User's Manual

FoCs

Formal Checkers - a Productivity Tool

Version 1.0

with Sugar2 support (EDL flavor)

Formal Methods and Technologies Group
IBM Research Lab in Haifa
April 2003

Notices

FoCs User's Manual

Date modified April 2002

For information regarding FoCs, contact: Gil Shapir (shapir@il.ibm.com)
Tel: +972-4-8296258

Table of Contents

CHAPTER 1

Introduction

1.1 Overview

FoCs (short for Formal Checkers, pronounced “fox”) is a productivity tool that greatly aids design and verification engineers in the complex, costly task of developing simulation test benches.

FoCs automatically generates simulation checkers, also known as monitors, from properties specified in the language Sugar 2.0[†]. These properties, also called “rules” or “assertions”, describe the legal behaviors of the design under test. Typically, the user of FoCs derives properties from the design specification documents where they are written informally in English, and writes them as Sugar formulas. Using FoCs, these properties are translated into checker code in the desired target language—Verilog or VHDL[‡]. The checker code is then connected to the simulation environment. During simulation, the checkers track and report violations of the properties.

†. “Sugar” is an industry-standard language for assertion-based verification. It was selected as a basis for an IEEE standard by Accellera on April 22, 2002.

‡. In the future, FoCs will also generate checkers in C/C++.

As an example of the power of FoCs, consider an arbiter which must abide by the following property: *upon the completion of five consecutive cycles where the request signal is asserted and the acknowledge signal not asserted, the busy signal should be asserted*.

The Sugar formulation for this property is:

- `{[*]; {request & ! acknowledge}[5]}|-> {busy_flag}`

Once this property is fed into FoCs, the tool produces a corresponding VHDL or Verilog checker that can be integrated into a simulation environment and monitor the design behavior on a cycle-by-cycle basis for violation of the property. It is often the case that a one-line or two-line Sugar property is automatically translated by FoCs to a checker which spans hundreds of lines of HDL code†. The benefit—in terms of programmer time that would have otherwise been spent in manually coding the checker—is evident. There are other benefits to using FoCs—such as reduced debugging time, portability, and reuse of properties. The checking code produced by FoCs is synthesizable, so this code can be used in emulation as well. By virtue of these advantages, FoCs increases engineering productivity in a very notable manner.

A complimentary application of FoCs is the generation of coverage monitors for coverage analysis. When FoCs is used for this purpose, the user specifies combinations and/or sequences of events which he or she wants covered in simulation. FoCs then automatically generates a checker to track the occurrence of these events during simulation.

For further information on FoCs, see the FoCs website at

www.haifa.il.ibm.com/projects/verification/focs/index.html

1.2 About This Manual

This manual is intended to serve as a guide for using FoCs (including the definition of design properties using Sugar). The manual is organized as follows:

- **Chapter 1 Introduction** – introduces the FoCs tool.

†. The checker produced by FoCs for the above Sugar property is included, for reference, in Appendix A.

- **Chapter 2 Installation and Setup** – explains how to install and set up the FoCs tool.
- **Chapter 3 Linking the Checkers with your Design** – explains how to link the checkers with the design you are testing.
- **Chapter 4 Tutorial** – guides you through a short example, where you can get hands-on experience in creating checkers using FoCs.
- **Chapter 5 Sugar** – provides details on how to start using the Sugar and EDL specification languages to specify the design properties, including the structure of the rules file, creating formulas, expressions, and satellites. This chapter also presents examples of real-life formulas that can be used with FoCs.
- **Chapter 6 Customizing FoCs Settings –** describes the settings and tabs that can be customized for your use.
- **Chapter 7 Using FoCs for Functional Coverage** – explains how to enhance the quality of tests by providing a means for measuring test coverage.
- **Chapter 8 Defining Bugspray Events** – for users who want to use FoCs with Bugspray instrumentation. Explains how to define Bugspray events in the FoCs rules file.
- **Chapter 9 FoCs for RuleBase Users** – explains the methodology of working with FoCs with properties used for Formal Verification.
- **Appendix A** – Shows examples of checker code in VHDL and Verilog.
- **Appendix B** – Documents and explains common error messages.

CHAPTER 2

Installation and Setup

2.1 Installation

This section explains how to install the FoCs tool and get it up and running.

If FoCs is not already installed on your computer or network, proceed as follows:

1. Create a new directory and copy the installation file focs.tar.gz into the directory.
2. Type the command gzip -d focs.tar and press Enter.
3. Type the command tar xvf focs.tar and press Enter.
4. Type the command rm focs.tar and press Enter.

This will unzip the tar file and copy the installation files into their appropriate location.

2.1.1 Personal Setup

To customize FoCs for your individual environment, you need to set an environment variable and create an alias for FoCs.

The environment variable FOCS_DIR should point to the FoCs installation directory.

```
In csh: setenv FOCS_DIR the_installation_directory
In ksh: export FOCS_DIR=the_installation_directory
```

Create an alias “focs” for $FOCS_DIR/focs as follows:

```
In csh: alias focs '$FOCS_DIR/focs'
```

or
add $FOCS_DIR to your search path.

2.2 Running FoCs

The following sections provide tips on how to begin working with FoCs.

2.2.1 Checker Generation

Before you begin, you should create a working directory from which you run FoCs.To begin using the FoCs tool:

1. Type the command: `focs`
2. If this is your first FoCs session in this directory, you have to set up FoCs for your project. Click **Settings** to open the Settings dialog.
3. Select the target language (VHDL or Verilog), the target simulator (only for VHDL), the clock signal name, and the reset signal name (unless you ask FoCs to generate an internal reset).
4. Select the rules file—the file in which you write your rules. The language in which you specify rules (Sugar) is described in Chapter 4 of this guide.
5. You can browse through the other Settings tabs and fields if you want to have more control over the generation process. Use the tool-tips to see short field descriptions.
6. When you are done, close the Settings dialog and return to the main window.
7. Select a rule to be translated into a checker and click **Generate**. Errors are reported in the Messages window below. If generation is successful, the checker filename will appear in the Messages window.
8. If you wish to translate several rules into one checker file, select these rules (using control/mouse-button or shift/mouse-button or the All button) and press Generate. You will be asked to provide a name for the checker file, and for the entity name if the generated checker is in VHDL.

2.2.2 Batch Mode

Checkers can also be generated by the command line without invoking the GUI. To generate a checker for a specific rule, you must type in the command line: **focs -batch <rule name>**. To generate a checker for all rules in the rules file, you must type in the command line: **focs -batch all**. To generate several rules, you must type in the command line: **focs -batch <rules names>**. In this case, the settings for the generated checker are those defined in the file focs.setup. The best way to update this file is by defining the settings through the GUI. When exiting FoCs, the settings are saved to this file. You can also use the following flags in the batch mode:

focs -batch <rule name or rules names or all> **-rule** file> **-o** <output_file_name>

-s <setup_file_name>.

Flag	Explanation
-r	This flag reads the rule file from the command line instead of reading the rule file from the setup file
-o	This flag gives a specific name for the output file (checker name).
-s	This flag reads the setup file from the command line instead of reading the default setup file focs.setup (from the current directory).

CHAPTER 3

Linking Checkers with your Design

3.1 Introduction

In order to monitor design behavior during simulation, the generated checkers must become part of the simulated model. This occurs through the following three steps:

1. **Linkage**

   The checker module (in Verilog) or entity/architecture (in VHDL) must be compiled and linked to the design under test. The actual commands are specific to the compilation/simulation environment, and are not within the scope of this document.

2. **Instantiation**

   A call to the module (in Verilog) or instantiation of the entity (in VHDL) must (usually) be included in the design. FoCs assists in this step by generating the calling statements.

3. **Signal Connection**

   The checker signals must be connected to the real design signals. The solution depends on the language and on the simulation environment. FoCs provides a standard language solution for port mapping, and specific signal-connection solutions for several simulations.

In the following sections, we describe the mapping method and linkage.

3.2 Signal Mapping

To keep your formulas simple and readable, you can use short and meaningful signal names. You can then map these simple names to the real signal names. FoCs will map those signals to real ports if an instantiation statement was created, when a mapping for MTI is defined, or when Bugspray is used.

For example, use the name `request` rather than `BXX_ARB_REQ`. You can create a mapping file in your working directory and let FoCs map the aliases to the real names. The basic format of the mapping file is a list of pairs of signal names, one pair in each line.

```
alias1     realname1
alias2     realname2
...
```

For example:

```
request    BXX_ARB_REQ
signal1    /MY_DESIGN/BLOCK1/BLOCK11/SIGNAL1
```

You should define the deliminators according to your mapping option, for example, dots for port mapping or slashes for MTI Signal Spy.

The mapping file should be pointed to by Settings/SignalMapping/MappingFile.

3.2.0.1 Nested Design Signals

If design signals are nested and all signals are declared in the same entity, it is possible to define the Design Signals Prefix parameter in the settings options, under Signal Mapping. The value of this field is added to every design signal that appears in the mapping file. In case of Automatic Mapping, where the port name is considered to be the corresponding design signal name, the Design Signals Prefix is added to all port names that appear in the checkers entity (see Section 3.2.0.2 on page 14).

For example:

Design Signals Prefix is defined to: design.buf

content of file mapping.dat :

```
clk clock
rec receive
trans transmit
```

Is equivalent to:

```
clk design.buf.clock;
rec design.buf.receive;
trans design.buf.transmit;
```

To map ports to signals that appear in different parts of the design (the signals are nested, but the path is different from signal to signal), the following syntax exists in the file mapping.dat:

#path < path for the signal >

All of the signals that appear after this line, and until the next line with the same syntax, will receive the string in arrow brackets (<>) as a subpath, in addition to the global path defined by a environment variable. It is possible to use this syntax without defining a global path.

Example 1:

Design Signals Prefix is defined to: design

content of file mapping.dat:

```
#path buf_1
clk clock
rec receive
#path buf_2
trans transmit
```

Is equivalent to:

```
clk design.buf_1.clock;
rec design.buf_1.receive;
```

```
trans design.buf_2.transmit;
```

Example 2:

Design signals Prefix is undefined.

content of file mapping.dat:

```
#path design.buf_1
  clk clock
  rec receive
  #path design.buf_2
  trans transmit
```

Is equivalent to:

```
clk design.buf_1.clock;
rec design.buf_1.receive;
trans design.buf_2.transmit;
```

3.2.0.2 Automatic Signal Mapping

There is one more possibility to map a port signal. If the port signals have the same names as the signals in the design and are not nested or have the common path, they can be mapped by defining Warn Incomplete Mapping to No (in the Settings options, under Signal Mapping), and defining Design Signals Prefix to the path needed.

In this case, every port name in the checker that does not appear in the mapping file (or if a mapping file does not exist at all) will be mapped to the signal with the same name and with the path defined by Design Signals Prefix. A mapping that is defined in a mapping file, overrides the Automatic Mapping. As before, defining the path is optional.

Example 1:

Warn Incomplete Mapping No

Design Signals Prefix design

content of mapping.dat

```
#path buf
clk clock
```

```
checker ports : clk, rec, trans
```

Is equivalent to:

```
clk design.buf.clock;
rec design.rec;
trans design.trans;
```

Example 2:

Warn Incomplete Mapping No

Design signals Prefix is undefined

file mapping.dat doesn’t exist

checker ports : clk, rec, trans

Is equivalent to:

```
clk  clk;
rec  rec;
trans trans;
```

Example 3:

File mapping.dat doesn’t exist

Warn Incomplete Mapping No

Design signals Prefix design

Is equivalent to:

```
clk design.clk;
rec design.rec;
trans design.trans;
```

3.2.0.3 Using Hierarchical Signal Names

It is possible to use hierarchical signal names when writing Sugar formulas. For such signals, FoCs will automatically create a unique, non hierarchal signal name, and map it to the hierarchical name.

For example, it is possible to write the formula:

```
formula
{
       always(u1.u2.aa)
}
```

In this case, FoCs will generate a checker signal named **focs_u1_u2_aa**, and a mapping between it and u1.u2.aa.

If a design signal prefix is defined, it will be added to the signals mapping. In the previous example, if we had design signal prefix set to main we would get the mapping:

focs_u1_u2_aa => main..u1.u2.aa

For this kind of mapping, “Automatic Mapping” should be chosen, i.e., “Warn Incomplete Mapping” should be set to “No”.

It is possible to override the default mapping that is created by defining a different mapping for the signal name that was created in the mapping.dat file.

In the above example, it is possible to define in the file mapping.dat the following mapping:

```
focs_u1_u2_aa "design.my_block.aa"
```

3.2.0.4 Mapping Vectors

When a checker port is a vector, the signal that will be mapped to it should also be a vector. In the file mapping.dat, the syntax for vector is:

```
vector_name(index1..index2)
```

This is a syntax only for design signals because the range of a checker port is known (the size of design vector has to correspond to the appropriate checker port).

Example:

content of file mapping.dat:

```
#path buf_1
 clk clock
 rec "receive"
 #path buf_2
 trans transmit
 bus "vector(10..41)"
```

Where “bus” is the port in the checker which is vector 0..31, the above is equivalent to:

```
clk buf_1.clock;
rec buf_1.receive;
trans buf_2.transmit;
bus buf_2.vector(10..41);
```

If the range of the vector is not defined, the range of the checker port will be used. All the options that were discussed in the previous subsections are relevant for both vectors and signals.

3.3 Linkage

The following sections discuss linkage, which describes how to link the generated checkers with the design for several simulation environments.

3.3.1 Verilog

By default, two files are generated—the checker module and a file that contains a call to this module. You should embed the call statement in the design. The actual parameters in the call statement are the design signal names mentioned in the

formulas. If a name is mapped, as described in the Signal Mapping section above, the post-mapping name will be used as an actual parameter in the call statement.

You can choose to generate a bare Verilog by using Settings/GenerationStyle/ GenerateModule=NO. In this case, the checker will not be encapsulated in a module and you will have to embed its body in the design.

3.3.2 Pure VHDL

By default, three files are generated—the checker entity+architecture, a file that contains a component statement, and a file that contains an instantiation statement. You should embed the latter two statements in the design (automatically, if possible). The actual parameters in the port map of the instantiation statement are the design signal names mentioned in the formulas. If a name is mapped, as described in the Signal Mapping section above, the post-mapping name will be used in the instantiation.

3.3.3 Bugspray (IBM only)

The generated checker contains instrumentation directives that help Bugspray link the checker to the design and connect the checker signals to the design signals. The component and instantiation statements are not required and are not generated. Settings/GenerationStyle/DesignEntityName must specify the design entity to which the checkers refer. Port mapping names are used in Bugspray “ --!! inputs” section.

3.3.4 Model Sim®

Three files are generated—the checker entity+architecture, a file that contains a component statement and a file that contains an instantiation statement. You should embed the latter two statements in the design.

There are three possible signal mapping methods, controlled by Settings/ SignalMapping/MappingMethod. If you choose None, the generated checker will be regarded as a Pure VHDL checker and linked to the design as such (see above). We recommend that you choose None. If force freeze or signal spy is used, appropriate mapping directives will be added to the generated checker. In the case of force-freeze, the directives are written to a separate file, with the extension “mon” (shortcut for monitor). When using signal spy, every design signal name must be double quoted

except for generic ports, described in the next section. This is because signal spy mapping deals with strings.

3.3.4.1 Signal Spy Mapping Using Generic Ports

It is possible to define a mapping from signals to generic ports.

To map a signal to a generic port, simply write the name of the generic port, in the mapping file, without double quotes. It is also possible to use a concatenation of strings, using generic ports (exactly as you do when applying signal spy on your VHDL).

For example:

`aa,bb,cc` are checker ports.

/main/u1/aa, main/u2/bb, main/u2/cc are design-under-test signals

to which we want to map checker ports (correspondingly).

Two examples of possible mappings FoCs can generate:

a. without generic ports:

```
init_signal_spy("aa", "/main/u1/aa");
        init_signal_spy("bb", "/main/u2/bb");
        init_signal_spy("cc", "/main/u2/cc");
```

b. with generic ports:

```
generic (
              x : STRING := "";
              y : STRING := "" ) ;

          init_signal_spy("aa", "/main/u" & x & "/aa");
          init_signal_spy("bb", "/main/u" & y & "/bb");
  init_signal_spy("cc", "/main/u" & y & "/cc");
```

It is the user's responsibility to set `x` to be "1" and `y` to be "2" in checker instantiation.

**** Note: two mappings are give the same result under condition x = "1" and y = "2".***

Mapping file for (a) - without generic ports:

```
aa  "/main/u1/aa"
    bb  "/main/u2/bb"
    cc  "/main/u2/cc"
```

Mapping file for (b) - with generic ports:

```
aa  "/main/u" & x & "/aa"
    bb  "/main/u" & y & "/bb"
    cc  "/main/u" & y & "/cc"
```

**** Note: x and y appear in mapping file without double quotes so that they are interpreted by FoCs as generic ports.***

For more information about force freeze and signal spy, see the MTI documentation. The directives are derived from the mapping mechanisms described in the Signal Mapping section above. This means that you have to provide a mapping file and/or use the Design Signal Prefix option.

If the mapping method is something other than None, the generated entity, component, and instantiation have an almost empty port map because the actual signal hooking is done through the special directives. Only the clock and reset signals are explicitly referenced.

CHAPTER 4

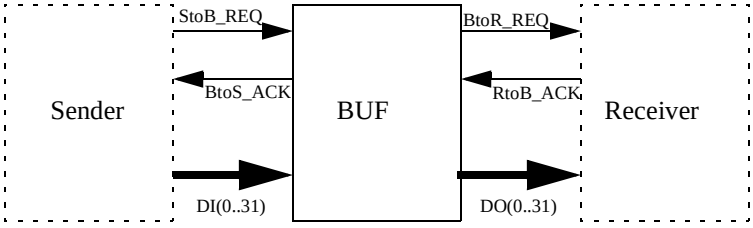
Tutorial

4.1 Introduction

This chapter guides you through an example of a simple design and how FoCs can be used to enhance its verification productivity. The tutorial presents a small design named BUF and a list of rules which the design must abide by, and shows you how to generate checkers from these rules.

4.2 Design Description

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



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

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

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

4.3 The Rules File

For the tutorial, a rules file with some rules regarding the BUF design has already been created, and can be found at: $FOCS_DIR/tutorial/rules.

The rules are written in Sugar. See Chapter CHAPTER 6: The Sugar Specification Language for more details.

The rules file contains the following rules:

```
vunit ack_interleaving {

  assert "No overflow: RTOB_ACK is asserted between any two BTOS_ACK
assertions"
```

```
  { [*] ; !RST & rose(BTOS_ACK) ; true }( rose(RTOB_ACK) before
rose(BTOS_ACK) );


  assert "No underflow: BTOS_ACK is asserted between any two RTOB_ACK
assertions"
  { [*] ; !RST & rose(RTOB_ACK) ; true }( rose(BTOS_ACK) before
rose(RTOB_ACK) );



}




vunit four_phase_handshake_left{
    assert "A request can not be raised when ack is high "
    never{ [*] ; !STOB_REQ & BTOS_ACK ; STOB_REQ };

    assert "A request can not be lowered when ack is low"
    never{ [*] ; STOB_REQ & !BTOS_ACK ; !STOB_REQ };

    assert "An acknowledge can not be raised when req is low"
    never{ [*] ; !BTOS_ACK & !STOB_REQ ; BTOS_ACK };

    assert "An acknowledge can not be lowered when req is high"
     never{ [*] ; BTOS_ACK & STOB_REQ ; !BTOS_ACK };

}
```

```
vunit four_phase_handshake_right{

    assert "A request can not be raised when ack is high"
    never{ [*] ; !BTOR_REQ & RTOB_ACK ; BTOR_REQ };

    assert "A request can not be lowered when ack is low"
     never{ [*] ; BTOR_REQ & !RTOB_ACK ; !BTOR_REQ };

    assert "An acknowledge can not be raised when req is low"
     never{ [*] ; !RTOB_ACK & !BTOR_REQ ; RTOB_ACK };

    assert "An acknowledge can not be lowered when req is high"
never{ [*] ; RTOB_ACK & BTOR_REQ ; !RTOB_ACK };

}
```

```
vunit checking_data{
   VAR tmp(0..31):boolean;
   ASSIGN init(tmp(0..31)) := 0;
   ASSIGN next(tmp(0..31)) :=
    if rose(BTOS_ACK) then DI(0..31)
    else tmp(0..31)
    endif;
```

```
    assert "The data sent to the receiver is the same data received
from the sender in the last write"
     { [*] ; !RST & rose(RTOB_ACK) }( DO(0..31) = tmp(0..31) );

}
```

4.4 Initial Setup for a Working Environment

If you are running FoCs for the first time:

1. Add the following setting to your .cshrc file (or the shell with which you are working):

   ```
   setenv FOCS_DIR location_of_focs_executable

   alias focs $FOCS_DIR/focs
   ```

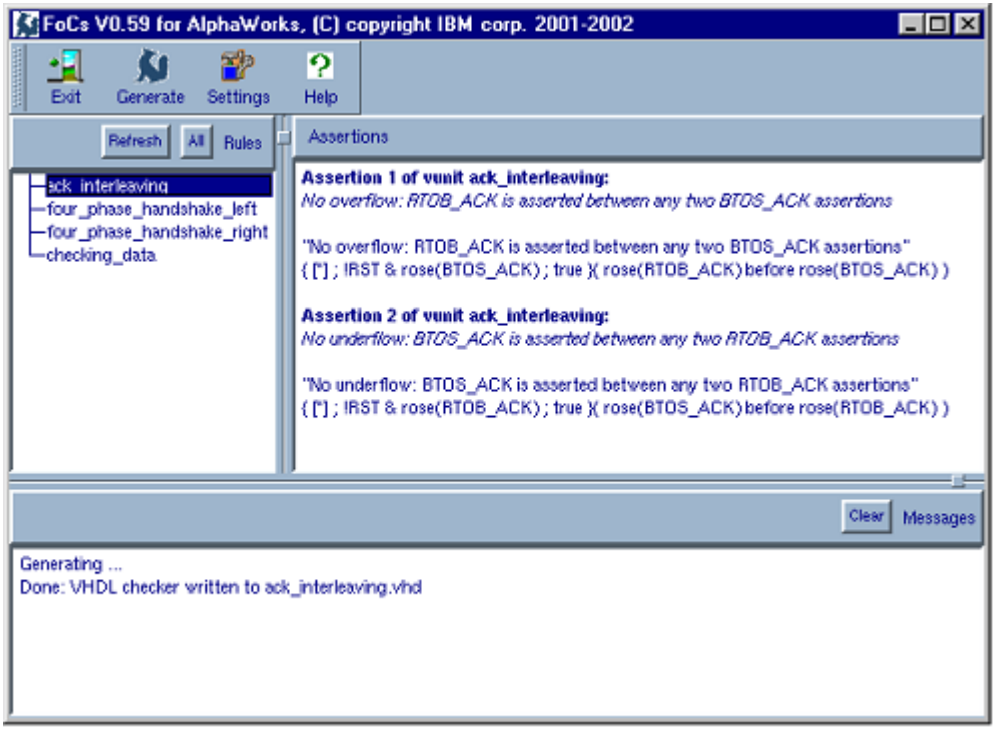
2. Create a directory called focs_tutorial.
3. Copy $FOCS_DIR/tutorial/rules into your focs_tutorial directory.
   The rules file includes four rules about BUF.
4. Invoke the FoCs GUI by typing **focs**.

4.5 Generating Checkers

The FoCS GUI has three different windows:

- **Rules window** – displays a list of rule names from the rule file (select rule file from the settings window).
- **Formulas window** – displays all formulas from the selected rules in the rules window.
- **Messages window** – displays all relevant messages from FoCs.

The following is a screenshot from the tutorial.



To display the Settings window:

1. Click **Settings**.
2. Update the **Clock Name and Reset Name** in the Clock/Reset Tab.
   You can browse through the other Settings tabs and fields if you want to have more control over the generation process. Use the tool-tips to see short field descriptions.
3. Choose the target language: VHDL/Verilog
   When you open FoCS for the first time, all settings are set to their default values.
4. Click **OK.** This saves your settings and closes the Settings window.

To generate a checker **from one rule:**

1. Select rule **ack_interleaving**
   The Formulas window displays two formulas.
2. Click **Generate**.
   One of the following messages (according to the target language) appears in the Message window:

   Generating ...

   Done: VHDL checker written to ../ack_interleaving.vhd
   or
   Done: Verilog checker written to ../ack_interleaving.v

You can open the file (ack_interleaving.vhd or ack_interleaving.v) created in your current directory, and see the checker that was generated.

To generate a checker **from several rules**:

1. Select rule **ack_interleaving**.
2. Press the Ctrl button and select rule **checking data**.
   The Formulas window displays all of the formulas for these two rules.
3. Click **Generate**.
   The **Choose an output filename** window will open.
4. Enter the name of the desired checker (**checker.vhd or checker.v)** and click **OK**.
   The following message appears in the Messages window:

   Generating ...

   Done: VHDL checker written to ../checker.vhd
   or
   Done: Verilog checker written to ../checker.v

To generate a checker **from all rules**:

1. Click **All** from the Rules window.
   This selects all rules in this window.

2. Click **Generate**.
   The **Choose an output filename** window opens.
3. Enter the name of the desired checker (all.vhd or all.v) and click **OK**.
   The following message appears in the Messages window:

   Generating ...

   Done: VHDL checker written to ../checker.vhd
   or
   Done: Verilog checker written to ../checker.v

CHAPTER 5 *Customizing FoCs Settings*

5.1 Overview

FoCs can be customized using the Settings dialog box. The Settings dialog box consists of several tabs—Main, Clock and Reset, Checker Generation Style, Reporting, and Signal Mapping—each of which contains several options. When you give an option a new value, the new value is active as soon as you set it. The value remains active for the rest of the session. When you quit FoCs, your settings are saved and used the next time you run FoCs. The options are written to the file focs.setup.

5.2 Main Tab

The following sections describe the options available within the Main tab.

5.2.1 Rules File

In this file, you write the specification—the properties to be checked. For information on the specification language, see CHAPTER 6: The Sugar Specification Language.

5.2.2 Target Language

This option allows you to select the target language for the automatically generated checkers.

- VHDL
- Verilog

Future versions of FoCs may include more target languages, e.g., C and/or C++.

5.2.3 Target Simulator (VHDL only)

Some simulation environments only support a subset of the VHDL language. Some environments include special instrumentation for linking the checkers with the unit under test (UUT). If your simulation environment appears below, select it. Otherwise, use Pure VHDL and tailor the checkers with the UUT to suit your needs.

- Pure VHDL
- MTI - FoCs generates a “spy” file used for linking checker names with UUT names. See Section 3.2 on page 12.
- Mvlsim/Bugspray - FoCs generates the checkers in a Bugspray file that are to be included in the simulation build process.

5.2.4 Output File Name

This is the name of the resulting checker file. By default, the rule name is used with the extension v (verilog) or vhd (VHDL). If you select User Defined, you will be asked to provide a name each time a checker is generated.

- Use rule name – the rule name will be used as the file name.
- Use entity name (VHDL only) – the entity name will be used as the file name.
- Use module name (Verilog only) – the module name will be used as the file name.
- User defined – you will be asked to provide a name during translation.

5.3 Clock and Reset Tab

The following sections describe the options available within the Clock and Reset tab.

5.3.1 Clock Name

This is the design clock that drives the checker's clock (mandatory). Currently only a single clock is supported.

5.3.2 Clock Polarity

This is the clock edge in which signals are sampled during simulation.

- Rising edge
- Falling edge
- Both edges

5.3.3 Simulation Delay (Verilog only)

This option allows you to add a delay of n nano-seconds between the active edge of the clock and the sampling of signals.

- No
- Yes

5.3.4 Reset Mode

This option allows you to provide an external reset to the checker or ask for an internally-generated reset.

- External – the reset signal will be provided as an input to the checker.
- Internal – the reset signal will be generated internally.

It is mandatory to either define the external reset signal name or to choose the option of an internal reset signal.

5.3.5 Checker Reset Name (External Only)

If you choose "External" for the Reset Mode option**,** you should provide the reset signal name (mandatory).

5.3.6 Number of Reset Cycles (Internal Only)

If you choose "Internal" for the Reset Mode option, you may select the number of cycles during which the internal reset signal remains active at the beginning of

simulation. The default is one cycle.

5.3.7 Reset Polarity

- Active high
- Active low

5.4 Checker Generation Style Tab

The following sections describe the options available within the Checker Generation Style tab.

5.4.1 Checker Entity Name (VHDL Only)

This is the entity name of the resulting checkers. By default, the rule name is used. You can select User Defined and provide a name.

- Use rule name – the rule name will be used as the entity name.
- User defined – you will be asked to provide a name during translation.

5.4.2 Checker Module Name (Verilog Only)

This is the module name of the resulting checker. By default, the rule name is used. You can select User Defined and provide a name.

- Use rule name – the rule name will be used as the module name.
- User defined – you will be asked to provide a name during translation.

5.4.3 Generate Module (Verilog only)

This option allows you to choose whether or not to encapsulate the checker within a module.

- Yes
- No

5.4.4 Produce Instantiation Code

- FoCs can provide automatic generation of instantiation statements—Component and Instance in VHDL, Call in Verilog. The instantiation code is produced according to the defined Mapping Options (see CHAPTER 3: Linking Checkers with your Design).
- Yes
- No

5.4.5 Interface Filename

By default, FoCs regards signals that are only referenced (not assigned) in the rules as design signals. The file defined as "Interface Filename" can be used to add signals that you want to regard as design signals. If the signal has auxiliary EDL definitions in the rules file, they will be ignored. This option is useful when using the rule file for both model-checking and checker generation. The file is either a VIM/DEF file or list of lines, one signal name in each line.

5.4.6 Design Entity Name (Mvlsim/Bugspray Simulation Only)

This option allows you to define the design entity to which the checkers refer. This is mandatory when using Bugspray.

(It produces the Bugspray code “`--!! design entity <name>`”.)

5.4.7 Vector Direction

This option allows you to define the bit order in which FoCs generates vectors. All vectors in the checker are written in the same way.

- Ascending – [0..n]
- Descending – [n..0]

5.4.8 Logic Signal Type (VHDL only)

This option allows you to choose the standard logic support package to use.

- std_logic
- std_ulogic

5.5 Reporting Tab

The following sections describe the options available within the Reporting tab.

5.5.1 Severity of Assertion (VHDL Only)

This option allows you to define the severity of the generated VHDL assertion.

- Note
- Warning
- Error
- Failure

5.5.2 Report Template File Mechanism (Verilog and VHDL-93 full)

5.5.2.1 Report template file

In order to configure the checker output, you can write a report template file. In this file, write the VHDL / Verilog commands you want executed when the formula fails (report template).

You can define different sets of commands for different formula types. You can write the report template file in the following style:

VHDL	Verilog

IF (focs_formula_type = ERROR) THEN
<VHDL code>
ELSEIF (focs_formula_type = COVER) THEN
< VHDL code>
END IF;
	if (focs_formula_type==‘ERROR)
	<Verilog code>
	else if (focs_formula_type == ’COVER)
	<Verilog code>

This style is only a recommendation. Every legal VHDL / Verilog code is allowed, because this code is simply copied to the checker. Note that there is no previous

syntactical or semantical checking done on this code, so that every error in it will be determined only when compiling the checker with the VHDL / Verilog compiler.

Both in VHDL and Verilog, two additional commands with special treatment can be used—printf and fprintf. Their semantics and usage in the report template file are defined later in this document. Verilog commands $display and $fdisplay have the same treatment as printf and fprintf and can be used for checkers generated in VHDL as well as Verilog (FoCs will create equivalent code in the checker's language).

The content of the report template file is copied to the checker inside the IF statement used instead of assertion, as shown in the pseudo-code example below:

```
IF(formula failed) THEN
      focs_formula_type <= ERROR / COVER
      report template file content
   END IF;
```

5.5.2.2 Formula type definition

You can define the type of every formula as follows:

`#type = ERROR# / COVER#.`

If no type is defined, the default type value is ERROR.

Example:

```
formula {
           "#type = COVER# Standard formula description"
           {formula body}
         }
```

When generating checker code, FoCs adds a variable named "focs_formula_type" for every formula that contains the value ERROR or COVER, according to the type defined in the formula description. ERROR and COVER are integer constants whose definition is implicitly generated by FoCs.

You may choose not to use this mechanism, by means of not mentioning the

formula type in the report template file.

The only side effect of this mechanism is the definition of the variables ERROR and COVER in the checker, which implies that redefining these variables is forbidden.

5.5.2.3 Usage and semantics of printf / $display and fprintf / $fdisplay

The syntax and semantics of the $display and $fdisplay commands are similar to those of printf and fprintf, respectively.

Syntax: printf (“format”, parameters) or fprintf (fp, “format”, parameters). (The explanation for “fp” -file pointer appears below.)

The purpose of these commands is to configure the output which is written during the simulation to the log (file or stdout). The semantics are like the C semantics of these commands. FoCs will identify these commands and generate VHDL / Verilog code which will write output to the output stream defined in print command.

The syntax of the format is like C syntax—every parameter should be defined in format by its prototype (%d, %s, etc.) and every string can be written in the format. The only difference from C is that there is no need to add a backslash before special characters (like quotation marks), and the use of double quotation marks ("") and “%”.

Allowed prototypes, inside the format string, are %s - string, %d - integer, %b - boolean, and %l - std_logic. Print commands should start at the new line, cannot be broken by the new line in the middle, and must not end with a semi-colon (“;”). Among the parameters, there are two parameter names with special semantics—`rule_name` and `desc` (description), both of which are strings. When FoCs finds these parameters in the printf command, the following is performed:

1. `rule_name` will be replaced in the format string by the appropriate rule name.
2. `desc` will be replaced in the format string by the description of the appropriate formula. In the formula description, you can define signals whose values you want outputted when the formula fails. The syntax is: `< <signal_name> >`. Every signal which appears in the formula description in <> brackets is replaced by its current value in the simulation output when the formula fails.

For example:

Formula description:

```
formula {
```

```
    "#type = COVER# Values of signals : foo1 = <foo1>"
   { formula body}
}
```

FoCs finds the signals in <> brackets in the description, and adds them to the parameters list of the printf command. If the signal is not a checker signal (neither a checker port nor checker internal signal), its type is assumed to be std_logic in VHDL, or boolean in Verilog.

Note: The use of the special names rule_name and desc is optional.

5.5.2.4 Writing output to the file streams

There are two options for defining file descriptors:

Option 1: The user controls file opening and closing outside the checker. There is no special definition for file descriptors—FoCs finds them in the fprintf commands and adds them to the checker ports (as port of type FILE in VHDL or port of type reg[31:0] in Verilog). In this option, the user opens and closes the files, and maps the file descriptors to the appropriate checker ports.

Option 2: Implicit file opening and closing within the checker. In this option, the user defines the file descriptors connections in the file section of the report template.

The syntax of the file section:

```
#FILES
   <file descriptor>    <file name on disk>
   <file descriptor>    <file name on disk>
#END
```

i.e, every line in the file section consists of the file descriptor, which can be used in print commands as described above, and the file associated with it. For example:

```
#FILES
   error_fd     /proj/simout/error_file
    cover_fd    /proj/simout/cover_file
#END
```

FoCs generates the VHDL / Verilog code for opening and closing files and connecting them to their descriptors. There is no need to predefine standard output stream. The commands printf and $display create the code which writes output to stdout according to the HDL subset.

When using implicitly-generated opening/closing of files, FoCs generates the port “focs_finish_signal”.

The user must connect this port to the signal which gets the value “1” when the simulation ends. The purpose of this port in the checker is to tell the checker when to close the files.

5.5.2.5 Including files, libraries, and packages

When describing the use of file descriptors, the term file section was mentioned. There are three special sections which can be defined in report templates: the library section, the use section, and the file section. This paragraph will describe the use of the library section and the use section.

Syntax: library section

```
#LIBRARY
   <library 1>
   <library 2>
#END
```

Syntax: use section

```
#USE
    <package 1>
    <package 2>
#END
```

In VHDL, the library section contains libraries, which the user wants defined in the final VHDL code by the “library” clause (like ieee), while the use section contains the full package names, including the appropriate library name (for example, ieee.std_logic_1164.all). In the use section in Verilog, the user must write Verilog file names which he wants included in the final Verilog code by using the Verilog

compiler directive “include”. The library section is ignored in Verilog.

The main purpose for including external libraries in checker code is to allow the use of functions and variables defined in these libraries. If the function was defined in the external package (file in Verilog) and the package was included, the user can use this function in his/her report template.

FoCs does not perform syntactical or semantical checks on the included user packages (files), so if the package contains errors, or if the parameters to the function call in the report template are not checker signals or predefined signals in included packages (files), FoCs will not report any error, but the created checker will fail during the compilation.

For VHDL users only: the only libraries which should not be defined in the library sections are ieee, ibm, work, modelsim_lib, and std. These libraries are included in the checker automatically. The packages which are included automatically are: ieee.std_logic_1164.all, modelsim_lib.util.all, ibm.std_logic_support.all, and std.textio.all. Each library / package is included according to FoCs settings, so that not all these packages are included in every checker.

The use of the libraries section, the use section, and the files section is optional. You can write as many library/package/file sections as you wants. Every new library/ package/file descriptor name should appear on a new line. The report template directives (#LIBRARY, #USE, #FILE, #END) should also appear on a new line. All text outside these special sections is treated as report template, and is copied to the checker.

Comments

Lines which starts with “--” are treated as comments and thus are ignored by FoCs (either in the report template or in the special sections).

5.5.2.6 Examples

1. The following is an example for output configuration using the report template file and formula description:

VHDL example:

Report template file:

```
#LIBRARY
    utils1                          Libraries section
    utils2
#END

#USE
    utils1.print_functions_package1.all          Use section
    utils2.print_function_package2.all
#END

#FILES
     cover_file    /proj/simulation_output/cover        File section
#END
```

Template body:

```
    IF ( focs_formula_type = ERROR ) THEN
    printf ( " ERROR in %s at cycle %d : %s ", rule_name, cycle, desc)
cycle is signal defined by user in one of the packages
     error_func();           error_func is the function written in one
of the predefined packages
    ELSEIF ( focs_formula_type = COVER )
      fprintf (cover_file," COVERAGE EVENT : rule %s at cycle %d : %s
", rule_name, cycle, desc)
    END IF;
```

Formula description:

```
    formula {
```

```
      "#type = COVER# Values of signals : foo = <foo>"
     formula body
   }
```

Generated code:

Assuming that the rule name is `CheckRule` and `foo` is of type `std_logic`, the VHDL code generated by FoCs will be:

```
 .
 .   other libraries
 .
 library utils1;                      Library section
 library utils2;
 .
 .   other packages
 .
use utils1.print_functions_package1.all;       Use section
use utils2.print_function_package2.all;




ENTITY ...
 PORT (
   .
  .   checker ports
  .
  foo        :IN std_logic;
     focs_finish_signal :IN std_logic);      Special port added for
```

```
file commands
  END ...


  ARCHITECTURE ...


      TYPE focs_ftype is (ERROR,COVER);


    .
    .   checker internal signals
    .
       FILE cover_file : TEXT;
   SHARED VARIABLE focs_string_0 : STRING(1 TO 10) := " ERROR in ";
    SHARED VARIABLE focs_string_1 : STRING(1 TO 10) := " at cycle ";
   SHARED VARIABLE focs_string_2 : STRING(1 TO 3) := " : ";
   SHARED VARIABLE focs_string_3 : STRING(1 TO 1) := " ";
   SHARED VARIABLE focs_string_4 : STRING(1 TO 23) := " COVERAGE EVENT
: rule"          ;
   SHARED VARIABLE focs_string_5 : STRING(1 TO 10) := " at cycle ";
   SHARED VARIABLE focs_string_6 : STRING(1 TO 3) := " : ";
   SHARED VARIABLE focs_string_7 : STRING(1 TO 1) := " ";
    SHARED VARIABLE focs_checkrule : STRING(1 TO 9) := "checkrule";
   SHARED VARIABLE focs_string_8 : STRING(1 TO 27) := "Values of
signals : foo = "
     ;
   SIGNAL   focs_file_handle_enable_0 : std_logic;
   SIGNAL   focs_file_open : std_logic := i2l( 1 );
    .
```

```
   .   checker internal signals
   .

BEGIN
 PROCESS               Process for files opening / closing
 BEGIN
   WAIT UNTIL clk'EVENT AND clk = '1';
    IF (focs_file_open = '1') THEN
      focs_file_open <= '0';
     file_open(cover_file,"/proj/simulation_output/cover",WRITE_MODE);
      focs_file_handle_enable_0 <= '1';
    END IF;
    WAIT UNTIL focs_finish_signal = '1';
    IF (focs_file_open = '0') THEN
      focs_file_handle_enable_0 <= '0';
      file_close(cover_file);
    END IF;
 END PROCESS;


  PROCESS
      .
      .  other process variables
      .
      VARIABLE focs_line_1 : LINE;
      VARIABLE focs_formula_type  : focs_ftype;
     BEGIN
```

```
        .
        .    VHDL checker body
        .
    focs_formula_type := COVER;
   IF ( < formula failure condition>) THEN        Report template
     IF ( focs_formula_type = ERROR ) THEN
      WRITE(focs_line_1,focs_string_0);         Translation of print
command
       WRITE(focs_line_1,focs_checkrule);
              WRITE(focs_line_1,focs_string_1);
              WRITE(focs_line_1,cycle);
              WRITE(focs_line_1,focs_string_2);
              WRITE(focs_line_1,focs_string_8);
              WRITE(focs_line_1,to_bit(foo));
              WRITELINE(OUTPUT,focs_line_1);
       error_func();
           ELSEIF ( focs_formula_type = COVER )
              WRITE(focs_line_1,focs_string_4);
              WRITE(focs_line_1,focs_checkrule);
       WRITE(focs_line_1,focs_string_5);
              WRITE(focs_line_1,cycle);
              WRITE(focs_line_1,focs_string_6);
              WRITE(focs_line_1,focs_string_8);
              WRITE(focs_line_1,to_bit(foo));
              WRITELINE(cover_file,focs_line_1);
           END IF;
       END IF;
    END PROCESS;
 END ...
```

Verilog example:

Report template file:

```
 #USE
       print.v
   #END

   #FILES
       cover_file    /proj/simulation_log/cover
   #END

  if (focs_formula_type == ‘ERROR) begin
  $display (" ERROR in %s at time %d : %s ", rule_name, $time, desc)
end
  else
    if (focs_formula_type == ‘COVER) begin
  $fdisplay(cover_file," COVERAGE EVENT : rule %s at time %d : %s ",
rule_name,
 $time,desc)
    end
```

Formula description:

```
    formula {
   "#type = COVER# Values of signals : foo = <foo>"
   formula body
 }
```

Generated code:

Assuming that the rule name is `CheckRule` and `foo1` is of type `boolean`

The Verilog code generated by FoCs will be:

```
module ... (
      .
      .   checker ports
      .
      foo,
      focs_finish_signal
 );
      .
      .    checker signals
  .
  input foo;
      reg focs_file_handle_enable_0;
      input focs_finish_signal;

‘include "print.v"
 ‘define ERROR 1
 ‘define COVER 2
      initial
```

```
       begin
      cover_file=$fopen("/proj/simulation_log/cover");
      if (cover_file==0)
      begin
       $monitor("Fatal error : Can't open file /proj/simulation_log/
cover");
       $finish;
      end
           focs_file_handle_enable_0 <= 1'd1;
           wait (focs_finish_signal);
           focs_file_handle_enable_0 <= 1'd0;
           $fclose(cover_file);
       end
       .
       .   checker body
       .
       if ( < formula fail condition> ) begin :assert0
           integer focs_formula_type = 'COVER;
           if (focs_formula_type == 'ERROR) begin
       $display(" ERROR in CheckRule at time %0d : Values of signals :
foo1 = %b",$time,foo);
           end
           else
               if (focs_formula_type == 'COVER) begin
        $fdisplay(cover_file," COVERAGE EVENT : rule CheckRule at time
%0d : Values of signals : foo = %b",$time,foo);
        end
       end
  endmodule
```

2. VHDL Example: counting events and stopping the simulation, using report template file:

Report template file:

```
#library
          utils
     #end
     #use
          utils.countevents.all
   --Package utils.countevents contains definition of integer
counter with initial value 0
   --and the definition of constant STOP_VALUE
#end
     IF (focs_formula_type = ERROR) THEN
         counter <= counter + 1;
         IF (counter = STOP_VALUE) THEN
              < stop the simulation >
  END IF;
     END IF;
```

5.5.3 Maximal Number of Fails

This option allows you to limit the number of reported errors for one formula (during run time), inactivating the relevant part of the checker. Moreover, the formula will be disabled after the defined number of fails.

5.6 Signal Mapping Tab

The following sections describe the options available within the Signal Mapping tab.

5.6.1 Mapping File

It is possible to define a mapping file, that defines a mapping between checker ports and the actual design signals. For more details, see Section 3.2 on page 12.

5.6.2 Mapping Method (VHDL – MTI Simulator only)

Selecting a Mapping Method allows you to select which method to use for mapping checker signals to design signals.

- Signal spy
- MTI- force freeze
- None

5.6.3 Design Signals Prefix

If design signals are nested and there is a common path for all signals, it is possible to define the common path here. In the mapping of checker ports to design signals, this path will be added as a prefix to the names of design signals. See examples in CHAPTER 3: Linking Checkers with your Design.

5.6.4 Checker Signals Prefix (MTI – force-freeze only)

When using force-freeze, specify the location of the checker relative to the monitor file.

5.6.5 Warn Incomplete Mapping

FoCs can supply warnings about checker ports that do not have a mapping defined in the mapping file, or complete these values itself by mapping such checker signals to signals with the same name (with the design signals prefix if such was defined).

- Yes – FoCs will warn about signals that do not have a mapping defined for them.
- No – FoCs will create a default mapping for signals that do not have a mapping defined for them.

CHAPTER 6

The Sugar Specification Language

6.1 Introduction

Sugar is the specification language used by FoCs. It is used to describe properties that are required to hold in the design under test (DUT). FoCs can very easily create powerful checkers from Sugar properties using only a small number of Sugar constructs.

In this document, the term 'property' refers to a specification, described in informal English, that must hold true for the design under simulation. The term 'formula' refers to the coded representation of properties in Sugar[†].This chapter introduces Sugar in a way that allows FoCs users to start describing properties in Sugar and generating checkers from those properties simply.

The chapter is organized as follows:

Section 5.2: Getting Started with Sugar – introduces the basic Sugar constructs. This is an informal description of these constructs, designed to give you the notion of how to describe the required design behaviors using Sugar.

†. In the Sugar documentation, the term 'property' refers to both the coded representation and the informal English statement. For the sake of clarification, in this book the term 'property' refers to the informal English statement, and the term 'formula' refers to the coded formulation of the Sugar property.

Section 5.3: The Building Blocks of a Sugar Formula – this is a more precise description of the Sugar properties building blocks.

Section 5.4: Writing a Rules File – describes the notion and structure of the Rules File, from which FoCs reads Sugar properties to create checkers.

Section 5.5: State Machines – this section describes how to use a larger subset of the Sugar language and gain further expressive power, for defining complex properties.

6.2 Getting Started with Sugar

The Sugar specification language lets you describe properties to which the design under simulation must adhere. Many properties can be easily described using the following constructs.

6.2.1 Always (p)

This Sugar construct enables you to assert that some property p is true on every cycle of the simulation. P can be any boolean expression composed of signal names, constants, and operators.

For example, you may want to check that signals `grant1` and `grant2` are not asserted together. This property can be expressed in Sugar by the following formula:

```
always (!(grant1 & grant2))
```

which states that at every cycle of a simulation, it is never the case that both `grant1` and `grant2` are asserted.

The following are more examples of properties that can be expressed using **always**(p):

- The property “Whenever `ack` is asserted, `req` was asserted in the previous cycle,” can be expressed as

  ```
  always (ack->prev(req))
  ```

  prev(x) is a built-in function, which is true if x was true in the previous cycle. A list of built-in functions you can use appears in Section 6.3.3 on page 64.
- "Variables `state1` and `state2` never have the same value."

  ```
  always (state1 != state2)
  ```

  “`!=`” denotes inequality. A list of relational operators appears in Table 2.
- "If `busy` is true then `working` is also true"

```
always (busy -> working)
```

"->" denotes "implies".

"At most one of the signals `x, y,` or `z` is 1 (mutual exclusion)."

```
always (x+y+z <=1)
```

A list of mathematical operators appears in Section 6.3.1 on page 59.

- "If the `head` pointer of a queue is equal to the `tail` pointer, `queue_empty` must be true":

  ```
  always((head(0..3)=tail(0..3)-> queue_empty))
  ```

  Both `head` and `tail` are 4-bit arrays and the expression `head(0..3)=tail(0..3)` denotes equality of arrays. The symbol ".." denotes a range of array bits. `head(0..3)` denotes bits 0 through 3 in the array `head`. You can refer either to a range of entries of an array `head (0..3)` or to one entry of an array (e.g., `head(2)`). A reference to a whole vector should explicitly include its range (`vec1(0..16)` rather than `vec1`).

- "The bitwise and of vectors `vec(0..7)` and `mask(0..7)` has at least one bit set":

  ```
  always((vec(0..7) & mask(0..7)) != 00000000b)
  ```

  We need the parentheses around `vec(0..7) & mask(0..7)` because "`!=`" has a higher precedence than "`&`". Table 1 shows the operators precedence.

6.2.2 never (p)

The construct **never**(p) allows us to specify conditions that should never hold. For example, to express that the signals `enable1` and `enable2` are mutually exclusive, it is possible to write

```
never (enable1 &enable2)
```

6.2.3 The next operators

It is possible to define that some property should hold at some next cycle. This is done by defining the following formula:

- **`next (p)`** - the property p should hold at the next cycle

- **`next [N] (p)`** - the `Number N` indicates at which next cycle the property p should hold, that is for `Number i` the property holds at the ith cycle.

For Example:

- always (request -> next(acknowledge))

  States that a request is always followed by an acknowledge on the next cycle.
- always (request -> next[3](acknowledge))

  States that a request is always followed by an acknowledge after 3 cycles.

It is possible to define that a property holds at all cycles of a range of cycles by defining:

- **`next_a [i : j] (p) -`** `property p` holds at all cycles between `ith` and `jth` next cycles, inclusive.

It is also possible to specify that a property holds at least once in a range of future cycles.

- **`next_e [i : j] (p)`** - `property p` holds at least once between `ith` and `jth` next cycles, inclusive.

6.2.4 Sugar Extended Regular Expression – SERE

The construct **always(p*)*** can refer only to an expression that spans one cycle. Sometimes we want to check events that span over a period of time, and not just one cycle.

To this end, we can express properties of multi-cycle traces using Sugar Extended Regular Expressions—SEREs. SEREs can be used to describe sequences of boolean expressions over time.

For example, a SERE describing any occurrence of `start,ready,ready` can be written as: `{[*];start;ready;ready}`.

The `[*]` at the start of the sequence is an event that denotes “skip any number of cycles”; `start` between two semi-colons (`;start;`) means a cycle in which `start` is asserted, and in the following two cycles, `ready` is asserted. So this sequence represents many possible traces. For example:

- traces in which `start` is asserted at the beginning and then followed by two `ready`: `start,ready,ready`,...

- traces in which `start` happens at the second step and then followed by two `ready`: `true,start,ready,ready` (true represents “skip one cycle”), etc.

If we omit `[*]`, the sequence would describe only traces that start with `start,ready,ready`. It is important to write `[*]` at the beginning of the sequence because we usually want to refer to any occurence of `start,ready,ready` in the trace, and not just at the beginning of the execution.

Writing SEREs is an extension to writing regular expressions. Regular expressions are simple to write but still very expressive. Defining a regular expression is very intuitive, when keeping the desired timing diagram in mind.

6.2.4.1 The constructs {SERE} |-> {SERE} and {SERE} |=>{SERE}

SERE’s can be used to described properties that span several cycles. For example, the property “whenever the sequence `start, ready, ready` appears, the second `ready` after `start` is accompanied by `result = ok` and followed by `done`” can be written as

```
{[*]; start; ready; ready} |->{result=ok ;done}
```

The meaning of this construct is as follows: if the sequence on the left side is encountered during simulation, then the right side should be true, starting at the last cycle of this sequence.

It is possible to use two kinds of implication operators. Using `|->` between the two sequences means that the right hand sequence must begin at the last cycle of the left hand side. Using `|=>` means that the right hand side sequence must begin one cycle after the left hand side.

The following are examples of properties that can be expressed using SEREs implication:

- `{[*];start; busy[*]; end}|->{success; done}`
  states that if signal `start` is asserted, at the next cycle or later in the future, signal `end` is asserted, and in the interim signal `busy` holds, then `success` is asserted with end, and at the next cycle `done` is asserted.
- `{[*];request}|=>{request[*]; grant}`
  states that if there is a `request`, than it must remain active until `grant`.

- All properties in the form of {SERE} |-> {SERE} and {SERE} |=> {SERE}, can be described in English as "If the right side occurs, then the left side must occur." For example, the property "If during get tag=1, then in the next get tag=2 and in the next get tag=3," is expressed by:

  ```
  { [*] ; get&tag=1 }|=>{ !get[*];  get&tag=2; !get[*] ; get&tag=3 }
  ```

  And the property "If during get tag=1 and in the next get tag=2 then in the next get tag=3," is expressed by:

  ```
  { [*]; get&tag=1; !get[*]; get&tag=2}|=>{ !get[*]; get&tag=3 }
  ```

- When we want to express that a signal p is asserted at least once, we can use the event p[+].

  For example, {[*]; start; busy[+]; end}|->{success; done}
  states that if signal start is asserted, signal busy is asserted for one or more cycles, and finally signal end is asserted, then success is asserted with end, and followed by done.

- Instead of writing true[+], you can write [+].

  {[*]; start; [+]; end}|->{success}
  states that if signal start is asserted, and two or more cycles later signal end is asserted, then success is asserted with end, followed by done.

- There is a special way to describe an exact number of consecutive repetitions. For example, writing ready[*8] expresses eight consecutive cycles in which ready is asserted.

  {[*]; start; ready[*8]}|->{result=ok}

  states that if start is followed by eight consecutive cycles in which ready is asserted, then at the eighth ready result=ok.

  To say that ready is asserted at most eight consecutive cycles we write ready[*..8].

  {[*]; start; ready[*..8]; !ready}|->{ok} states that if start was asserted, and starting the next cycle, ready was asserted for at most eight cycles until ready was false, then ok is asserted with !ready. The other types of consecutive repetitions are listed in Table 2.

- Assume we want to write "the second ready after start should be accompanied with success." Here we want to check also executions in which the start and the ready signals are not necessarily consecutive. This can be expressed by

  ```
  {[*]; start; !ready[*]; ready; !ready[*]; ready}
  |->{success}
  ```

The expression `!ready[*]` means zero or more consecutive steps in which ready is false.

- There is a special shorthand for non-consecutive repetition. For example, the property "Whenever we see eight non-consecutive data transfers between `start_trans` and `end_trans`, the signal `error` is not asserted with `end_trans`" can be described by:

  ```
  {[*];start_trans;data[=8];end_trans}|->{!error}
  ```

  The event `data[=8]` represents eight non-consecutive repetitions of `data`.

6.2.4.2 Subsequences

A sequence may contain a subsequence in curly braces.

- For example the sequence

  `{[*]; {start; [*]; end}[*]}` describes a trace in which there are zero or more occurrences of the scenario in which `start` is asserted and in the next cycle or later `end` is asserted.

- The other consecutive repetitions can also be applied to subsequences. For example:
  `{[*]; {start; [*]; end}[+]}` describes a trace in which the subsequence `{start,[*],end}` occurs at least once.
- `{[*]; {start; [*]; end}[*8]}` describes a trace in which the subsequence `{start; [*]; end}` occurs exactly eight times.
- `{[*]; {start; [*]; end}[*..8]}` describes a trace in which the subsequence `{start; [*]; end}` occurs at most eight times.

The following are examples that use subsequences:

- After `start`, if we see 8 `gets` (not necessarily consecutive) and there is no `abort` during this period, then one cycle after the last `get` must begin 8 `puts` (not necessarily consecutive before `done` (`done` may not come at all)

  ```
  { [*]; start; {!get&!abort[*]; get&!abort}[*8] }|=>{ {!put&!done[*]; put&!done}[*8] }
  ```

- If `req1` is active, it will be granted (`gnt1`) within no more than 7 `gnts`

  ```
  { [*]; req1 }|->{ {!gnt[*]; gnt&!gnt1}[*0..6]; !gnt[*]; gnt&gnt1 }
  ```

- `p` is true in cycle 0 and every fifth cycle

  ```
  { true; {true[*4]}[*] }=>{ p }
  ```

For details on all subsequence options, see Section 6.3.2.2 on page 62.

6.2.4.3 The | and && operators

It is possible to define an AND and OR relation between subsequences, using the | and &&, respectively.

- For example, to express the property:

  "If there is a `request` that is followed either by `read` and no `cancel_read` or `write` and no `cancel_write` until `done`, then `ok` is asserted with `done`"

  it is possible to write:

  ```
  {[*]; request; {read; !cancel_read; !done[*]}|{write; !cancel_write;
  !done[*]}; done} |-> {ok}
  ```

Writing an or between two subsequences means that only one of them must happen for the sequence to hold. The subsequences may be of different length.

- To express the property:

  "If there is no `abort` during `{start; a; b; c[*]; d; end}`, `success` will be asserted with `end`"

  it is possible to write:

  ```
  {[*]; {start; a; b; c[*]; d; end}&&{!abort[*]}}|->{success}
  ```

The subsequences that have a && operator between them occur simultaneously. This means that they begin and end together, and must have a non-contradicting length.

The && operator cannot appear in the right side sequence.

6.2.4.4 Applying never to SERE's

We can use a sequence to check that a bad trace never happens. This is done by applying the **never** operator to the forbidden sequence.

Examples:

- If `request` is asserted it will remain active until (inclusive) `grant`.

  ```
  never{[*]; ,request& !grant; !grant[*]; !request}
  ```

- If `request` is asserted, it will remain active until (not inclusive) `grant`:

  ```
  never{[*]; request; !grant[*]; !request & !grant}
  ```

- If `grant` is active, and there is no `retry` in the next cycle, `busy` must become active two cycles after `grant`:

  ```
  never {[*]; grant; !retry; !busy}
  ```

6.2.4.5 Counting Boolean events

It is possible to count boolean events that are not necessarily consecutive. This is particularly useful when combined with &&.

Examples:

- Request will be serviced within the 5 coming `acknowledges`

  ```
  never{[*]; req; ack[=5]&&serv[=0]}
  ```

- It is forbidden to have 15 `writes` during which there are less then 2 `reads`.

  ```
  never{[*]; write[=15]&&read[<2]}
  ```

6.2.5 Suffix Implication {SERE}()

It is possible to combine SEREs with temporal properties. This form of writing means that starting from the last cycle of the sequence, the temporal property must hold.

6.2.5.1 {SERE}(p until q)

Using this construct, it is possible to check that some signal holds until another signal is asserted. For example: "Between a `request` and its `acknowledge` the `busy` signal must remain asserted" can be described by

```
{[*]; req; true}(busy until ack)
```

`{SERE} ( a until b)` means `a` should be true on the last cycle of the sequence and continue to be true until (but not including when) `b` is true. `a` and `b` are boolean expressions, and `b` may never happen.

`{SERE} ( a until_ b)` means `a` should be true on the last cycle of the sequence, and continue to be true until (and including when) `b` is true. `a` and `b` are boolean expressions, and `b` may never happen.

For example:

- `{[*]; req; !retry; !retry}(busy until end)`
  states that for every request (assertion of signal `req`) that is not retried (signal `retry` is not retried in the next two cycles), signal `busy` must be asserted until signal `end` is asserted.

6.2.5.2 The Construct {SERE} (p before q)

If we want to describe that a specific signal must be asserted before another signal, we can use the **before** operator. For example: "Always if `req` then `ack` will happen **before** the next `req`".

```
{ [*]; req; true}(ack before req)
{SERE} ( a before b)
```

expresses the requirement that on the last cycle of all traces that match `SERE`, the first occurrence of `a` must happen before the first occurrence of `b`. There is no requirement that `b` eventually happen.

```
{SERE} ( a before_ b)
```

expresses the requirement that on the last cycle of all traces that match `SERE`, the first occurrence of `a` must happen before or together with the first occurrence of `b`. There is no requirement that `b` eventually happen.

For example:

- If `start` is activated, it must not be active again before `stop` is activated.

  ```
  { [*]; start; true }( stop before start)
  ```

The `true` at the end of the sequence skips one cycle because we want to check the property on the cycle after start.

6.3 The Building Blocks of a Sugar Formula

6.3.1 Boolean Expressions

The basic building blocks of a Sugar formula are Boolean expressions. A Boolean

expression consists of signal names of the design under verification, numbers, constants, and operators.

6.3.1.1 Signal Names

For integers i and j, the following are the signal names:

- a simple name: `signal_name`
- A bit of a vector: `signal_name(i)` (bit i of signal `signal_name`)
- `signal_name(i..j)` (bits i through j of signal `signal_name`)
- `signal_name(b: i..j)` (bit b of signal `signal_name`, where the range is given by i and j. b must be an integer, and the relevant signal must be defined in the rules file)

6.3.1.2 Numbers

- A *decimal number* 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 less than eight hexadecimal digits, and ends with 'H' (e.g., 7FFFH, 0FFH)

A reference to a whole vector should explicitly include its range (`vec1(0..16)` rather than `vec1`).

6.3.1.3 Operators

The operators appearing in boolean expressions in decreasing precedence are described below.

TABLE 1. Operators

`()`	parentheses
`!`	not
`*  /`	multiplication and division
`+  -`	addition and subtraction
`= != > < >= <=`	relational operators
`&`	Boolean and
`|`	Boolean or
`xor`	Boolean xor
`<->`	Boolean implication
`<<  >>`	vector shift; the right operand should be an integer

Examples:

- `request` is a boolean expression asserting that the environment signal `request` is set.
- `op=READ | op=WRITE` is a boolean expression asserting that the design signal `op` currently has either the value `READ` or the value `WRITE`.
- `(counter>32) <-> queue_is_full` is a boolean expression asserting that the user-defined signal `counter` has a value greater than 32, if and only if the design-signal `queue_is_full` is asserted.

6.3.2 Temporal Properties

6.3.2.1 Temporal Constructs

The temporal constructs of Sugar used by FoCs are summarized below. These constructs provide for the definition of temporal behavior across multiple cycles.

- **always** (p) – p is true on every cycle
- **never**(p) - p is forbidden on every cycle.
- **next**(p) - p should hold on the next cycle.
- **next**[i](p) - p should hold on the ith next cycle.
- **next_a**[i:j](p) - p should hold between the next i to j cycles, iclusive.
- **next_e**[i:j](p) -`property` holds at least once between `ith` and `jth` next cycles, inclusive
- **{SERE}** |-> **{SERE}** – if the left side sequence occurs, the right side sequence must hold, starting from its last cycle.The formula does not require that the second SERE will conclude.
- **{SERE}** |=> **{SERE}** – if the left hand side sequence occurs, the right hand side sequence must hold, starting from the following cycle. The formula does not require that the second SERE will conclude.
- **never{SERE}** – The sequence may never occur.
- **{SERE}** (p **until** q) – Starting at the last cycle of the sequence, p must hold until q occurs, not including the cycle q is asserted. The formula does not require that q must eventually occur (in that case, p must be true forever).
- **{SERE}** (p **until_** q) – Starting at the last cycle of the sequence, p must hold until q occurs, including the cycle q is asserted. The formula does not require that q must eventually occur (in that case, p must be true forever)
- **{SERE}** (p **before** q) – Starting at the last cycle of the sequence, p must happen before the first q. The formula does not require that q eventually happen.
- **{SERE}** (p **before_** q) – Starting at the last cycle of the sequence, p must happen before or together with the first q. The formula does not require that q eventually happen.

For the full Sugar 2.0 documentation, see http://www.haifa.il.ibm.com/projects/verification/sugar/fp_lrm_0912.pdf.

6.3.2.2 Sequence Operators

The building blocks of sequences are boolean expressions and operations on them. The following tables summarize the possible operators.

Using b to represent a boolean expression, Table 2 lists legal operators and their meaning.

TABLE 2. Simple Operators

`b[*]`	`b` occurs in 0 or more consecutive cycles
`b[+]`	`b` occurs in one or more consecutive cycles
`b[*i]`	`b` occurs in exactly `i` consecutive cycles
`b[*i..j]`	`b` occurs in at least `i` consecutive cycles, but in no more than `j` cycles
`b[*i..]`	`b` occurs `in i` or more consecutive cycles
`b[*..i]`	`b` occurs in no more than `i` consecutive cycles
`[*]`	Zero or more cycles are skipped
`[+]`	One or more cycles are skipped
`[*i]`	i cycles are skipped
`[*i..j]`	At least i cycles, but no more than j cycles, are skipped
`[*i..]`	At least i cycles are skipped
`[*..i]`	At most i cycles are skipped

`b[=i]`
`b[=i..j]`
`b[=i..]`
`b[=..i]`
	A sub-sequence in which `b` occurs the number of times indicated (not necessarily consecutively).

TABLE 3. Subsequence Operators

If sere, sere1, and sere2 represent sequences, then the below are possible operators:

Operator	Name	Description
`sere1&&sere2`	And	Both sere 1 and sere2 occur simultaneously
`sere1|sere2`	Or	Either sere1 occurs or sere2 occurs
`sere[*]`	Any repetition	sere occurs 0 or more times.
`sere[+]`	Positive repetition	sere occurs one or more times.
`sere[*i]`	Exact repetition	sere occurs exactly i times.
`sere[*i..j]`	Range repetition	sere occurs at least i times, but not more than j times.

Operator	Name	Description
`sere[*i..]`	At least repetition	sere occurs i or more times.
`sere[*..i]`	At most repetition	sere occurs no more than i times.

6.3.3 Built-in Functions

Sugar has several built-in functions, which are described below:

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.

prev(x) is true if x was true in the previous cycle.

next (x) is true if x is true in the next cycle. This construct can not be used in rules, since rules shouldn't relate to the future. It can be used when defining auxiliary variables and behaviors, as is explained in section 5.5.

6.4 Writing a Rules File

FoCs is rules (now called vunit) oriented. A vunit is the basic component for which FoCs can generate a checker. A vunit defines a group of related properties, represented as Sugar formulas, which are translated into one checker. (It is possible to translate multiple rules into one checker.) It is also possible to define auxiliary variables and state machines by defining Finite State Machines (FSMs), as explained in Section 5.5, and to include macro definitions, as explained in Section 5.4.3.

6.4.1 The Structure of the Rules File

Before beginning, you should plan the hierarchical structure of the rules files and how it can best represent the design properties.

A rules file consists of a set of verification units (vunits), where each verification unit may include one or more assertions (at least one assertion), macros, and FSM statements. An assertion is a Sugar formulation of a property of the design at hand. Macros and FSM statements can be used to define auxiliary variables and state machines that ease expressing properties.

The structure of the rules file is:

```
vunit vunit_name1 {
```

```
  <Macro definitions (#define,%if,%for)>
  <FSM statements (var, assign, define, module, instance)>

  assert
    “textual description”
     Sugar-formalation  ;

  assert
    “textual description”
     Sugar-formalation  ;
  ...

}

vunit vunit_name2 {
  ...
}
...
```

The vunit names are any meaningful names that begin with a letter and consist of letters, digits, or underscores. The textual description is text delimited within double quotes that describes the property in informal English†. Every assertion may have a textual description associated to it. This description must appear after the **assert** keyword. The description is followed by the Sugar formulation of the property, that ends with a semicolon (“;”).The syntax of the FSM statements and the macros are defined in the following sections of this chapter.

The rule syntax is as follows:

```
vunit name {
    <Macro definitions (#define,%if,%for)> (optional)
   <FSM statements (var, assign, define, module, instance)>
(optional)
```

†. Textual description in double quotes is currently only supported for Sugar 2.0. for FoCs, and is not part of the language definition.

```
assert “textual description”  Sugar-formulation ;
assert “textual description” Sugar-formulation ;
...

}
```

A vunit must contain at least one assertion. All the other parts are optional. The order of statements in a vunit is unimportant, and each type of a statement may appear several times. It is important to fill in the textual description of formulas. It is possible to have this description displayed when the checker generated from this formula detects an error during simulation.

6.4.2 A Methodology for Writing Rules Files

When designing a rules file, you should consider both readability and efficiency issues.

- Go over the block outputs, one by one, and write (in English) all the things you can check on that signal—its shape, its valid values, its relations with other signals, etc.
- From a methodological standpoint, it is useful to divide rules into three levels:
  Level 1: Every signal to itself (e.g., `pulse, constant zero`...).

  Level 2: Relations between signals at the same interface (e.g., `request, ack`).

  Level 3: Cross-design signals or very complex rules.
- Write an English explanation for every verification unit, specifying exactly what you are checking. If you have an English documentation of the verification units—use the same description for both, and write the rule names in the relevant places in the documentation as well.
- Keep your rules file as readable and simple as possible—you’ll return to them when you don’t expect it! Using advanced mechanisms such as “module” and “%for” is not always the most readable approach.
- Write your verification units short as well—A long/complex verification unit can be easily fragmented. It’s good for generic reasons and also for readability.
- Partitioning your verification units into several files will make it easier to work with when you have dozens of them.
- Use special naming conventions for auxiliary variables defined by FSM statements to distinguish them from signals of the design.

6.4.3 Comments, Macros, and Preprocessing Directing

There are two types of comments that can be written in the rules files:

1) Text beginning with "--" and ending at the end of line.

2) Text beginning with "/*" and ending with "*/".

Comment text is ignored by FoCs A comment can be inserted anywhere a space is legal (except in text strings).

Before processing the rules file, FoCs 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, etc.) and #include. See "man cpp" on your unix system for more details.

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

%for

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

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

or:

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

-- step can be negative

or:

```
%for <var> in { <item> , <item> , ... , <item> } do
...
%end
```

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

stripped.

Be aware that **%for** generates a formula for each iteration of the **%for** loop.

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

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

The current value of the loop variable can be accessed using simply "`ii`" if "`ii`" is a separate token in the text. For example:

```
%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;
```

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

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

In the current version, operators work only on numeric values (i.e., it’s OK to write the following):

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

But it is not possible to write

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

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**. The syntax of the **%if** construct is as follows:

```
%if <expr> %then
```

```
...
%end
```

or:

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

6.5 State Machines

Although Sugar increases expressiveness capabilities, there are still properties that cannot be expressed, and others that are too complicated to formulate. **State machines** may provide solutions in many of these cases. The state machine records events that occur in the design under verification. Sugar Formulas can then refer to these events by accessing the state machine's internal state. State machines do not affect the design because information flows only from the design to the state machine. In this section, we describe the special statements for writing state machines.

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 than k data items. Formulation of this property in Sugar is difficult, but it becomes easy with a satellite. An up/down counter is defined, with a range of 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 properties might have become easier if one could talk about past events. Assume 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 a state-machine inside the rule that will help us express this property as follows:

```
vunit if_p_then_q_since_r{
 var state:boolean;      -- defining a boolean variable state
```

```
  assign init(state):= 0   -- initialising the variable
  assign next(state):=    -- assigning a value to state in the next
    case                   -- cycle
      !q :0;              -- if q is false then next(state) is false
     q & r :1;           -- if q and r are true then next(state) is true
     else :state;        -- if the above conditions are false then state
    esac;                -- does not change.
  define q_since_r :=(q & r)|(q & state); -- q_since_r means q is
                 -- active since the last occurance of r.
  assert
    "If p occurs, then at that time q should be active since the last
occurrence of r"
    {always (p ->q_since_r )}
}
```

6.5.1 Additional Expressions Used in State Machines

State machines may include boolean expressions, if-then-else expressions, and case expressions.

6.5.1.1 if and case Expressions

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

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

```
case
   condition1 : expr1 ;
   condition2 : expr2 ;
   ...
   else : exprn ;
esac
```

A **case** expression is evaluated as follows: `condition1` is evaluated first. If it is true,

`expr1` is returned. Otherwise, `condition2` is evaluated. If it is true, `expr2` is returned, and so forth. The **else** part is essential FoCs—in order to define the behavior as deterministic, it is advisable to the behavior 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, and it 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.

6.5.2 Statements for State Machines – FSM Statements

The following statements are required for defining auxiliary variables writing state machines:

var

define

assign next, assign init

module, instance

The order of the statements is unimportant. We now describe each of these statements in detail.

6.5.2.1 The var Statement

A **var** statement declares auxiliary variables required for the state-machine and the formulas in a rule, and has the following format:

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

The type can be one of the following:

boolean

{ enum1, enum2, ... }

number1 .. number2 (range between integers)

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, or 3. The fourth statement declares a variable called “length” which can take on any of the values between 3 and 15, inclusive.

We can also define arrays using the var statement:

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), ... ,
```

A **var** statement only declares auxiliary variables. The **assign** and **define** statements, described below, define the behavior of these variables. FoCs does not allow non-determinism—the value of a variable at each cycle should be explicitly defined using the assign and define statements.

6.5.2.2 The Assign Statement

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

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

```
assign name := expression;
```

The first statement assigns an initial value to an auxiliary variable. The second statement defines the value of an auxiliary variable in the next cycle. The third statement assigns a value to a variable in the current cycle.

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

```
assign init(state) := idle;
assign next(state) :=
     case
         reset : idle1;
         state=idle & !start : idle;
        state=idle & start : 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;
```

6.5.2.3 The Define Statement

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

```
define name := expression;
```

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

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

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

assign must refer to a variable defined with **var**.

define must NOT refer to a variable defined with **var**.

6.5.2.4 The Module and Instance Statements

A **module** statement is used to group together the statements of a state machine. Instead of writing them directly inside the rule, this module can be instantiated inside the rule using the **instance** statement.

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

```
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);
}
```

This module can be defined in the rules file (outside a rule) and instantiated and used in a rule as follows:

```
vunit if_p_then_q_since_r{
instance i1 :since(q,r)(q_since_r);
assert
```

"If p occurs, then at that time q should be active **since** the last occurrence of r."

```
    {always (p ->q_since_r)}
}
```

A module statement is used to define a module which 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** 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, FoCs will issue a warning.

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

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
outputs is a list of output parameters, actually connecting the instance outputs to real signals of the design. An instance name is optional.

6.5.2.5 Advanced Operations on Arrays

It is often convenient to apply operations to entire arrays or to ranges of indices. Boolean arrays are the only arrays supported by FoCs. These arrays are commonly used for buses and bundles.

6.5.2.6 Defining Arrays

An array of state variables is defined as follows:

```
var name ( index1 .. index2 ) : boolean ;
```

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

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

For example:

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

6.5.2.7 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.

Other operations that can be used with arrays are:

:= != if case & | ^ ! -> <->

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

In boolean operands, 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. For example:

```
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;
```

Conversion of Bit Vectors to Integers and Vice Versa

The following are built-in functions for converting bit vectors to integers and vice versa.

Bit vector to integer:

```
bvtoi( a_vector )
```

Integer to bit vector:

```
itobv( an_integer )
```

Example:

```
always ( Addr(0..31) = itobv(256) ->  bvtoi(data(0..3)) = 9 )
```

Note that constant integers are converted to bit vectors implicitly. There is no need to apply `itobv`.

Construction of Bit Vectors from Bits or Sub-vectors

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

```
expr ++ expr
```

Example:

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

If expr is a constant, it should be either 0 or 1. Wider constant vectors should be 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**(expr,N) can help to construct arrays of repeated elements:

For example, defining an array of 8 1's

```
assign arr(0..7) := rep(1,8);
```

Shorthands:

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

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

6.5.2.8 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`".

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 FoCs will identify ..N as a token and will not replace N by 7.

6.5.2.9 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(0..2) & 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 init( a(0..2) ) := 0
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;
```

CHAPTER 7

Using FoCs for Functional Coverage Analysis

7.1 Functional Coverage

FoCs can be easily used to automatically generate monitors for tracking user-defined coverage events. The only difference between a functional checker and a coverage checker is in the interpretation of the “error” message in simulation. For coverage purposes, the message is a positive indication that the desired event happened.

We recommend using the following formula styles for coverage analysis with FoCs:

1) `always !( b )`

The boolean expression b can never be true.

2) `never{ [*]; event1; event2; ..., eventn }(`

The sequence of events can never happen.

In the checker which FoCs generates, a message indicates that the event, or sequence of events, did happen.

Examples:

```
vunit requestWithFullQueue {
   assert “a request comes when the queue is full”
      always !( request & queue_full ) }
```

```
vunit coverRWR {
   assert “read and later write and later read again
        never{ [*]; read,; [*]; write; [*]; read }

   assert “read and later write and later read again,
      and there was no other read between the two reads”
     never { [*]; read; !read[*]; !read & write; !read[*]; read }
```

More coverage options, such as event counting, will be added to FoCs in the future.

CHAPTER 8 *Defining Bugspray Events*†

†. For IBM users only

8.1 Introduction

When using Bugspray, it is possible to define events of type fail, count, and harvest. It is possible to associate a Bugspray event to every Sugar assertion. The definition is made in the description of the assertion.

8.2 Syntax

The description of the Sugar assertion starts with the Bugspray definition. A Bugspray definition is separated from a standard description by "`--!!`" at the beginning and a semicolon ("`;`") at the end:

```
assert
  "--!!<Event type(fail,count,harvest)> <Special flags>;
   standard assertion description "
    Sugar formulation ;
```

If no bugspray definition is used, the formula will be a fail event.

The following are special flags used by Bugspray. All special flags are optional.

Flag	Explanation
-t	In count events, this flag defines a trigger signal (a signal that says when to increase a counter for event). If no signal name is defined (in this case there is no need for writing -t in special flags), the keyword “no_trigger” is used as the trigger signal.
-Tp	This flag denotes whether to use the keyword “TracePoint” for a count event.
-vn	This flag allows you to specify a variable name for count and harvest events.
-cn	This flag allows you to specify a class name for count events.
-c	This flag denotes whether to use the keyword “Cycle” for a harvest event.

8.3 Events

8.3.1 Fail Events

Syntax:

```
assert
“--!!fail; standard assertion description”
 Sugar formulation ;
will be translated in checker to :
--!!fail outputs
 --!!   1:
```

```
  .
  .
 --!!   i: “standard formula description”;
   .
 --!!end fail outputs;
```

Standard formula description is used for error message string.

8.3.2 Count Events

Syntax:

```
assert
“--!!count <-t signal name> <-Tp><-cn class name><-vn variable name>;”
 Sugar formulation ;
```

Note: <xxx> is used for optional values supplied by the user.

For each count event, if the class or variable name are not defined by the user, the checker automatically produces a class name and a variable name using the name of checker’s entity:

class name : `count_class_<checker entity name>`

variable name : `count_<checker entity name>_i`, where i is an index of the count event

Examples for possible definitions: (assuming checker entity name is “chkr”)

```
formula
“--!!count;”
{ Sugar formula }
will be translated to :
--!!count outputs
--!!  1
   .
    .
```

```
--!!  i: count_class_chkr count_chkr_i no_trigger;
    .
--!!end count outputs;

assert
 “--!!count -t clk -cn user_class_name;”
 Sugar formulation ;
will be translated to :
 --!!count outputs
 --!!  1:
        .
        .
 --!!  i: user_class_name count_chkr_i clk;
        .
 --!!end count outputs;

assert
“--!!count -t clk -Tp -cn user_class_name -vn user_var_name;”
 Sugar formulation ;
will be translated to :
 --!!count outputs
 --!!  1:
      .
     .
 --!!  i: user_class_name user_var_name clk, TracePoint;
     .
 --!!end count outputs;

assert
```

```
“--!!count -Tp;”
 Sugar formulation ;
will be translated to :
--!!count outputs
--!!  1:
       .
       .
 --!!  i: count_class_chkr count_chkr_i no_trigger, TracePoint;
       .
--!!end count outputs;
```

8.3.3 Harvest Events

Syntax :

```
assert
“--!!harvest <-c> <-vn variable name>; standard formula description”
 Sugar formulation ;
```

Note: <xxx> is used for optional values supplied by the user.

For each harvest event, if the variable name is not defined by the user, the checker automatically produces a variable name using the name of checker’s entity:

variable name : `harvest_<checker entity name>_i;`

Examples for possible definitions : (assuming checker entity name is "chkr")

```
assert
“--!!harvest; standard formula description”
 Sugar formulation ;
will be translated to :
--!!harvest outputs
--!!  1:
       .
```

```
        .
--!!  i: harvest_chkr_i “standard formula description”;
        .
--!!end harvest outputs;

assert
“--!!harvest -c -vn user_var_name; standard formula description”
     Sugar formulation
will be translated to :
--!!harvest outputs
--!!  1:
       .
       .
--!!  i: user_var_name “standard formula description”, Cycle;
        .
--!!end harvest outputs;
```

In all types of events, indexes are assigned automatically.

CHAPTER 9 *FoCs for RuleBase Users*

9.1 Tips for Users of RuleBase

RuleBase is an industrial-strength formal verification (FV) tool, developed by the IBM Haifa Research Laboratory. RuleBase is especially applicable for verifying the control logic of hardware designs, and uses Sugar for design specification. There are several advantages to using FoCs-generated checkers in the simulation of designs which were formally verified against the same Sugar properties.

- Often, the input constraints (a.k.a. “the environment model”) defined for formal verification are more restricted than the real environment modeled in simulation. Thus, in simulation, one can exercise the design against inputs whose effect on the design has not been explored by FV. It follows that the enhanced checking capability of FoCs provides better coverage and confidence in verification quality.
- FoCs checkers can help find problems in the input constraints defined for FV. For example, if the results of FV and simulation—relative to the same properties—do not agree, it is likely that the input constraints have not been defined correctly.
- Using FoCs, one can validate the assumptions made in the FV process. These assumptions can be formulated as Sugar properties, translated by FoCs into checkers, and checked during simulation.

It is possible to use the same rules file for both RuleBase and for FoCs; however, FoCs only allows deterministic signal definitions. The RuleBase rules file may include non-

deterministic definitions that are relevant for environment definitions. These definitions are irrelevant for the generated checker, which receives its input from the simulation inputs. For FoCs to ignore such definitions, you must set the Interface Filename (in Settings, under Generation Style) to your VIM DEF file, located in your vimdbase/DEF directory. Using this file, FoCs knows what signals are actual design inputs.

CHAPTER 10

Appendix A - Checker Code Examples

10.1 Examples of Checker Code in Verilog and VHDL

Below is the checker code generated by FoCs for the Sugar rule:

```
vunit check_busy_flag{
assert
     {[*]; {request&!acknowledge}[*5]} |-> {busy_flag}

}
```

10.2 Checker Code in Verilog

The following is the FoCs checker code in Verilog:

```
module check_busy_flag (
    clk,
    rst,
    request,
    acknowledge,
```

```
    busy_flag
);
    input clk;
    input rst;
    input request;
    input acknowledge;
    input busy_flag;
    reg focs_enable_0;
    reg focs_ok_0;
    reg [0:6] focs_v_0;
    reg [0:6] focs_vout_0;


 initial
 begin
 end
    always @(posedge clk)
        if (rst) begin
            focs_enable_0 <= 1'd1;
            focs_ok_0 <= 1'd1;
            focs_v_0[0:6] <= 7'b1110000;
        end
        else
            if (focs_enable_0) begin
                focs_ok_0 <= !((focs_v_0[6] & ((request &
!(acknowledge))
       & !(busy_flag))));
                focs_vout_0[0:6] = {1'd0, (focs_v_0[1] & 1'b1),
(focs_v_0[2
      ] & (request & !(acknowledge))), (focs_v_0[3] & (request & !(
      acknowledge))), (focs_v_0[4] & (request & !(acknowledge))),
(focs_v_0
```

```
      [5] & (request & !(acknowledge))), (focs_v_0[6] & ((request & !(
      acknowledge)) & !(busy_flag)))};
                focs_v_0[0:6] <= {focs_vout_0[0], (focs_vout_0[0] |
      focs_vout_0[1]), (focs_vout_0[0] | focs_vout_0[1]),
focs_vout_0[2],
      focs_vout_0[3], focs_vout_0[4], focs_vout_0[5]};
               if (!((!(clk) | ((!(focs_enable_0) | focs_ok_0) | rst))
      )) begin
                    $display("formula 1");
                    $finish;
                end
            end
endmodule
```

10.3 Checker Code in VHDL

The following is the FoCs checker code in VHDL:

```
library ieee;
library ibm;
use ieee.std_logic_1164.all;
use ibm.std_logic_support.all;
ENTITY check_busy_flag IS
   PORT (
      clk :IN std_logic;
      rst :IN std_logic;
      request :IN std_logic;
      acknowledge :IN std_logic;
      busy_flag :IN std_logic);
END check_busy_flag ;
```

```
ARCHITECTURE checker OF check_busy_flag IS
   SIGNAL   focs_enable_0 : std_logic;
   SIGNAL   focs_ok_0 : std_logic;
   SIGNAL   focs_v_0 : std_logic_vector(0 TO 6);


    function i2l (src: integer)
return std_logic is
variable R: std_logic;
    begin
if src = 0 then
    R := '0';
else
    R := '1';
end if;
return R;
    end i2l;


    function l2i (src: std_logic)
return integer is
variable R: integer;
    begin
if src = '0' then
    R := 0;
else
    R := 1;
end if;
return R;
    end l2i;
```

```
    function b2l (src: boolean)
return std_logic is
variable R: std_logic;
    begin
if src then
    R := '1';
else
    R := '0';
end if;
return R;
   end b2l;


    function l2b (src: std_logic)
return boolean is
variable R: boolean;
    begin
if (src = '1') then
    R := true;
else
    R := false;
end if;
return R;
    end l2b;


    function b2i (src: boolean)
return integer is
variable R: integer;
    begin
```

```
if src then
    R := 1;
else
    R := 0;
end if;
return R;
    end b2i;


    function i2b (src: integer)
return boolean is
variable R: boolean;
    begin
if src = 0 then
    R := false;
else
    R := true;
end if;
return R;
    end i2b;


    function reverse (arg: std_Logic_vector) return std_Logic_vector is
    variable result : std_Logic_vector( arg'range );
  begin
    for i in arg'range loop
      result(result'right - i + result'left) := arg(i);
            end loop;
            return (result);
    end reverse;
```

```
  -- convert an integer to an STD_LOGIC_VECTOR
  function i2vl(ARG: INTEGER; SIZE: INTEGER) return STD_LOGIC_VECTOR
is
    variable result: STD_LOGIC_VECTOR (SIZE-1 downto 0);
    variable temp: integer;
    -- synopsys built_in SYN_INTEGER_TO_SIGNED
    begin
    -- synopsys synthesis_off
    temp := ARG;
    for i in SIZE-1 downto 0 loop
        if (temp mod 2) = 1 then
        result(i) := '1';
        else
        result(i) := '0';
        end if;
        if temp > 0 then
        temp := temp / 2;
        else
        temp := (temp - 1) / 2; -- simulate ASR
        end if;
    end loop;
    return result;
    -- synopsys synthesis_on
    end i2vl;


  -- convert a boolean to an STD_LOGIC_VECTOR
  function b2vl(ARG: BOOLEAN; SIZE: INTEGER) return STD_LOGIC_VECTOR
is
```

```
    variable result: STD_LOGIC_VECTOR (SIZE-1 downto 0);
    begin

    if (ARG = false) then
     result(SIZE-1) := '0';
    else
     result(SIZE-1) := '1';
    end if;
    if (SIZE = 1) then
       return result;
    end if;
    for i in SIZE-2 downto 0 loop
       result(i) := '0';
    end loop;
    return result;
    end b2vl;
```

-- convert a logic vector to integer

```
function vl2i (src: STD_LOGIC_VECTOR)
return integer is
  variable R : integer;
  variable mult : integer;
begin
  R := 0;
  mult := 1;
  for i in src'high downto src'low loop
    if src(i) = '1' then
    R := R + mult;
  end if;
```

```
  mult := mult * 2;
  end loop;
  return R;
end vl2i;
```

```
BEGIN
   PROCESS
      VARIABLE focs_vout_0 : std_logic_vector(0 TO 6);
   BEGIN
      WAIT UNTIL clk'EVENT AND clk = '1';
      IF ( l2b( rst ) ) THEN
         focs_enable_0 <=  '1' ;
         focs_ok_0 <=  '1' ;
         focs_v_0(0 TO 6) <= "1110000";
      ELSIF ( l2b( focs_enable_0 ) ) THEN
         focs_ok_0 <= NOT( ( focs_v_0(6) AND ( ( request AND NOT(
acknowledge )
             ) AND NOT( busy_flag ) ) ) );
         focs_vout_0(0 TO 6) := ( ( ( ( ( (  '0'  & ( focs_v_0(1) AND
'1' ) )
            & ( focs_v_0(2) AND ( request AND NOT( acknowledge ) ) ) )
& (
           focs_v_0(3) AND ( request AND NOT( acknowledge ) ) ) ) & (
           focs_v_0(4) AND ( request AND NOT( acknowledge ) ) ) ) & (
           focs_v_0(5) AND ( request AND NOT( acknowledge ) ) ) ) & (
            focs_v_0(6) AND ( ( request AND NOT( acknowledge ) ) AND
NOT(
            busy_flag ) ) ) );
```

```
         focs_v_0(0 TO 6) <= ( ( ( ( ( ( focs_vout_0(0) & (
focs_vout_0(0) OR
           focs_vout_0(1) ) ) & ( focs_vout_0(0) OR focs_vout_0(1) ) )
&
            focs_vout_0(2) ) & focs_vout_0(3) ) & focs_vout_0(4) ) &
            focs_vout_0(5) );
      END IF;
   END PROCESS;
   ASSERT ( NOT( ( ( NOT( clk ) OR ( ( NOT( focs_enable_0 ) OR
focs_ok_0 ) OR
      rst ) ) =  '0'  ) )
          )
      REPORT   " FAILURE EVENT: rule: CHECK_BUSY_FLAG, formula: 1 :
formula 1"

      SEVERITY NOTE;
END checker ;
```

CHAPTER 11

Appendix B - Common Error Messages

11.1 Common FoCs Error Messages

Below are some common error messages and their meanings:

11.1.1 Settings Errors

Fatal error: In "Settings" clock name must be supplied

Defining the Clock in the Settings is obligatory.

Fatal error: In "Settings" reset name must be supplied

Defining the reset signal, when reset is defined as External, is obligatory.

Fatal error: In Bugspray mode, the name of the design must be specified

When selecting Target simulator to be "Bugspray", defining the design entity name is obligatory.

11.1.2 Sugar Errors and Warnings

Warning: Assertion 1: Assertion does not begin with "always"

When an assertion begins with `always`, or a sequence begins with `[*]`, it is checked at every cycle. Otherwise, it is checked only at the first cycle. Such a message is likely to appear if there is a formula with `{e1,e2,...}`. This assertion doesn't start with `[*]`, and therefore will only be checked on the first cycle.

Warning: Assertion 1: Operation AF cannot run in Safety on-the-fly mode

Warning: Assertion 1: Running this assertion with OnTheFly = No.

Warning: Assertion is not safety. Translation failed. Check if there is no use of AF or ECTL operators in the formula

Assertion not on-the-fly

If one or more of the above messages appears, it means that the Sugar assertion is not supported by FoCs. Since checkers run during simulation, and the properties are checked at each cycle, only properties that can be verified at each cycle can be written. Such properties are called Safety properties.

Properties that refer to "sometime in the future", such as "liveness" properties, that ensure the occurrence of some event, can not be verified during simulation, and therefore cannot be used in FoCs.

See Sugar_v1.4 -with FoCs notes for further explanation on unsupported Sugar operators.

Fatal error : Nondeterministic operator is used

Since the Sugar properties are translated to HDL code, it is forbidden to use non-determinism in the assertions and state machines. Such a message may mean that there is a variable that has received a non-deterministic value.

Fatal error: Environment is nondeterministic: behaviour of signal x is undefined

In this case, there must be some variable that was defined but not assigned.

Fatal error: Environment is nondeterministic : in signal x next state is defined without init state

This means that there is an assign statement for variable x, but no init statement. This is forbidden, because behaviour must be deterministic.

Fatal error : Nondeterministic environment - case without else in file...

A case statement must always contain an else part (even if all cases are covered), Otherwise the behaviour is considered as non deterministic.

Index