;
assign next(name) := expression;
assign name := expression;

The first statement assigns an initial value to a state variable. The second statement defines the next-state function of a state variable. A state variable is simply a memory element, or register (flip-flop or latch). The third statement assigns a value to a combinational variable.

The following are examples of legal **assign** statements:

```
assign init(state) := idle;
assign next(state) :=
    case
       reset : idle;
       state=idle : busy;
       state=busy & done :  idle ;
       else : state;
    esac
```

The keyword **assign** may be omitted for the second and following consecutive **assign** statements. Thus, the following:

```
assign var1 := xyz;
          init(var2) := abc;
          next(var2) := qrs;
```

is equivalent to:

assign var1 := xyz;
assign init(var2) := abc;
assign next(var2) := qrs;

A.1.8 The define statement

A **define** statement is used to give a name to a frequently-used expression, much like a macro in other programming or hardware description languages. The **define** statement has the following format:

define name := expression;

For instance, the following are legal **define** statements:

define adef := (q | r) & (t | v);
define bb(0) := q & t; cc := 3;

As with the **assign** statement, the keyword **define** may be omitted in second and following consecutive **define** statements.

A.1.9 %for

The **%for** construct replicates a piece of text a number of times, with the possibility of each replication receiving a parameter. The syntax of the %for construct is as follows:

%for <var> %in <expr1> .. <expr2> %do
...
%end

or:

%for <var> in <expr1> .. <expr2> step <expr3> do
...

```
%end
  -- step can be negative
```

or:

```
%for <var> in { <item> , <item> , ... , <item> } do
...
%end
  -- where <item> is either a number, an identifier, or a string in double-
quotes.
  -- When the value of an item is substituted into the loop body (see below),
  -- the double quotes will stripped.
```

In the first case, the text inside the %for-%end pairs will be replicated expr2-expr1+1 times (assuming that expr2>=expr1). In the second case, the text will be replicated (|expr2-expr1|+1)/expr3 times (if both |expr2-expr1| and expr3 are positive or both are negative). In the third case, the text will be replicated according to the number of items in the list.

During each replication of the text, the loop variable value can be substituted into the text as follows. Suppose the loop variable is called “ii”. Then, the current value of the loop variable can be accessed from the loop body using the following three methods:

- The current value of the loop variable can be accessed using simply “ii” if “ii” is a separate token in the text. For instance:

```
%for ii in 0..3 do
  define aa(ii) := ii > 2;
%end

is equivalent to:
define aa(0) := 0 > 2;
define aa(1) := 1 > 2;
```

```
define aa(2) := 2 > 2;
define aa(3) := 3 > 2;
```

- If “ii” is part of an identifier, it can be accessed using %{ii} as follows:

```
%for ii in 0..3 do
  define a%{ii} := ii > 2;
%end
```

is equivalent to:

```
define aa0 := 0 > 2;
define aa1 := 1 > 2;
define aa2 := 2 > 2;
define aa3 := 3 > 2;
```

- If “ii” needs to be used as part of an expression, it can be accessed using %{<expr>} as follows:

```
%for ii in 1..4 do
  define aa%{ii-1} := %{ii-1} > 2;
%end
```

is equivalent to:

```
define aa0 := 0 > 2;
define aa1 := 1 > 2;
define aa2 := 2 > 2;
define aa3 := 3 > 2;
```

The following operators can be used in pre-processor expressions:

```
=  !=  <  >  <=  >=  -  +  *  /  %
```

In the current version, operators work only on numeric values, i.e. it’s ok to write

```
%for i in 0..3 do
   i   %if i != 3 %then   +   %end
%end
```

But it is not possible to write

```
%for command in {read, write} do
...
   %if command = read %then-- doesn't work!
...
```

A.1.10 Reserved words

The following words are keywords and should not be used as identifiers:

a abf abg af ag always as_in assign ax before before! before!_ before_ boolean bvtoi case define e ebf ebg ef eg else endif env envs esac ex fairness false fell forall formula formulas if in init inherit instance itobv mod mode module next next_event next_event! override rep zeroes ones nondets rose rule test_pins then true u union until until! until!_ until_ var w whilenot whilenot! within within! xor

If a keyword is prefixed with the '\' character, it becomes a regular identifier.

A.2 Arrays

It is often convenient to define arrays of state variables and to apply operations to entire arrays or to ranges of indices. Boolean arrays (buses, bundles) are the most common, but other types of arrays (integer sub-range, enumerated constants) are also useful. Hence RuleBase is oriented mainly toward boolean arrays, but supports other types of arrays also.

A.2.1 Defining arrays

An array of state variables is defined as follows:

var name (index1 .. index2) : type ;

It actually defines (|index2-index1|+1) state variables named name(index1), ..., name(index2), where index1 can be either greater or less than index2.

Examples:

```
var
  addr(0..7) : boolean;    -- 8 boolean variables, addr(0), addr(1), ... , addr(7)
  counter(4..5) : 0..3;    -- 2 integer variables, each can have the values 0,1,2,3
  status(3..0) : {empty, notempty, full };
                           -- 4 variables, each can have the values empty, notempty, full
```

An array can also be defined with a **define** statement:

define name(index1 .. index2) := <expr>;

Example:

define masked_sig(0..3) := sig(0..3) & mask(0..3);

A.2.2 Operations on arrays

Reference:

The simplest operation on an array is a reference to a bit or a bit range. One bit of an array is referenced as *array_name(N)* where *N* is a constant. A range of bits is referenced as *array_name(M..N)*. It is always necessary to specify the bit range when referencing an array.

It is possible to access an array element using variable index:
array_name(V: index1..index2) where *V* is a integer variable, and *index1..index2* are constants indicating its range. Example:

var source(0..7): **boolean**; V: 0..7;

define destination := source(V:0..7); -- assuming that the behavior of V is define elsewhere

Other operations that can be used with any type of arrays are:

:= = != if case

Example: aa(0..7) := **if** bb(0..2)=cc(0..2) **then** (dd(0..7) **else** ee(1..8) **endif**;

The rest of the operators can be applied to boolean arrays (bit vectors) only.

Boolean connectives (bitwise): & | ^ ! -> <->

Both operands must be of the same width (unless one of them is constant). The result will have the same width as the vector operands.

Example: v(0..7) := x(0..7) & y(0..7) | !z(0..7);

Relational: < > <= >=

Both operands must be of the same width (unless one of them is constant). The result will be a scalar boolean value.

Examples: c := v(0..7) > x(0..7); d := v(0..7) <= 16;

Arithmetic (unsigned): **+ - ***

Both operands must be of the same width (unless one of them is constant). The result will have the same width as the vector operands.

Examples:

```
define cc1(0..7) := aa(0..7) + bb(0..7);
       cc2(0..7) := aa(0..7) + 1;
       cc3(0..7) := 10 * aa(0..7);
```

In order not to lose the most significant bits of the result, pad the operands with zeroes on the left. Examples:

```
define aa(0..7) := zeroes(4) ++ bb(0..3) * zeroes(4) ++ cc(0..3);
       co++sum(0..7) := 0++a(0..7) + 0++b(0..7);
```

(++ is the concatenation operator, described below. zeroes(4) is a vector of four zeroes)

Shift: >> <<

The first operand must be a boolean vector and the second operand must be an integer constant or variable. The result is a boolean vector of the same width as the first operand. These operations perform the logical shift, i.e vacated bit positions are filled with zeroes.

Examples:

```
define cc(0..7) := aa(0..7) << 2;
var shift_amount: 0..5;
define dd(0..7) := bb(0..7) >> shift_amount;
       ee(0..8) := 0++ff(0..7) << 1;
```

A.2.3 Conversion of bit vectors to integers and vice versa:

Bit vector to integer:

bvtoi(a_vector)

Integer to bit vector:

itobv(an_integer)

Example:

assign next(counter(0..7)) := **itobv**(**bvtoi**(counter(0..7)) + 1);

Note that constant integers are converted to bit vectors implicitly - there is no need to apply itobv. It is recommended to use bit vectors instead of big integer variables, if possible.

A.2.4 Construction of bit vectors from bits or sub-vectors

The concatenation operator (++) is used to make bit vectors out of bits or smaller vectors:

expr ++ expr

Example:

define wide(0..5) := narrow(2..3) ++ bit1 ++ bit2 ++ another_narrow(0..1);

If expr is a constant, it should be either 0 or 1. Wider constant vectors should be splitted into separate bits.

define x(0..5) := y(0..2)++1++0++z; -- allowed
define x(0..5) := y(0..2)++10B++z; -- not allowed

The concatenation operator can also appear on the left-hand-side of an assign or define statement. For instance, the following statement:

define a ++ b ++ c(0..2) := d ++ 1 ++ 0 ++ e(0..1);

is equivalent to the following four statements:

define a := d; b := 1; c(0) := 0; c(1..2) := e(0..1);

The built-in construct **rep**() can help to construct arrays of repeated elements:

rep (expr, N) is equivalent to expr concatenated with itself N times. For instance, to make each bit of array 'arr' non-deterministic, the following assignment could be used:

```
assign arr(0..3) := rep({0,1},4);          --  {0,1}++{0,1}++{0,1}++{0,1}
```

Shorthands:

zeroes(N) is equivalent to **rep**(0,N)

ones(N) is equivalent to **rep**(1,N)

A.2.5 Array Notes

- **The exact range must be specified in the operation**. "a = b" is not equivalent to "a(0..3) = b(0..3)". b(0..3) represents variables b(0) through b(3) while b represents one variable with no index.
- Operands can take any ranges, provided that their widths are compatible. For example, "a(0..3) & b(1..4)" is legal, but "a(0..3) & b(0..4)" is not.
- If one of the operands is a boolean vector and the other is a numeric constant, the constant is considered an array of bits. For example, "a(0..1) = 10B" is equivalent to "a(0)=1 & a(1)=0" and "a(1..0) = 10B" is equivalent to "a(1)=1 & a(0)=0".
- "**var** v(0..3): { 5, 7, 13 }" defines 4 state variables, each of them can take the values 5 or 7 or 13. This is sometimes confused with "**var** v(0..3): **boolean**; **assign** v(0..3) := { 5, 7, 13 };" that defines a vector of 4 bits, and the whole vector can take the values 5 or 7 or 13.
- Arrays can be used as formal parameters of modules and as actual parameters of instances. The actual parameter width must match the width of the formal parameter.
- If you write "#define N 7" and later "a(0..N)", leave a space around the two dots: a(0 .. N). Otherwise the standard preprocessor (cpp) used by rulebase will identify ..N as a token and will not replace N by 7.

A.2.6 More array examples

```
var a(0..3), b(0..8), c(0..2) : boolean;
define d(0..3) := b(5..8);-- different sub-ranges
define e(0..2) := b(2..0) & c(0..2);-- different directions

var x_state(0..2), y_state(0..2): {s1, s2, s3 };
define same_state :=  x_state(0..2) = y_state(0..2);

assign next( a(0..2) ) :=
  case
    reset : 0;
    a(0..2) = b(0..2) : c(1..3);
    a(0..1) = 10B : d(0..2);
    else : a(0..2);
  esac;

var counter(0..7) : boolean;
assign
  init( counter(0..7) ) := 0;
  next( counter(0..7) ) := counter(0..7) + 1;
```