User's Guide

RuleBase

Formal Verification Tool

Version 1.4

Verification Technologies Group
IBM Haifa Research Laboratories
June 2002

Provided to _____ on _____

by special agreement with IBM

Notices

RuleBase User's Guide
First edition (1996)
Date modified June 2002
For information regarding RuleBase, contact: Gil Shapir (shapir @il.ibm.com)
Tel: 972-4-8296258

Contents

CHAPTER 1

Introduction

1.1 Overview

Traditionally, logic verification is done by simulation. In simulation, a test vector is applied to the logic model, and the results of the simulation are examined. Both the generation of the test vectors and the examination of the results can be done either automatically using a special-purpose tool, or by hand.

Coverage is one of the major problems associated with simulation. Since we cannot exhaustively simulate all possible sequences of input vectors, we need a way to decide when enough input vectors have been applied in order to give us reasonable confidence that our design functions as intended.

Formal verification is a novel technique for logic verification of hardware designs. It attempts to address the problem of coverage by mathematically proving that a design is correct with respect to its specification. There are many approaches to formal verification. RuleBase, a formal verification tool developed by IBM, uses an approach known as "model checking", which is equivalent to exhaustive simulation of the circuit for every possible input sequence. In model checking, the specification consists of a set of properties to be

proved. For example, “if signal x is asserted, then within three clocks, signal y will be de-asserted”, or “signals z and w will never be asserted together”. If the property is true, the designer is notified. If the property is false, a counter-example is provided. The counter-example is a waveform that shows a simulation sequence that proves that the property is false.

The main advantage of model checking over simulation is that it frees the designer from the need to generate test vectors. Model checking checks the properties specified for every possible input sequence. However, most chips are not designed to accept every possible input sequence, so if a given property fails for an illegal input sequence, it is of no interest. Thus, we need a way in which to specify all the legal input sequences to the formal verification tool. We can do this by specifying a model of the expected environment. This model describes the legal input sequences to the design under test.

One of the practical problems of model checking is known as “the size problem”. Because of the size problem, complete model checking runs can verify designs that have a few hundred state variables (latches or flip-flops). This is not enough to be useful in real hardware designs.

The RuleBase formal verification tool solves the size problem by renouncing the proof of truth that is possible with model checking on small designs. By renouncing the proof of truth, RuleBase can verify designs that contain up to a few thousand state variables. Although an answer of “true” to a specification is no longer a firm indication that the design is correct, an answer of “false” with a counter-example is an indication of a bug in the design (or specification or environment). This way, RuleBase can be used to obtain much better verification than is possible using simulation alone, even for designs that are too large for complete model checking.

One of the ways of dealing with the size problem is to reduce the design under verification. Reduction is accomplished by analyzing the environment description provided by the user as well as the specification to be checked, and eliminating any logic that has no bearing on the specification under the environment. Using the techniques of reduction in combination with renounce-

ment of the proof of truth is known as over-reduction. For instance, instead of describing the complete environment of the design under test, the user may choose to describe a subset of that environment. RuleBase uses the environment to reduce the design to a size that is suitable for model checking. Then, another subset of possible behaviors can be described. Thus, the user has complete control over the reduction process. An answer of “true” for a specification under a specific environment indicates that in this specific environment, the specification is true, but it does not indicate anything about the truth or falsity of the specification under other environments.

1.1.1 About This Document

The remainder of this document is structured as follows:

- **CHAPTER 2: Getting Started –** explains how to access RuleBase, how to set up a verification environment, and how to prepare a design for formal verification.
- **CHAPTER 3: Tutorial –** provides a hands-on introduction to RuleBase in the form of a tutorial. The tutorial presents a small design of a buffer and shows how to verify it under RuleBase.
- **CHAPTER 4: Describing the Environment –** explains how to specify environment behavior and discusses arrays, non-determinism, fairness, clocks, and more.
- **CHAPTER 5: Sugar – The RuleBase Specification Language –** describes both CTL and Sugar, and the models on which they operate.
- **CHAPTER 6: Sugar – Formula Examples –** includes a list of useful formula patterns, mainly for novice users.
- **CHAPTER 7: Managing Rules, Modes, and Environments –** suggests how to manage verification projects.
- **CHAPTER 8: Size Problems and Solutions –** discusses the techniques used to extend the design size limit as far as possible.
- **CHAPTER 9: Debugging Aids –** describes various debugging aids that are part of the RuleBase tool.
- **CHAPTER 10: Graphical User Interface: Tool Controls and Options –** describes the tool controls and options available, and how to set them from the graphical user interface.

- **CHAPTER 11: Design for Formal Verification –** presents some practical design guidelines to aid in formal verification.
- **CHAPTER 12: Coverage Methodology –** describes some ways to approach the problem of completely covering the block when proof of truth is not possible because of size problems.
- **CHAPTER 13: Advanced Verification Engines (RuleBase Premium) –** describes additional verification engines that are included in the RuleBase Premium version.
- **APPENDIXAPPENDIX T: Option tables –** describes different settings and options which can be adjusted in order to enhance RuleBase's performance.

CHAPTER 2

Getting Started

2.1 Accessing RuleBase

Before running RuleBase for the first time, perform the following:

Note: *The instructions below are for csh users; if you are using another shell, use the appropriate replacements.*

- In your home directory, in the file *.cshrc*, add the following lines:
  setenv RBROOT <directory>
  alias rb "$RBROOT/guirb.bat"
  where <directory> is the full path to the directory in which RuleBase binary files are installed.
- To bring these settings into effect, enter **source .cshrc**.
- Check to make sure you have access to $RBROOT.
  If you do not, call the local RuleBase focal point, or contact us (see the cover page for our email address).
- Copy the following files from $RBROOT to your home directory:
  Guirb, **Scope**, **Cctdag, Analyze**

2.2 Preparing the Verification Environment

This section provides an example of how to quickly build a working environment. These instructions should help you create an initial environment with which to experiment, and are not meant to give you a complete understanding of working with RuleBase.

Note: *Some of the file names (e.g., envs and rules) are only recommendations and you may select other names for these files.*

You must first create a new directory in which the verification process will take place. Your verification files will be located in this directory, and RuleBase will also create various files and sub-directories in this directory.

Before running the first rule, prepare:

- The design to be verified
- File *rulebase.setup*
- File *envs*
- File *rules*
- File *run*

We describe each of these items in the following sections.

Note: *RuleBase supports several hardware description languages (VHDL, DSL, Verilog) and several translation/synthesis paths. "Design Translation" on page 20 details how to prepare the design for verification. If your design environment is not mentioned, please contact us. CHAPTER 11: Design for Formal Verification suggests design rules that can ease the verification process (e.g., proper partitioning).*

2.2.1 "rulebase.setup"

This file should exist in the verification directory and must include (at least) the following four lines:

- **setenv entity <DESIGN_NAME>**
  This is the name of the top-level entity of your design (in upper case). “Design Translation” on page 20 explains what is considered an entity in each of the translation paths.
- **setenv name <design_name>**
  This is the name of your top-level design file (without the extension). “Design Translation” on page 20 explains what is considered a name in each of the translation paths.
- **setenv SYNTHESIS <path>**
  This is your translation path: it can be either DSL, HIS, HIS_VERILOG, SYNOPSYS, or VIM.
  See “Design Translation” on page 20 to determine which of these to use.
  If you need the Compass translation path, please contact us for instructions.
- **setenv database *envs***
  The file envs is where your environment models and rules are written. The file is described below.

RuleBase only reads the rulebase.setup file once, at the beginning. Any change to this file requires that you either exit and re-start RuleBase or select the “File/ Read rulebase.setup” menu option.

2.2.2 "envs"

This file should include environment models. Although it is possible to mix models and specifications, we recommend that you separate them. Hence, environment models are in the *envs* file and specifications are in the *rules* file.

On the first line of file *envs* write:

#include “rules”

This way, RuleBase knows it should read the file *rules*. If you wish to write your environment models in several files, connect the other files to *envs* using the **#include** command.

To start working, you must give a behavior to every **input** signal of your design. To provide full legal behavior for each of your input signals, see CHAPTER 4: Describing the Environment before proceeding. In addition, we recommend that you read CHAPTER 4: Describing the Environment before beginning real verification work.

If you just want to try out RuleBase, you can give a simple (possibly incorrect) behavior to your input signals. For each signal choose one of three possible behaviors:

```
define SIGNAL1 := 0;
define SIGNAL2 := 1;
var SIGNAL3 : boolean;
```

The first two possibilities assign a constant value to the input signal. The third one gives an input signal totally free behavior: SIGNAL3 may change on every cycle. A signal given this behavior is called “a free variable”. At this stage, you do not want to leave too many variables free because it may cause a quick explosion of the state space. However, if you are only interested in seeing your design function, it is reasonable to leave five to ten signals as free variables at this stage.

Note: *In some translation paths, all signal names of the design are converted to upper case.*

Pay special attention to the **reset** and **clock** signals. For a complete discussion on how to model these signals, see “Modeling Reset” on page 105 and “Modeling Clocks” on page 102. For now, the simplest clocking scheme - one clock, is assumed to be sufficient.

1. Assign the clock signal the constant value ‘1’:

```
define CLOCK := 1;      -- where CLOCK is the clock name in your design.
```

2. Assign the reset signal the following behavior:

```
var reset_state : 0..3;     -- Assuming that three cycle reset is required
assign
    init(reset_state) := 0;
    next(reset_state) :=
      case
        reset_state < 3 : reset_state + 1;
        else            : 3;
      esac;
 define RESET := (reset_state != 3);        -- where RESET is your reset signal
```

If your design needs more than three cycles of active reset, you may increase the cycle length by changing ‘3’ to the desired number.

2.2.3 "rules"

Write your specifications in the *rules* file. Here, as in the *envs* file, you may write the rules in several files and use the **#include** directive to connect them.

Each rule should have the following format:

```
rule  <name> {
   “ <a comment describing the rule>”

   formula
      “ <a comment describing the formula>”
      {  <sugar-formula> }
   ...
   formula
      “ <a comment >”
      { <another sugar formula > }
}
```

A rule must have a unique name and may contain any number of formulas. Comments are optional in both the rules and formulas. In addition, a rule may contain environment models that override the default environment. For more information, see “Defining Rules and Modes” on page 149.

The most important part of the rule is its specification, written as a Sugar formula. We describe Sugar, the RuleBase specification language, in CHAPTER 5.

To get started with writing rules for RuleBase, choose an (important) **output** signal of your design, and write the following rule in your *rules* file:

```
rule start {
   "getting started"
   formula
     "just to see a rule running"
     { AG AF (<output-signal>) }
}
```

The above formula states that on every path, always, a state will exist in which <output-signal> has the value one.

You may write more formulas (either in rule *start* or in separate rules) to check real properties of your design. The most simple form of a formula is

```
formula { AG !( <some-bad-event> ) }
```

where <some-bad-event> stands for a Boolean expression that should never be true in your design. For example, if *enable1* and *enable2* are two signals that should never be active at the same time, the following formula can be used:

```
formula { "enable1 and enable2 are mutually exclusive" AG( !enable1 | !enable2 ) }
```

For additional formula patterns, see CHAPTER 6: Sugar – Formula Examples.

2.2.4 "run"

The *run* file is only needed for batch runs. However, we recommend that you prepare it now. Copy this file from $RBROOT to the working directory.

2.3 Running RuleBase

After preparing the four items described in the previous section, you are ready to run RuleBase.

1. Type **rb** in your verification directory.
   The RuleBase window will appear. A list of the rule names you defined (See “"rules"” on page 16) should appear on the left side. In our case, the rule “start” will appear.
   To the right of the rule list is a column of yellow push buttons that activate commonly used commands. There is also a large text area for displaying files. At the top of the window, there is a menu bar and a message line.
2. Select the **start** rule from the rule list and press the **Run** push button. RuleBase will start to run your rule.

Watch the log of your run as it appears in the RuleBase window. If the log scrolls too fast, you can use the scroll bar on the right hand side. When you touch the scroll bar, the bottom right **Freeze** button turns red and changes to **Frozen**. To see the updated log and free the display, press the red **Frozen** button.

The following describes the verification process:

First, the design is translated into an internal representation. The translated design is kept on the disk for use in future runs. The translation process will only be repeated for a new version of the design.

Next, RuleBase loads the design, the environment models, and the formulas into memory. At this time, RuleBase performs many types of checks, and gives warning messages where necessary.

Press the **Warning** push button to see a list of all the warning messages produced during the run. After you press the **Warning** button (or any other button), press the **Log** push button to display the log again.

Note: *Pay attention to the warning messages as they may indicate serious problems.*

Then, Reduction takes place. Reduction removes parts of the design that are not required for the verification of the formulas of the current rule. It also links those environment models that resolve essential input signals to the design. Information regarding the size of the design (in terms of flip-flops and gates) is displayed before and after the reduction.

After reduction, the actual verification process begins. During verification, two types of messages appear continuously: 'nodes allocated <number>' and 'iteration <number>'. Whenever 'nodes allocated' grows too much, dynamic BDD ordering will try to reduce the number of nodes. Hopefully, at this stage of experimentation the reduced design is fairly small. That is, 'nodes allocated' are less than 500,000, 'iteration' is less than 200, and the total run time is a few minutes. Otherwise, you will have to reduce the design further by restricting some free inputs, or employ more advanced methods, as described later in this manual.

3. At the end of the run, press the **Results** quick button.
   Your rule will be displayed with one of three possible results: 'failed', 'passed', or 'vacuously'. If you get an 'unknown' result, it means that you pressed the **Results** button too early and your run was not finished.
   Press the **Log** push button to see the log again.

If the result is 'failed', it means that your formula does not hold true in your design, and a counter-example was produced. If the result is 'passed', your formula holds true. To see the counter-example of a failed formula, click the left mouse button near the word 'failed', then drag the mouse and choose **Show timing diagram.** See CHAPTER 9: Debugging Aids for instructions on how to use the timing diagram browser. If the result is 'vacuously', no timing diagram exists for this formula. This result may indicate a problem in the formula, environment or design (see "Debugging Aids" on page 166 for an explanation of vacuity).

Some of the formulas may have failed because the environment behavior is wrong, as some of the free inputs have unexpected behavior. We suggest that you use this opportunity to refine the environment model. You can try to make use of your short experience with the tutorial, or read CHAPTER 4: Describing the Environment to learn more about environment modeling.

4. After changing your environment or adding formulas, save the editor's buffer. Then, select the rule that you want to run from the rule list (if it is not already selected), and press the **Run** button again.
   If the name of a newly created rule does not appear in the rule list, select the "File/Refresh" menu option.
5. Repeat the process of refinement and analysis until all the rules that should pass, do pass.
   You may also add rules and formulas to cover all the interesting properties of your design.

Note: *To learn more about formulas and rules, read CHAPTER 5: Sugar – The RuleBase Specification Language, CHAPTER 6: Sugar – Formula Examples, and CHAPTER 7: Managing Rules, Modes, and Environments. Browse through the other chapters to learn more about tools and methods that RuleBase provides to ease successful verification.*

To exit RuleBase, select the "**File/Quit**" menu option.

2.4 Design Translation

RuleBase supports several Hardware Description Languages (HDLs) and several translation paths. Wherever possible, it uses existing tools of the design environment—compilers and synthesizers—to translate the HDLs into a lower level representation that only consists of basic gates and flip-flops. The following sections describe how to translate the design in some of the environments. If none of the environments described here meet your needs, please contact us.

2.4.1 CLSI and HIS / VHDL

The following section describes setting environment variables for CLSI and HIS/VHDL users.

2.4.1.1 Setting Environment Variables

To set the environment variables, add the following lines to the *rulebase.setup* file in your verification directory:

setenv name <TOP>

<TOP> is the top-level entity in your design (in capital letters)

setenv entity <TOP>

<TOP> is the top-level entity in your design (in capital letters)

setenv SYNTHESIS HIS

setenv SRC <directory>

The directory in which the VHDL files are located (optional)

setenv sources “<VHDL-files >”

<VHDL-files > is a list of VHDL file names separated by spaces.
The files should appear in bottom-up reference order.
The entire list should be written as one line.

It is usually enough to set the above environment variables in order to work with HIS/VHDL.

Note: *You only need to read the next two sections if you encounter problems. We recommend that you review the HIS compilation messages in order to locate possible problems.*

2.4.1.2 Setting CLSI and HIS Variables

The RuleBase focal point usually only needs to perform this setup once per site, in which case you may skip the rest of this section.

The $RBROOT/../his_aix/clsi.local file stores site-specific settings. It contains the following information:

setenv VTIP <vtip>

<vtip> is the directory in which the clsi compiler is located.

setenv LM_LICENSE_FILE <CLSI licence file>

For each of the VHDL libraries you use, add the following two lines:

setenv dls_<lib> <directory>

<lib> is the library name in lower-case

setenv <LIB> dls_<lib>

<LIB> is the library name in upper-case

Check to make sure that the libraries are CLSI-compiled, and that compilation is performed in bottom-up reference order.

You can have your own copy of the clsi.local file. If clsi.local exists in the verification directory, it is read instead of the central clsi.local.

2.4.1.3 Hints

1. If the VHDL attribute BTR_NAME is used with an entity, this entity will be synthesized as a black box, unless attribute RECURSIVE_SYNTHESIS is set to 1. RECURSIVE_SYNTHESIS can either be specified in an entity def-

inition or in a component instantiation. There is no way to specify it globally.

2. The GEN directive **range** is not supported; use **left** and **right** instead.
   Wrong way: GEN: for I in DataIn'range generate
   Right way: GEN: for I in DataIn'left downto DataIn'right generate

3. HIS needs to know all the pins that should be treated as bidi's. You can do this in one of the following two ways:
   - Attach attribute IO_PHYSICAL_DESCRIPTION = BI_DIRECTIONAL to each inout port.
   - Attach attribute PHYSICAL_PINS = TRUE to the Entity (then all its inout ports are considered bidi's).
     HIS will split every bi-directional signal into two signals (input and output) for the purpose of formal verification with RuleBase
4. Look for the string "DUMMY" in the compilation log file. If it appears, a cell library was missing and the compilation considered the cell to be a black box.

2.4.2 HIS/Verilog

The following section describes setting environment variables for HIS/VERILOG users.

2.4.2.1 Setting Environment Variables

Add the following lines to file *rulebase.setup* in your verification directory:

setenv name <TOP>

<TOP> is the top-level module in your design

setenv entity <TOP>

<TOP> is the top-level module in your design (in capital letters)

setenv SYNTHESIS HIS_VERILOG

setenv SRC <directory>

The directory in which the Verilog source files are located (optional)

setenv sources “<Verilog-files >”

<Verilog-files > is a list of Verilog file names separated by spaces.
The entire list should be written as one line.

2.4.3 TexVHDL

Note: *The following information may become inaccurate as a result of compiler changes. In cases of doubt, consult the TexVHDL Compiler Reference Manual.*

setenv name <ENTITY>

<ENTITY> is the top-level entity in your design

setenv entity WORK.<ENTITY>.<ARCHITECTURE>

<ENTITY> is the top level entity in your design and <ARCHITECTURE> is the top level architecture (both in capital letters)

setenv SYNTHESIS TEXVHDL

setenv sources <makefile>

<makefile> is the name of a file that contains a list of VHDL source file names, one name in each line. Files are listed in bottom-up order - referenced files appear before the referencing file.

setenv VHDLPATH <path>

<path> is a list of directory names, separated by colons, in which the source VHDL files, **not** including library source files, reside.

setenv DBIN <path>

<path> is a list of directory names, separated by colons, in which the compiled protos, including library protos, reside.

setenv TEXSIM_DIR <dadb_install_dir>

If you define TEXSIM_DIR, RuleBase will load DaDb tools from this directory. Otherwise, the tools will be loaded from $RBROOT (the RuleBase installation directory).

2.4.3.1 Working with TexVHDL Libraries

If the libraries are not yet compiled, compile them by following the instructions in the TexVHDL Compiler Reference Manual. Then, for each library add the following to the “rulebase.setup” file in the verification directory:

1. Define an environment variable whose name is the library name (in upper case) that points to the library source files. For example:
   setenv IEEE .../vhdl/source/ieee
2. Add the directory with the compiled protos to DBIN. For example:
   setemv DBIN “${DBIN}:.../vhdl/protos/ieee”

Library source files should not be included in the makefile.

2.4.4 Synopsys / VHDL

With the Synopsys translation path, you must compile the design into a gate-level description outside of RuleBase. The result should be a single gate-level VHDL file, <name>.vhdl, that only consists of **not** and **and** VHDL operators, and the component SYNOP_BASIC_FF. You can use the following dc_shell commands to create gate-level VHDL.

1. vhdlout_write_components = false
2. vhdlout_equations = true

3. verilogout_equation = true
4. verilogout_write_components = false
5. target_library = “gtech.db”
6. read -format vhdl { <vhdl_file1>, <vhdl_file2>, ... } /* Read VHDL files */
7. current_design = <top_level_entity_name> /* Specify name of top-level entity */
8. compile -no_map /* Compile with low effort */
9. replace_synthetic -ungroup
10. ungroup -all -flatten /* Sometimes more flattening is needed */
11. write -no_implicit -format vhdl -o <name>.vhdl /* Write gate-level VHDL */
12. quit

Add the following lines to the *rulebase.setup* file in your verification directory:

setenv name <name>

<name> is the gate-level VHDL file name without the extension

setenv entity <NAME>

<NAME> is the same as <name> but in capital letters

setenv SYNTHESIS SYNOPSYS

2.4.5 Synopsys / Verilog

The Synopsys/Verilog path is very similar to the Synopsys/VHDL path. In fact there are two paths:

- **Verilog to gate-level Verilog**
  Use the instructions in “Synopsys / VHDL” on page 25, but replace ‘-format vhdl’ by ‘-format verilog’ in lines 6 and 11. Then, translate gate-level Verilog to gate-level VHDL using the v2v tool provided with RuleBase. (This is a temporary workaround.)
- **Verilog to gate-level VHDL**
  Use the instructions in “CLSI and HIS / VHDL” on page 21, but replace ‘-format vhdl’ by ‘-format verilog’ in line 6.

2.4.6 DSL

RuleBase can read standard DSL files, including DSB library files. The only special preparation needed is for latches and flip-flops.

If you use the master outputs of a master-slave latch, or if you do not use master-slave latches or edge-triggered flip-flops, contact us for instructions to learn how to map your memory elements to standard RuleBase elements.

If you use master-slave latches and only use slave outputs, or if you use edge-triggered flip-flops, follow the directions in “Mapping Master-slave Latches and Edge-triggered Flip-flops” on page 27, to map your latches and/or flip-flops to standard RuleBase elements.

After mapping your memory elements, follow the directions in “Setting Environment Variables” on page 29 to set up a DSL environment for formal verification.

2.4.6.1 Mapping Master-slave Latches and Edge-triggered Flip-flops

1. Your DSL file should instantiate a device that represents the memory element (it should not make direct use of the “Register” statement of DSL).
2. Replace the desblo file that represents your basic memory element with a desblo file that instantiates the standard RuleBase register NBITREG.

An NBITREG is a 1-32 bit memory element that represents a simple master-slave latch or D-flip-flop. It has the following inputs:

- CLK – the clock.
- DATA_IN(0..N) – the data input.
- ASYNC_SET – an asynchronous set.
- ASYNC_RESET – an asynchronous reset

and the following output:

- DATA_OUT(0..N) – the data output.

If your master-slave latch or flip-flop has scan pins or other circuitry not directly related to the functionality of the memory element, they should be ignored (left unconnected).

Below is an example that maps a master-slave latch called “latch4l” with scan pins to the standard RuleBase memory element NBITREG. As you can see in the example, the scan pins and the slave clock are left unconnected.

```
/* 1 TO 32 BIT SRL REG  */
 SIM = SYN
 CALL CHECKPARM  ‘width’  1 32 1
 CALL CHECKPARM   ‘set’      0 1 0
 CALL CHECKPARM  ‘reset’     0 1 0
 CALL CHECKPARM  ‘nl2’       0 1 0
 CALL CHECKPARM  ‘hide’      0 2 0
 CALL CHECKPARM  ‘bhc’       0 3 0
 CALL CHECKPARM  ‘type’      0 6 0

 GENERATE
 BLOCK

  INPUT
      DI #bitrange#,
```

```
    SHIFTCLK        IS B"0",
    MASTERCLK,
    SLAVECLK        IS B"0",
    SCIN            IS B"0",
    SETL1           IS B"0",
    RESETL1_        IS B"1";

OUTPUT
    PL2OUT #bitrange#;

DEVICE   U  :  NBITREG (width=#width#)
MASTERCLK. . . . . | CLK              |
DI#bitrange#. . . . . .| DATA_IN      |
SETL1. . . . . | ASYNC_SET     |
RESETL1_. . . . . .| ASYNC_RESET  |
                   | DATA_OUT      | PL2OUT#bitrange#;
END BLOCK
END GENERATE
END SIM=SYN
```

If you have neglected to perform this step (mapping of memory elements to standard RuleBase memory elements), RuleBase will notify you with the following message:

Unknown box type: <a lowest-level register name>

2.4.6.2 Setting Environment Variables

These instructions use the following notation:

- <top>.dessrc – the top-level DSL file.
- <dir> – the directory in which formal verification will take place.

Add the following lines to the *rulebase.setup* file in the verification directory.

```
setenv name <top>
setenv entity <TOP>       # Same as <top> but in capital letters
setenv SYNTHESIS DSL
setenv database envs      # Name of environments file
setenv sources <top>.dessrc
setenv DSLPATH .:<directories containing relevant DSL files>
setenv DSBPATH .:<directories containing relevant DSB files>:$RBROOT
setenv DSLOUT .           # Directories in the above lists are separated by colons
```

2.4.6.3 Flip-flop Initialization

If you use a reset signal for initialization, connect it to the ASYNCH_SET or ASYNCH_RESET appropriately. If another initialization scheme is used (e.g., through the scan chain), it can be translated to a set of EDL statements (see CHAPTER 4: Describing the Environment). If you use Boeblingen-style srl initialization files, contact us.

2.4.6.4 Compilation Errors

In the case of compilation errors, see file compile.msg.

2.4.7 Koala Verilog Compiler

RuleBase comes with a native Verilog front-end, Koala, which can be invoked using the following settings:

```
setenv SYNTHESIS KOALA_VERILOG

setenv entity <TOP-LEVEL-MODULE-NAME>
    # the value of this environment variable is the name of the topmost
    # block in the design under test
```

setenv name <TOP-LEVEL-MODULE-NAME>

```
# for historical reasons, there should also be a definition of environment
# variable $name, with exactly the same value as for $entity
```

setenv sources <SOURCE-FILE-LIST>

```
# the value of this environment variable is a blank-separated
# list of verilog source files.
```

Example:

```
setenv SYNTHESIS KOALA_VERILOG
setenv entity dunit
setenv name dunit
setenv sources “dunit.v mux16_4.v arbiter.v”
```

If there are a lot of files in the model, it is convenient to create a wrapper file, for example “all_files.v”, which contains ‘include’ directives for the Verilog preprocessor to include all the model files. Then, use the following:

```
setenv sources all_files.v
```

2.4.8 Compass

If you wish to use the Compass translation path, contact us for instructions.

CHAPTER 3

Tutorial

3.1 Introduction

This tutorial presents a small design of a buffer, and explains how to verify it under RuleBase. Since RuleBase supports both VHDL and VERILOG, we cover both in this tutorial.

Note: *You can find a more comprehensive tutorial on our web site at: http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/tutorials.html*

After completing this tutorial, you should feel comfortable enough to begin using the RuleBase tool. However, we assume you have basic knowledge in logic design. It is important that you don't override the special options settings with which the tutorial comes, and that you perform all the steps in the specified order. Moreover, since the tutorial does involve code modifications, make sure that you start to work on a fresh unmodified copy.

All files referred to in this chapter can be found in the tutorial directory.

- The VHDL tutorial usually is located: $RBROOT../tutorial/tutorial_vhdl.

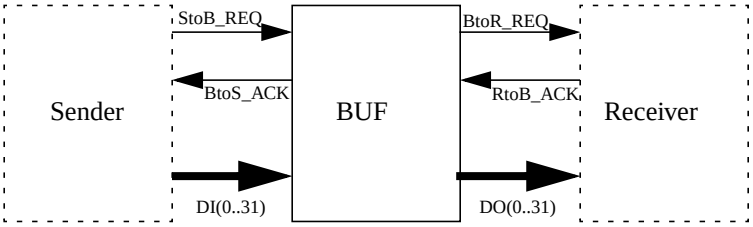
- The VERILOG tutorial usually is located:
  $RBROOT../tutorial/tutorial_verilog.

Make a private copy of this directory from which to run the tutorial.

We assume that you have access to $RBROOT and that you performed the initial setup as described in “Accessing RuleBase” on page 12.

3.2 Specification

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 (sender to buffer request). If BUF is free, it reads the data and asserts BtoS_ACK (buffer to sender 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.

3.3 BUF Implementation

An implementation of BUF, written in VHDL, is described in the *BUF.vhd* file (See “BUF implementation in VHDL” on page 44).
The VERILOG implementation resides in the *buf.v* file (See “Implementing BUF in VERILOG” on page 47).

From the source file, you can see that it consists of four parts:

1. State machine SENDER_INTERFACE – controls the interface with the sender state machine.
2. RECEIVER_INTERFACE – controls the interface with the receiver.
3. OCCUPIED_FLAG – a flag that indicates whether BUF has data.
4. DATA_BUFFER – a register that holds the 32 bit data.

Knowledge of implementation details is not mandatory, unless you want to fully understand the bug fix in the sequel. In this case, we suggest you read the source file: VHDL *BUF.vhd* or the VERILOG buf.v.

Depending on the user’s design environment, RuleBase supports several automatic translation paths of the implementation to a lower level format suitable for verification.

- **For VHDL users:**
  No specific VHDL translation path is set for this tutorial, and the VHDL file is already translated for you.
- **For VERILOG users:**
  RuleBase implicitly translates this file.
  To make the appropriate design available for verification, type **setup1** in the unix command line.

3.4 Modeling the Environment

This section explains how to assign behavior to primary inputs. If inputs are left unspecified, unexpected input sequences may induce incorrect behaviors of the implementation. These are called *false negatives* - “bugs” which result from a behavior that is impossible in the real environment.

Environment models are described in EDL (Environment Description Language). Models for this example are in the *envs* file. For the sake of clarity, all models are written in a uniform style. First, a module that describes the behavior is defined, and then the module is instantiated. This is similar to defining a function and then calling it. Both the sender and receiver require models.

The sender model (see below) has one state variable with two states: idle and busy. It begins in the idle state, in which it has no data to send. If the previous transaction has terminated (BtoS_ACK=0), the sender non-deterministically decides if it wants to send data. When it decides to send data, it goes to the busy state and raises StoB_REQ. It stays there for an arbitrary amount of time, at least until BUF acknowledges the acceptance of data (BtoS_ACK=1). This delay is arbitrary because the specification doesn’t force the sender to release StoB_REQ immediately. The sender then returns to the idle state.

```
module sender ( reset, ack )( req )   -- two inputs and one output
  “The sender initiates data transfers ‘at random’ and stays active for
    an arbitrary long time.”                -- a textual description
  {
    var state : { idle, busy };              -- has two states
    assign
      init(state) := idle;                   -- begins in the idle state
      next(state) :=                         -- next-state function
        case
          reset : idle;                      -- remains idle during reset
          state=idle & !ack : { idle, busy };    -- if idle and ack is inactive,
                                             --    can go to busy
          state=busy &  ack : { idle, busy };    -- if busy and ack is active,
```

```
                                        --    can return to idle
        else : state;                    -- else stay in the same state
      esac;
    define req := state=busy;            -- req is active when sender is busy
  }
instance sender : sender ( RST, BtoS_ACK )( StoB_REQ ); -- instance of module sender
```

By using non-determinism, **all** possible situations are checked. It is not a random selection of one or a few execution paths. The simple, abstract model represents all possible variations of a real sender, no matter how complicated they are, provided that they adhere to the specified protocol.

The receiver model (below) is surprisingly similar to the sender (in fact this tutorial could use the same module but they are left separate for clarity.)

```
module receiver ( reset, req )( ack )
 {
  var state : { idle, busy };
  assign
    init(state) := idle;
    next(state) :=
      case
        reset : idle;
        state=idle &  req : { idle, busy };
        state=busy & !req : { idle, busy };
        else : state;
      esac;
  define ack := state=busy;
 }
instance receiver : receiver ( RST, BtoR_REQ )( RtoB_ACK );
```

A behavior is assigned to the reset (RST) signal; it is asserted for one cycle at the beginning of execution.

```
module reset1 ( )( RST )
```

```
  "A one cycle reset at the beginning"
  {
    var RST: boolean;
    assign
      init(RST) := 1;
      next(RST) := 0;
  }
instance reset : reset1 ( )( RST );
```

Since RuleBase runs the clock itself (in the case of a design with a single clock), the clock (CLK) is stuck at '1', as follows:

```
define CLK := 1
```

See "Modeling Clocks" on page 102 for a complete explanation of clocks.

Note that we didn't assign behavior to data inputs, since the first rules that we are going to write do not refer to data, and control is not affected by data. The 32 bit register and the data inputs will be dropped automatically during reduction.

3.5 Specifying Properties for Verification

Now we want to verify certain properties (rules) of BUF. You can find the full text of these rules in the *rules* file. The first property claims that neither overflow (two reads without a write in between) nor underflow (two writes without a read in between) can occur. Actually, this example claims that the input acknowledge and output acknowledge operations are interleaved. The first formula says the following: "it is always true that if RST is not active and BtoS_ACK is asserted, then beginning from the next state, RtoB_ACK will be asserted before BtoS_ACK is asserted again". The second formula is similar. As you can see, explanatory comments may be embedded for the rule and for each formula.

```
rule ack_interleaving {"input acknowledge and output acknowledge are interleaved"
```

formula

"No overflow: RtoB_ACK is asserted between any two BtoS_ACK assertions"

{ **AG** (!RST & **rose**(BtoS_ACK) ->
AX (**rose**(RtoB_ACK) **before rose**(BtoS_ACK))) }

formula

"No underflow: BtoS_ACK is asserted between any two RtoB_ACK assertions"

{ **AG** (!RST & **rose**(RtoB_ACK) ->
AX (**rose**(BtoS_ACK) **before rose**(RtoB_ACK))) }

}

3.6 Performing Verification

The *rulebase.setup* file describes the verification environment: VHDL or VERILOG file name, VHDL or VERILOG entity name, the file that contains environment models and rules, and the translation path.

Note for VHDL users: *To avoid specific compiler dependency, the translation path assumes an input generated by Synopsys.*

To activate RuleBase, type **rb**.
RuleBase will then run as a background process.
(To exit from RuleBase, select the menu option **File/Quit**.)

Look at the RuleBase front panel. We only use four parts of the front panel for this tutorial:

1. A status window with red lines at the upper part.
2. A list of rules to be verified (on the left). This list currently has three entries. Sometimes the rule name is preceeded by a status letter, such as 'D' (Done), 'R' (Running), or 'K' (Killed).
3. A column of yellow push buttons to control verification and its options.
4. A big text window that occupies most of the work area, and is used to display important information.

To verify the property specified above, select the rule **ack_interleaving** and press the **Run** button.

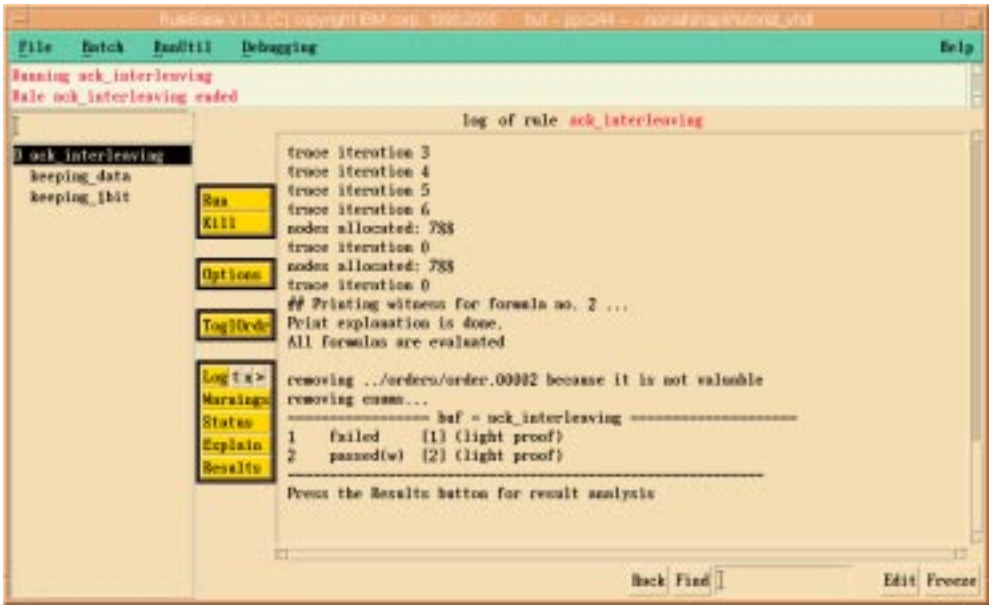


FIGURE 1. Front panel of RuleBase

While the rule is running, a log of its execution is displayed in the text window. At present, the only interesting information is the final result, which is: the first formula failed and the second one passed. The term “light proof” has to do with the specific verification algorithm. The failure means that there is a case in which there are two consecutive BtoS_ACK with no RtoB_ACK in between.

3.7 Problem Analysis

To view the results, press the **Results** button. Information about the two formulas will be displayed in the text window. The area for each formula consists of

three parts: verification results, an English description of the verified property (a display of the comment coded by the user), and the actual formula.

1. Click the mouse button anywhere in the area of the first formula. A pop-up menu will appear.
2. To select **Show timing diagram,** drag the mouse to this entry and release the button. Wait a few seconds until the timing diagram appears and displays a counter-example. All relevant signal names are shown. (To display diagrams for additional signals, click their names in the left list. For a detailed description of the waveform display, see CHAPTER 9: Debugging Aids.)

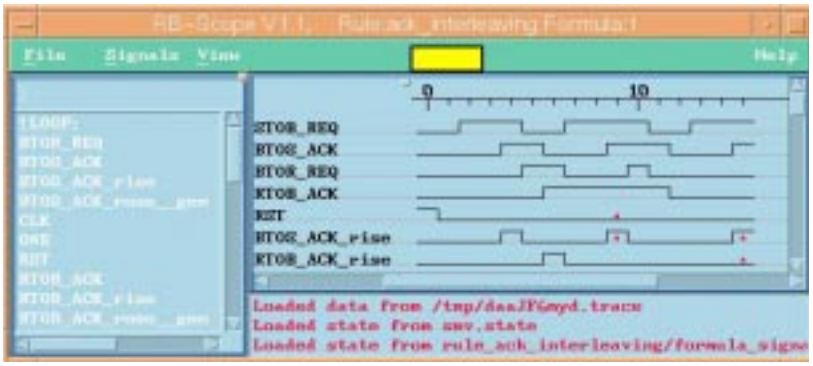


FIGURE 2.

A counter-example is a trace that demonstrates a failure of the design to fulfill a specified requirement.

Here we see an example in which the first formula fails: BtoS_ACK is asserted in both cycles 9 and 15 while RtoB_ACK is constantly high.

To better understand the problem, look at the interaction between BtoR_REQ and RtoB_ACK. The four-phase handshaking is broken in cycle 10, in which BtoR_REQ is asserted although RtoB_ACK is active. This occurs because the condition under which BUF can initiate a new transaction to the receiver is

incorrect (the relevant line in BUF.vhd or buf.v is marked as a comment). BUF only looks at the OCCUPIED flag and it also has to wait for RtoB_ACK to become inactive.

3.8 Fixing Problems and Rerunning Rules

To fix the problem

1. Type **setup2** at the unix command line.
2. Close the timing diagram (using the **File/Quit** menu option) and press the **Run**. Wait a few seconds. Both formulas will now pass (as tautologies).

This replaces the incorrect line with the line next to it (currently a comment) and recompiles the design.

3.9 Witness

Suppose that for some reason (due to a problem either in the design or the environment models), BtoS_ACK can never be asserted, or that it is only asserted once and RtoB_ACK is never asserted. In both cases, the formula will pass because there was no violation of the property.

This hides a problem of which you should be aware. It is called a *vacuous pass*, and it is a form of a *false positive* answer. To show that the pass was not vacuous, a *witness* is generated. A witness is a timing diagram that exhibits an interesting execution trace that demonstrates one case in which the formula is true. An interesting execution trace is one in which each event mentioned in the formula appears.

In our example, there is a “(w)” near the “passed” message. This means that a witness is available. To display the witness, press **Results**, click the mouse anywhere in the area of the first formula, and select **Show timing diagram**. This time the diagram displays a witness, rather than a counter-example. That is, this trace is an interesting positive example of the truth of the formula.

Close the timing diagram.

3.10 Data-Path Rule

1. If you have not already run ’setup2’, run ’setup2’ now.
   Next, verify that the data sent to the receiver is the same data received from the sender. The value of DI (data in) when BtoS_ACK is asserted (moment of transfer to BUF) must be the same as the value of DO (data out) at the next time RtoB_ACK is asserted (moment of transfer to receiver).

   forall x(0..31): **boolean**:

   formula

   { **AG** (!RST & rose(BtoS_ACK) & DI(0..31)=x(0..31) ->
   next_event(rose(RtoB_ACK))(DO(0..31)=x(0..31))) }

   (The operators are described in Chapter 5.)

2. Select rule **keeping_data** from the rule list and press the **Run** push button. RuleBase stops with a fatal error: design inputs DI(0) to DI(31) are unresolved. Since DO is referred to in the formulas and the value of DI influences DO, you must declare the DI vector. At first, it is given a fully free behavior, which means that it can always change its value.

   var DI(0..31) : **boolean**;

3. This environment model already exists in the *envs* file. To activate it, remove the two dashes in front of the line **#define WRONG_DATA** at the beginning of the *envs* file.
4. Press the **Run** push button again and wait a few seconds.
   Both formulas failed.
5. Press the **Results** push button, click the formula and select **Show timing diagram**.

You can see that the value of DO when RtoB_ACK is asserted is different from the value of DI when BtoS_ACK is asserted. This happened because our server environment model is not adhering to the requirement of keeping the data stable while StoB_REQ is active. You will often see “bugs” that result

from incorrect modeling of the environment. A common practice to avoid such problems is writing rules that verify the correct behavior of the environment models.

The following is a fixed version of DI:

```
var DI(0..31) : boolean;
assign
 next(DI(0..31)) :=
  case
   !StoB_REQ : nondets(32);
   else : DI(0..31);
  esac;
```

To activate the fix

1. Remove the two dashes in front of the line **#define CORRECT_DATA** at the beginning of the file *envs* and add two dashes in front of the line **#define WRONG_DATA**.
2. Press the **Run** push button again.
   Both formulas passed.
3. To see a witness, press **Results**, click the first formula and select **Show timing diagram**.

3.11 Reducing the Size of the Data Model

Sometimes testing the data consistency of all the vector's 32 bits at once may work very well, especially in large models. One technique is to test a single bit instead of the whole vector. So, instead of comparing DI(0..31) with DO(0..31), you can compare DI(0) and DO(0), while setting all other DI input vector elements to a constant, for example, of 0.

The above rule can be simplified as follows:

```
rule keeping_1bit {
```

```
forall x: boolean:
formula
  { AG ( !RST & rose(BtoS_ACK) & DI(0)=x ->
          next_event(rose(RtoB_ACK))(DO(0)=x ) ) }
}
```

To test the rule

1. Remove the two dashes in front of the line #define WRONG_DATA
2. Add two dashes in front of the line #define CORRECT_DATA.
   The rule keeping_1bit should fail quickly .

3. Remove the two dashes in front of the line:
   #define CORRECT_ONE_BIT,and add two dashes in front of the line #define CORRECT_DATA. The keeping_1bit rule should now pass very quickly.

3.12 Exiting RuleBase

To exit from RuleBase, select the **File/Quit** menu option.

3.13 Exercise

So far in this tutorial, we have not mentioned rules that cover the entire buffer specification. We encourage you to think of additional properties and formulate rules accordingly.

We recommend reading the remainder of this manual, or at least reviewing Chapter 5, which describes the specification language Sugar.

3.14 BUF implementation in VHDL

```
library IEEE;
 use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_unsigned.all;


entity BUF is port (
  CLK, RST     : in     std_logic;
  StoB_REQ, RtoB_ACK : in     std_logic;
  DI           : in     std_logic_vector (31 downto 0);
  DO           : buffer std_logic_vector (31 downto 0);
  BtoS_ACK, BtoR_REQ : buffer std_logic );
end BUF;


architecture RTL_VIEW of BUF is

  type S_STATES is (S_IDLE, S_READ, S_DONE);
  signal S_STATE : S_STATES;

  type R_STATES is (R_IDLE, R_SEND);
  signal R_STATE : R_STATES;

  signal OCCUPIED, READ : bit;

  begin

    SENDER_INTERFACE: process
    begin
      wait until CLK'event and CLK='1';
      if (RST = '1') then
        S_STATE <= S_IDLE;
      elsif (S_STATE = S_IDLE) then
        if (StoB_REQ = '1' and OCCUPIED = '0')
          then S_STATE <= S_READ;
        end if;
```

```
    elsif (S_STATE = S_READ ) then
      S_STATE <= S_DONE;
    elsif (S_STATE = S_DONE ) then
      if (StoB_REQ = '0')
        then S_STATE <= S_IDLE;
      end if;
    end if;
  end process;

  RECEIVER_INTERFACE: process
  begin
    wait until CLK'event and CLK='1';
    if (RST = '1') then
      R_STATE <= R_IDLE;
    elsif (R_STATE = R_IDLE) then
      if (OCCUPIED = '1')                -- !
--     if (RtoB_ACK = '0' and OCCUPIED = '1')
        then R_STATE <= R_SEND;
      end if;
    elsif (R_STATE = R_SEND) then
      if (RtoB_ACK = '1')
        then R_STATE <= R_IDLE;
      end if;
    end if;
  end process;

  OCCUPIED_FLAG: process
  begin
    wait until CLK'event and CLK='1';
    if (RST = '1') then
      OCCUPIED <= '0';
    elsif (OCCUPIED = '0') then
      if (READ = '1')
```

```
        then OCCUPIED <= '1';
       end if;
      elsif (OCCUPIED = '1') then
       if (RtoB_ACK = '1' and BtoR_REQ = '1')
        then OCCUPIED <= '0';
       end if;
      end if;
    end process;

    DATA_BUFFER: process
    begin
     wait until CLK'event and CLK='1';
     if (READ = '1')
      then DO <= DI;
     end if;
    end process;

    READ  <= '1' when (S_STATE = S_READ) else '0';
    BtoS_ACK <= '1' when (S_STATE = S_DONE) else '0';
    BtoR_REQ <= '1' when (R_STATE = R_SEND) else '0';

end RTL_VIEW;
```

3.15 Implementing BUF in VERILOG

```
 module buffer (CLK, RST, STOB_REQ, RTOB_ACK, DI, DO, BTOS_ACK,
BTOR_REQ);

   input CLK, RST, STOB_REQ, RTOB_ACK;
   input [31:0] DI;
   output [31:0] DO;
```

```
   reg [31:0] DO;
   output BTOS_ACK, BTOR_REQ;

   parameter S_IDLE = 2'h0, S_READ = 2'h1, S_DONE = 2'h2;
   reg [1:0] s_state;

   parameter R_IDLE = 1'h0, R_SEND = 1'h1;
   reg r_state;

   wire read;
   reg occupied;

   always @(posedge CLK)
     begin :SENDER_INTERFACE
if (RST)
s_state <= S_IDLE;
else
case (s_state)
S_IDLE:
if (STOB_REQ && !occupied)
s_state <= S_READ;
S_READ:
s_state <= S_DONE;
S_DONE:
if (!STOB_REQ)
s_state <= S_IDLE;
endcase
     end

always @(posedge CLK)
      begin :RECEIVER_INTERFACE
if (RST)
r_state <= R_IDLE;
```

```
else
case (r_state)
R_IDLE:
if (!RTOB_ACK && occupied)
  // wrong:
// if (occupied)
r_state <= R_SEND;
R_SEND:
if (RTOB_ACK)
r_state <= R_IDLE;
endcase
   end

always @(posedge CLK)
      begin :OCCUPIED_FLAG
if (RST)
occupied <= 0;
else
case (occupied)
1'b0:
if (read)
occupied <= 1;
1'b1:
if (RTOB_ACK && BTOR_REQ)
occupied <= 0;
endcase
   end

always @(posedge CLK)
      begin :DATA_BUFFER
if (read)
DO = DI;
   end
```

```
assign read = (s_state == S_READ);
assign BTOS_ACK = (s_state == S_DONE);
assign BTOR_REQ = (r_state == R_SEND);

endmodule
```

CHAPTER 4

Describing the Environment

4.1 Overview

This chapter describes the syntax and semantics of EDL constructs, suggests modeling techniques and demonstrates them with some examples. Before starting to write your environments, we recommend you read CHAPTER 7: Managing Rules, Modes, and Environments.

4.1.1 Describing Environment Models

RuleBase checks the properties specified for every possible input sequence. However, most chips are not designed to accept every possible input sequence. Designers often assume a correct behavior of the target environment and simplify the design by ignoring illegal behaviors.

RuleBase must be made aware of the environment's legal behavior, otherwise it might produce "false negatives", which are counter-examples that result from illegal input sequences. The way to specify environment behavior is to write environment models, which are the logic that drives the inputs of the design to be verified.

Every input of the design must be assigned some behavior. Some inputs are kept constant (e.g., configuration inputs), others remain completely free (non-deterministic), while the control signals of interest are usually assigned detailed and exact behavior.

Environment models are written in the RuleBase Environment Description Language (EDL), a dialect of the SMV language. EDL is somewhat similar to common hardware description languages (HDLs), but it also supports non-determinism and multiple environments.

Environments are linked to the design and to other environments by signal names. Signals produced by the environment will match and drive design signals that have the same name even if they are internal to the design, which is a way to abstract by overriding, described later. Signals (both output and internal signals) produced by the design will match and drive environment models that require these signals. In some translation paths, design signals are converted to upper-case.

Writing good environment models is an art. Good environments should be **small** and **simple**, while allowing **all and only** the legal behaviors. Environments should be small to avoid overloading the model-checker, and simple in order to be easily written, read, and maintained. Good environment models should not produce illegal behavior, or else false-negative results will be produced. On the other hand, they should model all the legal behaviors because an un-modeled behavior is a good place for bugs to hide. An attempt should be made to hide as much detail as possible using abstraction techniques (as explained later).

The following are the stages of environment modeling:

- Study the block interfaces in detail. The behavior of every input and every relevant output must be understood. This information can be gathered from standard bus protocols, design documents, and communication with the designers.

- Plan the hierarchical structure of the environment models, grouping related signals and reusing components where possible.
- Decide how to model each input. Some inputs are held constant, at least during the initial stages of verification. Usually there is a set of interesting control inputs that need detailed modeling. We have to design and implement logic to drive these signals.
- Code the logic in EDL.

4.2 Language Constructs

An environment is made up of a few type of statements. These statements are described in the following sections. The order of the statements is unimportant and keywords are not case-sensitive; they can be in either upper or lower case.

4.2.1 Expressions

4.2.1.1 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). RuleBase infers the width of constants from the context in which they are used and **not** from their format. For example, 0010B can be assigned to any bit vector that has at least two bits.

One of the symbolic values a variable can take on is an enumerated constant. For instance, if we declare the following:

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

then each of the five symbolic values “idle”, “st1”, “st2”, “st3”, and “waiting” is 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
- prev(name) -- refers to the value of name on the previous cycle

The use of **prev** results in additional state variables, one for each variable to which it refers. However, multiple references to the same variable will only add one extra variable.

For more information on variables see Section 4.2.2.
For more information on arrays, see in Section 4.3.

4.2.1.2 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 (since / is also legal in a signal name, make sure to surround it with spaces)
expr **mod** exprmodulo
(Arithmetic operators can be applied only to integer and boolean expressions.)

4.2.1.3 Operator Precedence and Associativity

The following operators are listed in decreasing order of precedence and strength:

++ (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.

4.2.1.4 Case and If Expressions

EDL provides two constructs that 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, we recommend you 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.

Note: *This section deals with* ***if/case*** *expressions rather than statements (****if/case*** *statements are allowed only inside sequential processes. See Section 4.4). You* ***cannot*** *write, for example:*
if *c* ***then assign*** *a := x; b := y;* ***else assign*** *a := z; b := w;* ***endif****;*
Instead, you should write:
assign a := ***if*** *c* ***then*** *x* ***else*** *z* ***endif****;* *b :=* ***if*** *c* ***then*** *y* ***else*** *w* ***endif****;*

4.2.1.5 Non-deterministic Choice

RuleBase uses non-determinism to describe many possible behaviors at once. Section 4.8 describes non-determinism in detail. In this section, the non-deterministic constructs are briefly mentioned for completeness. The non-deterministic constructs of RuleBase have the following format:

{ $expr_1$, $expr_2$, ... , $expr_n$ }a non-deterministic choice

$expr_1$ **union** $expr_2$ another way to express {$expr_1$, $expr_2$}

n_1 .. n_2 another way to express {n_1, n_1+1,..., n_2}

4.2.1.6 Other Expressions

The following are also expressions:

(expr) a parenthesized expression
expr **in** {v_1, v_2, ... , v_n} shorthand for ((expr = v_1) | (expr = v_2) | ... (expr = v_n))

4.2.1.7 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 use 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 only add one extra variable.

4.2.2 Var Statement

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

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

***Note:** The variables are always state variables as long as the declaration is not within a sequential process (see Section 4.3).*

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

The type can be one of the following:

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

(For information on arrays, see Section 4.3.)

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.

4.2.3 Assign Statement

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

- **assign init**(name) = expression; assigns an initial value to a variable (combinational or state.
- **assign** name = expression; assigns a value to a variable (combinational or state).
  assign next(name) = expression; defines the next-state function of a state variable.

A state variable is simply a memory element, or register (flip-flop or latch).

Note: *Using* ***assign next*** *within a sequential process causes the variable to be a state variable (see section 4.3). Variables outside a process are already state variables by definition.*

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

```
assign init(state) := idle;
assign next(state) :=
   case
      reset : idle;
      state=idle : { 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;
```

4.2.4 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 the second, and following, consecutive **define** statements.

4.2.5 The Difference Between Assign and Define

A state variable (flip-flop or latch) must always be declared with the **var** statement. If assigned an explicit value, the **assign init**() and **assign next**() statements are used (if either is omitted, the initial and next values, respectively, are considered to be completely non-deterministic).

For a combinational variable (output of combinational logic), you may use either **assign** or **define**. Users of SMV or of previous versions of RuleBase may recall that there were subtle differences between the **assign** and **define** statements which made it more efficient to use one or the other in certain situations. These differences are no longer present in RuleBase, which will convert from one to the other as needed in order to make the model checking more efficient.

Only the following semantic distinctions exist between **assign** and **define**:

- **assign** must refer to a variable defined with **var**.
- **define** must NOT refer to a variable defined with **var**.
- An **assign** statement can be thought of as a variable assignment, while a **define** statement should be thought of as macro text substitution. Thus, in the following:

```
VAR v,v1,v2,d1,d2: boolean;
assign v := {0,1};
assign v1 := v;
assign v2 := v;
define d := {0,1};
assign d1 := d;
assign d2 := d;
```

It is true that v1=v2, because both are equal to the value of the variable v. However, it is not true that d1=d2, because the macro text substitution has made the assignments to d1 and d2 equivalent to:

```
assign d1 := {0,1};
assign d2 := {0,1};
```

so that each non-deterministic assignment is completely independent of the other. If you code something similar to the above, RuleBase will issue a warning that a non-deterministic **define** expression is used multiple times.

4.2.6 Module Statement

An environment file can be totally flat, with no hierarchy at all. In this case, all statements are considered to be enclosed by one big main module. However, it is usually more appropriate to write a modular and hierarchical environment. The **module** and **instance** statements are used for this purpose.

A **module** statement is used to define a module that can be instantiated a number of times, as in hardware description languages. It has the following format:

```
module module_name ( inputs ) ( outputs )
{
   statement;
   statement;
   ...
}
```

where *inputs* is a list of formal parameters passed to the module, *outputs* is a list of formal parameters produced by the module, and *statements* is any sequence of **var**, **assign**, **define**, **fairness,** and **instance** statements. The input/output parameters can be thought of as input/output signals. Input parameters are produced elsewhere, and they drive the module, while output parameters are produced by the module itself and can be used elsewhere. A signal that appears as an output parameter of a module must be defined and assigned a value in that module (**var** or **define** or **instance** output). If a signal that appears as an input parameter of a module is not used in that module, RuleBase will issue a warning.

For instance, the following is a legal **module** statement:

```
module delayed_and (s1, s2) (out)
{
  var out : boolean;
  assign
     init(out) := 0;
```

```
    next(out) := s1 & s2;
}
```

Modules cannot be *declared* inside other modules but they can be *used* (instantiated) by other modules.

4.2.7 Instance Statement

A **module** statement is only a definition—it has no effect until it is instantiated (called). The **instance** statement instantiates a module using the following format:

instance instance_name : module_name (inputs) (outputs);

where *instance_name* is the name of the specific instance (one module can be multiply instantiated), *module_name* is the name of the module being instantiated, *inputs* is a list of expressions passed as inputs to this instance, and *outputs* is a list of output parameters that connect the instance outputs to real signals of the design or the environment. An instance name is optional.

For example, the following is a legal **instance** statement, instantiating the two-input and-gate defined in Section 4.2.6:

instance da : delayed_and(q,r)(t);

4.2.8 Fairness Statement

A **fairness** statement is used to describe a fairness constraint. We describe the use of fairness in detail later in this chapter. Briefly, a **fairness** statement describes a condition that must be met an infinite number of times. It is an important tool in specifying abstract environment models. The **fairness** statement has the following format:

fairness expression;

The following is a legal **fairness** statement:

fairness state != busy;

Currently, the fairness expression cannot contain temporal operators. This limitation protects users from commonly encountered mistakes. RuleBase supports other types of fairness constraints, which are detailed in Section 4.8.3.1, Advanced Fairness Types.

4.2.9 Scope Rules

Statements inside a module cannot reference variables outside that module (no *global* symbols). External signals and variables needed by the module must be passed as parameters to the instance. A module can only assign values to external signals and variables by passing them as output parameters.

On the other hand, it is possible to reference internal signals of an instance from outside that instance. For example, if module M has an internal signal Sig, and Ins is an instance of module M, one can refer to signal Sig as Ins/Sig ('/' is the hierarchy character). This allows formulas to refer to the internal state of instances without the burden of exporting state variables. It also allows you to easily override parts of existing modules without changing the module definition. For further detail on overriding, see Section 4.6.

4.2.10 Comments, Macros, and Preprocessing

There are two types of comments in environment description files:

1. Text beginning with "--" and ending at the end of line.
2. Text beginning with "/*" and ending with "*/".

RuleBase ignores comment text. You can insert a comment anywhere a space is legal (except in text strings).

Before processing the environment description files, RuleBase calls a standard preprocessor, cpp, to filter these files. The mechanisms provided by cpp can be used to facilitate the development of environment models. The most useful

mechanisms are macros, conditional compilation (#ifdef, #if, #endif, ...) and #include. See “man cpp” on your unix system for more details.

The cpp preprocessor may issue errors when the ’ character appears inside commented text; therefore, we recommend you avoid the use of this character within comments.

RuleBase provides additional preprocessing abilities in addition to cpp. These are the %for and %if constructs described below.

4.2.10.1 %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
```

- <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 be stripped.

In the first case, the text inside the %for-%end pair 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 simply using “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 aa%{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;
```

4.2.10.2 %if

The syntax of the %if construct is as follows:

```
%if <expr> %then
...
%end
```

or:

```
%if <expr> %then
...
%else
...
%end
```

The **%if** construct is similar to the #if construct of the cpp preprocessor. However, %if must be used when <expr> refers to variables defined in an encapsulating %for.

4.2.10.3 Operators in Preprocessor Expressions

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

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

In the current version, operators work only on numeric values, for example, it is acceptable to write:

```
%for i in 0..3 do
      %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!
...
%end
```

4.2.11 Reserved Words

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

^ ~ < << <= <-> = => > >= >> | |=> || |-> - -> -- , ; : := ! != / .. ( ) [ ] { } * & && + ++ A ABF ABG AF AG always AND ANDVECTOR AR as_in assign assume attributes AWR AX before before_ before! before!_ boolean bvtoi case coverage define E EBF EBG EF EG else endcase endif env envs ER esac eventually EWR EX fairness false fell fg forall formula formulas gf goto hint hint_ holds_until holds_until_ if in inherit init instance invar itobv max min mod mode module never next next_event next_event! next_event_f next_event_f! next_event_g next_event_g! nondets NOT ones OR original ORVECTOR override prev process reduce_instance rep restrict rose rule stable_until stable_until! test_pins then trans true U union until until_ until! until!_ V var W whilenot whilenot_ whilenot! whilenot!_ within within_ within! within!_ xor zeroes

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

4.3 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 primarily oriented toward Boolean arrays, but also supports other types of arrays .

4.3.1 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);

Note that the following line

var x(0..3) : { 5, 7, 13 };

defines an array of four integer variables, each of them can have the values 5, 7, or 13. This is **not** a non-deterministic bit vector. To define a bit vector and assign to it the three values non-deterministically, do the following:

var x(0..3) : **boolean**; **assign** x(0..3) := { 5, 7, 13 };

4.3.2 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 an integer variable, and *index1..index2* are constants that indicate its range. Example:

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

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

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

:= = != if case prev

Examples:

aa(0..7) := **if** bb(0..2)=cc(0..2) **then** dd(0..7) **else** ee(1..8) **endif**;
aa(0..7) := **prev**(bb(2..9));

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

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;
```

4.3.3 Converting 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 );
```

Constant integers are converted to bit vectors implicitly, you do not need to apply itobv. We recommended that you use bit vectors instead of big integer variables, if possible.

4.3.4 Constructing 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 split 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 construct arrays of repeated elements:

rep (expr, N) is equivalent to expr concatenated with itself N times. For example, you can use the following assignment to make each bit of array ‘arr’ non-deterministic:

```
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)

nondets(N) is equivalent to **rep**({0..1},N)

4.3.5 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 four state variables, each of them can take the values 5, 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, 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.

4.3.6 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);

var nda(0..2): boolean;
assign nda(0..2) := {001b, 010b, 111b};          -- non-deterministic assignment to a vector
```

```
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;

module and_or ( a(0..7), b(0..7), c(0..7) )( d(0..7) )
{ define d(0..7)  := a(0..7) & b(0..7) | c(0..7); }

instance a1 : and_or( x(0..7), y(7..0), z(0..7) )( w(7..0) );
```

4.4 Sequential Processes

Process constructs of EDL are similar to “process statements” of VHDL and to “always blocks” of Verilog. They can be useful in situations when it is awkward to write explicit concurrent definitions for signals. Using process constructs, you can write your code in the form of sequences of statements, which are “executed” in each cycle to compute the needed values of signals. The only statements allowed in a process are variable declarations, variable assignments, IF statements, and CASE statements.

As a simple example,

```
process {
   var foo: boolean;
   foo := d1;
```

```
    if c then foo := d2 endif;
}
```

is equivalent to the concurrent assignment

```
assign foo := if c then d2 else d1 endif;
```

(Of course, in this example the concurrent form is simpler than the process construct).

As a slightly more realistic example, suppose that we need to model a ripple-carry adder in EDL, but for some reason cannot use the '+' operator:

```
process {
  var sum(0..7): boolean;
  var carry: boolean;
  carry := 0;
  %for i in 7..0 step -1 do
     sum(i) := x(i) ^ y(i) ^ carry;
     carry := (x(i) & y(i)) | (x(i) & carry) | (y(i) & carry);
  %end
}
```

The carry signal is assigned several times in the process, and each stanza of the loop refers to the value of carry valid for this specific stanza. if some code outside this process refers to the carry signal, it will refer to the "final" value of carry, which in this case is the overflow bit of the adder.

It is convenient to think about processes as sequential code which is "executed" each cycle, but what happens technically is that RuleBase analyzes the process construct, keeping track of interim assignments, and generates concurrent definitions for signals driven by the process. This means, for example, that in the wave browser you will only be able to see the "final" values of signals.

If you are familiar with VHDL or Verilog, you will notice that EDL processes are not explicitly associated with some clock signal or a sensitivity list. Instead, they are implicitly clocked on the "system clock", just like the concurrent "assign next" construct.

The following provides a closer look at the building blocks of a process construct.

- **Variable declarations**
  The process construct should contain **var** declarations for all signals that are assigned within the process. The **var** declaration of each signal should appear before the first assignment to it. Currently there is a restriction on chains of **var** declarations within a process: each **var** declaration should start with **var** keyword, for example:

  "**var** foo: **boolean**; bar: **boolean**;"

  is not allowed, but both

  "**var** foo, bar: **boolean**" and "**var** foo: **boolean**; **var** bar: **boolean**"

  are allowed.

  While using a var outside a process defines a state variable, this is not the case here, unless **assign next** is used (see 2. below).

- **Assignments**
  The three usual forms of RuleBase assignments are supported:

  assign S := expr;

  assign next (S) := expr;

  assign init (S) := expr;

  The keyword **assign** can be omitted. **define** constructs are illegal within a process. S is a signal or a concatenation of signals.

  The assignment of the first form:

  S := expr;

  is similar to the VHDL variable assignment and to the blocking Verilog assignment, in that references to S, which are "executed" after this assignment, will already refer to the new value of S. Therefore, the order of the statements is important. For example,

  foo := 0;

  bar := foo;

  foo := 1;

  will assign 0 to bar (even though foo is re-assigned later on).

The assignment of the form:

next (S) := expr;

behaves more like the VHDL signal assignment and to non-blocking Verilog assignment, in that it doesn't influence the values of S that can be observed in this cycle.

The use of **next** makes S a state variable.

next (foo) := 0;

bar := foo;

will assign the current-cycle value of foo, which is not necessarily 0, to bar. The next-cycle value of foo will be 0 (in the absence of further assignments to ``next (foo)" in the process).

The assignment of the form:

init (S) := expr;

is very special in that it will only be "executed" in the very first cycle, and will have no effect on subsequent cycles.

- **CASE statements**

case

$guard_1$: $stat_1$;

$guard_2$: $stat_2$;

...

$guard_n$: $stat_n$;

else: $stat_e$;

esac;

Each $guard_i$ is a Boolean expression. The else clause is optional. Each $stat_i$ is either a single assignment, or an **arbitrary sequence of statements enclosed in braces**.

- **IF statements**

The IF statement is less general than the CASE statement and can take one of two forms:

```
   if condition then
      statements
   endif;
or
   if condition then
      statements
   else
      statements
   endif;
```

The following is an an example of a process construct that makes use of different statements:

```
module server (start,grant)(request,done)
{
   process {
      var state: { idle, wait, busy };
      init(state) := idle;
      next(state) := state;  -- default behavior

      var request, done: boolean;      -- state machine outputs
      request := false; done := false; -- their default behavior

      case
         state=idle & start:
            next(state) := wait;

         state=wait: {
            request := true;
            if grant then
```

```
            next(state) := busy
          endif;
        }

        state=busy: {
          done := {true,false};
          if done then
            next(state) := busy
          endif;
        }

      esac;
    } -- process
  } -- module
```

4.5 Environment Constraints

Trans, Invar, **assume**, **restrict,** and **hints** are environment constructs that enable you to set constraints on signals. They allow you to describe the environment by declarative means instead of giving each signal a functional behavior. These environment constraints can be combined with other environment constructs such as var, assign, define, etc.

4.5.1 Initially and Trans

The **initially** statement enables you to specify a boolean expression that must hold true at the very first cycle. RuleBase will cut off from the model all initial states that do not hold the boolean expression specified within the initially statement.

The syntax of the initially construct is as follows:

initially <expr>
-- Where <expr> is a boolean expression

<expr> can include both environment and design signals.

The **trans** statement enables you to specify a set of legal transitions between states of the design, thus ignoring all other transitions, which are illegal. RuleBase will **force** the model to hold the boolean expression specified within the trans statement at every cycle.

The syntax of the trans construct is as follows:

trans <expr>
-- Where <expr> is a boolean expression

<expr> can include both environment and design signals.

The statement trans <expr> implies that all transitions which do not comply with <expr> are cut off.

Note that <expr> should not be just any boolean expression, but a boolean expression that describes transitions between states of the design. If <expr> does not contain any next() expression, then it does not refer to any transition, therefore nothing will be cut off; in other words, such a trans statement will not have any effect. This implies that <expr> should contain at least one 'next'.

For example:

```
rule transexamp {
    var a,b,c: boolean;
    trans next(a) = next(b);
    initially  (a=c)
    formula { AG (a | b | c) }

}
```

4.5.2 Invar

The **invar** statement enables you to specify a boolean invariant that you want to be true at any cycle. RuleBase will **force** the model to hold this invariant at every cycle.

The following shows the syntax of the invar construct:

```
invar <expr>
 -- Where <expr> is a boolean expression.
```

The Boolean expression within the invar can include both environment and design signals.

Example:

Given a design with the inputs request1, request2, and request3, the design should only work properly under the constraint that one request, at most, can be active at any given cycle.

This can be specified by:

```
var request1, request2, request3: boolean;
invar (request1 + request2 + request3 <= 1)
```

request1, request2, and *request3* signals can have any non-deterministic behavior that holds the above invariant.

4.5.3 Assume

Assume can be seen as an extension of the invar construct. It enables you to write more expressive assumptions on your model, which tell RuleBase to force your model to hold those assumptions. The assumptions are written as Sugar properties.

The syntax of the assume construct is as follows:

assume {sugar_formula}

Most of the Sugar safety formulas can be used within the assume. These Sugar formulas are the same formulas that can be verified on the fly.

Examples:

- *read* and *write* are inputs to a design.
- *read* should not be followed by *write* (one or two cycles later).
  This can be specified by:

  var *read, write*: **boolean**;
  assume {**AG** (*read* -> **ABG**[1..2] (!*write*))}

***Note**: **AG** and **ABG**, and **AX** and **before_** and sequences, mentioned in the sequel, are constructs of the Sugar specification language, described in CHAPTER 5.*

Additional requirements:

- The first input command must be a write.
  assume {*write* **before_** *read*}
- A sequence of three consecutive *write*s is illegal.
  assume { {[*], *write*[3]}(false) }

Assume can help you define complex behavior of inputs.

Using the assume construct, you can start the verification process with an initial free environment, and you can add environment assumptions when you encounter “false negatives” (counter-examples that result from illegal input sequences).

Writing an environment with assumptions enables you to apply compositional verification to your design using the assume-guarantee approach.

The assume-guarantee approach is as follows:

1. Assume that the input signals obey some assumptions.
2. Take those assumptions and guarantee they hold by turning them into rules and verifying them on the blocks that produced them.

Consider a design that is partitioned into two blocks: block1, block2 (see Figure). In the verification of block1, one can write environment assumptions on the input signals generated by block2 using the assume construct. Later, when proceeding to the verification of block2, the already written assumptions can be turned into formulas and verified on block2.

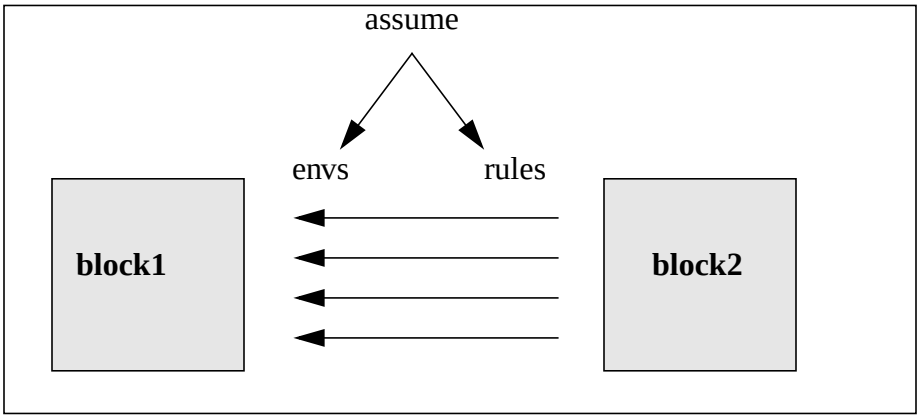


FIGURE 3. Design partitioned into two blocks that uses the assume construct for compositional verification

Assumptions cannot be written inside the module or process.

Notes:

- In some cases, the assume construct can cause state space explosion problems by introducing many variables. (These variables are needed to construct a deterministic automata that represents the assume construct.) In

such cases, it may be more convenient for the user to use other constraint constructs, such as invar or restrict, or define the behavior in the usual way using define and assign.

- Non-deterministic variables may cause false negatives, if they are used in the same assume but in different points of time, for example:

```
var x,y,z: boolean;
assign next(x) := x;
  assume {AG(y=x -> AX(z=x))};
 . . .
```

The user may get a counter example to some formula in which the value of y for one cycle differs from the value of z for the next cycle (i.e., violates the assume).

The other example (with the same meaning and same problems) is:

```
forall x  assume {AG(y=x -> AX(z=x))}
```

The two cases can be rewritten as follows, without causing false negatives:

```
assume {AG(y=0 -> AX(z=0))};
assume {AG(y=1 -> AX(z=1))};
```

In general, assumes are most useful in free environment.

- There is an additional case of false negatives, as seen by users when the counter example does not show the restricted behavior, but is final (i.e., would necessarily show such behavior if prolonged). Such counter examples can be eliminated by writing the same assume on the “causing variables”. The following provides a simplified example of eliminating such a counter example by writing the same assume on the “causing variables”:

```
var x, y: boolean;

assign init(x) := {0,1};
assign next(x) := x;
```

assing init(y) := 0;
assing next(y) := !x;

assume {**AG**(!y) }
rule dummy {
formula {**AG** (y) }
}

The user gets the counter example of length 1 where x=0 and y=0. This trace only has the next state in which y=1 and x=0 is forbidden according to the assumption. Thus, the trace the user gets in this example is not real. By tracing back to the forbidden behavior of y, one can see that it is forbidden for x to be 0. By replacing the existing assume with

assume {**AG**(x) }

on the same formula, the user gets the real counter example of length 1 where y=0 and x=1. (In this case, real means that it can be prolonged to any length.)

4.5.4 Restrict

The restrict environment construct is used to limit the state space exploration to certain paths. The restrict looks like a regular expression, and its semantics resemble the semantics of a regular expression. Only paths that match the regular expression will be checked.

The syntax of the **restrict** construct is as follows:

restrict {regular_expression}

- Where the regular expression events can be any of the **sequence** events.

Example:

restrict { !read[*], read, !read[*] }

- Restrict RuleBase to check only paths with at most one read command.

Restrict does not have any meaning when it starts with a [*], since every computation path is a prefix of such a restrict; hence, this restrict will not force a limitation on the model.

There are several motivations for the use of restrict, including:

1. **A 'guide/direct' search to start with specific behavior.**
   Example: Every path should start with two reads followed by a write
   restrict {read[2], write, [*]}
   Note: *The [*] at the end of the above restrict is necessary; if you omit it, RuleBase will only check paths with the length of three.*
2. **An easy way to define an input that behaves according to a specific pattern.**
   Example:
   Bus is defined as follows:
   var bus: {idle, BOP, data, EOP};
   We restrict the bus behavior to the following pattern:
   restrict { {bus=idle[*], bus=BOP, bus=data[4], bus=EOP}[*] }
   That is, there can be any number of transactions, with any number of idle cycles in between, in which each transaction starts with BOP, followed by four cycles of data and terminated with EOP.
3. **A quick way to verify that a specific design failure does not exist in the 'design after fix':**
   Given a formula that failed and a trace that shows a counter example for the formula, we fix the design and would like to verify (quickly) that we will not get the same failure that we had before.
4. **To convert the inputs of the trace into a restriction that describes the inputs value in each cycle**
   - Click the *Results* button
   - Click the formula that failed and select '*Generate restrict (inputs)*'
   - Run the formula again including the file which contains restrict you generated.

   The GUI will give you its name; it resides in your working directory.

If the counter example no longer exists, you will receive a vacuous result for the new run. This run is quick since it restricts the search space to a very specific pattern of inputs.

4.5.5 Hints

The **hint** list can be seen as a generalization of the **invar** construct. RuleBase uses each hint in the list to restrict the search of the state space in the same way as invar, by switching to the next hint in the list when no additional states can be explored using the current one.

Syntax:

hint *expr*, ..., **hint** *expr*[*NUM*], ...;

Semantics:

- **hint** *expr* – continues to search reachable states with the transition relation constrained by *expr* until the fixpoint is reached.
- **hint** *expr*[*NUM*] – only performs *NUM* of steps with the constraint.
- RuleBase automatically adds **hint** *TRUE* at the end of the list, so you do not need to do it.
- When a hint ends (either the fixpoint is reached or a given number of cycles has passed), it passes on to the next one.

In the case of a liveness formula with hints, every liveness formula is checked on the fly at every fixpoint that is reached that has a hint.

Examples:

var *cmd*: {*read, write, flush, stall* };

hint *cmd = read*, **hint** *cmd = write*, **hint** *cmd* != *flush*[5];

Hints may be combined with liveness and counters as follows:

- **liveness + hints**
  RuleBase does not execute liveness algorithms that use hints in the calculated fixed point as some may expect (CAV99 - Ravi & Somensi).
  Instead, it performs the following algorithm, which is related to liveness on the fly for all hints (according to their order):
  1. Compute the reachable states using the hint (exactly like in on-the-fly mode).
  2. Before changing to the next hint, build the transition relation, simplified according to the reachable states (like in liveness on-the-fly).
  3. Evaluate the formula with the classic algorithm on this partial model.
  4. If the formula fails, generate a counter example; otherwise, move to the next hint.
- This is exactly like liveness on the fly, which is executed after every hint fixed point instead of after every 'n' iterations.
  counters mode + hints
  RuleBase turns off during the on the fly model checking.
  It executes the liveness + hints algorithm described above with the following change: Every time the transition relation is built, the counters are added to the model. When RuleBase continues reachability with hints, the counters are removed again.

4.5.6 Additional Environment Constraint Examples

- *cmd, busy* are design inputs. *busy* is active one cycle after *cmd*:
  var *cmd, busy* : **boolean**;
  assume {**AG** (*cmd* -> **AX** *busy*) }
- When sending a command, *cmd* should be active for three cycles, and then inactive for at least two cycles.
  var *cmd*: **boolean**;
  assume { {[*], !*cmd*, *cmd*}(**ABG**[1..2] (*cmd*) }
  assume { {[*], *cmd*[3]}(**ABG**[1..2](!*cmd*)) }
- The above environment written with restrict:
  var *cmd*: **boolean**;
  restrict { {!*cmd*[*], *cmd*[3], !*cmd*[2]}[*] }

- Consider a design block that should work properly under the following assumptions:
  - *start*, *end* (the input signals) are pulses.
  - *start* and *end* are interleaving (i.e., there is always a *start* between two *end*s and vise versa).
  - The first *end* will be proceeded by a *start.*

    var *start*, *end*: **boolean**;
    assume { **AG** (*start* -> (**AX** *end* **before** *start*)) }
    assume { **AG** (*end* ->(**AX** *start* **before** *end*)) }
    assume { **AG** !(*start* & *end*) }
    assume { *start* **before** *end* }
- The above environment written with restrict:
  var *start*, *end* : **boolean**;
  restrict { {(!*start* & !*end*)[*], *start* & !*end*, (!*start* & !*end*)[*], !*start* & *end*}[*] }

4.6 Linking the Environment to the Design

In RuleBase, the name connects (links) the design and environment. Thus, to give behavior to an input signal of the name “reset” in your design, give a signal, of the same name, behavior in your environment, using either the **define** statement (see “Define Statement” on page 61), or the **var** statement (see “Var Statement” on page 59) in combination with the **assign** statement (see “Assign Statement” on page 60).

It is important to make sure that you use the name of the signal exactly as recognized by RuleBase (including capitalization). A list of the design signals that RuleBase recognizes can be found under the “Debugging/Signals before reduction” menu option.

4.7 Overriding Design Behavior

The environment can be used to override the behavior of part of the design. To override the behavior of an internal design signal, give it behavior using the

define statement, or the var statement in combination with the assign statement, which specifies **override** as follows:

```
define override sig := ...
```

or:

```
var override sig: boolean;
assign  init(sig)   := ...
        next(sig) := ...
```

Overriding design behavior is especially useful if you have implemented a specific behavior of a signal, but want to make sure the design works for any behavior of the signal. For instance, suppose that we have a signal called “predict” that implements a complicated predict function. Some other piece of logic uses the “predict” signal in its calculations. Suppose our formula is the following:

```
AG (predict -> AX[2] !low_priority_request)
```

Also suppose that this formula should be true regardless of the implementation of the predict function. We can make the job of RuleBase easier by eliminating all of the logic driving “predict”, and overriding it with a totally non-deterministic behavior, as follows:

```
var override predict: boolean;
```

Now, predict can have any behavior. For another example of overriding internal signals, see “Abstraction of Internal Parts” on page 159.

When overriding design signals, it is important to make sure that you are using the name of the signal exactly as recognized by RuleBase (including capitalization). A list of the design signals that RuleBase recognizes can be found under the “Debugging/Signals before reduction” menu option.

4.7.1 Overriding Initial Values

Sometimes, it is necessary to override the initial value of a flip-flop in the design without modifying its next-state function. In these cases, specify the initial value as follows:

```
assign init(abc) := 1;
assign init(def) := {0,1};
```

The first statement above assigns an initial value of 1 to signal abc. The second statement assigns a non-deterministic initial value to signal def. In other words, the value of signal def at power-on is not known.

4.7.2 Using Original Design Behavior

When design behavior is overridden with an **override** statement, it is sometimes necessary to use the original behavior of the design in addition to the overriding one. In such cases, the original design behavior can be accessed by specifying **original**, as follows.

For example, suppose ack is a signal in the design.
If you write the following:

```
mode check_override {
  var override ack:boolean ;
  assign ack := 0 ;
}
rule override_original {
  envs check_override ;
  formula { AG (original(ack) = 0) }
}
```

RuleBase will check the formula according to the original behavior of ack and not according to the behavior that overrides it.

When the original behavior of ack is accessed, an auxiliary signal NET_original_ack is created and it takes the original behavior of ack. However, the behavior of ack shown in the scope will be the overriding behavior, not the original one, and NET_original_ack will not appear in the scope.

Note for advanced users*: The reason that NET_original_ack is not shown in the scope is that it is filtered out of the SMV log and the scope (like all signals whose names begin with NET_).*

To see NET_original_ack in the scope

- Insert the following two lines to your relubase.setup file:
  setenv SMVFLAGS “$SMVFLAGS -no_filter_synopsys”
  setenv rb_dont_filter_names 1

Or:

- Add the following definition to your envs file:
  define orig_ack := NET_original_ack

You will then be able to view orig_ack in the scope.

4.8 Using Non-determinism and Fairness

It may not yet be clear to you how an environment is used to describe *every possible* input sequence. This is important if we are to fulfill the promise made that formal verification is equivalent to exhaustive simulation. To achieve this exhaustiveness, we use non-determinism.

This section discusses non-determinism and its uses. It is necessary to understand this subject thoroughly in order to use formal verification. Afterwards, we discuss fairness, a closely related concept. Fairness is a way of limiting non-determinism so that the paths that we filter out the paths that we do not want to consider.

A non-deterministic environment is an environment in which we specify more than one possibility for the behavior of a given variable. When we make a non-deterministic assignment, we are indicating to RuleBase that **all** possibilities must be considered. Do not confuse a non-deterministic assignment with the X value sometimes used in simulation, or with a don't care assignment as used in synthesis. A don't care assignment gives a measure of freedom to the synthesis tool—it indicates that any value chosen by the tool is acceptable. In synthesis, the actual logic will either have one value or the other. A non-deterministic assignment, on the other hand, does not give any freedom. Rather, it forces RuleBase to consider the exact outcome of all possible choices.

This section assumes that the rules checked are of the form "for all possible execution paths, some property holds true." If rules of this type are proven in an abstract system, it will also hold true in every concrete system that implements the abstract system. Experience has proven that most of the rules used in practice are of this type.

4.8.1 Coding Non-determinism

4.8.1.1 Free Variables

A free variable is any variable that is declared, but not assigned a behavior using an **assign** statement. For instance, assume the following is part of an environment that models a CPU driving a memory bus:

```
var command: {read, write, none};
```

Since we have not specified any behavior for the *command* variable, RuleBase must consider all possible sequences of commands.

A non-deterministic choice between values of a variable can also be made by enumerating all possible values. Thus, we could have made the command variable free, as follows:

```
var command: {read, write, none};
assign command := {read, write, none};
```

4.8.1.2 Non-deterministic Choice

Many times, we do not want a variable to be completely free, but rather constrained in some way while still exhibiting non-deterministic behavior in certain cases. For this purpose, we can use non-deterministic choice among expressions. The non-deterministic choice is an expression that indicates a choice between a number of values. For instance, the following expression:

```
{write, none}
```

indicates a non-deterministic choice between the values “write” and “none”. Suppose that our CPU contains a MESI-state cache. Then, it will never issue the read command unless it is in an invalid state. However, the write command may be issued in any case. We would then model our CPU as follows:

```
var command: {read, write, none};
assign command :=
  case
    mesi_state = invalid:   {read, write, none};
    else                :   {write, none};
  esac;
```

In this environment we have specified that the command can be any of the three declared values if the variable mesi_state equals invalid. Otherwise, the variable command can take on either the value “write” or the value “none”.

Example:

Say we have an arbiter that receives two commands: c1 and c2. If both commands have the value “none”, then the arbiter outputs “none”. If one of the command is something other than “none”, then that command is chosen. If both commands are something other than “none”, then the arbiter may choose either command non-deterministically. We can model this as follows:

```
module an_arbiter (c1, c2) (output_command)
{
  var output_command: {read, write, none};
  assign output_command :=
```

```
        case
            (c1 = none) & (c2 = none):   none;
            (c1 = none):                 c2;
            (c2 = none):                 c1;
            else          :              {c1 , c2};
        esac;
}
```

4.8.1.3 Auxiliary Non-deterministic Variables

The arbiter shown above is rather simplistic. To complicate things, let us assume that command c1 comes with address a1, and command c2 comes with address a2. Then, if we choose command c1, it makes no sense to choose a2. In this case we must choose a1. One way to associate one non-deterministic choice with another is to use an auxiliary non-deterministic variable. The following example illustrates this point.

FIGURE 4. Another arbiter

```
module another_arbiter (c1, a1, c2, a2) (output_command, output_address)
{
  var choose: {1,2};
  var output_command: {read, write, none};
  var output_address: boolean;

  assign output_command :=
      case
          (c1 = none) & (c2 = none): none;
          (c1 = none): c2;
          (c2 = none): c1;
          else          : case choose = 1: c1; 2: c2; esac;
      esac;
  assign output_address :=
      case
          (c1 = none) & (c2 = none): {0,1};
          (c1 = none): a2;
```

```
            (c2 = none): a1;
            else        : case choose = 1: a1; 2: a2; esac;
        esac;
    }
```

By using the free auxiliary variable “choose”, we have tied the non-deterministic choice between c1 and c2 to that between a1 and a2. Notice that in the case where both c1 and c2 are none, we let the address be free. This is an accurate picture of an arbiter in which the address is undefined in the case that no command is chosen.

4.8.2 Using Non-determinism to Create an Abstract Model

Suppose we need to model an arbiter that uses a round-robin or other algorithm in order to ensure that every requestor gets a turn. Now, assume that this arbiter is not part of the model under test, but a piece of logic that we know is correct. Creating an exact model of the arbiter will be time-consuming and error-prone. We would probably spend a good amount of time debugging the model rather than verifying our design under test.

If the properties to be verified only depend on the fact that the arbiter eventually gives every requestor a turn, and not on the specific algorithm used by the arbiter, then we may want to use non-determinism to make our modeling job easier. By using a non-deterministic arbiter, as shown in Section 4.8.1, we ensure that any property we prove will be true in the case that the real arbiter is used. This is because a non-deterministic arbiter models all possible sequences of events wherever the non-deterministic choice appears. Since the real behavior is one of the possible choices, it follows that anything proved for the non-deterministic arbiter is true for the real arbiter. A model that includes more behavior than the entity being modeled is called an abstract model.

There is one catch, however. Since our non-deterministic arbiter models all possible behaviors, it also models the behavior in which c1 is always chosen whenever a non-deterministic choice is to be made. We need a way to filter out this possibility and the way to do so is through **fairness**.

4.8.3 Fairness

Recall that the **fairness** statement has the following format:

fairness expression;

The meaning of the **fairness** statement is that we are only interested in sequences in which the expression specified will happen infinitely often. That is, we are not interested in input sequences in which, at some point in time, the expression becomes false and stays that way forever.

By making the following two fairness constraints within the arbiter of Figure 4

fairness choose = 1;
fairness choose = 2;

we indicate to RuleBase that we are only interested in input sequences in which *choose* takes on the values 1 and 2 infinitely often. That is, we filter out sequences in which, at some point in time, *choose* gets stuck at either value.

4.8.3.1 Advanced Fairness Types

In the example above, suppose that we want *choose* to take on the value 1 infinitely often only on those paths in which c1 is not stuck at value ‘none’ forever. (That is, for paths on which c1 is stuck at ‘none’ forever, we have no requirements from *choose.*) For this purpose, the **fairness** statement described above is too strong; we need a weaker type of fairness to filter out the paths we want. The statement

GF->GF c1 != none, *choose* = 1;

will leave the paths we want in the model.

***Note**: **GF** and **FG**, mentioned in the sequel, are constructs of the Sugar specification language, described in CHAPTER 5: Sugar – The RuleBase Specification Language.*

The following additional fairness types are supported:

- **FG p ;**
  Leaves in the model only paths on which, from some point onward, *p holds forever.*
- **FG->FG p , q ;**
  Leaves in the model only paths on which if a point from which *p holds forever*, then a point also exists from which *q holds forever.*
- **FG->GF p , q ;**
  Leaves in the model only paths on which, if a point from which *p holds forever* exists, then *q holds infinitely* often .
- **GF->GF p , q ;**
  Leaves in the model only paths on which if *p holds infinitely* often, then *q also holds infinitely* often.

4.8.3.2 Danger of Fairness

Fairness is a powerful, but dangerous tool. The danger of fairness is that too many paths may be unintentionally filtered out, some of which may include violations of our formulas. Here is an example:

```
module server (start) (ready)
{
  var state : { idle, busy, done };
  assign
      init (state) := idle;
      next (state) :=
        case
            state=idle & start : busy;
            state=busy : { busy, done };
            state=done : idle;
            else : state;
        esac;
  define ready := state=idle;
  fairness state = done;
}
```

In the above example, we give the variable “state” a non-deterministic behavior while it is busy. We also constrain RuleBase with a **fairness** statement so that it only checks paths on which the machine does not stay busy forever. However, this is a dangerous formulation of the fairness requirement. Since for each “done” there is one “start”, the paths left in the model also have “start” infinitely often. If some deadlock condition in the verified design prevents start from being asserted, this deadlock will not be detected, because the fairness constraint filtered out paths on which start is not asserted infinitely often.

To overcome the problem in the above example, the **fairness** statement should be formulated in a way that prevents state from staying busy, while having no side effects:

fairness state!= busy;

4.9 Using Counter Files

Counters in the design may induce many iterations during reachability analysis, because only one counter state is reached at each step. If you have big counters and experience this problem, try the following:

If your design is small, first try to run without reachability: Set options/verification/reachability=no and verify-safety-OnTheFly=no. This may solve your problem.

If your design is not small, or if the above solution resulted in BDD explosion, try the “counters trick”:

1. Set options/verification/reachability=yes and verify-safety-OnTheFly=no.
2. Create file “counters” in the verification directory that contains the names of the counter variables, one at each line (vectors should be split into single bits).
3. Add the following line to rulebase.setup:
   setenv SMVFLAGS “$SMVFLAGS -counters_file ../counters”

Note: *This will result in approximate reachable state space, so errors cannot be detected on the fly.*

If you expect errors in early iterations, we recommend that you clean them first in OnTheFly mode (options/verification/reachability=yes and verify-safety-OnTheFly=yes), and only then use the counters trick.

4.10 Modeling Clocks

To use formal verification properly, it is essential to understand the way RuleBase deals with clocks, and to choose the proper clock scheme. This section assumes that the clock signal is generated externally and drives the verified design through input clock pins.

The most simple case is a design that only has one clock, in which only one level or edge of the clock is used in the design. In this case, the clock input should be held constant at the value '1':

```
define CLK := 1;    -- CLK is the clock input pin                    (*)
```

RuleBase understands it as the clock being active in every cycle. This works even when some of the flip-flops are gated. The gated flip-flops will work only when the gate is active.

The next scheme has one clock, but both levels (or edges) are used in the design. In this case, we define the clock as having alternate values 0 and 1, as follows:

```
var CLK: boolean;    assign init(CLK) := 0;    next(CLK) := !CLK;        (**)
```

Notes:

- If your design uses master-slave latches, then the master latches will change on one level of the clock, and the slave latches on the other. However, if the only use of the master latches is to drive the slaves (i.e., there is no use of the master latch output other than by its slave), then you can still use the simpler clock scheme described above, which will give you better perfor-

mance. To do this, you must model the master-slave pair as a single edge-triggered flip-flop or latch (see CHAPTER 2: Getting Started for modeling of latches).

- Although (**) may be used in designs that have one clock with one phase, model-checking of (*) is more efficient.
- When the clock is defined as in (**), formulas should include explicit references to the clock signal. For example, the following formula:

  "**AG**(p -> **AX** q)"

  should be rewritten as:

  "**AG**((p&**CLK**) -> next_event**(CLK)(**q))"

  This rewriting may also be necessary in the more complicated clock schemes described below. If all signals in the formula refer to the same clock, as in the examples above, RuleBase can rewrite the formula automatically. To do that, write

  AG(p -> **AX** q) :: clk = CLK

  See section 5.4: "Multiple-Clocks in Formulas" for more details.

Before continuing further with more clock schemes, it is important to note that complex schemes usually contribute to size problems more than simpler ones. When planning the micro-architecture of the design, it is advised to partition the design in a way that each part will have the simplest scheme possible, preferably one clock.

The next scheme has multiple synchronized clocks. For example, assume that there are two clocks, with a 1:3 ratio between their frequencies. In this case, we fix the faster clock at value '1' (always active), and generate a pulse every third cycle for the slow clock:

```
define FAST_CLOCK := 1;
var clock_counter: 0..2;
assign next(clock_counter) := (clock_counter +1) mod 3;
define SLOW_CLOCK := clock_counter = 0;
```

In contrast to clocks in real systems, whose duty cycle is usually 50%, slow clocks in RuleBase should only be active for **one** cycle each time. (If this is a problem because the clock is generated internally, contact us.)

A similar case is a ratio of 2:3 :

```
var clock_counter: 0..5;
assign next(clock_counter) := (clock_counter +1) mod 6;
define SLOW_CLOCK := clock_counter in { 0, 3 };
define FAST_CLOCK := clock_counter in { 0, 2, 4 };
```

If the clocks are not synchronized, some tricks are necessary in order to work in a synchronous framework. One case is presented to demonstrate the range of possibilities. In general, even when the clocks are not synchronized, the ratio of frequencies is kept within a limited range. Assume, for example, that the ratio can range from 1:2 to 1:3, which means (among other things) that sometimes the faster clock beats twice (and possibly three times) before the slower clock beats once. One possible solution is to model a slow clock that non-deterministically generates a pulse once every two or three cycles:

```
define FAST_CLOCK := 1;
var clock_counter: 0..2;
assign next(clock_counter) :=
    {(clock_counter +1) mod 2 , (clock_counter +1) mod 3};
define SLOW_CLOCK := clock_counter = 0;
```

Even if the clock scheme in your design is a complex one, we recommend that you begin verification with the simplest scheme possible. **WHAT/WHO** is likely to detect some of the design errors regardless of the scheme.

Only after the simplified design seems to be error free, should you move to a more complex and realistic scheme and hunt for the problems that otherwise cannot be detected.

4.11 Modeling Reset

Another important signal that appears in most of the designs is the reset signal. Usually, reset is activated for some time after power-up, and then deactivated for normal operation. Reset must be active long enough to initialize all memory elements with the correct values. In many designs, a few cycles (1 to 10) are enough. The following example shows an environment model that generates a 4-cycle active-high reset:

```
var reset_counter : 0..4;
assign
  init(reset_counter) := 0;
  next(reset_counter) := if reset_counter=4 then 4 else reset_counter+1 endif;
define RESET := reset_counter != 4;
```

It is important to identify the optimal duration of reset. It should be long enough for correct operation, but not too long. A big counter may contribute to the size problem inherent to formal verification and may result in unnecessarily long counter examples.

CHAPTER 5

Sugar – The RuleBase Specification Language

5.1 Overview

Sugar is the specification language of RuleBase. It is used to formally describe properties to which the design under verification must adhere. Sugar is an extension of the temporal logic CTL (Computational Tree Logic). CTL is designed with academic orientation, and needs some adjustments in order to be used in industry. Particularly, complex CTL specifications are difficult to read and write. Sugar adds, on top of CTL, a small set of new operators that simplify formulation of complex properties. It is fully backward compatible with CTL.

The following sections describe both CTL and Sugar. Section 5.2, Semantic Model, provides background on the underlying model on which CTL and Sugar operate (it is not necessary for the understanding of the rest of the chapter and you can skip it if you like). Section 5.3, CTL Operators and Section 5.4, Sugar Operators describe the CTL and Sugar operators, and the remaining sections offer some practical advice. Before you start to write formulas, we recommend that you read CHAPTER 7: Managing Rules, Modes, and Environments.

5.2 Semantic Model

RuleBase is used to verify that a given finite-state machine satisfies a given list of properties. The machine consists of a design, usually written in a hardware description language, composed with an EDL (Environment Description Language) description of the target environment in which the design is expected to run. There are cases, such as in protocol verification, where both the design and the environment are written in EDL. The finite state machine has no free inputs; every input of the design is driven by some signal of the environment, and every input of the environment is driven by the design. While there are no free inputs, the machine usually has multiple choices when moving to the next state, because some of the state-variables, mainly those that model the environment, have non-deterministic behavior.

A non-deterministic finite state machine can be unfolded into an infinite tree that represents the machine's computations. The tree root represents the initial state of the machine, each tree node corresponds to a state in the machine, and the edges that emanate from a state are the possible transitions to other states. The infinite paths of the tree, beginning at the root, are the machine's computations. A machine with multiple initial states is unfolded into multiple trees. In the unfolded tree, different nodes may correspond to the same state of the machine. Figure 5 shows an example of a machine and its computation tree.

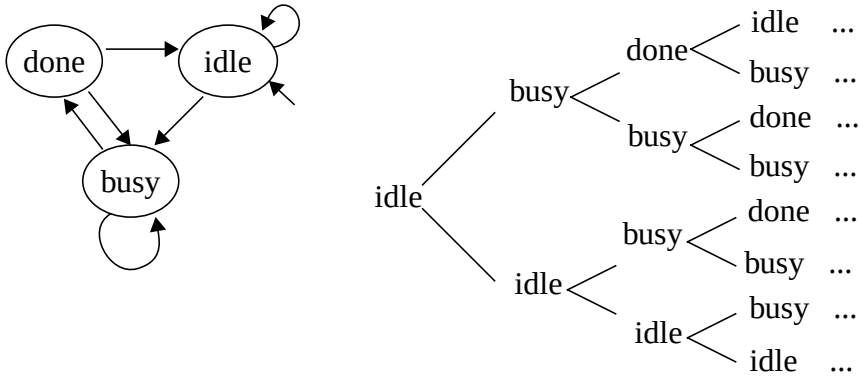


FIGURE 5. Example: A finite-state machine and part of its computation tree

It may be useful to keep this computation tree structure in mind when writing rules, because RuleBase formulas are interpreted over such trees.

Within RuleBase, rules are written in the specification language Sugar. Sugar is built on top of CTL (Computational Tree Logic). CTL, and hence Sugar, is specially designed to work with the computation trees described in the previous paragraphs. In the temporal logic CTL, time is discrete, and the world consists of a current state, mapped to a specific node in the computation tree, and of many possible futures (all computation paths emanating from this state). CTL has no way to refer to the past. The only way to reason about the past is to have information stored in state variables.

An important premise in CTL is that time is infinite. A computation is an infinite sequence of points in time, which start at the current state. Thus, from any point in time (any “current state”), there are many infinite computations (branches) into the future. In Figure , the beginning of one path (recall that all paths are infinite) is shown in bold. In this figure, and in later figures, the only reason that some points do not show a future is lack of space. **Every point in**

time has a future. To simplify the figures, state names are sometimes omitted from tree nodes.

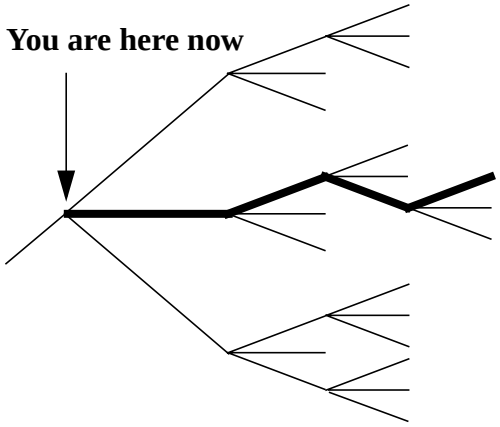


FIGURE 6. Beginning of one possible path

5.3 CTL Operators

CTL formulas have the following syntax:

1. Signal names and constants are CTL formulas.

2. CTL formulas combined with boolean operators are also CTL formulas:
   !f1, f1 & f2, f1 | f2, f1 -> f2, f1 <-> f2, f1 xor f2,

3. If f, f1, and f2 are CTL formulas, then the following are also CTL formulas:
   AX f, EX f, AG f, EG f, AF f, EF f, A[f1 U f2], E[f1 U f2]
   These eight operators are called **temporal operators**.

Boolean operators have their usual meaning.

Temporal operators are used to reason about events that take place along some time interval. Each temporal operator consists of two letters. The first letter is either **A** or **E**, where **A** means “the formula holds in all paths beginning in the current state”, and **E** means “the formula holds in at least one path beginning in the current state”. The second letter is **G**, **F**, **X,** or **U**, where **G** means “the formula holds from now on”, **F** means “the formula holds now or will hold in the future”, **X** means “the formula will hold in the next point of time”, and “f1 **U** f2” means “f2 holds now or f1 will hold until (but not necessary including) f2 holds”.

The temporal operator letters and their meanings are:

- **A** =All
- **E** = Exists
- **G** = Globally
- **F** = Future
- **X** = neXt
- **U** = Until

The following sections detail the eight temporal operators.

Temporal operators take precedence over Boolean operators. Therefore, you should use parentheses to enclose the formula to which the temporal operator is applied.

5.3.1 AG and EG

By combining the meaning of **A** with the meaning of **G**, the resulting **AG** means “for all paths, from now on”. This is depicted in Figure 7 below. The points in time affected by the operator are marked with a black triangle.

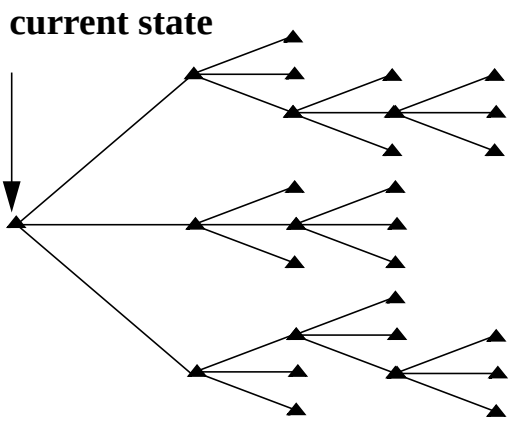


FIGURE 7. **AG**

As can be seen by looking at Figure 7 , all points in time on paths that start in the current state are marked. Consider an example with two signals, “read” and “write”, which should never be active simultaneously. This fact can be stated in CTL as follows:

AG !(read & write) **(For. 1)**

Because the Boolean formula “**!**(read & write)” is prefixed by **AG**, it will be checked at every point in time starting at the current state.

EG, on the other hand, means “for some path, from now on”. This is depicted in Figure 8 below.

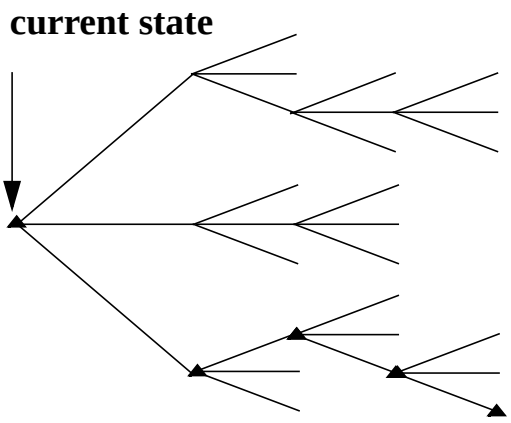


FIGURE 8. EG

In Figure 8 , you can see that all points in time along one infinite path are marked. This illustrates the fact that in order for **EG** to be satisfied, you need **at least** one path where every point in time satisfies the demand. For example:

EG (transaction_starts -> read) **(For. 2)**

states that there is a possible computation (infinite branch) in which all the transactions are reads.

5.3.2 AF and EF

By combining the meaning of **A** with the meaning of **F**, we find that **AF** means “for all paths, now or at some future point in time”. This is depicted in Figure below.

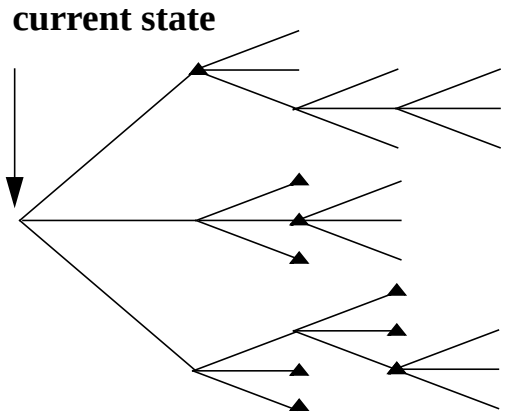


FIGURE 9. **AF**

By examining Figure , we can see that starting at the current state, along every possible path, at least one future point is marked. For example, say that at the current state, a request has been made and it requires an acknowledge. The acknowledge may take place at different points in time, depending on the circumstances, but it must always eventually take place. This can be expressed in CTL as:

AF ack **(For. 3)**

The above formula is not very useful, since in real life a request is made at many points in time and under many circumstances. In real life, our world would probably look more like this:

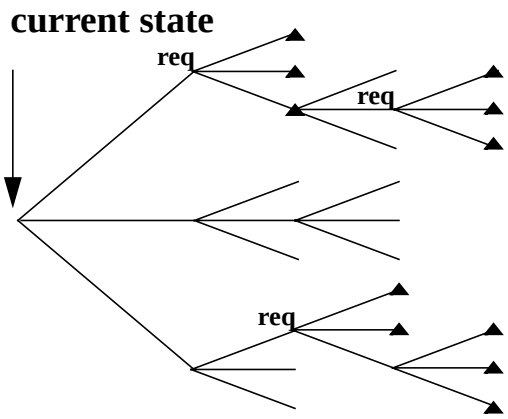


FIGURE 10. AF in the real world

In Figure 10 , a request is made at three different points in time. Starting at each point where a request is made, there are many infinite paths. For each one of those paths, at least one future point is marked. This can be expressed in CTL as:

AG (req -> **AF** ack) **(For. 4)**

where "->" is the "implies" operator. Thus, this formulas can be read as: for all paths, at every point in time, if there is a request, then for all paths emanating from that point, at some future time, we must receive an acknowledge. In simpler terms: whenever there is a request, eventually there is an acknowledge.

There are still some open questions regarding Formula 4. Why is **AG** required in Formula 4? Why not simply state:

req -> **AF** ack **(For. 5)**

The answer is that Formula 5 refers only to the initial state. For a hardware model, the initial state is located at power on. Thus, Formula 5 refers only to a request that occurs at power on. In order to express events that take place after power on, you must always enclose the formula in one of the eight basic temporal operators (**AG, AF, AX, AU, EG, EF, EX, EU**). Specifically, in order to express a request that can happen at any time, you must enclose Formula 5 in the temporal operator **AG**. **)**

EF, on the other hand, means “for some path, at some point in time”. This is depicted in Figure below.

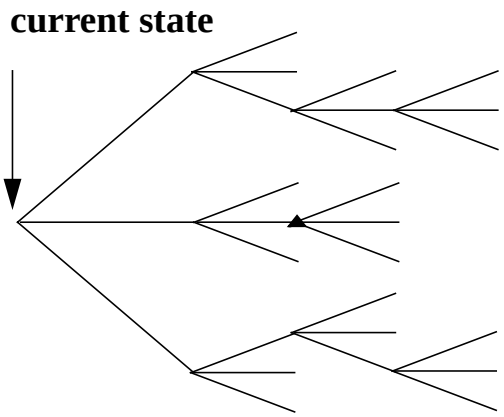


FIGURE 11. EF

By examining Figure , you can see that there is some point in some future path from the current state which is marked. For example, **EF** can be used to express that it must always be possible for our state machine to return to state “idle”, as follows:

AG EF (state = idle) **(For. 6)**

which reads as: for all paths, at all points in time, there is some path in which, at some point in time, the state will be idle. In simpler terms: it is always true that a path exists to idle. Thus, **EF** can be used to express a lack of deadlock.

5.3.3 AX and EX

AX means “for all paths, at the next point in time”. This is depicted in Figure below.

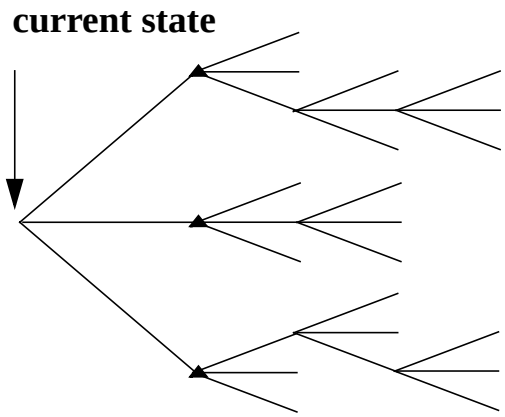


FIGURE 12. AX

In Figure 12 , along all paths that start in the current state, the very next point in time is marked. For example, if a request is made at the current state, and an acknowledge is required at the very next time step. This is expressed as:

AX ack **(For. 7)**

As is the case with **AF** described above, Formula 7 is not practical, since in real life a request is made at many points in time and under many circumstances. In real life our world would probably look more like Figure 13 .

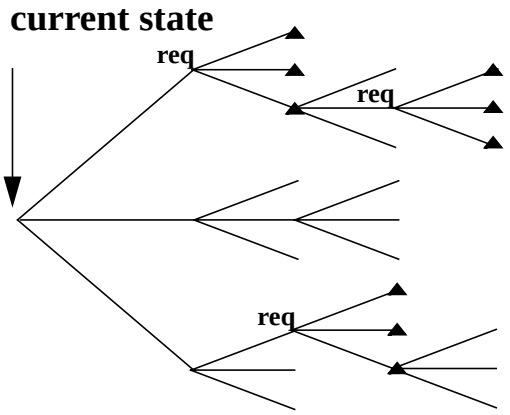


FIGURE 13. AX in the real world

In Figure 13 , a request is made at three different points in time. Starting at each point where a request is made, there are many infinite paths. For each one of those paths, the very next point in time is marked. This can be expressed in CTL as:

AG (req -> **AX** ack) **(For. 8)**

Formula 8 can be read as follows: for every request, we must get an acknowledge at the next point in time.

It is worthwhile to compare Figure 10 with Figure 13 . In the former, a request must be acknowledged eventually. In the latter, a request must be acknowledged at the very next point in time.

EX means “for some path, at the very next point in time”. This situation is depicted in Figure 14 below.

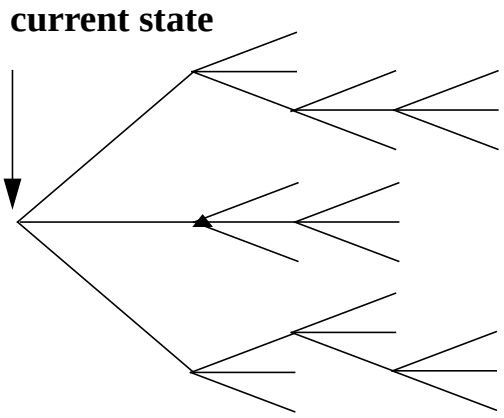


FIGURE 14. EX

Once again, by studying Figure 14 , you can see that for some path from the current state, the very next point in time is marked.

5.3.4 AU and EU

AU has two operands, and is used as follows:

A [q **U** r] (For. 9)

which reads: for all paths, q is true until r is true. Note that:

- r **must** occur eventually.
- r can occur in the current state, in which case q may not appear at all.
- q need not hold at the time r holds.

This is depicted in Figure below.

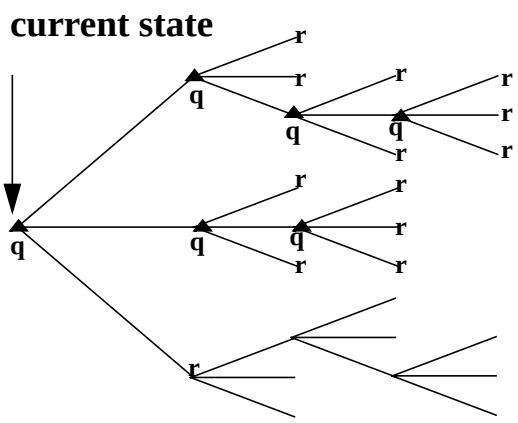


FIGURE 15. **AU**

By examining Figure , you can see that from the current state, all points on all infinite paths are marked until a point where r holds is reached. The marked points are those in which q must be true. For example, suppose that you want to ensure that a busy signal is asserted from the moment a request is made up until the time that an acknowledge is received. This is expressed in CTL as:

AG (req -> **A**[busy **U** ack]) **(For. 10)**

In this case, Figure represents a subset of the complete time tree, with a request that occurs at the current state.

The **AU** operator requires that the terminating condition eventually happen. That is, there are two ways Formula 10 can fail. First, if the busy signal is inactive somewhere between req and ack, and second, if the ack never occurs. Because it makes a demand on its terminating conditions, **AU** is known as a **strong** operator.

EU means “for some path, until”. The computation tree for **EU** is left as an exercise for the reader.

At this point, the eight basic CTL operators A**G, EG, AF, EF, AX, EX, AU,** and **EU** have been covered. While combinations of the basic CTL temporal operators presented here can provide a lot of expressive power, complex CTL formulas are difficult to read and write. To overcome this limitation, RuleBase provides higher-level operators that add more expressive ability.

5.4 Sugar Operators

Sugar adds several operators on top of CTL in order to answer real needs that arise in practical formal verification. Although many Sugar formulas can be expressed in pure CTL, many other formulas are practically impossible to express in CTL because they would be too complex. Sugar is also stronger than pure CTL in the theoretical aspect, mainly in its ability to express any regular expression, as described in Section 5.4.6.

Experience shows that in CTL it is easy to write formulas that are syntactically correct, but their meaning is completely different from what the user had in mind. Sugar protects you from making these kinds of mistakes in two ways. One way is to limit the formulas syntactically. For example, in some fields of certain Sugar operators only Boolean expressions are allowed. The other way is to produce a warning when a formula is suspected of having a wrong meaning. For more details see Section 5.7, Writing Correct Formulas.

Experience indicates that almost all useful formulas fall into the ACTL [SG90] subset of CTL, (i.e., they require that properties will hold along **all** paths rather than in **some** paths). For this reason, the new Sugar operators should be interpreted as being applied to **all** paths (as if there is an **A** in front of them).

Another observation is that the strong versions of the CTL **until** operator (**AU** and **EU**) are not suitable for the formulation of many properties. Expressing a weak **until** (in which there is no demand that the terminating condition must eventually occur) in CTL is laborious and error prone. Sugar provides the weak

until operator, and in addition, provides both weak and strong versions of some higher-level operators (**next_event**, **within**, etc.). A strong operator name has '**!**' as its last character (e.g., **within!**).

The following sections describe Sugar operators, beginning with the simpler ones.

5.4.1 Bounded-Range Operators

5.4.1.1 AX[n]

The first Sugar operator is **AX**[n]. This is simply shorthand for n times **AX**. For example,

AG(req -> **AX**[3] ack) **(For. 11)**

is equivalent to

AG(req -> **AX AX AX** ack) **(For. 12)**

This can be read as "whenever there is a request, an acknowledge will be received three clocks later".

5.4.1.2 ABF

The operator **ABF**[i..j](f) constrains the future of the operator **AF** between i and j clocks from where it is applied. For instance, the following example exhibits the rule "whenever there is a request, an acknowledge will be received within 1 to 3 clocks":

AG (req -> **ABF**[1..3](ack)) **(For. 13)**

The equivalent CTL expression of this simple fact is:

AG (req -> **AX** (ack | **AX** (ack | **AX** ack))) **(For. 14)**

5.4.1.3 ABG

The operator **ABG**[i..j](f) constrains the future of the operator **AG** between i and j clocks from now. For example, the following expresses the rule “whenever there is a request, the busy signal is locked and stays locked for the next 4 clocks”:

AG (req -> **ABG**[0..4](busy)) **(For. 15)**

The equivalent CTL expression is:

AG (req -> (busy & **AX** (busy & **AX** (busy & **AX** (busy & **AX** busy))))) **(For. 16)**

5.4.2 Until Operators

5.4.2.1 until

As discussed in Section 5.3.4, the **AU** operator is a strong operator. That is, the formula

A [p **U** q] **(For. 17)**

means that q must eventually occur, and that p must be true on all paths until q occurs. The **until** operator is the weak version of the **AU** operator. It is written:

p **until** q **(For. 18)**

and means that for all paths, p is true until q occurs. However, the weak **until** does not require that q eventually occur (in that case p must be true forever). For example, to express the rule “always, once a transaction starts, there will be no additional transaction starts before the end of the first transaction”, you can use the following Sugar formula:

AG (trans_start -> AX (!trans_start **until** trans_end)) **(For. 19)**

Formula 19 does not require that every transaction end, only that a new one does not start before the first one ends.

Another way to write the weak **until** operator is:

A [p **W** q] **(For. 20)**

which uses syntax that mimics that of CTL.

5.4.2.2 until!

The **until**! operator is a strong version of the **until** operator. It is equivalent to the CTL operator **AU**.

5.4.2.3 until_

Formula 18 requires that p be true until, but not including, the cycle on which q is true (if there exists such a cycle). The statement

p **until_** q **(For. 21)**

means “p until q” and also requires that at the first cycle where q is true (if at all), p is also true.

5.4.2.4 until!_

The **until**!_ operator is a strong version of the **until_** operator.

5.4.3 Before Operators

5.4.3.1 before

The **before** operator has the format

p **before** q **(For. 22)**

and means that on all paths, the first p must happen before or together with the first q. The **before** operator is a weak operator, that is, it does not require that p eventually happen.

5.4.3.2 before!

The **before**! operator is a strong version of the **before** operator. Thus, the formula:

AG (req -> (data_receive **before**! ack)) **(For. 23)**

requires that after request, data_receive is asserted before or together with ack, and data_receive must eventually be asserted.

5.4.3.3 before_ and before!_

(p **before_** q) and (p **before_!** q) are similar to (p **before** q) and (p **before**! q), but require that the first p happen strictly before (and not together with) the first q.

5.4.4 Next_event

next_event is a conceptual extension of the **AX** operator. While **AX** talks about the next cycle, **next_event** talks about the next time a certain event occurs. Variations of **next_event** are extensions of the **AX**[n] and **ABG**[i..j] operators.

5.4.4.1 next_event(p)(q)

The operator **next_event**(p)(q) means that the next time that p occurs, q will occur. For instance, imagine an arbiter in which requests are processed in the order they are received, unless there is a high priority request, in which case it must be processed immediately. For simplicity's sake, assume that there is only one requestor that can send high priority requests. Then a rule might be:

"whenever a high priority request is received, the next grant must be to the high priority requestor". Here is the rule in Sugar:

AG ((req & high_priority) -> **AX (next_event**(grant)(dst = high_priority_requestor))) **(For. 24)**

It is important to note that the operator **next_event**(p)(q) does not require that the event p eventually happen. It only states that if p does happen, then q must happen. Thus, Formula 24 can be more precisely read as: "whenever a high priority request is received, if there is eventually a grant, then the first grant must be to the high priority requestor". Because this operator does not make any demands on the eventual occurrence of p, it is known as the weak **next_event** operator. The strong **next_event** operator, presented in Section 5.4.4.2, has the added semantics of "p must eventually occur".

There is one limitation on the use of **next_event**(p)(q) and all its incarnations. While q can be any Sugar formula, p must be a Boolean formula, (i.e., a formula with no temporal operators).

5.4.4.2 next_event!(p)(q)

The operator **next_event**!(p)(q) is called the strong **next_event** operator. It means the same as **next_event**(p)(q) with the additional meaning that p must occur. Thus, the strong version of Formula 24:

AG ((req & high_priority) -> AX (**next_event**!(grant)(dst = high_priority_requestor)))**(For. 25)**

states that "whenever a high priority request is received, a grant must eventually occur, and the next grant must be to the high priority requestor".

5.4.4.3 next_event(p)[n](q)

The operator **next_event**(p)[n](q) means "on the nth time that p occurs, q will occur". For example, suppose that for every request, 4 ready signals must be sent, and that on the last one, a signal called last_ready must be sent. That is, after a request, the 4th ready signal must be accompanied by the signal last_ready. This can be expressed in Sugar as:

AG (req -> AX (**next_event**(ready)[4](last_ready))) **(For. 26)**

As with **next_event**(p)(q), this operator is a weak operator—it does not require that p occur the specified number of times. For the corresponding strong operator, see Section 5.4.4.4.

5.4.4.4 next_event!(p)[n](q)

This is the strong version of the **next_event**(p)[n](q) operator. It has the same meaning as the corresponding weak operator of Section 5.4.4.3, with the additional meaning that p must occur at least n times. Thus, the strong version of Formula 26:

AG (req -> AX (**next_event**!(ready)[4](last_ready))) **(For. 27)**

states that after a request, there must be at least 4 ready signals, and the 4th ready signal must be accompanied by the signal last_ready.

5.4.4.5 next_event(p)[i..j](q) and next_event!(p)[i..j](q)

The formula

next_event (p)[2..4](q) **(For. 28)**

states that in the second, third, and fourth times that p occurs, q occurs as well. Formula 29 is a stronger version of Formula 28, which also requires that p occur at least 4 times on every possible path.

next_event!(p)[2..4](q) **(For. 29)**

5.4.4.6 next_event_f(p)[i..j](q) and next_event_f!(p)[i..j](q)

The formula

next_event_f(p)[3..4](q) **(For. 30)**

states that in one of the third or fourth times that p occurs, q should occur as well. Formula 31 is a stronger version of Formula 30, which also requires that p should occur at least 3 times on every path where q occurs on the third p, and at least 4 times on others.

next_event_f!(p)[3..4](q) **(For. 31)**

5.4.5 Within and Whilenot

The behavior of many reactive systems is repetitive, and consists of a few basic types of transactions that take place again and again. In such systems, there are properties that are only interesting within transaction boundaries. The **within** and **whilenot** operators can help formulate such properties by limiting the scope of formulas to given intervals. By handling boundary conditions, **within** and **whilenot** let you focus on the actual properties to be checked, without worrying about extreme cases.

5.4.5.1 within(p,q)(r)

The operator **within**(p,q)(r) means that "formula r is true in the period of time starting when p is true and ending one cycle before q is true". For instance, we can express the requirement "between a request and its acknowledge, the busy signal must remain asserted" as follows:

AG(**within**(req,ack)(**AG** busy)) **(For. 32)**

Compare this formula with Formula 15, where we knew exactly how long busy should be asserted. In Formula 32, we express the fact that the busy signal should remain asserted for a period of time without knowing in advance exactly how many clocks that will be, or whether it is exactly the same number of clocks each time.

within is a weak operator—it does not require that either of the conditions p or q ever happen. But, if in some computation, p occurs and q never follows, then the formula r should hold at p and remain true forever. For the corresponding strong operator, see Section 5.4.5.2

The effect of the **within**(p,q)(r) operator on **AF** is more interesting. Recall that the standard meaning of **AF** is “for all paths, at some point in the future”. By restricting the **AF** operator with the **within**(p,q)(r) operator, **AF** means “for all paths, at some point (in the future) between p and q”. For instance, suppose you want to express the fact that before an acknowledge can be sent, data must be received. This can be done using the following Sugar formula:

AG (**within**(req, ack)(**AF** data_receive)) **(For. 33)**

Because the **AF** operator is restricted by **within**, its scope ends at the acknowledge. Thus, the formula expresses the fact that for all paths that start at the time of a request, at some time in the future but before an acknowledge signal is asserted, data is received.

In the general case, **within**(p,q)(r) trims the tree of computations and checks the validity of formula r on this trimmed tree rather than on the full tree of computations. The trimmed tree only contains the cycles of every path between p and a cycle before q. Thus, the trimmed tree could have finite paths as well. In fact, in the strong **within**!(p,q) operator (Section 5.4.5.2) one may think of the trimmed tree as only having finite branches.

5.4.5.2 within!(p,q)(r)

This is the strong version of the **within**(p,q)(r) operator. It has the semantics of the **within**(p,q)(r) operator, with the additional requirement that p must eventually occur and q must eventually follow (p may occur at the same time as q). Thus, the strong version of Formula 32:

AG (**within**!(req,ack)(**AG** busy)) **(For. 34)**

states that “after every point in time there is a request that is followed by an acknowledge signal, and between the request and its acknowledgment, busy should be active”.

5.4.5.3 whilenot(q)(r)

The operator **whilenot**(q)(r) means that in every computation formula r is true now and stays true at least until a clock before q is true. If q is true now then **whilenot**(q)(r) is also true. For instance, Formula 32 can also be expressed as:

AG (req -> **whilenot**(ack)(**AG** busy)) **(For. 35)**

The operator **whilenot**(q)(r) is a weak operator, that is, it does not require that q eventually happen.

whilenot(q)(r) is a derivative of the **within** operator. It may be thought of as **within**(now,q)(r).

5.4.5.4 whilenot!(q)(r)

This is the strong version of the **whilenot**(q)(r). It has the same meaning as the **whilenot**(q)(r) operator, with the addition that q must eventually happen.

5.4.6 Sequence

The sequence is a Sugar construct used to describe computation paths on which some formula must hold. It looks like a regular expression, and its semantics resemble the semantics of regular expressions. The sequence suits the world of hardware design. It can be regarded as a textual representation of a timing diagram, or as a generalized control program for simulation. Its main advantage is the simplicity of writing certain properties that are difficult to formulate using other CTL and Sugar operators.

The sequence has two parts, a list of events {e1, e2, ...} and a Sugar formula (f).

{ e1, e2, ... , en } (f) **(For. 36)**

The sequence can be regarded as an **if** statement, in which the event list is a condition that indicates when to check the formula. It means "if at some com-

putation path all the events take place in the order they are defined, then the formula must hold on this path at the last cycle of the last event in the list" (an event may last more than one cycle). A comma between two events denotes a move of one cycle forwards (however, if an event takes zero cycles, a comma either before it or after it is ignored).
An event can be one of the following:

1. **p**
   A Boolean expression ‘p’.
   The expression ’p’ holds for one cycle.
2. **p[=i]**
   p is a Boolean expression.
   p occurs exactly i times, not necessarily consecutively.
   p[=3] is equivalent to {!P[*], P, !P[*], P, !P[*], P !P}
   Example
   {read, write[=3], cancel}
3. **p[>=i]**
   p is a Boolean expression.
   p occurs at least i times, not necessarily consecutively.
   p[>=3] is equivalent to {!P[*], P, !P[*], P, !P[*], P, true[*]}
4. **p[>i]**
   p is a Boolean expression.
   p occurs more than i times, not necessarily consecutively.
   Examples
   {read, write[>3], cancel}
5. **p[<=i]**
   p is a Boolean expression.
   p occurs at most i times, not necessarily consecutively.
6. **p[<i]**
   p is a Boolean expression.
   p occurs less than i times, not necessarily consecutively.

7. **p[>i,<j]**

   p is a Boolean expression.

   p occurs more that i times but less than j times, not necessarily consecutively.

   i and j are natural numbers (i>=0 , j>i+1).

8. **p[>=i,<j]**

   p is a Boolean expression.

   p occurs at least i times but less than j times, not necessarily consecutively.

   i and j are natural numbers (i>=0 , j>i).

9. **p[>i,<=j]**

   p is a Boolean expression.

   p occurs more that i times but at most j times, not necessarily consecutively.

   i and j are natural numbers (i>=0 , j>i).

10. **p[>=i,<=j]**

    p is a Boolean expression.

    p occurs at least i times but at most j times, not necessarily consecutively.

    i and j are natural numbers (i>=0 , j>=i).

    true.
    Skip one cycle. Equivalent to “AX”.

    ‘**[*]**’.
    Skips zero or more cycles. Equivalent to “**AG**”.

11. ‘**goto p**’

    p is a Boolean expression.
    Go to the next time that p occurs.

    Equivalent to {!p[*], p}.

    Example

    {req, **goto** ack, **goto** busy, end} (done)

12. ‘**p holds_until q**’,

    p and q are Boolean expressions.

p holds (true) until q occurs
Equivalent to {(p & !q)[*], q}.

Example

{req, busy **holds_until** done} (ack)

The following sequences match the above one:

{req, busy, busy, busy, done}

{req, done}

13. '**p holds_until_ q**',

p and q are Boolean expressions.

q holds until (inclusively) q occurs.
Equivalent to {(p & !q)[*], p&q}.

14. **Q[n]**

A sub-sequence Q followed by '**[n]**', where n is a positive integer.
The sub-sequence holds n consecutive times.

15. **Q[*]**

A sub-sequence Q followed by '**[*]**'.
The sub-sequence holds zero or more consecutive times.
(Note: If Q is not a simple Boolean expression, then this kind of event must be followed by a simple Boolean event.)

16. **Q[+]**

A sub-sequence Q followed by '**[+]**'.
The sub-sequence holds one or more consecutive times.
(Note: If Q is not a simple Boolean expression, then this kind of event must be followed by a simple Boolean event.)

17. **P || Q**

Two sub-sequences P and Q separated by **'||'.** (or-between-sequence).
Either the first sub-sequence holds, or the second sub-sequence holds. For example, the formula

AG ({p,{q,r} || {s,t},u}(v))

is equivalent to:

(**AG** ({p,q,r,u}(v))) & (**AG** ({p,s,t,u}(v)))

18. **P && Q**

Two sub-sequences P and Q separated by **'&&'.** (and-between-sequence). P and Q must occur at the same time (start and end at the same cycle).

P and Q must be of the same length (same number of cycles).

Examples

If read arrives before write and both read and write are not cancelled (and get a grant) then

read will be serviced before write.

{[*],

{read, (!cancel)[*], grant_read} **&&**

{true, write, (!cancel)[*], grant_write}}

(operate_read **before** operate_write)

Exactly 3 write events should occur during the sequence:

{..... {req, read[+], flush, cancel} **&&** {write[=3]} .....}(...)

19. **P[i..j]**

P is a subsequence

i and j are natural numbers and i>=0, j>=i, j!=0

P holds between i to j times.

Examples

{read, write[7..10], flush}

{read, write[0..3], flush}[1..4]

20. **P[i..]**

P is a subsequence.

i is a natural number and i>=0

P holds at least i consecutive times.

Example

{read, write[7..], flush}[2..]

21. **P ~ Q**

Two sub-sequences separated by '~' (P~Q).

The first cycle of Q starts when P reaches its last cycle.

Examples

{ start, req, **goto** busy~done, end} (ack)

{ start, {{read, busy[*]} || {write, flush }} ~ {done, ready} } (ack)

is equivalent to:

{start, {{read&done}||{read,busy[*], busy&done}} || {write, flush&done} ,ready } (ack)

22. **P ->Q**

Two sub-sequences separated by '->' (P->Q).

If a path that is compatible with P occurs, it must be followed (starting at the same cycle where P ends) by a path whose prefix is compatible with Q.

Examples

req -> (ack **until** (ready **until** (busy **until** end)))

is equivalent to:

{req} -> {ack[*], ready[*], busy[*], end}

{start, data1, data2, error} ->
(**AX**(cancel_data1, &
(**AX**(cancel_data2 &
AX(idle **until** error)))))

is equivalent to:

{start, data1, data2, error} ->
{true, cancel_data1, cancel_data2, idle[*], error}

23. **P =>Q**

Two sub-sequences separated by '=>' (P=>Q).

If some path compatible with P occurs, then it must be followed (starting one cycle after P ends) by a path whose prefix is compatible with Q.

Examples

{start, data1, data2, error} =>

(**AX**(cancel_data1, &
(**AX**(cancel_data2 &
AX(idle **until** error)))))

is equivalent to:

{start, data1, data2, error} =>
{ cancel_data1, cancel_data2, idle[*], error}

24. **P ->Q!**

P, Q are sub-sequences.

If a path that is compatible with P occurs, then it must be followed (starting at the same cycle where P ends) by a path that is compatible with Q and so, reaches Q's end (i.e. reaches the last cycle of Q):

Comments: Strong version of P->Q

Example:

{a,b} -> {c,d[*],e}! - e must happen

{a,b} -> {c,d[*], e} - e may not happen (if d is 'forever') i.e.
a, b&c, d, d, d, d, d,d ,d ..... - is a valid sequence

req -> (ack **until** (ready **until** (busy **until!** end)))

{req} -> {ack[*], ready[*], busy[*], end}!

25. **P => Q!**

P, Q are sub-sequences.

The same as P->Q!, with the difference that Q starts one cycle after P reaches its end.

Examples:

- Additional ways to express **AG** (waiting -> **AX** next_event (done)(**AX** idle)):
  - {[*], waiting, !done[*], done, true}(idle)
  - **AG** {waiting, **goto** done}(**AX** idle)
  - {[*], waiting, **goto** done, true}(idle)
- The fourth *ready* after *start* should be accompanied with *result=ok:*
  {[*], start, { !ready[*], ready }[4] }(result=ok)

The next example is interesting from a theoretical point of view. It is a Sugar formula that cannot be expressed in bare CTL. It expresses the fact that f is true at every even cycle (0, 2, ...):

{ {true, true}[*], true} (f)

Sequences may be useful for showing interesting paths, even if you don't intend to find bugs. Suppose that you want to see a scenario in which a cache line is modified, and later becomes exclusive without being invalidated in between. The following sequence claims that this path is impossible, and its counter example will demonstrate such a path (if one exists):

{ [*], modified, !invalid & !exclusive[*], exclusive } (false)

False is a formula that can never be true, so a counter example will be provided if the sequence in braces is possible.

5.5 Multiple-Clocks in Formulas

Sometimes, the design under verification has more than one clock, and it should be verified in several clock ratios. Assume for example that there are two clocks, clk_a and clk_b, that we want to verify in two ratios: 1:1 and 1:2. Assume also that the following formula is written for ratio 1:1.

AG(p-> **AX** (q-> **AX** (r **until** s))) **(For. 37)**

If signals p,q,r,s, only depend on the slower clock, clk_b, then the formula should be written differently for ratio 1:2.

AG((p &clk_b) -> **AX**[2] (q -> **AX**[2] ((r|!clk_b) **until** (s&clk_b)))) **(For. 38)**

To avoid the need to change formulas when the clock ratio is changed, the user can specify the clock according to which the formula should behave, and the translation will be done automatically. In our example, the user should specify the clock as follows:

AG(p-> **AX** (q-> **AX** (r **until** s))) :: **clk**=clk_b **(For. 39)**

5.6 Quantification Over Data Values

When specifying the behavior of data, it is often necessary to refer to specific data values. For example, suppose that we want to say that the data read in during a *read* operation will be written out in the next *write* operation. One way to do this is to write a formula for each data value:

```
%for i in 0..31 do          -- assuming that the data type is 0..31
  formula { AG( (read & data_in=i) -> next_event(write)(data_out=i) ) }
%end
```

This may be inefficient and even impossible if there are too many values. The above can be done in one formula using the **forall** construct as follows:

```
forall i: 0..31:
 formula { AG( (read & data_in=i) -> next_event(write)( data_out=i) ) }
```

The syntax of **forall** is:

forall variable : type :

where *variable* is an EDL variable that is defined only for the purpose of quantification. It should not be defined elsewhere. *type* is any legal type, including a bit vector.

More examples:

```
forall i(0..31): boolean:
 formula { AG( (read & data_in(0..31)=i(0..31)) ->
                  next_event(write)(data_out(0..31)=i(0..31)) ) }

forall i: 0..15:
 formula { AG( counter=i -> AX counter=(i+1) mod 16 ) }
```

Although it looks natural to use **forall** with formulas, it is also possible to use it anywhere else in EDL. For example:

```
forall i(0..31): boolean:
  define data_in_is_i   := data_in(0..31)=i(0..31);
         data_out_is_i := data_out(0..31)=i(0..31);
  formula { AG( (read & data_in_is_i) -> next_event(write)(data_out_is_i)
) }
```

forall adds extra state variables. In many cases, this will not cause size problems, provided that you have a good BDD order that includes these variables.

5.7 Writing Correct Formulas

The semantic model of CTL and Sugar, described in Section 5.2, is sometimes counter-intuitive. While reasoning about computation **trees** has its benefits, users often think in terms of **paths**. Sugar operators are designed to prevent problems that result from misunderstanding the semantic differences. However, there are still cases in which you should be careful. This section attempts to characterize some of these cases.

In many cases, formulas that are not *causal* have a meaning that does not coincide with the intention of the user. By *causal*, we mean formulas in which an event B depends on event A only if event A occurs no later than event B. For example, assume that you want to state the following rule: “every grant is immediately preceded by a request”. Since CTL cannot reason about the past, one may be tempted to write:

AG((**AX** grant) -> request) **(For. 40)**

This formulation relies on the future and is incorrect; it means “if grant holds in **all** the next states of some state, request must be active in this state”. It misses all the states that have grant active on some, but not all, of their successors. The correct way is to write the following causal formula:

AG(!request -> **AX** !grant) **(For. 41)**

We recommend that you do **not** use CTL formulas that contain the **E** operator (**EG**, **EF**, **EU,** and **EX**) unless a property cannot otherwise be formulated (for example, "**AG EF** p**"** can find a weak form of deadlock). The main reason for this recommendation is that it is impossible to produce a counter-example when an **E** formula fails. The negation of an **A** formula (**AG**, **AF**, **AU,** and **AX**), or of a Sugar formula, is equivalent to some **E** formula, so we also recommend that you do not negate such formulas.

RuleBase employs two methods in order to protect users from these, and other common mistakes. One method is to limit Sugar operators in a way that will prevent unintended use. For example, the **within** operator can take only Boolean expressions (no temporal operators) in its *start* and *end* fields. The other method is to issue warnings for suspected formulas. The cases in which such warnings are issued are:

- For any type of **until** or **before** operator with two temporal operands.
- If the right operand of an **until** operator contains the '->' operator.
- If the operand of an **AF** or an **EF** operator contains the '->' operator.
- For a temporal sub-formula on the left side of an "->" or on any side of the '<->' operator.
- When the operator '|' (boolean **or**) has two temporal operands.

It should be emphasized that **there are** correct formulas that do not obey the above rules. However, it is important to write these formulas very carefully, and to use them only if you are a very experienced user. Most of the properties that are needed in daily use can be formulated while adhering to these rules.

RuleBase can produce textual explanations of Sugar formulas as a formula debugging aid. To see formula explanations select a rule and click the **Explain** push button. These explanations may sometimes help find errors in formulas, by presenting them in a different manner.

5.8 Satellites – More Expressiveness

Although Sugar increases expressiveness capabilities, there are still properties that cannot be expressed, and others that are too complicated to formulate. *Satellites* may provide solutions in many of these cases. A satellite is a state-machine that records events that occur in the design under verification. Formulas can then refer to these past events by accessing the satellite's internal state. Satellites do not affect the design because information only flows from the design to the satellite (except when fairness is used in certain ways).

For example, assume that a queue of depth k reads data on one side and writes it on the other side. Assume that we want to prove that the queue never contains more that k data items. Formulation of this property in Sugar is difficult, but it becomes easy with a satellite. An up/down counter is defined, whose range is 0 to k, and which is incremented on reads and decremented on writes. It is now necessary only to verify that the counter never exceeds k. We can use the same counter to check for an underflow: Its value should never be less than 0.

Some formulas might have become easier if one could talk about past events. Assume that we want to state that "if p occurs, then at that time q should be active **since** the last occurrence of r". We can define the operator **since** as a module:

```
module since( e1, e2 )( e1_since_e2 )
{
  var state: boolean;
  assign next(state) :=
            case
              !e1 : 0;
              e1 & e2 : 1;
              else : state;
            esac;
```

```
  define e1_since_e2 := (e1 & e2) | (e1 & state);
}
```

and use it to formulate the required property:

```
instance i1 : since( q, r )( q_since_r );
formula { AG ( p -> q_since_r  ) }
```

CHAPTER 6

Sugar – Formula Examples

6.1 Overview

In this chapter we present a list of useful formula patterns. Its main purpose is to help the novice user, but experienced users may also find interesting patterns. We want to emphasize the fact that one does not need to know all of these patterns to perform successful verification work. Most of the formulas in an average project only employ a small set of patterns. However, you may find ideas that will simplify your work.

The following list is dynamic and we expect it to continue to grow. If you have additional patterns that may help others, send them to us and we will add them to this list.

Note: *This chapter is brought here in a very preliminary form.*

6.2 Basic Formulas

- *ok* is always true:
  AG *ok*

- *some_requirement* is always true when reset is inactive:
  AG (!reset -> *some_requirement)*
  Note: *Many designs begin in an unspecified state, and are being stabilized during reset. Failure of a formula during reset is not interesting, so we filter this time interval as shown above.*
- Variable *state* can never have the value *error*:
  AG(*state* != *error*)
- Variables *state1* and *state2* are never in the same state:
  AG(*state1* != *state2*)
- Variables *state1* and *state2* are never in state *critical* together:
  AG(*state1* != *critical* | *state2* != *critical*)
  or
  AG !(*state1 = critical* & *state2 = critical*)
- If *busy* is true then *working* is also true:
  AG(*busy -> working*)
- If *almost_done* is true now, *done* will be true in the next cycle:
  AG(*almost_done* -> **AX** *done*)
- If *hold* becomes active, it remains active for at least one more cycle:
  AG(**rose**(*hold*) -> **AX** *hold*)
  Note: rose*(hold) is true if hold is currently 1 and was 0 in the last cycle.*
- *got* should rise 3 cycles after *get* rises:
  AG(**rose**(get) -> **AX**[3](**rose**(got)))
- If we are *going_to_abort* now, we *abort* within 0 to 4 cycles:
  AG(*going_to_abort* -> **ABF**[0..4](*abort*))
- If *master1_needs_bus* becomes active, *master2_accesses_bus* should be inactive for at least 3 cycles, beginning from the next cycle:
  AG(*master1_needs_bus* -> **ABG**[1..3](!*master2_accesses_bus*))
- *Counter* is always between 3 and 7:
  AG(*counter* >= 3 & *counter* <=7)
  or
  AG(*counter* **in** { 3,4,5,6,7 })

- *Status* never has the values *warning* or *error* or *fatal*:
  AG !(*status* in { *warning*, error, fatal })
  or
  AG(*status* != *warning* & *status* != *error* & *status* != *fatal*)
- At most one of the signals *x*, *y* or *z* is 1 (mutual exclusion):
  AG(*x*+*y*+*z* <= 1)
- If *error* becomes active, it will remain active forever:
  AG(*error* -> **AG** *error*)

6.3 Arrays

- Define a bit vector vec of 4 bits that may have at any moment any of the values 3, 8, or 14:
  var *vec(0..3)*: **boolean**; **assign** *vec(0..3)* := {3,8,14};
  Note: *The above is* ***NOT*** *equivalent to* ***"var*** *vec(0..3): {3,8,14};" which declares an array of four enumerated signals, each of them may have one of the values 3, 8, or 14.*
- If the *head* pointer of a queue is equal to the *tail* pointer,*queue_empty* must be true:
  AG((*head(0..3) = tail(0..3)*) -> *queue_empty*)
- The bitwise **and** of vectors *vec(0..7)* and *mask(0..7)* has at least one bit set:
  AG((*vec(0..7)* & *mask(0..7)*) != 0)
- Exactly one bit of the bit vector *v(0..7)* is 1:
  AG((%**for** ii **in** 0..7 **do** *v(ii)* + %**end** 0) = 1)
- The above is expanded to:
  AG(*v(0)+v(1)+v(2)+v(3)+v(4)+v(5)+v(6)+v(7)* = 1)

6.4 Before

- If a *request* occurs, then an *ack* should occur (strictly) before the next *request*:
  AG(*request* -> **AX**(*ack* **before_** *request*))

  Notes:

  - The second *request* may not occur, in which case *ack* is not required.

- **before_** (with an underscore) means strictly before: *request* will come (if at all) at least one cycle after *ack.*
- The **AX** means that we expect *ack* to come at least one cycle after *request*.

- Another way to formulate the above requirement, which allows more explicit specification of boundary conditions:

  { [*], *request*, !*ack*[*], *request* }(**false**)

  Notes:

  - The path begins with any sequence of events. Then a *request* occurs, and (beginning from the next cycle) *ack* is inactive for zero or more cycles. Finally, there is another *request*. The *false* on the right hand side means that if such a sequence exists then the formula should fail.
  - A technique we use is:
    Instead of specifying what should happen, specify what should not happen (as a bad sequence of events), and require false to be satisfied at the end of this sequence. Since false is a formula that may never be satisfied, existence of the bad sequence in our design will cause RuleBase to produce a counter-example.

6.5 Until

- If *request* is asserted, it will remain active until (inclusive) *grant*:
  AG(*request* -> (*request* **until_** *grant*))

  Notes:

  - *grant* may never occur after this *request*, in which case *request* must stay active forever.
  - **until_** (with an underscore) means that *request* must also hold at the first cycle where *grant* holds.
- Another way to formulate the above requirement:
  { [*], *request* & !grant, !*grant*[*], !*request* }(**false**)
- If *request* is asserted, it will remain active until (not inclusive) *grant*:
  AG(*request* -> **AX**(*request* **until** *grant*))

- Other ways to formulate the above requirement:
  { [*], *request*, !*grant*[*], !*request* & !*grant* }(**false**)
  or
  { [*], *request*, !*grant*[*] }(*request*)

6.6 Forall

- If *data_in(0..7)* has some value during *read*, in the next time that *write* is active *data_out(0..7)* will have the same value:
  forall x(0..7): **boolean**:
  AG((*read* & *data_in*(0..7)=x(0..7)) -> **next_event**(*write*)(*data_out*(0..7)=x(0..7)))

Notes:

- **forall** is a means for applying a formula to multiple values at a time. It is equivalent to writing a separate formula for each value that the **forall** variable can take:
  AG((*read* & *data_in*(0..7)=0) -> **next_event**(*write*)(*data_out*(0..7)=0))
  ...
  AG((*read* & *data_in*(0..7)=255) -> **next_event**(*write*)(*data_out*(0..7)=255))
- **forall** has its penalty—an extra state variable (8 bits in the example above)—but this variable does not usually contribute excessively to the size problem, if the BDD order is reasonable.

6.7 Eventuality

- If *request* is asserted, *ack* should be asserted in the future, beginning from the next cycle:
  AG(*request* -> **AX AF** *ack*)
- If *request* rises, *ack* should be asserted at the same cycle or in the future:
  AG(**rose**(*request*) -> **AF** *ack*)
- No matter what is the current state, it is always possible to reach a state where *mstate=idle*:
  AG AF *mstate=idle*

6.8 More Sequences

- If *grant* is active, and there is no *retry* in the next cycle, *busy* must become active two cycles after *grant*:
  { [*], *grant*, !*retry* }(**AX** *busy*)
  or
  { [*], *grant*, !*retry*, **true** }(*busy*)
  or
  { [*], *grant*, !*retry*, !*busy* }(**false**)
  or
  AG(*grant* -> **AX**(!*retry* -> **AX** *busy*))
- The fourth *data_ready* after *start* should be accompanied by *last_data*:
  { [*], *start*, {!*data_ready*[*], *data_ready*}[4] }(*last_data*)
- The fourth *data_ready* after *start* should be accompanied by *last_data*, unless there was an *abort* in the middle:
  { [*], *start* & !*abort*, {!*data_ready* & !*abort*[*], *data_ready* & !*abort*}[4] }(*last_data*)

CHAPTER 7

Managing Rules, Modes, and Environments

7.1 Overview

There are many possible ways to structure a verification project. The basic elements of all structures are the same: EDL statements, formulas, modes, and rules. However, as the project becomes more complicated, spans a longer period, and more people become involved, it becomes more important to use a standard methodology.

The main contributor to project complexity is Behavioral partitioning (see “Behavioral Partitioning” on page 157). Behavioral partitioning is an effective method to attack the size problem. In this method, the environment is degenerated in various ways to reduce the size of the design to be verified. Formulas should then be run in multiple reduced environments to cover the full environment. Unless managed carefully, these multiple environments may get out of control.

This chapter suggests a methodology of managing multiple rules, modes, and environments. The methodology is a result of our experience in many formal verification projects. Section 7.2, Defining Rules and Modes describes the syn-

tax and semantics of rules and modes. Section 7.3, Using Modes to Limit the Environment shows an example of how to approach the size problem by using modes, and Section 7.4, Verification Project Management suggests how to structure a verification project that has multiple environments.

7.2 Defining Rules and Modes

RuleBase is rule oriented. A rule is the basic entity that can run. A rule defines a group of related formulas to be verified in one run. It may also re-define parts of the design or environment, thereby overriding the default behavior for the specific run.

The rule syntax is as follows:

rule name {
  "optional textual description of the rule"

  -- at least one formula
  formula "optional textual description" { Sugar-formula }
  formula "optional textual description" { Sugar-formula }
  ...

  -- the rest of the statements are optional:

  envs rule-name, rule-name, ... ;
  formulas rule-name, rule-name, ... ;
  test_pins signal-name, signal-name, ..., rule-name, rule-name, ... ;
  inherit rule-name, rule-name, ... ;

  <EDL statements (**var**, **assign**, **define**, **instance**, **fairness**)>
}

A **mode** is a rule that cannot be run by itself, and is used to group and name formulas and/or environments. It can only be inherited by rules or by other modes. The syntax of **mode** is exactly the same as the rule syntax, except that it begins with the keyword **mode** instead of **rule**.

A rule must contain at least one formula (not required in mode). All the other parts are optional. The order of statements in a rule is unimportant, and all kinds of statements may appear numerous times. We recommend that you fill the textual description of formulas and rules. This description may help during the analysis of verification results and facilitate maintenance.

Rules and modes can inherit formulas and EDL statements from other rules and modes:

- The **formulas** statement inherits formulas.
- The **envs** statement inherits EDL statements.
- The **test_pins** statement forces RuleBase to keep some signals during the reduction stage, even if they are not needed for verification of the specific rule. Sometimes these signals are needed to provide a better understanding of counter-examples. Test pins can also be inherited. The statement

  test_pins enable, command;

  forces RuleBase to keep track of signals enable and command, even if they are not needed for verification. These signals can later be viewed in Scope windows (the Scope waveform display tool is explained in “Scope Waveform Display Tool” on page 166). The statement

  test_pins <rulename>;

  inherits all **test_pins** statements that appear in rule <rulename>. If <rulename> is also a name of a signal in your design, then the above statement is ambiguous, and RuleBase will issue the following error message:

  Name collision: <rulename> is both a rule and a signal
- The **inherit** statement can be used to inherit the environments, formulas, and test pins. The statement

  inherit rule_name;

  is equivalent to:

envs rule_name; **formulas** rule_name; **test_pins** rule_name;

Rules and modes may include EDL statements (**var**, **assign**, **define**, **fairness** and **instance**). The behavior assigned to signals by these statements overrides the signals' behavior in the default environment (all EDL statements outside rules or modes are considered as the default environment). A rule may inherit EDL statements from other rules or modes using the **envs** statement. Inherited statements override the default environment, but are overridden by statements written directly in the body of the rule. The exact hierarchy of behavior is as follows:

1. Signal definition in the default environment overrides the definition in the design (HDL).
2. Inherited signal definition overrides the definition in the default environment.
3. Signal definition in the running rule overrides inherited signal definition.

7.3 Using Modes to Limit the Environment

One way to approach the size problem is to limit the behavior of the environment, as mentioned in "Behavioral Partitioning" on page 157. RuleBase uses information from the restricted environment to automatically reduce the size of the model to be verified. To help reductions, some signals in the environment may be set to constant values, or restricted to some other simple behavior. This over-reduction is usually done by using modes, rather than in the default environment, as shown in the example below.

Suppose that a design obtains a command and an address from the environment, in addition to other things. The default environment will include the following lines:

```
var command: { load, store, add, jmp };
define CMD(0..2) :=          -- these are the actual command inputs of the design
  case
    command=load  : 010b;
```

```
      command=store  : 111b;
      command=add   : 011b;
      command=jmp   : 100b;
   esac;

var CMD_VALID: boolean;

var ADDR(0..15): boolean;
assign next(ADDR(0..15)) :=
   if CMD_VALID then ADDR(0..15) else nondets(16) endif;
   -- ADDR is stable when CMD_VALID is active, and is free to change otherwise
```

Now, suppose that the design is too large, or verification takes too long, even though you have used all basic methods to cope with size problems (see CHAPTER 8: Size Problems and Solutions). In this case, you may want to perform behavioral partitioning, and define modes that restrict the default behavior. Several possibilities of such modes are shown below:

```
mode load_add {
   "two commands only.  CMD(0..1) become constant"
   var command: { load, add };
}

mode eight_addr_bits {
   "bits 0..7 are 0.  bits 8..15 retain their behavior"
   define ADDR(0..7) := 0;
}

mode load_add.eight_addr_bits {
   "combining the above two modes"
   inherit load_add, eight_addr_bits;
}
```

```
mode another_way_to_do_the_same {
   var command: { load, add };
   define ADDR(0..7) := 0;
}
```

Now, rules can run in the restricted environment by inheriting the above modes. For example:

```
rule some_property {
   inherit load_add.eight_addr_bits;
   formula { ... }
}
```

Since over-reduction limits the model checking run to only a subset of the possible input sequences, multiple runs of the same rule using different environments are sometimes necessary to provide good verification. Managing these multiple environments is described below in Section 7.4.

7.4 Verification Project Management

A well-formed verification project usually consists of the following elements:

- Default environment
- Modes that define restricted environments
- Modes that group related formulas
- Rules

Default environment:
The default environment should model the full behavior of the environment. When writing the default environment, we recommend that you “forget” the small details of how you intend to attack the size problem. This does not mean that the environment is written without considering this problem—on the contrary, the environment models should be abstract and small. Specific reductions should only be reflected in modes, which are to be written at a later stage.

Modes that define restricted environments:
In many cases the default environment does not cause enough reduction of the design to be verified. Behavioral partitioning is one of the methods that may help in these cases. In behavioral partitioning, multiple reduced environments are defined, each of them is represented as a mode. Then each formula is run in all these modes. (See **Section 7.3**.)

Modes that group related formulas:
The necessity to run each formula in multiple environments suggests that you keep formulas in separate modes, to be inherited by rules.

Rules:
In this methodology, the list of rules is a matrix of environment modes and formula modes, in which each formula may run in many environment.

Example:

```
-- environment modes
mode read_only {
   define command := read;
}
mode write_only {
   define command := write;
}

-- formula modes
mode no_starvation {
   formula { AG AF grant1 }
   formula { AG AF grant2 }
}
mode no_collision {
   formula { AG !(grant1 & grant2) }
}
```

```
-- rule matrix
rule read_only.no_starvation    { inherit read_only, no_starvation; }
rule read_only.no_collision     { inherit read_only, no_collision; }
rule write_only.no_starvation   { inherit write_only, no_starvation; }
rule write_only.no_collision    { inherit write_only, no_collision; }
```

CHAPTER 8

Size Problems and Solutions

8.1 Introduction

Size is one of the major obstacles to using formal verification for any design. RuleBase is limited to designs that have several hundred state variables (flip-flops) after reduction, or several thousand before reduction. The number of state variables is a rough estimate of design complexity; the size limit depends on the complexity of the logic as well as the number of memory elements. This chapter discusses techniques you can use to push the size limit for your design as far as possible.

8.2 Design Partitioning

The simplest method to overcome size problems is design partitioning. Thus, instead of trying to verify the entire design at once, you may verify it unit-by-unit. (See also “Design Partitioning” on page 205) The partitioning methodology is as follows:

1. Split the design into manageable partitions, whose interfaces are well defined and easy to model.

2. When verifying a partition, replace its neighbors by abstract models. These models should only represent the interfaces with the verified partition, hiding unnecessary details.
3. Verify the correct behavior of the abstract models of the neighbors by writing specific rules for this purpose.

While partitioning can be quite effective, there are obviously properties that can only be verified when the entire design is considered. Partitioning also requires extra effort in studying internal interfaces and writing models for neighboring blocks.

8.3 Rule Partitioning

Before beginning model checking, RuleBase performs static analysis of the design, and discards any signals that do not affect the rule being run. For example, assume that the design has two outputs, each of which is affected by a different (possibly overlapping) set of input signals. If you run formulas that check these two signals under the same rule, RuleBase will have to build a representation of the entire model. However, if you separate the formulas into two groups, in which one group checks the first output and the second group checks the other output, RuleBase can build a partial representation in each case.

In effect, by partitioning the formulas into different rules, you enable RuleBase to automatically partition the design by only using that part of the design required to check the specific rule.

Note: *Accumulating related formulas in one rule may save time if the formulas refer to the same part of the design.*

8.4 Behavioral Partitioning

Behavioral partitioning is a technique in which the input sequences of a design are restricted to a subset of the legal input sequences. In this way, you allow RuleBase to remove parts of the design that deal with behaviors that are not

seen under the restricted inputs. For instance, if a design has two modes: read and write, controlled by an input signal *command*, you can verify each of the modes separately. You can do this by declaring two separate environments: one *command* is constantly read, and the other is constantly write. This is the only action that you need to take. When input signals are set to a constant value, RuleBase will automatically eliminate the logic that is made redundant. For example, if you set *command* to read, then RuleBase will know how to eliminate all logic that is only activated under mode write.

8.5 Abstraction of the Environment

As we explained in CHAPTER 4 , writing environment models requires extreme thought and attention. Models should be very abstract and general, representing all possible behaviors of the real environment, while remaining simple and small. Models with too much detail are not an advantage and may result in unnecessary growth of the model.

For example, assume that the verified design is a cache controller, connected to a CPU on one side. It is not necessary to create a detailed model of the CPU. Rather, you can create an abstract model of the CPU to model enough of it to produce legal sequences of commands and control signals. Only a few dozen state variables are needed to model this behavior, as compared to the huge number required for the concrete CPU.

8.6 Gradual Enlargement

Attacking a new design with full blown environments is not very effective when the design is large. Experience suggests a gradual process, such as:

1. Begin with simple, restricted environment models that cause the design to be **over-reduced**.
2. Verify the reduced design, fix errors in the environment models, correct wrong formulas, and clean coarse design errors.
3. When the reduced design is stable enough, refine the environment. This usually increases the effective size of the design.

This method is most efficient during the development of environment models and rules, since at this stage the process is iterative and the turnaround time must be short.

A main reason that this method works is that the model built by RuleBase for a design that contains bugs is usually much larger than it is for a cleaner design in which the state space is less well-behaved. Thus, even if we could not verify the first ‘buggy’ design for all legal input sequences, perhaps it can be done after some of the bugs have been removed.

The following example is taken from an architectural-level verification. Consider a multi-processor system in which a number of CPUs are attached to one or more control units. During initial debugging, only one CPU is hooked up to clean major bugs out of the design (and environment). Once one CPU works, another is hooked up, and so on.

8.7 Abstraction of Internal Parts

If some part of the design is too complex or memory-intensive, and if the internal logic of that part is not directly involved in the property to be verified, it can be replaced by an abstract model. In effect, the part will now be regarded as an environment.

The replacement can be easily done in RuleBase. Define an abstract model to replace the part. This model should drive all the signals driven by the original part (it can also use signals used by the part). RuleBase does the remainder of the work, linking the model to the design and getting rid of the original part. Figure 16 illustrates this method.

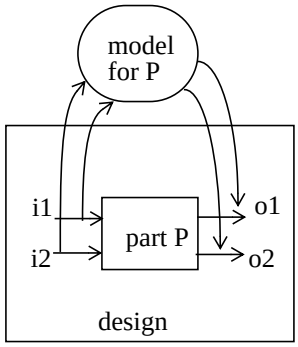


FIGURE 16. Abstraction of an internal part

For example, if a design includes an arbiter on which the rest of the design should work regardless of the exact arbitration algorithm, that arbiter can be replaced by an abstract one that only guarantees mutual exclusion. Such an abstract arbiter for N devices can be modeled using log(N)+1 bits.

8.8 BDD Ordering

RuleBase uses a data-structure called a Binary Decision Diagram (BDD) to represent the model. In a BDD, every state variable has a distinct level, from 1 to n, where n is the number of state variables. The order in which the levels are allocated to the state variables has a large impact on the size of the BDD. For example, a design whose verification with a good BDD requires 30 MB of memory, may require 300 MB or more with a bad order. Therefore, it is important to find a good order.

RuleBase can perform BDD reordering during model checking. This is known as dynamic BDD ordering. Because BDD ordering is extremely CPU-intensive, it is inactive by default. You should turn it on for initial runs, and feed the resulting order back into RuleBase for all consecutive runs.

Since reordering is time-consuming, it is good to reserve the final order for use in later runs of the same rule and even of other rules.

To do this, open the **BDD order** section of the **Options** dialog box. The **Copy Now** line has two fields.

- To copy the final order at the end of the run to the *<rulename>.order* file in which *<rulename>* is the name of the rule, select **to <rule>.order**.
- To use this order on the next run, set the **Use Order File** field to **<rule>.order**.
- To copy the order file back to a pool or orders that can be used by all rules, click **to orders pool**.
- To save the order for use by other rules using the same or similar reductions, set the **Use Order File** field to **orders pool** on subsequent runs.

To automate control over ordering, use the various fields in the **BDD order** section of the **Options** dialog box. See CHAPTER 10 Graphical User Interface: Tool Controls and Options for more details.

8.9 Verify-Safety-OnTheFly

In its normal mode of operation, RuleBase will compute the reachable state space before checking any formula (the reachable state space is the set of all states of the design that can be reached from the initial states). For a class of formulas known as *Safety* formulas, RuleBase can in many cases determine the falsity of the formula before it has completed the search of the reachable state space. This method is know as Verify-Safety-OnTheFly.

The Verify-Safety-OnTheFly method has several advantages:

- A counter example or witness is produced as soon as a state is found in which the formula does not hold true. Crude errors (that usually stem from incorrect formulas or environment models) are detected and displayed quickly.

- The iterative process of searching for the reachable states is often much more expensive (in terms of memory and time) for states located far from the initial state. For example, stopping after half the number of total iterations can sometimes save 90% of the total run time.
- It is not necessary to build a full Transition Relation (TR). In normal model checking, RuleBase builds a TR that represents all possible state transitions of the design. Since the TR is a bottleneck in large designs, you save a lot of time and energy since you don't have to build it.
- Design errors often increase the model built by RuleBase. Models of erroneous designs tend to grow because the reachable state-space may include many unexpected states. Finding and fixing errors in early iterations while the state space reached is still small may decrease design size and allow later runs to go farther.

If the **Verify-Safety-OnTheFly** option is enabled, RuleBase attempts to check as many formulas as possible during the search for the reachable state space. Formulas that cannot be verified in this mode will be identified automatically and checked with the normal algorithm. The formulas that **DO NOT** suit Verify-Safety-OnTheFly can be characterized as follows:

- Formulas that mix the **A** and **E** path quantifiers.
- Formulas that contain the temporal operators **AF**, **AU**, or **EG**, or any strong Sugar operator (an operator whose name ends with '!'); these are known as *liveness* formulas, rather than *safety* formulas.
- Formulas in which there is a '|' (the **or** operator) between temporal sub-formulas.
- Formulas in which a weak **until** has a temporal sub-formula on the right hand side.

The **Verify-Safety-OnTheFly** option will sometimes need to add auxiliary state variables. For this reason, the user can control the option. It is advisable to try this option and see if the additional state variables are a problem for RuleBase (because of size limitations). In most cases, this option can be a con-

siderable time and space saver. RuleBase will not add any state-variables for rules of the form **AG**(p), where p is a combinational formula.

A useful trick when using this technique is to “and” your formulas together into one big formula. The advantage of this technique is that the overhead for checking formulas on the fly is reduced considerably. RuleBase has an option to make this automatic and transparent to the user. To operate this option, add the following line to your rulebase.setup file:

```
setenv RB_BIG_AND 1
```

Using this option, you will “and” all safety formulas, but the results will be given as if they were run separately. The RB_BIG_AND option will only operate if all formulas in the rule are safety formulas; otherwise, it will run in the regular OnTheFly mode.

To access the **Verify-Safety-OnTheFly** option, press the **Options** push button and open the **Verification** section of the **Options** dialog box.

The **Verify Safety OnTheFly** option menu has two entries:

- **Yes.** All Safety formulas will be checked OnTheFly. You can specify a parameter (a two-digit integer number) that determines the trade-off between memory and time consumption during the run. The default for this parameter is 10. If a bigger number is specified, the run will consume less memory, but counter-example production will take longer. If a smaller number is specified, the run is likely to consume more memory, but producing a counter-example will be easier. Therefore, if many of the formulas are likely to fail, we recommend you specify a small number, and if most of them are likely to pass, a big number. A special case exists when specifying 0 as parameter, which is similar to specifying ‘infinite’: the run will consume as little memory as possible as long as no rule fails. However, if a counter-example is needed, it will consume more time and space.

- **No**. Verify-Safety-OnTheFly is disabled.

8.10 Using Real Memory Efficiently

Your memory quota is often much less than all of the real memory available in the system. The operating system may kill a running process when the quota is exceeded, although unused memory is still available. RuleBase users will usually want to override this limitation. In *cshell,* for example, the *limit* command can be used to control and display the user's quota. A possible setting for a computer with 256 MB real memory is*:*

```
limit datasize 230000
limit memoryuse 230000
```

To view current limits, use the following command:

```
limit
```

RuleBase is **very** slow when it runs out of real memory; therefore, it is not a good idea to increase data-size above the size of the real memory.

CHAPTER 9

Debugging Aids

9.1Overview

NEED INTRODUCTION TO THE CHAPTER

We will describe additional debugging aids in other chapters:

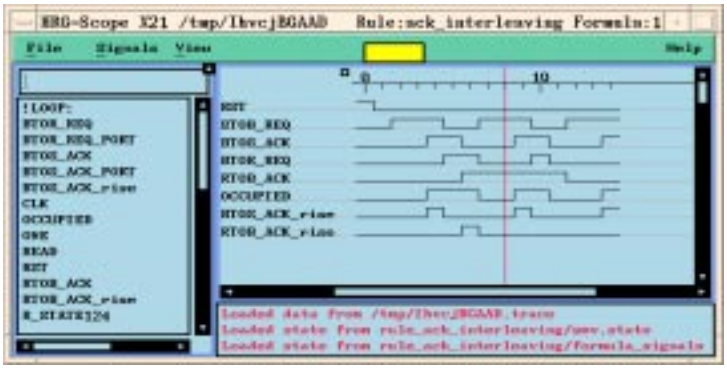
- Vacuity explanation – “Displaying Vacuity Explanation” on page 200
- Formula explanation
- Test generation
- Lists of variables and signals, before and after reduction

9.2 Scope Waveform Display Tool

You can display the counter-example or witness generated by RuleBase by invoking the Scope waveform display tool as described in Section 10.6.9. This section describes the Scope waveform display tool.

9.2.1 Main Window – Scope

The Scope main window is shown below. It consists of a number of areas:



- **Rule and formula display** – located on the top of the window frame. It displays the name of the rule and the formula number within the rule for which this trace was generated.
- **Menu bar –** located at the top of the window. It will be green if you have setup the default colors by copying the file Scope from $RBROOT to your home directory as described in CHAPTER 2: Getting Started.
- **Signal list** – the rectangular area on the left-hand side of the window.
- **Waveform display window** – the large rectangular area in which the waveform itself is displayed.
- **Message panel –** the small rectangular window below the waveform display window.

The following sections describe these areas in detail.

9.2.2 Menu Bar

The menu bar contains the following menu items:

9.2.2.1 File Menu Option

To open the sub-menu, click the **File** menu option. You will be presented with the following items:

- **Print screen** – prints a copy of the waveform display to the postscript file scope.ps.
  To print this file directly from Scope, add the following line to file Scope in your home directory: “Scope*printCommand: <your print command>”.
  For example “Scope*printCommand: qprt -Bnn -Pprt1 scope.ps”.
- **Load state** – prompts the user for a name of a Scope state file to replace the current state. (See Section 9.2.6, State Files for more information on the Scope current state and state files.)
  Create a state file using “Save state” or “Save state as” described below.
- **Append state** – prompts the user for a name of a Scope state file, whose signals will be appended to the currently displayed signals.
- **Save state** – saves the current Scope state, including the signals displayed, in the currently loaded state file. The saved state will be used the next time a waveform for this rule is displayed.(See Section 9.2.6, State Files for more information on the Scope current state and state files.)
- **Save state as** – prompts the user for the name of a Scope state file in which to save the state. This state can later be loaded using “Load state” described above.
- **Quit** – exits the Scope waveform display tool.

9.2.2.2 Signals Menu Option

To open the sub-menu, click the **Signals** menu option. You will be presented with the following items:

- **Add all** – adds all signals to the waveform display.
- **Remove all** – removes all signals from the waveform display.
- **Vertical text** – causes signals with text values (e.g., enumerated constants) to have a vertical display format.

- **Horizontal text** – causes signals with text values to have a horizontal display format.
- **Sort/Unsort** – users of RuleBase can ignore this option.

9.2.2.3 View Menu Option

To open the sub-menu, click the **View** menu option. You will be presented with the following items:

- **Zoom in** – zooms in on the waveform display.
- **Zoom out** – zooms out on the waveform display.
- **Show/Hide Toolbar** – users of RuleBase can ignore this option.

9.2.3 Signal List

The signal list contains a list of all signals in the design and environment that remained after reduction. If a signal does not currently appear in the waveform display, it can be added to the display by clicking it.

You can control the location of the display of a signal's waveform in the following manner.

1. To add signal 'a' above signal 'b' in the waveform (assuming that signal 'b' is already displayed), first mark signal 'b' by clicking its name (and NOT its waveform) in the waveform display window.
   Signal 'b' should now be marked by a rectangular box surrounding the signal name.
2. In the signal list, click the name of signal 'a'.
   The waveform of signal 'a' should now appear above that of signal 'b'.
3. To move the waveform of signal 'a', mark it as described above in step 1. Then drag and drop it into its new location.
4. To remove the waveform of a signal, right-click its name in the waveform display window.

5. To add signal ‘a’ to the end of the waveform display, unmark all signals by clicking the name of the currently marked signal (the easiest way to do this is to double-click the name of any signal, which first marks it and then unmarks it).
6. Click the name of signal ‘a’ in the signal list.
   The waveform of signal ‘a’ should now appear as the last signal in the waveform display.
7. To search for a signal name in the signal list, type its name (or part of its name) in the small window above the signal list, and press **Enter**. To quickly clear the search window, right-click anywhere in the search window.

9.2.4 Waveform Display Window

The waveform display window displays an execution trace that is a counter-example or a witness to a formula. The number bar at the top of the display counts the clock cycles of the fastest clock. Signals that have a textual display (e.g., enumerated constant values) only display a change in the signal value. If no value appears at time X, find the current value by looking to the left for the value at the last time the signal changed.

9.2.4.1 Displaying an Infinite Trace: !LOOP:

There are formulas whose counter-example or witness must be displayed as an infinite trace. For example, consider the following formula:

```
AG (p -> AF q)
```

If this formula is false, a valid counter-example is one in which ‘p’ is asserted, and then ‘q’ is never asserted. Never is an infinite amount of time, and thus an infinite trace is required to show that this formula is false. RuleBase displays an infinite trace by displaying a finite prefix, and then a set of states comprising a loop.

The special signal !LOOP:, which appears first in the signal list, is used to indicate the loop. A loop is indicated when the signal !LOOP: begins taking on the

values ‘=’ and ‘-’ alternately, so that the entire loop is marked by a string of the form “=-=-=-=-=-=”.

9.2.5 Message Panel

The message panel is used to display various errors, warnings, and informative messages.

9.2.6 State Files

The current state of Scope consists of various aspects of the waveform display, including the signals displayed, the zoom factor, and the window geometry. You can save the current state in a state file and use it in later sessions of Scope. To ease your work RuleBase performs basic management of Scope state files as follows:

1. You can choose if state files are saved in the rule’s directory, in which case different rules have different state files, or in the verification directory, in which case all rules share one state file. See “Per-rule state file” in Section 10.6.3.5.
2. When Scope is invoked, it loads the default state file, if one exists. If “Per-rule state file=No”, Scope loads smv.state from the verification directory. If “Per-rule state file=Yes”, Scope tries to load rule_<rulename>/smv.state. If this file doesn’t exist, Scope loads smv.state from the verification directory.
3. After loading the default state file, signals that appear in the formulas are appended.
4. You can save the current state in the current state file using the “File/Save state” menu option, or save it in another file using the “File/Save state as” menu option. (See “File Menu Option” on page 168.)
5. You can replace the current state by another state file using the “File/Load state” menu option. You can add Signals of another state file to the current state using the “File/Append state” menu option. (See “File Menu Option” on page 168.)

9.3 Vacuity

When a formula passes in a trivial manner it is called vacuity. If vacuity detection is enabled (see Section 10.6.3.4) the status of the rule as displayed in the results window (see Section 10.6.9) is ‘vacuously’. For instance, if the formula

```
AG (p -> AX q)
```

passes, but ‘p’ is never asserted, then the formula is said to pass vacuously. Vacuity occurs when a sub-formula does not affect the truth value of the formula. For instance, in the above formula, the sub-formula ‘AX q’ does not affect the truth value of the formula, because ‘p’ is never asserted. We can replace ‘AX q’ with any other sub-formula (even the sub-formula FALSE!), and the rule will remain true. Since a trivially true formula is not intentionally part of a specification, a vacuous pass usually indicates a problem in the rule, in the environment, or in the design under verification. For this reason, we strongly recommend that you do not turn off the vacuity checking option. If vacuity checking is enabled, but full witness generation (see Section 9.4, Witness) is turned off, minimal overhead is incurred.

In the above example, there is only one possible cause of vacuity. However, sometimes the situation is more complex. For instance, in the following formula:

```
AG (start_transaction -> next_event(acknowledge)(read_enable | write_enable))
```

the vacuity may be because ‘start_transaction’ is never asserted, or because ‘acknowledge’ is never asserted after ‘start_transaction’. In both cases, the sub-formula (read_enable | write_enable) does not affect the truth value of the formula.

To facilitate debugging of vacuous passes of this type, an explanation of the vacuous pass is available as described in “Displaying Vacuity Explanation” on page 200.

9.4 Witness

The knowledge that a formula passes only provides a measure of confidence in the correctness of the design under verification. One reason for lack of confidence is that the pass may be vacuous, as discussed above in Section 9.3, Vacuity. However, even if a formula passes non-vacuously, there is the possibility that the formula does not express the property that you intended. One way to achieve greater confidence that the formula does express your intentions is to examine a witness formula. A witness formula is a positive non-trivial example of the truth of the formula. A witness formula is created when full witness generation is enabled (see “Verification Control Panel” on page 194) and the rule passes non-vacuously. In this case, the status of the rule as displayed in the results window (see “Results” on page 198) is ‘passed (w)’. For instance, if the formula

```
AG (p -> AX q)
```

passes non-vacuously, then the witness trace will show a case in which ‘p’ is asserted and then the following clock ‘q’ is asserted.

We recommend that you enable witness generation at the beginning of the formal verification process, and you examine a witness trace at least once for every formula. Once a witness formula has been examined and the formula is determined to correctly express the desired property, you can turn off witness generation as described in “Verification Control Panel” on page 194. At this stage, model checking becomes a fully automated process in that it is enough to determine that each formula passes non-vacuously, without examining waveform displays for true formulas. We strongly recommend that you leave vacuity detection on at all times.

9.5 Reduction Analyzer

As explained in CHAPTER 8: Size Problems and Solutions, one way that RuleBase deals with the size problem is by behavioral partitioning and overreduction, in which parts of the design are eliminated based on the environment

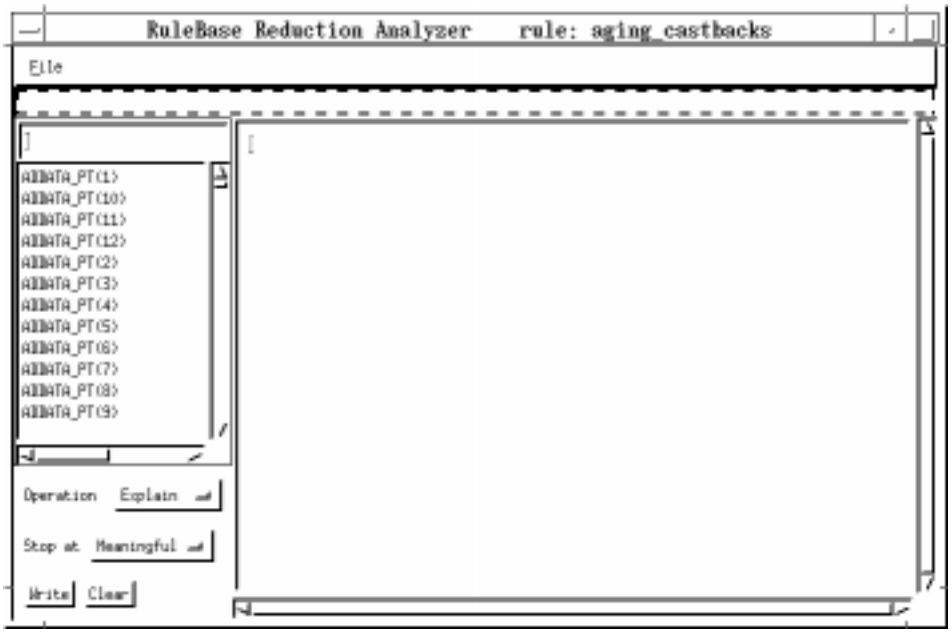
and the formulas to be checked. The reduction phase eliminates logic that is not in the cone of influence of the formulas to be checked, propagates constants from the environment forward, and eliminates redundant logic (that either was there from the start, or that became redundant because of constant propagation). The reduction analyzer allows insight into these reductions. Usually, there are two questions that the reduction analyzer can help answer:

- Why was signal X eliminated during reduction (why does it not affect the truth of the formulas in this rule)?
- Why wasn't signal Y eliminated during reduction (how does it affect the truth of the formulas in this rule)?

The reduction analyzer is invoked from the 'Debugging' menu option as described in "Debugging Menu Option" on page 188. The reduction analyzer can only be invoked if creation of a circuit file was enabled as described in "Debugging Control Panel" on page 196.

9.5.1 Main Window – Reduction Analyzer

The main window of the reduction analyzer is shown below.



It consists of the following areas:

- **Menu bar –** located at the top of the window. It will be light blue if you have setup the default colors by copying the file Analyze from $RBROOT to your home directory as described in CHAPTER 2: Getting Started.
- **Signal list** – the rectangular area on the left-hand side of the window.
- **Analysis display window** – the large rectangular area in which the wave-form itself is displayed.
- **Quick button menu** – the area below the signal list.

The following sections describe these areas in detail.

9.5.2 Menu Bar

The menu bar contains the following menu item:

9.5.2.1 File Menu Option

To open a sub-menu, click the **File** menu option.You will be presented with the following item:

- **Quit** – exits the reduction analyzer.

9.5.3 Signal List

The signal list contains a list of all signals in the design. To choose a signal, click the desired signal. To search for a signal name in the signal list, enter its name (or part of its name) in the small window above the signal list, and press **Enter**.

9.5.4 Analysis Display Window

The analysis display window is used to display the reduction analysis information.

9.5.5 Quick Button Menu

The quick button menu contains the buttons used to control the reduction analysis. The following sections describe each button in detail.

9.5.5.1 Operation

The operation quick button is used to select the reduction analysis operation to be performed. Choose a value, then click a signal from the signal list. The reduction analyzer performs the operation you requested. You control the depth of the analysis by the stop at quick button, described in the next section.

Possible values of the operation quick button include:

- **Explain** – asks the reduction analyzer to explain why a signal is dead, alive, or has a constant value. If a signal is dead (deleted by the reduction analyzer), the reason will be shown. If it is alive, its influence on one of the formula signals or on a test pin will be explained by showing a chain of influence from the selected signal through intermediate signals and finally to a formula signal or test pin. If a signal has a constant value, the derivation of that constant value from the environment through the design will be shown.
- **Cone** – asks the reduction analyzer to show the cone of influence of the selected signal. Some shortcuts may be taken. For instance, if signal xyz is defined as follows:
  define xyz := xx & yy & zz;
  and signal zz is a constant 0, then the cone of influence will not show logic beyond the first “and” gate.
- **Fullcone** – asks the reduction analyzer to show the cone of influence of the selected signal without shortcuts. For instance, in the above example, despite the fact that signal zz is a constant 0, the full cone of logic including that driving xx, yy, and zz will be shown.
- **Sources** – shows the signals that are inputs to the cone of logic that drive this signal.
- **Sinks** – shows signals that are driven by this signal.

9.5.5.2 Stop at

The stop at quick button controls the depth of the analysis of the operations described in the previous section. Possible values include:

- **Meaningful** – analysis is continued until a meaningful signal name is reached. A meaningful signal is one that was present in the original HDL code (i.e., was not added by the synthesis tool).
- **Flip-flops** – analysis is continued until a flip-flop (or latch) is reached.
- **Any** – analysis stops at any signal. In other words, analysis stops at the first logic gate that drives the signal.

9.5.5.3 Write

This button writes the current contents of the analysis display window to a file. You will be prompted for the file name.

9.5.5.4 Clear

This button clears the analysis display window.

9.6 Longest Trace

In model checking, the design and its environment are viewed as a finite state machine that is traversed. If reachability analysis is enabled, the first step of the verification is a breadth-first-search traversal of the state space of the model, starting from the initial state or states. The steps of this traversal are reported as “iterations” in the log file of the run. The set of initial states is iteration 0, the set of all states reachable in one step from some initial state is iteration 1, and so on. The last iteration includes all states that are as far from some initial state as possible. A path from some initial state to a state in the last iteration is called a “longest trace”.

A longest trace can be useful in gaining insight into the design under verification, as a trace to some state in the final iteration is in some sense one of the most complicated traces that can be generated. Frequently, examining a longest trace can uncover a bug in the design or in the environment.

You can control longest trace generation from the verification control panel as described in “Verification Control Panel” on page 194. You can view the longest trace using the debugging menu option as described in “Debugging Menu Option” on page 188.

9.7 Multiple Traces

You may sometimes want to see more than one counter-example for a formula and the more different the counter-examples are from another, the easier it is for you to debug the design.

RuleBase provides the multiple traces feature for this aid, which uses the Hamming distance heuristic to search different traces. The user asks RuleBase for the desired number of traces, and gets them (or less, if not enough traces exist).

You can enable this feature by adding "-multiple_traces n" flag to $SMV-FLAGS variable using File/Setenv menu as described in "File Menu Option" on page 186 (in which n is the number of traces that the user wants to see.)

Note: *Currently, you can see all the traces of a counter-example by pressing the Get Next Trace button in the new Scope, which is given by request and currently is not in the Rulebase package. The default Scope does not support this feature and only shows the first trace.*

9.8 Prolong Trace

Looking at the counter-example, which is given to the point that contradicts the formula, you may want to know "what happens next", (i.e., what are the values of signals on the following cycle, or even on the number of following cycles).

RuleBase makes it possible to know what happens next by providing the prolong trace feature. The user asks RuleBase to prolong the traces by n cycles, and each trace (if any) is prolonged by the given number of cycles. A warning is given if it is not possible, which occurs if the counter-example is finite and has no continuation of given length because of environmental constraints (invars, restricts, or assumes.)

You can enable this feature by adding the "prolong_trace n" flag to $SMV-FLAGS variable using File/Setenv menu as described in "File Menu Option" on page 186 (in which n is the number of cycles to be added to each trace.)

9.9 Reporting a RuleBase Bug to IBM

You may at some point believe that RuleBase itself has a bug. In this case, we ask that you report it to IBM. The report should include the files and directories that enable IBM to reproduce the bug, normally as a gzipped tar file. While it is possible to prepare, tar, and compress the appropriate files manually, the **utils** directory contains a Perl script named **bug_report.pl** that will do it automatically. When you run the script from the verification working directory, it will produce the following two files:

- A gzipped tar file, which contains all relevant files and directories.
- A template for the actual bug reporting.

  To run bug_report.pl, type 'bug_report.pl'.
  The script will guide you through the bug reporting preparation process.

Note: *We make the assmption that your Perl interpreter is installed as: /usr/local/bin/perl.*

9.10 Stand Alone Scope Utility

You usually get the Scope by choosing the result page of a specific rule from the RuleBase GUI, and then clicking the relevant formula. You can now launch the Scope as a standalone application from the command line, bypassing the GUI. In this case, the red dots do not appear.

To launch the Scope, issue the following from within the verification working directory:

mt_scp <rule-name> <formula-name> [trace-num]

The trace-num is set to default (1), which is the only valid option in which there are no multiple traces for the rule.

The scope should show up promptly; if it does not, the parameters are probably wrong.

9.11 RuleBase to VCD Converter

The **utils** directory contains the l**og2vcd.py** utility that enables you to convert a RuleBase log file into the **VCD** format. This utiilty is written in Python, and it requires you to have a Python 1.5.2 or higher interpreter installed on your system.

Usage instructions

1. To set the environment variable to PF_SCOPE, add the line 'setenv PF_SCOPE 1 ' to your rulebase.setup.

2. Run the rule.
3. Run the script from the verification directory:
   python $RBROOT/log2vcd/log2vcd.py

   --entity=<top-level entity>

   --vcd=<vcd file>

   --rule=<rule name>

   [--formula=<formula number>]

   [--trace=<trace number>]

Note: *All of the options must appear on the command line. They are separated for readability.*

- **<top-level entity>** – the name of the top-level entity of the design, which is the value of the 'entity' variable in rulebase.setup. You MUST specify this option.
- **<vcd file>** – the name of the output VCD file. You MUST specify this option.

- **<rule name>** – the name of the rule. You MUST specify this option.
- **<formula number>** – the formula whose trace you would like to convert to VCD. The default value is formula number 1.
- **<trace number>** – used in case there are multiple traces. The default value is trace number 1.

For example, say you have just run the rule XXX on a design whose top-level entity is named ZZZ and you would like to generate a VCD file for formula 1 to a file named xxx.vcd. Type the following command in your verification directory.

```
python $RBROOT/log2vcd/log2vcd.py --entity=ZZZ --vcd=xxx.vcd
      --rule=XXX --formula=1
```

Problems and limitations:

The following are the problems you may run encounter when setting the PF_SCOPE:

- Setting PF_SCOPE causes RuleBase to generate a file called PF_SIGNALS. It may take some time to generate (usually no more than a few seconds).
- Setting PF_SCOPE will cause the default Scope to be the PathFinder Scope (when you click the rule results in the RuleBase GUI). To prevent this from occuring, either remove this line from rulebase.setup and rerun the RuleBase GUI or use File/Setenv from the GUI to set the variable PF_SCOPE to 0.
- The format of the resulting VCD file may not be compatible with your Scope as the IEEE standard is not well defined.

9.12 Scope Resource File

The file Scope, located in the user home directory, is the resource file of the RuleBase Scope. Fonts, for example, are controlled from this file.

Each time a new user starts using RuleBase, this file is copied from $RBROOT to the user's home directory.

9.13 Additional Debugging Aids

Other debugging aids are available under the "Debugging" menu of the RuleBase main window. They are described in detail in "Debugging Menu Option" on page 188.

CHAPTER 10

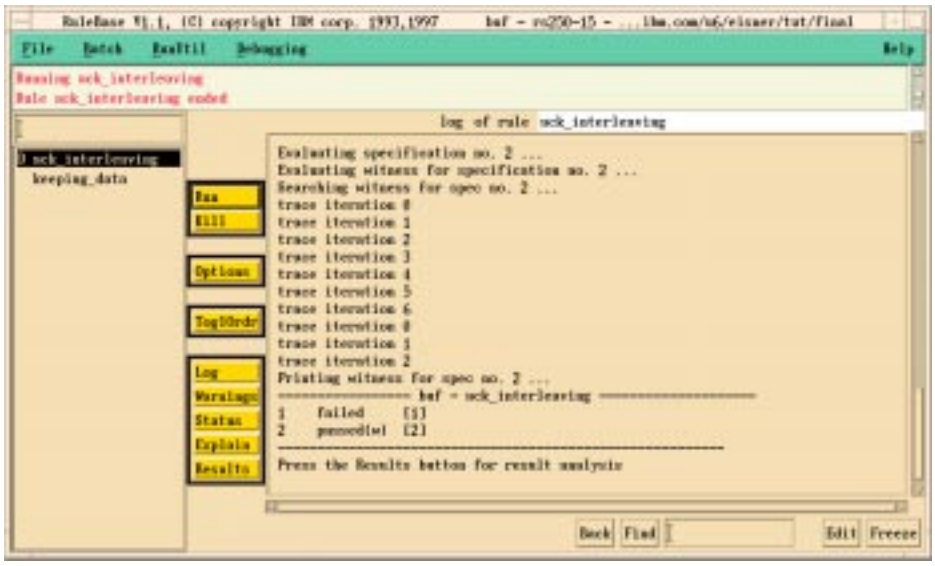
Graphical User Interface: Tool Controls and Options

10.1Introduction

This chapter describes both the graphical user interface of RuleBase, and the tool controls and options that it manipulates.

10.2 Main Window – Rule Base

The RuleBase main window is shown below. It consists of a number of areas:



- **Menu bar –** located at the top of the window. It will be green if you have setup the default colors by copying the file Guirb from $RBROOT to your home directory as described in CHAPTER 2: Getting Started.
- **Message panel** – located below the main control panel. It is off-white.
- **Rule list** – the rectangular area on the left-hand side of the window.
- **Quick buttons** – the dark yellow buttons to the right of the rule list.
- **Main text window** – the large rectangular area to the right of the quick buttons.
- **Text control panel** – located below the main text window.

The following sections describe each of the areas in detail. The most frequently used area of the RuleBase main window is the area that contains the quick buttons. If you are reading this document for the first time, we recommend that you skip to “Quick Buttons” on page 190 for a description of the quick buttons before reading the remainder of this chapter.

10.3 Menu Bar

The menu bar contains the following menu items:

10.3.1 File Menu Option

To open a sub-menu, click the **File** menu option. You will be presented with the following items:

- **reFresh** – updates the list of rules in the rule list. This is necessary if, for instance, you have added new rules since invoking RuleBase. There is no need to refresh if the changes you have made do not affect the rule list. Other changes will be seen automatically upon the next run of a rule. The one exception to this is the file rulebase.setup. As described in CHAPTER 2: Getting Started, the file rulebase.setup is read once upon invocation of RuleBase. Therefore, any changes to this file will not be seen automatically, nor will they be seen after choosing reFresh. To see changes to the file rulebase.setup, you must exit RuleBase and reinvoke.
- **Cleanup** – removes and/or compresses log and other files from previous runs.
- **Status of all rules** – creates a summary file that shows the status of each rule (formulas passed/failed on last run, run time, etc.).
- **Past status of rule** – displays the status of previous runs of this rule in the main text window, such as: formulas passed/failed, run time, etc.
- **Setenv –** allows you to set environment variables.You will be prompted for the name and value of the variable to set.
- **Read rulebase.setup –** reads the current rulebase.setup file. You can use it to update environment variables, instead of using the **Setenv** option described above. No ‘unsetenv’ is done, thus if a ‘setenv’ line was erased from rulebase.setup file, reading it will not change the environment variable.
- **Quit** – exits RuleBase.

10.3.2 Batch Menu Option

To open a sub-menu, click the **Batch** menu option. You will be presented with the following items:

- **Start** – starts a batch file. You will be prompted for the name of the batch file to be run.
  To run a batch file from the unix shell
  1. Create the batch file described below under “create batch file”.
  2. Copy the file $RBROOT/run to the current directory.
  3. Select options for the batch run, and save them as described in “File Menu” on page 191.
  4. From the shell, invoke the batch file.
- **Kill** – kills the currently running batch process.
- **Start at** – schedules a delayed start of a batch file. You will be prompted for a start time and the name of the batch file to be run.
- **Kill all at** – schedules the kill of all processes (batch or otherwise) of this window at a later time. You will be prompted for the time at which to kill all running processes.
- **Create batch file** – creates a batch file for later use. The batch file will contain all rules, and can be edited by an external editor if only a subset of the rules are desired. You will be prompted for the batch file name.
- **Failed batch file** – same as create batch file, but only rules in which at least one formula failed on a previous run will be included.
- **Aborted batch file** – same as create batch file, but only rules that did not complete the previous run will be included.

10.3.3 RunUtil Menu Option

To open a sub-menu, click the **RunUntil** menu option. You will be presented with the following items:

- **Pause/Continue** – freezes a running rule. Choose it again to continue the run. A paused rule has a **P** to the left of its name.

- **Undo run** – undoes the effects of a rule that was run by mistake. Works only if this rule has been killed, and no other rule has been run since this rule was started.
- **Adopt** – allows the run to be controlled by the RuleBase window from which adopt is performed. Usually if a rule was run from a unix shell or from another RuleBase window, it cannot be controlled by the quick buttons or the Options dialog.
  Note: *This will only work if the run that is to be adopted is on the same machine as the RuleBase window that wants to adopt it.*
- **Unlock** – forces lock deletion.
  During execution, RuleBase locks the rule to prevent multiple simultaneous executions. Sometimes, a run may abort without removing the lock. Choosing unlock forces lock deletion. Only unlock a rule if you are sure that it is not running (on any computer).

10.3.4 Debugging Menu Option

To open a sub-menu, click the **Debugging** menu option. You will be presented with the following items:

- **State variables** – displays the names of the state variables for this rule (valid after reduction).
- **Signals before reduction** – displays names of all signals in the design. In some translation paths some of the internal names disappear and others are added.
- **Signals after reduction** – displays names of signals after reduction that were categorized as Deleted/Constant/Alive. Signals that appear in “Signals before reduction” and do not appear here are not in the cone of influence of the formula. The information to the right of the “--” can be ignored; it is used by the reduction analyzer.
- **Circuit before reduction** – this option is currently not documented.
- **Circuit after reduction** – this option is currently not documented.

- **Reduction analyzer** – invokes the reduction analyzer, which is useful for debugging reductions performed by RuleBase. For more information, see CHAPTER 9: Debugging Aids. The reduction analyzer can only be invoked if creation of a circuit file was enabled as described in “Debugging Control Panel” on page 196.
- **Where signal is used** – displays the locations (file name and line number) in which a signal whose name matches a given pattern (wildcard is possible) is used in the environment by the selected rule.
- **Where signal is driven** – displays the locations (file name and line number) in which a signal whose name matches a given pattern (wildcard is possible) is driven in the environment by the selected rule.
- **Show longest trace** – presents a timing diagram generated by the verification option “Gen longest trace”. See “Options” on page 190.
- **Save longest trace** – similar to “Show longest trace” above, but instead of being displayed, the timing diagram is stored in the “longest.trace” file for inspection by the stand-alone scope tool.
- **GenTest from longest trace** – similar to “Show longest trace” above, but instead of being displayed, the timing diagram is stored as a control program for simulation.

10.3.5 Help Button

The **Help** option opens the on-line help documentation.

10.4 Message Panel

The message panel is used to display warnings, errors, and informative messages.

10.5 Rule List

The rule list contains the list of rules coded by the user. It is derived from the database file (usually called “envs”) pointed to by the rulebase.setup file.

10.6 Quick Buttons

The quick buttons are the most frequently used buttons during verification by RuleBase. The following sections describe each in detail.

10.6.1 Run

To run the currently selected rule, click the rule name in the rules list. A running rule has an **R** to the left of its name. The **R** becomes **D** when the rule ends.

10.6.2 Kill

To kill the run of the currently selected rule, click the rule name in the rules list. A killed rule has a **K** to the left of its name.

10.6.3 Options

This button opens the options box. The options box consists of the following areas, each of which is described in more detail in the sections below:

- **File** – allows you to store the options in a file, or load them from a file.
- **BDD order** – opens the BDD order control panel.
- **Reduction** – opens the reduction control panel.
- **Verification** – opens the verification control panel.
- **Debugging** – opens the debugging control panel.
- **Hide** – closes the options box.

Note: *Many of the option buttons have a yellow "**A**" (apply) button next to them. When you change an option during a run, you must click the corresponding apply button to see the change.*

10.6.3.1 File Menu

The file menu consists of the following options:

- **Load –** loads a previously saved options configuration.
- **Load <rule>.cfg**: **–** loads a previously saved per-rule options configuration.
- **Save –** saves the current options configuration for use by all rules.
- **Save <rule>.cfg –** saves the current options configuration for this rule only.
- **Default –** loads the default options configuration.

10.6.3.2 BDD Order Control Panel

The BDD order control panel consists of a number of fields, including:

- **Reorder**: Enable/disable reordering.
  When enabled, reordering will only begin when all the conditions below are fulfilled.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Algorithm**: Dynamic reordering algorithm.
  Options are one or a combination of the following:
  - **Cheetah** – quickest algorithm, but in many cases achieves the poorest results. Use this algorithm in combination with another to achieve the best combination of run time with results.
  - **Quick rudell** – slower than Cheetah, but may result in a better order.
  - **Rudell** – slowest algorithm, but frequently gives the best results.
    Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Iteration**: Two integers that are lower and upper bounds on the iterations between which the reordering algorithm should be active.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*

- **BDD size**: Two integers that are the lower and upper bounds of BDD size between which the reordering algorithm should be active. Reordering will be activated when the "nodes allocated" displayed in the log file has reached the lower bound, and has not yet exceeded the upper bound.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Low threshold**: An integer. If a low threshold is defined, a BDD ordering round is stopped when the BDD size falls below the low threshold.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Laziness factor**: A real number greater than 1 that controls the effort expended in reordering. The default is 3. If the movement of a variable requires more effort than the Laziness Factor permits, this variable will not be moved further, and RuleBase will prefix its name with a dot.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Growth factor**: A real number greater than 1 that controls the start of a second reordering round. A new round will only start when the BDD size reaches last-size * growth-factor, in which last-size is the BDD size (nodes allocated) at the end of the previous round. The default is 2.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Use order file**: RuleBase may use an existing order file as its initial order. The order file options include:
  - **Orders pool** – the best match in the orders pool is used.
  - **<rule>.order** – if such a file exists, it is used.
- **Copy back after run**: After every round of dynamic ordering, the order is written to a file called temp.ord located in the rule directory. This file may be used in later runs as the initial order. The options include:
  - **No** – do not save the new order.
  - **To <rule>.order** – at the end of the run, copy the new order to **<rule>.order**.

- **To orders pool** – at the end of the run, copy the new order to the **orders pool**.
- **To both** – at the end of the run, copy the new order to both the **<rule>.order** and to the **orders pool**.

- **Copy now**: Before a run has completed, a new order file may exist. This button allows the new order to be copied back immediately to either **<rule>.order** or to the **orders pool**.

10.6.3.3 Reduction Control Panel

The reduction control panel consists of the following fields:

- **Reduction effort**: Determines how much effort (CPU time and memory) are dedicated to performing the reduction. The options include:
  - **Low** – no BDD reductions are performed.
  - **High** – BDD reductions are performed. Applying all BDD reductions may require a lot of time and space. To control the time and space used by the reductions, try disabling one or more of the heavy reductions, as shown below, or give a BDD node limit:
    - Heavy reduction 1 NO
    - Heavy reduction 2 NO
    - **BDD node limit**: By limiting the number of BDD nodes and using high effort with heavy reductions, you will frequently get better reduction results than with no heavy reductions and no BDD limit. A typical number for this limit is 300000. Leave the limit unspecified if there are no reduction problems.

  For further insight into the reductions performed, see “Reduction Analyzer” on page 173.
- **SMV reductions**: When this mode is active, RuleBase performs over-approximations in order to find constants FF’s, and to apply reductions (based on the constants-search results). This mode is inactive by default and

should be activated when a size problem is encountered. Constants found (in this mode) are saved in the FF pool so it is usually necessary to activate this mode if no new constants are to be added.

10.6.3.4 Verification Control Panel

The verification control panel consists of the following fields:

- **Reachability** – determines if a search of the reachable state space of the circuit is to be done as the first step of verification. For most designs, this should be set to "Yes".
- **Verify Safety OnTheFly** – determines whether safety formulas (formulas that do not contain strong operators or the AF operator) should be checked during reachability analysis. Safety formulas can only be checked on the fly if reachability is enabled. The options include:
  - **Yes**: Check all safety formulas on the fly. You can give a parameter (an integer) that determines the trade-off between memory and time consumption during the run. If the user gives this parameter a small value, the run may consume more memory, but will produce counter-examples faster than a parameter with a large value.
  - **No**: Do not check Safety OnTheFly.

  For more information about Safety OnTheFly, see "Verify-Safety-OnTheFly" on page 161.
- **Verify Liveness OnTheFly –** determines whether the liveness formulas (formulas that contain AF, strong until, or sugar operators ending with!) should be checked during reachability. Unlike Safety OnTheFly, the check may take a long time. Therefore, if the liveness formulas are likely to pass, we recommend that you do not use use this option. The options include:
  - **Yes:** Check all liveness formulas every n iterations, where n should be specified by the user.
  - **No:** Do not check liveness on the fly.
- **Attempt light proof** – applies the classical model checking algorithm on the rule (all possible states are considered and not necessarily only the reachable ones)**.** The square area along the button specifies the time (in sec-

onds) that RuleBase should invest in this mode. If the formula does not fail/pass in this period of time, RuleBase will apply regular model checking on the rule.

Notes:

- If the formula passed (in light proof), RuleBase provides a witness and vacuity explanation (if asked for). Producing a witness or vacuity explanation may take more time than the other evaluation modes.
- If the formula failed (in light proof), the time needed to produce a counter-example will not be restricted.
- A light proof does not check vacuity. A vacuous formula be given the status 'pass'.
- The longest trace cannot be produced in this mode.

- **Witness** – controls whether vacuity (a trivial pass) is checked, and whether or not a witness trace is produced. For further information on vacuity and witnesses, see CHAPTER 9: Debugging Aids. The options include:
  - **Full witness** – checks formula for vacuity, and if the pass is non-vacuous, produces a witness trace.
  - **Vacuity only** – checks the formula for vacuity, but if the pass is non-vacuous, does not produce a trace.
  - **None** – does not check the formula for vacuity.
    Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Explain vacuous** – if active and a formula is found to pass vacuously, RuleBase will point to the pre-condition that cannot hold.
- **Gen longest trace** – if active, and reachability is also active, a trace will be produced to a state that is as far from the initial state as possible. If the "**Now**" button is pushed, and reachability is also active, RuleBase will complete the analysis of the current iteration, and then generate a trace that is furthest from the initial state. It will then continue with reachability analysis and check the remaining formulas.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*

- **Stop after iteration** – an integer that indicates how far RuleBase should go into the reachability analysis. RuleBase will stop reachability after the specified number of iterations (i.e., it will not look at states that are farther than X steps from the initial state). If the "**Now**" button is pushed, and reachability is also active, RuleBase will complete the analysis of the current iteration, and only check remaining formulas if they pertain to the states it has not yet seen.
  Note: *If you change this option during the run of a rule, you must click the yellow "**A**" (apply) button to see the change.*
- **Run only if changed –** provides the ability to run a rule only if the design, environment, or formula have been changed since the last run. Furthermore, if the change in the design or environment does not affect the formula, then it also will not run. When this mode is active, it creates a signature for each run to be compared with the next run.

  The following options are available:

  - **No** – mode is inactive.
  - **Yes –** mode is active. A rule will run only in the case of changes.
  - **Force –** mode is active. A rule will run in any case and a signature will be created.
  - **Fictitious –** RuleBase creates a signature for the rule as it was just running. If the previous log file is older than the design, does not exist, or is not completed (rule did not finish), RuleBase will not create a fictitious signature.

10.6.3.5 Debugging Control Panel

The debugging control panel consists of the following fields:

- **Explain timing diagram –** determines if explanations of the counter-example will be shown in the trace. Explanations are red dots that show you where to look for interesting events.
- **Show formula text** – determines if the counter-example or witness trace will also open a window that displays the formula to which it corresponds.

- **Per-rule state file** – determines if the signal display configuration of the trace will be saved per-rule or not.
- **Clock cycle** – length of the clock cycle for test generation[1] (see “Results” on page 198).
- **Clock name** – name of the clock for test generation.
- **Signal prefix** – prefix for signal names for test generation.
- **Create circuit file** – a circuit file must be created during reduction if the reduction analyzer (see “Reduction Analyzer” on page 173) is to be used.

10.6.4 ToglOrdr

This button will toggle BDD ordering for the currently selected rule.

- If BDD ordering is currently taking place, it will stop the current round.
- If BDD ordering is not currently taking place, it will start one round.

This button only affects one round of BDD ordering. To turn BDD ordering on or off permanently, see section “BDD Order Control Panel” on page 191.

10.6.5 Log

This button will display the log file of the currently selected rule. If the rule is currently running, this option will only work if invoked from the machine on which the rule is currently running. If the rule has completed, the log file of the completed run will be displayed. The following two sub-buttons are provided to ease the user’s analysis of the log:

- **n –** deletes all of the ‘nodes allocated’ lines from the log.
- **> –** deletes all BDD ordering lines.

1. Test generation is not usually needed, because RuleBase generates a simulation-like trace for debugging. This option is only needed in the case that the user wants to generate a simulation test out of the counter-example generated by RuleBase.

10.6.6 Warnings

This button displays any warnings of the currently selected rule.

10.6.7 Status

This button displays the status of the currently selected rule. If the rule is currently running, it will display the start time of the run and the name of the machine on which the rule is running. If the rule has completed, it will also display the results (pass/fail) of each formula and the CPU time and memory usage.

10.6.8 Explain

This button displays an explanation of the formulas of the currently selected rule. The explanation is a rudimentary translation of the formula into English.

10.6.9 Results

This button displays the results of the currently selected rule. It displays each formula, along with information on its status. The status of a formula can be one of the following outcomes:

- **failed –** formula is false, and a counter-example has been produced.
- **failed(c) –** formula is false, and is a contradiction in the model.
- **passed(w) –** formula is true non-vacuously, and a witness trace has been produced.
- **passed(nv) –** formula is true non-vacuously.
- **passed(ta) –** formula is true, and is a tautology in the mode, which means that RuleBase could combinatorically determine that the rule passed, without the need to search for all the reachable states of the model.
- **passed –** formula is true, but vacuity has not been checked.[1]
- **vacuously –** formula is true vacuously.

- **unknown –** formula has not yet been determined to be true or false

For an explanation of vacuity, see CHAPTER 9: Debugging Aids.

At the beginning of the run, the status of all formulas is unknown. If you choose Safety OnTheFly (see “Verification Control Panel” on page 194), some formulas may be determined to be true or false before the completion of the run. It is, therefore, possible to click on the **Results** quick button before a run has completed and see that some formulas have either passed or failed, while the status of others is still status.

10.6.9.1 Displaying Counter-examples and Witnesses

If the status of a formulas as described above is either “failed” or “passed(w)”, a trace is available for viewing. In the case of “failed”, this trace is called a counter-example. In the case of “passed(w)”, it is called a witness.

Click near the word “failed” or “passed(w)” and hold the mouse button down. A menu will be displayed with the following options:

- **Show timing diagram**: If chosen, the waveform display tool Scope will display the counter-example or witness. For an explanation of the Scope tool, see “Scope Waveform Display Tool” on page 166.
- **Save timing diagram**: If chosen, the counter-example or witness will be saved in a file named rule_name.N.trace, in which rule_name is the name of the rule and N is the formula number in the rule. You can view this trace by using the following command:

  ```
  $RBROOT/scope  -sf smv.state  rule_name.N
  ```

NOTE: *Even after exiting RuleBase, you can access the trace by re-invoking RuleBase, clicking* ***Results****, and selecting* ***show timing diagram*** *as above. You only need to save using the* ***save timing diagram*** *option if you want to keep a copy of the trace independently of RuleBase. For instance, you can save a copy*

1. We strongly recommend that you check vacuity.

of a failing trace to send it to a colleague by e-mail, or to keep in a database for documentation purposes.

- **Generate test** – generates a test for simulation that will produce the same trace shown in the counter-example or witness. The following formats are available: Synopsys and Cadence Verilog XL. The default format is Synopsys.
  To generate a test for Cadence Verilog XL, add the following line to your rulebase.setup file:
  setenv RB_TEST_VERILOG 1
  NOTE: *Normally, there is no reason to generate a test from a counter-example or witness, because the counter-example or witness itself can be used for debugging. In addition, the generated test will only check the specific failure that was found, whereas running the rule again will check all possible failures that violate the rule. In other words, as a regression check, re-running the rule under RuleBase provides much better coverage than re-running the simulation test generated from a previous fail.*
- **Propagate values** – sometimes, when analyzing counter-examples, it is desired to see the value of design signals that were removed by the reduction. If every input that drives these signals is available, RuleBase can add them to the generated trace after the fact.
  To do this, create a file called “propagate.names” in the directory from which you have invoked RuleBase. The file should contain a list of signals, one to a line, which you would like to add to your trace. You can use wildcards; for example ‘*’ stands for ‘all signals’, and ‘block2/*’ stands for all signals whose names begin with ‘block2/’. Then, click **Propagate values** and wait until RuleBase computes the values. Finally, choose **Show timing diagram**. The signals you requested should now be available in the menu of signal names on the right hand side of the Scope tool.

10.6.9.2 Displaying Vacuity Explanation

If the status of a formula is “vacuously” and the explain vacuity facility was enabled (see “Verification Control Panel” on page 194), an explanation of the vacuous pass is available. Click near the word “vacuously”. Choose Explain

vacuity (the only available option). RuleBase will display an explanation. **The vacuity explanation is the shortest prefix of the formula that is always true**. For a detailed explanation of vacuity, see “Vacuity” on page 172.

10.7 Text Control Panel

The text control panel contains buttons that control the display of text in the main text window. Each control button is described in detail below.

10.7.1 BackText

This button performs a backward search for the text typed in the text search window, located to the right of the Find Text control button. It does not support a wild-card search.

10.7.2 Find Text

This button performs a forward search for the text typed in the text search window, located to the right of the Find Text control button. It does not support a wild-card search.

10.7.3 Edit Text

This button opens a window in which an editor is invoked on the current file displayed in the main text window. The default editor is vi. To call your preferred editor, add the following line to file rulebase.setup:

“setenv RB_EDITOR *your-editor*”

Examples:

“setenv RB_EDITOR emacs” # rulebase calls “emacs *file* &”
“setenv RB_EDITOR aixterm -e vi -R” # rulebase calls “aixterm -e vi -R *file* &”

10.7.4 FreezeText

This button freezes the main text window (by default it is updated continuously as the run progresses), making it easier to read the text. When clicked, this button will change its color to red and display the message **Frozen**. To unfreeze, click it again.

The main text window will be automatically frozen, when using the scroll bar, located on the right, to scroll backwards. To unfreeze it, click the red **Frozen** button.

10.7.5 GUI Resource File

The file Guirb, located in the user home directory, is the resource file of the RuleBase GUI. Fonts, for example, are controlled from this file.

Each time a new user starts using RuleBase, this file is copied from $RBROOT to the user's home directory.

CHAPTER 11

Design for Formal Verification

11.1Introduction

RuleBase supports a wide variety of design styles and methodologies. While in many cases you are not required to make special adjustments to existing design methodology, the following design guidelines will further ease the verification process.

11.2 Separating Control from Data

Although RuleBase can check both control logic and datapath, it is more effective for verifying control logic. Datapaths usually have many memory elements, which may increase the size of the internal model representation beyond the capacity of RuleBase. When verifying a design that includes both control and datapaths, the datapath is often replaced by an abstract model with fewer memory elements. This abstraction is easier when there is a clear separation between control and data in the design.

11.3 Design Partitioning

Design partitioning, in which each partition is verified separately, is one solution to the size limitation inherent to formal verification. Another reason for partitioning is the desire to push asynchronous signals and tri-state buffers to block boundaries (explained below). In many cases, the natural partitioning defined by the designers can serve as a basis for formal verification. In cases in which design partitioning is too fine, several blocks are often combined to form a bigger partition, which is more interesting in terms of verification.

Partitioning has several consequences:

- The effort exerted in defining, documenting, and studying the internal interfaces.
- The development of environment models for neighbor partitions.
- The tendency for changes in internal interfaces.
- The lack of expression for some global rules when the design is partitioned.

In light of this, our recommendations for partitioning are:

- Use the same partitioning for design and formal verification.
- Use documented or easy-to-understand interfaces.
- Use interfaces with stable protocols.
- Verify groups of related blocks, if the basic design blocks are small.

Experience shows that blocks that have several hundred flip-flops of control logic are good candidates for formal verification.

11.4 Clocking Schemes

While RuleBase supports many clocking schemes, the preferred scheme is in which each partition to be verified uses one clock. Multiple clocks, particularly if they are not synchronized, increase the size of the internal model's represen-

tation, and are not recommended for large partitions. When using multiple clocks, a small frequency ratio is preferred.

11.5 Design Mapping

RuleBase supports several languages and synthesis paths. The existing design environment (synthesis) tools are usually used for translation into gate-level representation. The design should be written in such a way that it will not prevent the translators from mapping it into a basic library of gates and flip-flops. For example, we do not recommend that you include switch-level macros in the design; you should use their logic-level equivalents.

Edge-triggered latches, and master/slave flip-flops whose master's output is only connected to the slave, are most suitable for RuleBase. If level-triggered latches are used, or if the master's output drives logic, special modeling is required, depending on how they are used in the design.

The following memory elements are supported in the synthesized netlist: flip-flops without asynchronous reset, flip-flops with asynchronous reset and transparent latches.

- Flip-flops without asynchronous reset will have the following behavior:

```
var q: boolean;
assign next(q) :=
    case
      clock: data;
      else: q;
    esac;
```

except when rb_e_t_ff option is used, in which case they will have the following behavior:

```
var q: boolean;
assign next(q) :=
    case
      next(clock) & !clock: data;
```

```
    else: q;
  esac;
```

See Table 8 on page 241 for a complete explanation of the rb_e_t_ff option.

- Flip-flops with asynchronous reset will be modeled as:

```
define q := !reset & q1;
var q1: boolean;
assign next(q1) :=
    case
      reset: 0;
      clock: data;
      else: q1;
    esac;
```

- Flip-flops with asynchronous set are also supported and are handled similarly.

- Transparent latches are modeled as:

```
define q :=
    case
      clock: data;
      else: prev(q);
    esac;
```

11.6 Asynchronous Logic

Ideally, there should be no asynchronous logic in the parts to be formally verified. RuleBase supports the verification of models in which the changes are at the cycle level. Asynchronous signals, if present, are best handled when situated at the verified partition boundaries. Synchronizing elements should be replaced by a short-circuit. State machines should be synchronized by proper

hand-shaking. RuleBase does not support relying on absolute durations (e.g., 40 nano-seconds before response).

11.7 Tri-State Buffers

Ideally, tri-state buffers should be located in a separate module so they can be easily separated from the design before formal verification. This is a common design style; however, in some designs, for various reasons, tri-state buffers are mixed with other logic. In these cases, they should be situated at the partition boundaries. Future versions will be able to handle tri-state buffers everywhere.

11.8 Parametric Designs

When the design includes memory arrays that highly influence the logic (e.g., FIFOs), we may want to verify these arrays rather than their abstract model. However, checking them as a whole may cause the model to become too big. To solve the size problem, you can define the array size as a parameter that can easily be changed. In this manner, you can choose the largest size possible within the RuleBase capacity.

11.9 Implementation Rules

Properties to be verified can be divided into two categories: specification rules and implementation rules. While specification rules are usually extracted from written documents, implementation rules are often not documented. We strongly recommended that you write these rules while developing the design.

CHAPTER 12 *Coverage Methodology*

12.1Overview

While RuleBase addresses the coverage problem of verification by simulation, it does not solve it completely. There is the question, “Have I coded all necessary rules? In addition, due to the size problem, behavioral partitioning, as described in CHAPTER 8: Size Problems and Solutions, is frequently used, which adds the following question to the coverage problem, “Have I coded enough environments?” We discuss these two questions in the following sections.

Before proceeding, we would like to emphasize that despite the fact that the second coverage question sounds very similar to the coverage problem of simulation, it does not mean that using RuleBase with behavioral partitioning is comparable to verification by simulation. Despite the coverage problem, verification with RuleBase can still provide much greater coverage than verification by simulation. For example, think of the set of all possible execution paths as inhabiting a two-dimensional space. Then, a test suite covers a finite number of points of the test space, as shown in Figure 17 below.

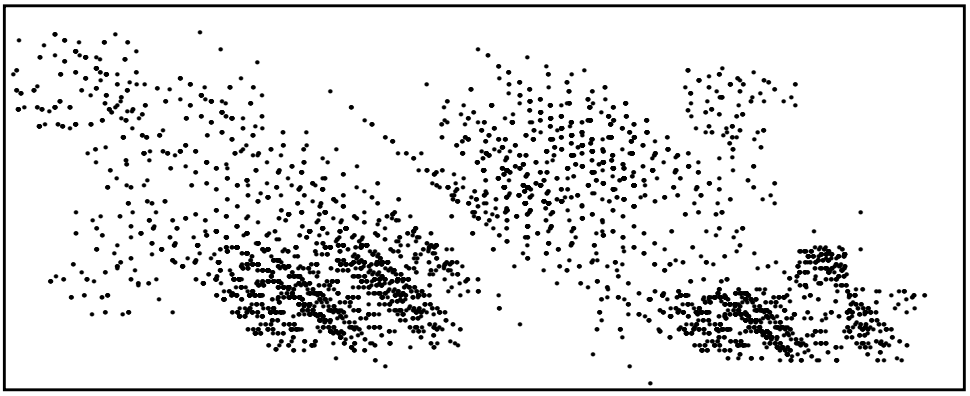


FIGURE 17. Coverage problem in simulation

With RuleBase, on the other hand, each environment covers an infinite number of points in the test space, as depicted below in Figure 18 .

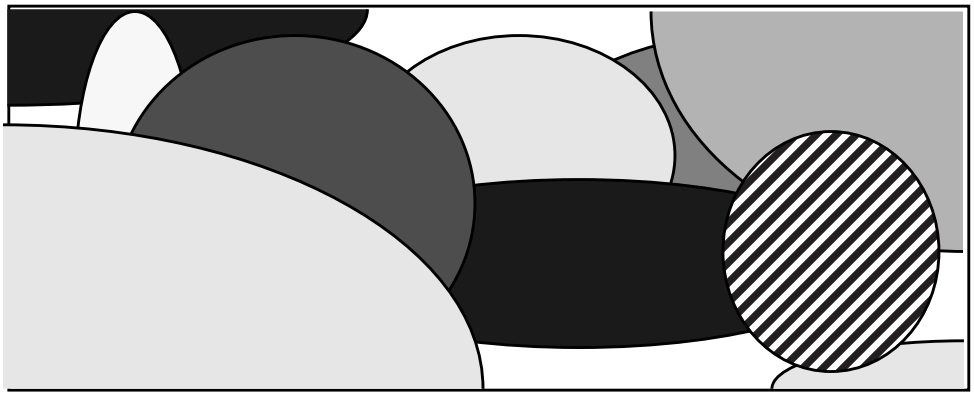


FIGURE 18. Coverage problem with RuleBase

While a complete solution to the coverage problem does not yet exist, we describe a methodology of rule and environment writing in this chapter that can help.

Note: *This chapter is brought here in a preliminary form.*

12.2 Coverage Model

The methodology is based on an attempt to obtain block Input-Output relationship coverage, which means:

- The block will be fed with every possible legal input sequence. Inputs are defined in the environments.
- All output signals will be systematically checked for correctness at all times. The rules check the outputs.
- Selected internal states of the system will be checked for correctness at all times.The rules check the internal states.
- The rules will check the functionality of the block. For instance, if the purpose of the block is to acknowledge requests with a “grant” signal, then this functionality should be covered, for example, with a formula of the format “AG(request -> AF grant)”.

12.3 Writing Rules

You should write rules in the following manner:

For **every** output signal and selected internal signals, and for **every** clock cycle:

1. Determine the relationships of the signal to all other signals (inputs and outputs).
2. Write the rules that check the preservation of these relationships.
3. Divide the rules for each signal into three types:
    - The signal **will always** change value when necessary.

- The signal **will not change** when it should not.
- The signal **will have** a specific value at times that it must have that value.

Experience with RuleBase does not necessarily indicate that rules that contain complex signals find design errors more often, and it doesn't mean that they cover all errors either. RuleBase is effective due to the careful and methodical coverage of all signals.

12.4 Writing Environments

When writing environments, we suggest you adhere to the following guidelines:

- Keep the input signal nondeterministic, whenever possible, even if this causes an illegal input sequence (as long as it doesn't confuse the block).
- Restrict inputs only when necessary. For example, when the logic is confused by illegal inputs, or when you want to restrict the environment to less than the legal input behavior because of size problems.
- Hold rule and environment reviews as described in "Planning and Reviewing Rule and Envioronment Writing" on page 214, since it is difficult to determine how risky a certain environment restriction is (in terms of missing design errors).
- Write rules that check the behavior of the environment, because the quality of the verification is dependent on the quality of the environment. Specifically, write rules that check that events in which the environment is expected to be able to generate are indeed generated. For instance, the following rules check that both read_enable and write_enable can be asserted by the environment (assuming these are environment signals):

  ```
  EF read_enable
  EF write_enable
  ```

12.5 Planning and Reviewing Rule and Envioronment Writing

Rule and environment writing are strict engineering activities and should be planned and reviewed. You should conduct reviews with teammates who understand block functionality on both the rules and the environments written. In addition, it is very important to go over environment restrictions with the block designers, in particular restrictions that were added in order to avoid state explosion. The designers may challenge the assumptions taken and may request more effort in verifying the restricted areas.

These reviews are also useful to describe the areas not covered by formal verification to the other teammates. A review can guide the simulation team to stress those areas with simulation tests.

CHAPTER 13

Advanced Verification Engines (RuleBase Premium)

13.1Introduction

The RuleBase Classical version (the common RuleBase version) includes only one verification engine, which is called Discovery. The RuleBase Premium version contains additional verification engines in addition to Discovery. This chapter describes these additional engines.

Note: *This chapter is brought here in a preliminary form.*

13.2 SAT Engine

SAT is a Bounded Model Checking engine. Similar to the standard model checking procedure conducted by Discovery, SAT searches for bugs in a design.

The main difference is that with SAT the user is required to specify a finite range of cycles for which the bug is searched. Bounded model checking with

SAT is normally conducted in a gradual manner since typically, the larger the range, the longer the search process.

First, SAT searches for bugs starting from the initial state, up to some user-defined bound *k*. If no bug is found, the next problem instance will be from *k*+1 to k+(interval size), and so forth. There are three possible terminations of this process:

- A bug is found. In this case a counter-example is presented to the user in the usual way.
- The problem becomes too difficult to solve in a reasonable amount of time. This means that the property was not proven to hold globally. The only guarantee that the user has in such cases is that the property holds up to the last cycle that SAT was able to prove.
  For example, if SAT proved that there is no bug in the range 0..20, and then timed-out while attempting to prove that there is no bug from cycle 21 to 30, the only guarantee that the user has is that there is no bug up to cycle 20. In this case, the user may try to look in cycles 21 .. 25, which is an easier problem (see below).
- No bug is found up to the *diameter D* of the design. If we compute the shortest path from an initial state to each of the reachable states of the system, the diameter is defined to be the longest of these paths. Since hardware designs have a finite number of reachable states, we are guaranteed that such a finite diameter exists (finding the diameter is difficult by itself, as we will later explain). Thus, if there is no bug up to cycle *D*, it means that there is no bug at all, and the property is verified. Before *D* is reached, there is no guarantee that the property holds globally.

13.2.1SAT Technology

SAT has a completely different underlying technology than Discovery, which works with a data-structure called BDD to represent the entire set of reachable states—which is why Discovery's bottleneck is typically the memory requirements that may grow exponentially

SAT first builds a Boolean formula, which is logically satisfiable if and only if there is a bug in the given range. Each signal in the design, in each cycle, is represented by a different variable in this formula. Then, SAT tries to satisfy the formula (i.e., find a single assignment to the formula that evaluates it to TRUE). If it finds one, it becomes the counter-example.

The satisfiability problem also requires exponential time to solve, but there are no exponential memory requirements. Thus, a fast CPU is the key for obtaining fast results with SAT rather than large memory. The size of the formula that is generated in the first step has a large effect on the speed of SAT. The formula grows linearly with the distance from the initial state. Thus, the distance from the initial state, rather than the range itself has the largest effect on the difficulty of the problem. For example, searching in cycles 20..30 takes more time than searching in cycles 10..20.

A thorough empirical study showed that many designs that cannot be verified with the standard Discovery engine can be efficiently solved with SAT, and vice versa. This is why the combination of these tools is very productive. SAT is usually better in finding bugs ('falsification'), especially if they can be reached in early cycles (typically up to cycles 40 – 50, depending on the design). Proving that no bug exists ('verification') is much harder for SAT, because this requires, as explained earlier, to reach the diameter *D.* Since this number is usually high (more than 100 even in small designs), it is normally beyond the capacity of SAT. The standard model checking engine of Discovery is much better in verification and in finding 'deep' bugs.

SAT has the following two restrictions:

- It can only check safety properties.
- It can only check one formula at a time.

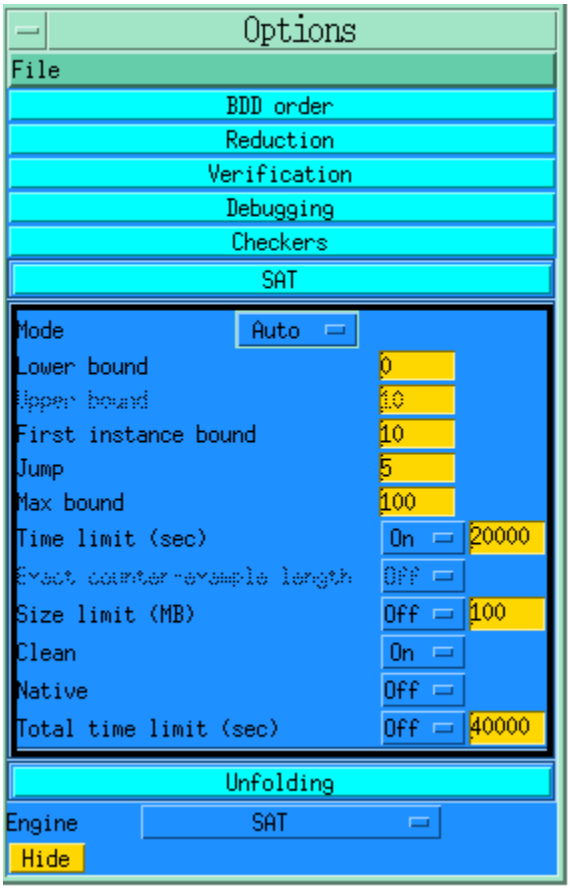
13.2.2 SAT GUI

As described above, there are several parameters that affect the way SAT operates. You can control the size of the bound *k*, the size of each 'jump', the time-out, and several other options described below.

To launch SAT from a RuleBase directory:

1. Set environment variable RB_SAT to 1 and launch the GUI.
   A new box, labeled SAT, will appear at the options and a new engine.
2. To run SAT, select SAT.
   You can run SAT in the following modes:
   - **Auto Mode**
     In Auto mode the range is incremented automatically. You can specify the initial cycle (the *lower bound*) from which to start, and the range of the search instance (sometimes referred to as the ‘jump’). Auto mode will stop if a bug is found or if the maximum cycle, you specified, is reached. It will also stop if the time limit is reached.
     For complete verification, the maximum cycle should be the diameter *D*. We recommend you specify a high number, thus forcing SAT to stop only when the time limit is reached. Auto mode has the following the options:

Name	Range	Default	Explanations
First Instance Bound	Natural	10	The Bound for the first instance.
Jump	Natural != 0	5	The size of the interval (number of cycles) between two consecutive runs.
Max Bound	Natural	100	The last cycle in which to look for a counter example. For full verification use the diameter (if it is known).
Total Time Limit	Natural	40000	The time limit for all the intervals.



- **Manual Mode**

  In this mode, you manually specify the range of the search.
  This option is useful in two cases. The first case is when searching for the exact cycle in which the bug first occurs (this is useful for finding the shortest possible counter example). For example, suppose that Auto mode finds a bug in the range 10 – 20. To find the exact cycle, use the Manual mode to search in cycles 10 – 15, and then, if the answer is positive, search again in cycles 13 – 15, and so forth, until the exact cycle

is found. Another typical case that makes Manual mode useful occurs during debugging. Once you find a bug and consequently change the design, the best way to know whether the bug was fixed is to perform another search for the bug in the same exact cycle. This is when the option of searching in a specific cycle only, which is only available in Manual mode, becomes very useful.
Manual mode has the following the options:

Name	Range	Restrictions	Default	Explanations
Upper Bound	Natural		10	Maximum cycle to search for a counter-example (or exact cycle for which to search, in case the previous option is on).
Exact Counter-example Length		(Sets Bound = Lower Bound)		Looks for a counter-example only in the cycle specified in Bound.

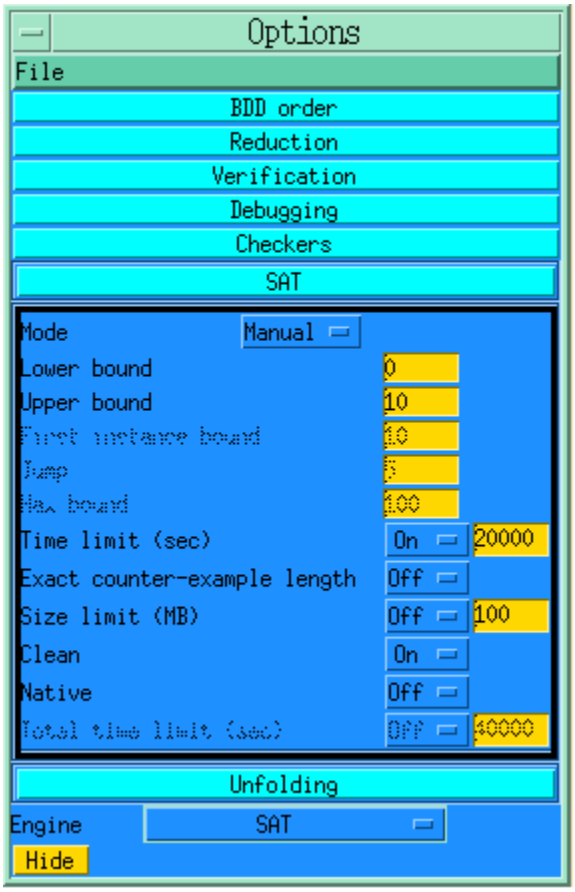

Options
File
BDD order
Reduction
Verification
Debugging
Checkers
SAT
Mode
Manual
Lower bound
0
Upper bound
10
First instance bound
10
Jump
5
Max bound
100
Time limit (sec)
On
20000
Exact counter-example length
Off
Size limit (MB)
Off
100
Clean
On
Native
Off
Total time limit (sec)
Off
40000
Unfolding
Engine
SAT
Hide

FIGURE 19.

Conjunctive Options:

Name	Range	Restrictions	Default	Explanations
Time Limit	Natural		20000 (Sec)	Time limit for a single interval.
Lower Bound	Natural	0 <= Bound	0	Minimal cycle to search for a counter example.
Size limit	Natural	0 < Size	Unlimited	Restrict the CNF formula file size. (MB).
Native			Off	Grasp's Native decision heuristic (DLIS)
Clean			On	Delete temporary files.

To remove SAT from the options, unset RB_SAT and restart the GUI.

13.3 Belzeebub Engine

The Belzeebub Engine incorporates classified IBM technology, and is only available from IBM under a confidentiality agreement. Contact technical support for details.

13.4 Unfolding Engine

Unfolding is a bounded model checking engine. It proves safety properties bounded to the first k cycles, using BDD-based procedures. If the property failed, Unfolding produces a Discovery-like counter-example.

The following are the advantages to this approach:

- State variables become wires, therefore Unfolding has an advantage in complex and shallow designs (many FFs, few inputs).
- Enables static BDD ordering – determines the initial order of variables in the BDD based on the circuit's structure. In many cases, this order is sufficient to complete calculation without the need to reorder.
- Unfolding calculates signal values as a function of inputs, rather than a randomly shaped set of states. Therefore, the function is more natural and reflects the circuit's structure.
- Eliminates false resource sharing along time. For example, if FF performs one task in even cycles and another task in odd cycles, Unfolding will have a different function for each task, and Discovery will have one big function that represents all of its tasks.
- Fine underapproximation – makes an input constant only at a specific cycle. Running with underapproximations is called Partial mode. In Unfolding, Partial mode can be thought of as an intermediate stage between simulation and formal proof (advanced simulation), when according to the parameters (the underapproximation threshold) it goes smoothly from one to another.

Unfolding is potentially suitable to wide and shallow designs (for example Boolean/sequential equivalence, datapath).

Unfolding has the following limitations:

- Unable, in most cases, to prove the correctness of formulas for all cycles; therefore, a witness is unavailable.
- Works only on Safety OnTheFly formulas.

13.4.1 Main Settings

The following are the main settings

- **cycles**
  - Number of cycles to be checked. If you have an idea that the location of a suspected bug is in cycle n, check n+10 cycles, since the bug may be easier to find in the further cycles.
  - Defaults to 50 cycles.

- **mode**
  - **EasyPartial (default)** – performs a quick check of the formula with low approximation bound (good for common bugs).
  - **Exact** – tries to prove the cycles (does not make approximations).
  - **HardPartial** – works harder than EasyPartial (higher approximation bound) and has a better chance of finding rare bugs.
  - **ContinuesRun (tmp name)** – tries to find the bug using EasyPartial, and if it can't find it, continues with HardPartial until the cycles are proven (or the bug is found).

- **Reorder (on/off) –** selects whether or not to perform reordering.
  - Defaults to "off".

- **Threshold –** defines when some special action should be taken (reorder or assumption). The number is the total number of BDD nodes currently in the calculated BDDs.
  - Defaults:
    Exact =1,000,000
    EasyPartial = 100,000
    HardPartial = 1,000,000
    (ContinuesRun - disabled)

- **ForeverMode** (available only for EasyPartial & HardPartial)
  - singleRun (default) – runs once on each cycle.
  - runForever – runs continuously until a counter-example is found or all cycles proven. Use it for nights/weekends.

- **AssumptionAgorithm** (available only for EasyPartial & HardPartial) – describes how to choose variables for underapproximation.
  - MaxLevel (N=0, default for EasyPartial) – quick algorithm, but potentially much more approximations will be made than with the second algorithm.
  - Estimator (N=2, default for HardPartial) – slow algorithm with good results.

3. **ReorderAlg**
   - Sift (N=4, default) – recommended. It takes a long time, but provides good results.
   - Random (N=2) – TBD
   - RandomPivot (N=3) – TBD
   - Linear(N=18) – TBD

- **ReadOrder (on/off) –** reads initial order from a file.

- **WriteOrder (on/off) –** writes order in the end of the run.

- **WriteOrderEvery N –** writes current order every N seconds.

- **timeOut: _____ (default=-1)** – provides timeout in seconds.

- **<show log>** – shows Unfolding log.

APPENDIX T

Option tables

Note: *This appendix is brought here in a preliminary form.*

SMV Options

TABLE 1. Reachability

USER VISIBILITY		DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	GUI - options				
-f	Reachability	Perform reachability. (In normal mode this means simplify assume with the reachable states.)	Works as Boolean. When Reachability is On, GUI sets -f -cp 1 -inc. Other options in this table only work when Reachability is set to Yes.		In GUI: Yes
-inc		Incremental: simplifies the transition relation, according to the last donut.			
-cp #n		Partitions transition relation part or reachability settings.			
-fly #a	Verify Safety OnTheFly (the number)	Saves each n donut while performing reachability.	Int >0 Only works when "Verify Safety OnTheFly" (in GUI) set to Yes.	Basic	In GUI: 10
-lfly #n	Verify Liveness OnTheFly (the number)	Performs liveness on_the_fly for each n iterations.	Int >0 Only works when "Verify Liveness OnTheFly" (in GUI) set to Yes.	Basic	In GUI: 10
-AF_on-the-fly_no w		Dynamic liveness on the fly.			
-longest_trace	Gen longest trace	Finds the longest trace that doesn't repeat the same state.	Boolean	Basic	In GUI: No
-no_longest_trace		see -longest_trace above. Only used when 'No' is applied dynamically.			
-longest_trace_no w	Gen longest trace (Now)	Longest trace to current donut.	dynamic option	Basic	
-early_termination #n	Stop after iteration	Stops after n iterations.	int >0	Advanced	Disabled

TABLE 1. Reachability

USER VISIBILITY		DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
-no_switch		Does not switch to full TR after reachability.	Boolean	Advanced	Disabled
-simplify_donut		Simplifies each donut according to !(reachable_states) OR Allows overlapping donuts.	Boolean	Advanced	Disabled
-fairness_in_safety		Provides fairness in Safety OnTheFly mode.	Boolean	Advanced	No
-simplify_in_reverse		Simplifies the BDDs according to the reachable states in a backward search.	Boolean	Very Advanced	No
-counters_file <file>		Counters mode on the signals in file.	File name	Basic	Not set

TABLE 2. Reorder

USER VISIBILITY		DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	GUI - options				
-reorder_minimum_size #n	<= BDD size	SMV will not reorder if it has less than n nodes allocated.	Int	Basic	In GUI: 100000
-reorder_maximum_size #n	BDD size <=	If SMV gets this flag, it will not reorder if it has more than n nodes allocated.	Int >0	Advanced	
-start_reorder_at_iteration #n	<= iteration	If SMV gets this flag, it will not reorder until iteration n.	Int	Advanced	
-stop_reorder_at_iteration #n	iteration <=	If SMV gets this flag, it will stop reordering when it reaches iteration n.	Int	Advanced	
-reorder_low_threshold #n	Low treshold	SMV stops reordering when the BDD size falls below the low threshold. Does not effect Cheetah algorithm.	Int	Advanced	In SMV: 0
-dont_swap_above #r	Laziness factor	r controls the allowed reordering effort for each variable reordering. A single variable is sifted in a specific direction, only as long as the BDD size does not exceed r*prev_size. (prev_size being the BDD size just before we started sifting the variable). Does not effect Cheetah algorithm.	Real Number > 1	Basic / Advanced	In GUI: 3.0
-growth_factor #r	Growth factor	r controls when the next reordering round will take place. A new round will only start when the BDD size reaches last_size*r. (last_size reflects the BDD size, at the end of the previous reordering)	Real Number > 1	Basic / Advanced	In GUI: 2.0

TABLE 2. **Reorder**

USER VISIBILITY		DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	**GUI - options**				
-reorder	Reorder	Performs dynamic BDD reorder.	Boolean. Other options in this table only work when reorder is set to yes.	Advanced	In GUI: Yes, In SMV: No
-rudell	Algorithm	The reorder algorithm.	Only one flag of these is supposed to be applied	Advanced	In GUI:Rudell+ Cheetah
-quick_rudell					
-cheeta					
-cheeta+rudell					
-cheeta+quick_rudell					
-no_reorder_in_AG_ce		Does not reorder while calculating AG counter example.	Boolean	Advanced	Disabled (Do reorder in ce)
-no_reorder_in_light_proof		Does not reorder in light proof.	Boolean	Advanced	Disabled (Do reorder)
-no_reorder_in_cone		If SMV gets this flag, all variables are reordered. Otherwise, only variables in cone are reordered.	Boolean	Advanced	Disabled (reorder only vars in cone)
-decrease_reorder _percent		Causes growth_factor to decrease gradually after 2 million nodes. At 2 million nodes, the percent is the original number. At 10 million nodes, it is about 1+ (original/10)	Boolean	Advanced	Disabled (no decrease)

Note: *All reorder options here can be changed dynamically.*

TABLE 3. Garbage Collection

USER VISIBILITY		DESCRIPTION	VALUES / CONSTRAINTS	LEVEL	DEFAULT
parameter	GUI - options				
-gcinfo		Provides more information about **garbage collection.**	Boolean	Advanced	No
-gcmin #n		The minimum size on which garbage collection is called.	Int	Very Advanced	0
-gcmax #n		The maximum size on which garbage collection is called.	Int	Very Advanced	100000
-gctime		The 'nodes allocated' printed with time stamp	Boolean	Advanced	No

TABLE 4. SMV Reductions

USER VISIBILITY		DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	GUI - options				
-reduction n	SMV reductions	Controls over-approximations and reductions in SMV. The number indicates the heaviness of the over-approximations. Controlled by SMV reductions.	(1..10)	Basic	In Gui: 8 (3 is probably Better) In Smv: Disabled
		No FF pool and equivalence pool.	Boolean	Advanced	Disabled
-no_after_reach_reductions		Removes the reductions after reachability.	Boolean	Advanced	Disabled
		Only removes reset reductions.	Boolean	Very Advanced	Disabled
-equiv_in_cuts		Adds equivalence to ff_pool.	Boolean	Very Advanced	Disabled

TABLE 4. SMV Reductions

USER VISIBILITY		DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	GUI - options				
-no_sat_reduction		Does not reduce after each formula evaluation.	Boolean	Very Advanced	Disabled
-no_reduction_in_counter_example		Does not perform the reduction before generating counter-example. Disabled in case of bug.	Boolean	Very Advanced	Disabled

TABLE 5. Other

USER VISIBILITY			DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	setenv	GUI - options				
-light_proof #n	RB_SPECN should be defined for RuleBase.		Tries to evaluate each formula without reachability (in normal mode) and with partition transition relation for n seconds.	Integer >0	Basic	In GUI: 30 in SMV: disabled
-light_proof_without_timer			Evaluates each formula without reachability and with partition transition relation.	Boolean	Basic	
-vacuous_check_only		Witness = Vacuity only.	Does not produce witness.		Basic	
-v #			Verbose level	1/2	Advanced	
-or_inside_simplify			Little change inside simplify assuming. Better for some models.		Very Advanced	
-ca			Checks all options. Does not run the formulas, only checks the model semantically.	Boolean	Advanced	
-or_before_recurse			Changes in r_collapse is good for some models. Provides a little optimization.		Very Advanced	

TABLE 5. Other

USER VISIBILITY			DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	setenv	GUI - options				
-efficient_backward_sort			This backwards sort does not yet support invars. If this parameter is present, no invars may be enabled.		Very Advanced	
- separate_reductions_in_counter_ example			Performs separate reduction for each formula. The default is common reduction only when set to counter-example reductions.		Very Advanced	
-osa			Optimized simplify assuming.		Very Advanced	
-suicide_if_swap			SMV kills itself if it has low CPU for a long time.	Boolean	Basic	
-first_fail			Quits after first failure.	Boolean	Advanced	
-i <file name>			Starts bdd_order	file name		
-log <file name>			Name of the log file.	file name		GUI sets it to 'log' being asked.
-d	Can be overriden by the RB_NODEL_SMV RB env. variable		Removes rb.smv (delete input in smv). Sets to No, and enables future run of stand alone SMV.	Boolean	Advanced	GUI sets it without being asked.
-no_filter_synopsys			Does not provide a filter on the signals printed to log.	One of these, at most, is supposed to be set.	Advanced	By default, none of these are set. SMV filters signals with NET_ in their names.
-extra_filter_synopsys			Provides extra filtering of the signals written to log.			
	RB_NAMES_FILTER <filename>		Filters the signals written to log. Used when it takes too long to print the trace.			

TABLE 5. Other

USER VISIBILITY			DESCRIPTION	VALUES/ CONSTRAINTS	LEVEL	DEFAULT
parameter	setenv	GUI - options				
-no_real_loop			Does not extract real loop. Used when trace generation blows up. May return a trace without a loop			Disabled
-multiple_traces #n			Find n traces for each formula.	int >1	Advanced	Not set (1 trace per formula).
-prolong_trace #n			Prolongs the trace with n cycles.		Basic	0
-dump_reachable <file>			Prints the reachable states to file.		Advanced	Not Set
-restore_reachable <file>			Takes the reachable states from the file.		Advanced	Not Set
-k # key table size			Very Advanced		Advanced	
-c #apply cache size			Very Advanced		Advanced/ Basic	
	CHECK_CONST_INIT		Disables Constant Propagation (CP) reduction.	Boolean	Very Advanced	0 (Cp Reduction Enabled)

RuleBase Options

TABLE 6. rulebase.setup

USER VISIBILITY		DESCRIPTION	VALUES / CONSTRAINTS	LEVEL	DEFAULT
parameter	GUI - options				
database		Configuration is controlled by rulebase.setup.			
entity		Configuration is controlled by rulebase.setup.			
SYNTHESIS		Configuration is controlled by rulebase.setup.	TEXVHDL, KOALA_VERILOG, DADB_VIM,...,NONE, VIM		
SOURCE		Located with VIM.controlled by rulebase.setup			../vimdbase
VIEW		Directory inside Vimdbase in which design is located. It is controlled by rulebase.setup.		Advanced	HISVHDL
RB_EDL_CASE		Defines case sensitivity of EDL, including rule names. The names of rule directories are produced from rule names. Therefore, changing this parameter in the middle of the project will "hide" directories of rules with capital letters in their names	SENSITIVE/ INSENSITIVE		SENSITIVE
RB_DESIGN_CASE		Depends on the design compilation (SYNTHESIS).	SENSITIVE/INSENSITIVE, RB_EDL_CASE is INSENSITIVE must be INSENSITIVE For VHDL must be INSENSITIVE.		SENSITIVE
RB_NO_IMPL		When set to 1, only EDL, no implementation. PERFORMED IN BACKEND OF RULEBASE.			Disabled

TABLE 7.

Parameter	setenv	GUI	DESCRIPTION	VALUES / CONSTRAINTS	LEVEL	DEFAULT
	rb_report_zero_clk		Checks if clock logics is 0.		advanced	
	RB_DETAILED	Explains vacu-ous.	Explains vacuous.		Shoud be removed from options. User always wants to explain vacuity.	In GUI: Yes
	RB_ASYNC_OUTPU T		Used for FFs with async reset. If ON, model FFs with async reset by ff with sync inputs		Advanced	On
	RB_BIG_ENDIAN		Useful when vector direction does not comply. If vectors used in format v(i..j), the value of RB_BIG_ENDIAN does not matter.		Advanced	RB:No GUI: Yes
	RB_COMPLEX_CLOC K_WARN		Warns about suspicious clock calculating	Boolean	Advanced	No
	RB_DOLLAR_FLAG		Switches $ to _ in name for SMV.		Advanced (When $s In Names)	
	RB_UNIQUE_NAME		Switches characters that con-flict with SMV to unique inter-nal representation.		Advanced	

TABLE 7.

	RB_REDUCTION_ONLY		RuleBase performs reductions, prints rb.smv, and quits. PERFORMED IN BACKEND OF RULEBASE	Boolean	Advanced	Disabled
	RB_TRANS_DSL_ARY		Translates DSL ARY to single FFs. If off, DSL ARY should be modeled in environment.	Boolean	Advanced	No
	RB_SPECN	Attempt light proof.	Causes Sugar to create normal mode formulas.			disabled
-witness - rulebase invocation option.		Witness	Sugar creates SPECs for checking vacuity/giving witness for pass formulas.		Basic	
	RB_AG_BOOLEAN	Verify Safety OnTheFly				
	rb_cct	Creates circuit file.	Creates circuit file for cctdag (Debugging). PERFORMED IN BACKEND OF RULEBASE?	Boolean	Basic	In GUI and RB: No
	init_impl_ffs		When ON, init every FREE IMPL DFF INIT to 0. PERFORMED IN BACKEND OF RULEBASE.	Boolean	Advanced	On
	rb_init_latches		If enabled, gives init value to latches.	Boolean	Advanced	No
	rb_constant_simulation		Constant signals pre-reduction simulation.		Very Advanced	0
	rb_report_unresolved		Provides more information on unresolved.		Very Advanced	1
	rb_ff_iterations		Number of FF EQUIV reduction iterations. RuleBase backend		Very Advanced	

TABLE 7.

	RB_ASSUMPTIONS		Translates assumptions.			1
	rb_a2d_boolean_only					

E.RuleBase

TABLE 8. Reductions Control

setenv	GUI	Meaning	Constraints	Level	Default
rb_bdd_reductions	Reduction effort	**Reduction** effort	1 - High 0 - Low	Basic	In GUI and SMV: 1(high)
rb_constant_propagation		Performs constant_propagation reduction.	Boolean	Advanced	1
rb_cone_verification		Simple Cone.	Boolean	Advanced	1
rb_domain_check		Domain reduction.	Boolean	Advanced	1
rb_reduction_report		Reports reduction related information.	Boolean	Advanced	No
rb_e_t_ff		Edge triggers FFs with long alternating clocks. Better to use clocks high 1 cycle. See also Section 11.5: Design Mapping.	Boolean	Advanced	Off

TABLE 8. Reductions Control

rb_check_nondet_define		Reports a warning for every nondeterministic **DEFINE** signal used more then once. PERFORMED IN BACKEND OF RULEBASE	Boolean	Advanced	1
rb_smart_constants	Heavy reduction 1	Is set in case too big to start SMV. Performed in backend of RuleBase. Useful when more than 1000 variables are generated, due to SMV limitation.	Boolean. If rb_bdd_reductio ns is On	Advanced	No
rb_smart_cone	Heavy reduction 2	Is set in case too big to start SMV.	Boolean. If rb_bdd_reductio ns is On	Advanced	Off
reduction_bdd_limit	BDD node limit	Maximal size of BDD in RuleBase. PERFORMED IN BACKEND OF RULEBASE	integer. If rb_bdd_reductio ns is On	Advan Ced	
simple_vars		Reduction of Assign to define. PERFORMED IN BACKEND OF RULEBASE	Boolean	Advanced	1
safe_fairness		No reduction when the fairness cone does not touch formula cone	Boolean	Advanced	1
RB_NO_REDUCTION		No reductions on envs.			
RB_EARLY_CONE_CH E CK		Early cone check.	Boolean	Advanced	No
RB_NO_ALL_AFTER_ RE DUCTION		Does not create all_after_reduction file.	Boolean	Advanced	Disabled

F.DP12 reduction

TABLE 9.

setenv	Meaning	Values/ Constraints	Level	Default
RB_CHECK_DPL1_L2	run **DP12**. Other flags in this table are only relevant if this variable is set.	Boolean	Basic	Off
RB_DPL12_REMOVE_ONLY	The layer to be removed.	L1:'1', L2:'2', Bigger:'0'	Advanced	Bigger(0)
RB_DP12_LEAVE_ORIG_CL K	Does not set all clocks to 1.	Boolean. if RB_DPL12_REMOVE_ ONLY is set to L1, model L2 clocks,...	Advanced	Off
RB_DP12_SAVE_DPVIM	Saves the VIM after DP12 and uses it as long as the design does not change.	Boolean	Advanced	No
RB_DPL1_L1_OUTPUT_RES TRICT	Aborts if c1_outputs is not full. Performs backward search.	Boolean	Advanced	No
RB_DPL1_STOP_AFTER_DP12	only DP12 check	Boolean	Very Advanced	
RB_DP12_NUM_OF_FAILS	The number of violations for which DP12 is looking.	Integer	Advanced	1
DP12_IGNORE_FILE_NAME		File Name	Very Advanced	C2_inputs
DP12_L1_OUTPUTS_FILE_N AME		File Name	Very Advanced	C1_outputs
DP12_L2_INPUTS_FILE_NA ME		File Name	Very Advanced	Ignore_inputs

TABLE 9.

DP12_CUT_INT_FILE_NAME	When points to a file, DP12 cuts all nets from that file.	File Name	Very Advanced	
RB_DP12_CUT_BITVECTOR E	Shorter problem message. When set after finding problem in name v(#) will cut all vector.	Boolean	Advanced	No
RB_DP12_ONE_WARNING_ FOR_PORT	Same as previous, for DSL_ARRAY.	Boolean	Advanced	No

G.GUI

TABLE 10.

parameter	setenv	GUI name	Meaning	Values/ Constraints	Lev el	Default
	PF_SCOPE		Calls pf scope via **GUI.**			
	RB_AUTO_LOAD _RUL E_CFG		Uses private config file from ./ cfg/<rule_name>.cfg			0
	RB_COND_RUN		Asks confirmation on run.			0
	RB_EDITOR		On edit press.			
	COPY_RUDELL	Copy back after run	Copies the result of the dynamic reordering at the end of the run.	To both: "yy" To <rule>.order: "yn" To orders pool: "ny" No:"nn"	Basic	In GUI: To both
	rb_find_order	Use order file	Prepares file bdd_oder before SMV starts.	orders pool: "pool" <rule>.order:"rule "	Basic	In GUI: To orders pool
	RB_ORDER_DIR		Orders dir for copy back order.			

H.Debbugging/Scope

TABLE 11.

parameter	setenv	GUI	Meaning	Constraint	Level	Default
	RB_EXPLAIN_CE	Explains timing diagram.	Explains timing diagram for old scope.			In GUI:Yes
	RB_SCOPE_MSG	Shows formula text.	Defaults show formula text - for old scope			In GUI:Yes
	RB_SCOPE_PER_RULE_STA TE	Per-rule state file.	Per rule state file - for old scope			In GUI:Yes
	RB_TRACE_NUMBER <n>		See the n'th trace.			

End of Document.