



FP6-IST-507219

PROSYD

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Support for Embedded PSL in Verilog Flavour - Observers

(Deliverable 3.2/16)

Due date of deliverable: June 30, 2006

Actual Delivery date: June 28, 2006

Start date of project: 01.01.2004

Duration: 3 years

Organisation name of lead contractor for this deliverable: IBM

Revision: 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2000-2006)		
Dissemination Level		
PU	Public	<input checked="" type="checkbox"/>
PP	Restricted to other programme participants (including the Commission Services)	<input type="checkbox"/>
RE	Restricted to a group specified by the consortium (including the Commission Services)	<input type="checkbox"/>
CO	Confidential, only for members of the consortium (including the Commission Services)	<input type="checkbox"/>

Notices

For information, contact pidan1@il.ibm.com

This document is intended to fulfil the obligations of the PROSYD project concerning deliverable 3.2/16, described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2004-2006. All rights reserved.

Table of Revisions

Version	Date	Description and Reason	By	Affected Sections
0.1	30.4.2006	Document template preparation	Dmitry Pidan	All
0.2	15.5.2006	Sections 2,3 draft	Sivan Rabinovich	2,3
0.3	18.5.2006	Full document draft	Dmitry Pidan	All
0.4	29.5.2006	Final document before managers review	Dmitry Pidan	All
0.5	13.6.2006	Revised after manager's review	Dmitry Pidan	All
1.0	28.6.2006	Final version	Cindy Eisner	Version number

Authors

Dmitry Pidan

Sivan Rabinovich

Executive Summary

One of the challenges verification engineers must address while doing assertion-based verification is attaching behaviour-checking properties to their correct context within the design.

The PSL language provides a method known as 'binding' for attaching properties to their context; binding attaches the verification units to their relevant design units. This method is supported by FoCs, the IBM tool for generation of simulation observers from PSL statements. The purpose of this deliverable is to extend FoCs with the ability to generate observers from the properties that are written explicitly in the design and to attach them to the correct context—in other words, to generate observers from *embedded assertions*.

Purpose

The purpose of this document is to describe the implementation of an extension made to FoCs in the framework of Deliverable 3.2/16.

Intended Audience

This document is intended for users of the IBM's FoCs tool.

Background

FoCs (Formal Checkers) is an IBM tool for generating simulation monitors from PSL specifications. The simulation monitor produced by FoCs is HDL code that can be ported into a simulation environment. FoCs supports the generation of Verilog/VHDL monitors from PSL/GDL flavour specifications and Verilog monitors from PSL/Verilog specifications. The specifications should be written in standard PSL language [1].

The purpose of this deliverable is to extend FoCs with support for PSL/Verilog flavour *embedded assertions*. PSL/Verilog flavour embedded assertions are the PSL assertions written directly inside the Verilog RTL code.

Contents

Table of Revisions	iii
Authors.....	iii
Executive Summary	iii
Purpose.....	iii
Intended Audience	iii
Background.....	iii
List of Tables	v
Glossary	vi
1 Introduction.....	1
2 Embedded PSL Assertions.....	2
Syntax	2
Implementation	3
Extracting Embedded Assertions from the Design.....	3
Displaying embedded assertions.....	3
Generating Instantiation Code	3
Display Enhancements.....	3
User Interface.....	4
FoCs GUI.....	4
Settings for Verilog flavor	5
<i>Embedded Vunits Display</i>	6
<i>Generating Observers for Embedded Assertions</i>	7
<i>Observers Instantiation Code</i>	7
3 Tool Properties and Development	9
Features List.....	9
Work Summary.....	10
Starting Point	10
Work Performed	10
Results.....	10
Supported PSL Subset	10
4 References.....	12

List of Tables

Table 1 List of features	9
--------------------------------	---

Glossary

Assertion

A property that is expected to hold true on a specific design.

Checker

See "Simulation monitor"

Embedded assertion

Assertion that appear in the body of the design implementation code.

FoCs

An IBM tool for generating simulation monitors from PSL specification.

HDL (Hardware Description Language)

One of several specialized high-level languages used by semiconductor designers to describe the features and functionality of chips and systems prior to handoff to the IC layout process. HDL descriptions are used in both the design implementation and verification flows. Currently, the two standard HDLs in use worldwide are Verilog HDL and VHDL. Several proprietary HDLs also exist, mainly for describing logic targeted for vendor-specific programmable logic devices.

HDL concurrent statement (assignment)

An HDL statement that is executed concurrently with other statements in the block. For the assignments, the value assigned to the left hand side of the assignment is considered to be valid at the next time point. See also [2].

Observer

See "Simulation monitor"

Simulation monitor

An HDL code unit that runs with the design in the simulation, checks the design under test for the user defined property, and reports an error if the property is violated. Sometimes also referred to as "checker".

Vunit

Verification unit – a PSL construct that can contain a number of assertions and additional modelling code.

1 Introduction

Assertion-based verification is a verification paradigm that plays a very important role in the design verification process. Under this paradigm, verification is performed using assertions, which are declarative statements that formally describe the specification of the design under test. In other words, they describe which design behaviours are legal and which are not.

In dynamic verification, the concept of assertion-based verification is interpreted as follows: assertions are evaluated together with the design and play the role of 'watchdogs'. They monitor design behaviour during the simulation and check that the design never enters illegal states.

One of the greatest advantages of assertion-based verification is its ability to perform 'white-box' testing. 'White-box' testing is directed at the internal states and variables of the design block, rather than being directed at the block interfaces, as is done for 'black-box' testing.

Writing assertions for white-box testing involves various challenges for the author. First, writing this kind of assertion requires a deep knowledge and understanding of the internal behaviours of the design under test. Second, these assertions must be precisely attached to the correct context within the design. This is because the assertion should follow the internal changes of the design and not only the changes that are visible from outside the block (interface changes). The concept of *embedded PSL assertions* addresses these challenges.

Embedded PSL assertions are placed directly in the RTL code. Writing assertions in this way provides a several advantages, among them the use of the most natural way to attach the properties to the correct context of the design. Another advantage to writing embedded assertions is that they can be inserted in the RTL code during the designer's development of the RTL, so they become part of the design development process. Hence, any change made to the design will also reflect these assertions.

The rest of this paper is organized as follows: Chapter 2 describes the syntax of embedded assertions, implementation details, and the FoCs user interface for support of embedded assertions. Chapter 3 summarizes the tool features and the work performed.

2 Embedded PSL Assertions

The IBM FoCs tool generates simulation monitors from PSL specifications. In the framework of this deliverable, FoCs was extended to support the PSL assertions embedded in the RTL code (Verilog [2])

Syntax

PSL assertions are embedded in the user-written Verilog code, using smart comments. Smart comments for embedded assertions are standard Verilog comments, marked by the reserved word "psl". Each assertion can be written in a single line or expanded over consecutive lines. Smart comments containing embedded assertions can also contain modelling layer code.

Single-line assertion:

- A single-line assertion is written as a Verilog single-line comment [2] that contains the reserved word "psl":

```
// psl <assertion text> ;
```

Multi-line assertion:

- A multi-line assertion is written as a sequence of Verilog single-line comments. The first line starts with the reserved word "psl", and the last line is terminated by ";". This is essentially one assertion segmented over a number of lines:

```
// psl <start assertion text  
// <continue assertion text>  
// <end assertion test> ;
```

- A multi-line assertion can be written also as a Verilog multi-line comment [2], starting with the reserved word "psl":

```
/* psl  
   <modelling code>  
   <assertion text>  
*/
```

The <> brackets appear in the syntax for readability purpose only and are not part of the syntax.

As can be seen from the syntax, embedded PSL assertions are written in smart comments. As a result, the design with embedded assertions can be used by both tools that support these embedded assertions and tools that do not.

Embedded PSL assertions can appear only inside Verilog concurrent code. PSL assertions that appear in sequential code are not supported by the tool. Signals used inside the assertions must be known in the block in which the PSL assertion appears, according to the Verilog scoping rules [2].

Implementation

Extracting Embedded Assertions from the Design

The processing of embedded assertions in FoCs starts with analysis of the design files. FoCs scans the design files and identifies those units that contain embedded assertions. Whenever an embedded assertion is encountered, the tool collects it. Finally when the tool reaches the end of a source module (Verilog module), it emits all the embedded assertions found so far to an external file. This file contains generated vunits based on the code found in the design modules. For each Verilog module, the tool generates one vunit.

The linkage between a generated vunit and its origin module is done by means of binding. Binding is the PSL method of associating a verification unit (vunit) with an HDL module. The binding rules define the scope of the Vunit: identifier resolution, instance replication, and more. The complete documentation on verification unit binding can be found in the PSL language reference manual [1]. For our purpose it is sufficient to note that a vunit bound to a module is synonymous with the actual PSL code written as embedded PSL inside that module itself.

Displaying embedded assertions

Before the tool can proceed with embedded assertions processing, the user interface displays the embedded assertions contained in the design (the user interface is further discussed in "User Interface" section). While FoCs processes the design to generate the embedded assertions file, it records the design hierarchy structure (modules and instances). This allows the tool to display a complete list of all the design modules and instances, where each module containing embedded assertions is highlighted.

Once FoCs has generated the vunits file, it handles it as if it was a regular PSL file.

Generating Instantiation Code

Another important feature developed is the generation of the instantiation code. The final observer code should be implanted into a simulation environment as part of the entire code that is simulated. Users can select the hierarchy location in which they prefer to place the instantiation code, and the tool then generates instantiation code based on this preference. For example, if the user asks to instantiate the observer code from the topmost module, the instantiated code will include hierarchical connection names starting at the top module.

Display Enhancements

Embedded assertions are displayed to the user in a tree view developed for this purpose. The tool's previous "flat" vunit list was upgraded, and now contains a hierarchical tree that displays the design hierarchy. By default, the tree root starts at the topmost module. The user can change this selection and position the root of the displayed tree in any other module in the list. The children of the selected root module in the tree view are all the instances of modules instantiated in the top module body. For each instance, the tree expands its children and so on.

User Interface

This section describes the part of FoCs user interface related to the displaying and running of the verification units. In the framework of deliverable 3.2/16, this interface was extended to support Verilog embedded assertions. Text in *italics* describes the new user interface features developed in the framework of deliverable 3.2/16.

Note: This section is based on the relevant sections of the FoCs User Guide [4].

FoCs GUI

The FoCs GUI comprises a toolbar and three panes:

Vunits List pane – displays a list of vunits names from the vunits file. The vunits file is selected in the FoCs Settings window.

Assertions pane – displays all the assertions for the selected vunit(s) in the vunits pane.

Messages pane – displays all relevant messages from FoCs.

The toolbar enables the following actions:

- Exit FoCs
- Generate the Checker(s)
- Killing the current FoCs run
- Refresh the Vunits List
- Choose all the vunits at once (toggle). The generally used multiple selection keys (control, shift) can be used instead
- Edit checker generation settings
- Choose an editor for editing files from within the GUI, by means of the **Edit** button. The editor path should be included in the user path

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

- Select the vunit. The Formulas window displays the vunit's formulas.
- Click **Generate**.

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

- Select a vunit. The Formulas window displays vunit's formulas
- Hold down the Ctrl button and select another vunit. The Formulas window displays all the formulas for these two vunits.
- Click **Generate**. The **Choose an output filename** window is displayed.
- Enter the name of the desired checker and click **OK**.

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

- Click the **Choose All** button on the toolbar. This selects all the vunits listed in Vunits List pane.
- Click **Generate**. The **Choose an output filename** window is displayed.
- Enter the name of the desired checker and click **OK**.

Settings for Verilog flavor

Users can customize FoCs settings with the Settings dialog box. The Settings dialog box consists of several tabs: Main, Clock and Reset, Checker Generation Style, Reporting, Signal Mapping, and Verilog Flavor. Each tab contains several options. When you change a value for an option, the new value is put into effect immediately. 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.

In order to generate a checker in Verilog flavor, FoCs needs information about the types of signals referenced in PSL code. This information can be provided using two possible options. You can either supply a file ("Signal Info File") in Verilog syntax with the necessary declarations, or have FoCs read the code of the design. *You must use the second option to process embedded assertions.*

Top Module Name

The Top Module Name setting determines the name of the top module in the design hierarchy.

Signal Info File Name

The Signal Info File contains declarations of the signals referenced in PSL code. You can select the name and the location of Signal Info File arbitrarily. It does not have to be related to the name and the location of the PSL code and design model files.

The syntax of Signal Info File is similar to normal Verilog syntax. The file is not partitioned into modules, and can contain declarations of signals from any number of modules in design hierarchy. Consequently, all names in declarations contained in the Signal Info File are hierarchical names, i.e., they must specify the name of the module where the declared signals are located in the design hierarchy. For example:

```
reg [7:0] my_module.my_signal;
wire another_module.some_instance.some_signal;
```

Recall that every PSL vunit should be bound to some Verilog module or instance and can reference signals (possibly using hierarchical names) that are supposed to be looked up in the module or instance to which the vunit is bound. The recommended way to code the Signal Info File is to declare all signals referenced from the vunit using a hierarchical name. This name should be a concatenation of the module or instance to which the vunit is bound and the (possibly hierarchical) name used within the vunit to reference the signal.

For example, for the following simple PSL vunit:

```
vunit simple_example(my_design)
{
    assert always (trigger -> next(count1 > count2));
}
```

the following Signal Info File might be used:

```
wire my_design.trigger;
reg [7:0] my_design.count1, my_design.count2;
```

For another, more complicated example, consider:

```
vunit complicated_example(my_design.a.b.c)
{
    assert always (x.y.z.trigger -> next(count1 > count2));
}
```

The Signal Info File might then appear as follows:

```
wire my_design.a.b.c.x.y.z.trigger; // note that we my_design.a.b.c and
                                   x.y.z.trigger are concatenated.
reg [7:0] my_design.a.b.c.count1, my_design.a.b.c.count2;
```

Getting FoCs to Read the Code of the Design

As an alternative to using the Signal Info File, you can specify to FoCs where the code of the design is located. FoCs will then be able to read the code and extract the type information automatically.

There are two ways to specify the information about the location of the design code. The first way is to use the following command-line flags:

- `+incdir+` the search path for a files included by ``include` directive
- `-y` with the search path for a missing Verilog modules

You should specify these flags in the Command Line field of the Verilog Flavor tab in the Settings dialog of the FoCs GUI.

The second option is to fill three separate fields of the tab intended for the Verilog preprocessor path, Verilog design path, and for the list of Verilog files to be read.

Verilog Preprocessor Path

The Verilog Preprocessor Path option is the search path for the files included by ``include` compiler directive (corresponds to `+incdir+` command-line flag).

Verilog Design Path

The Verilog Design Path option is the search path for definitions of missing modules (corresponds to `-y` command-line flag).

Design File

The Design File option includes the list of design files. The GUI includes a file selection dialog that enables the multiple selection of the files names.

Embedded Vunits Display

Embedded assertions are actually part of the design source text, therefore, it's natural to present them as part of the design display. The display of embedded assertions is part of a new user interface component that details the design hierarchy and enables users to select a desired embedded assertion according to its location in the design hierarchy.

The display is controlled by two GUI components:

1. *Design hierarchy tree view:*
This tree view holds information about all the modules and instances that appear in the design source files. The relations among modules in a design can be interpreted as a relation tree, in which the root node is the topmost module, its child nodes are the instances of modules instantiated inside it, and so on. Each node holds the name of the instance and, in parenthesis, the name of the module.
2. *Root module selection box:*
This list box enables you to control the tree view top module. Please note that this control only influences the display and doesn't change the design in any way. You can use this to simplify the display if you want to investigate a specific branch of the module tree.

The tree view enables you to select (or unselect) any node. When you select a node, the entire sub-tree of this node is automatically selected. This enables the generation of observers to all the modules instantiated in a specific node (instance) in the module hierarchy.

The tool displays the PSL contents for a selected part of the hierarchy in the central display area. When you select part of the hierarchy in the tree view, the central display area shows its PSL contents: all the assertions in the selected part and their translation into English.

Generating Observers for Embedded Assertions

The process of generating observers for embedded assertions is similar to generating observers for regular units. The following tutorial will help you become familiar with the process:

- 1. Start the tool and apply the settings for rules file, clock and reset names and details, generation style and most important – Verilog flavour settings. The settings for the Verilog flavour determine the top module and the source files of the design. All these parameters are configured in the Settings dialog, which you can access by clicking the **Settings** button.*
- 2. The design hierarchy view on the left pane of the display now shows the design details and hierarchy. You can change the starting point of the display by selecting a different value in the **Start from module** list box.*
- 3. You can browse through the different instances and look at the assertions inside them. Any instance that has embedded assertions will be marked with bold text.*
- 4. After browsing through the instances, you can select the desired modules for which you want to generate observers and click the **Generate** button. The tool processes the request and emits the observers as Verilog files in the tool's current working directory. Each observer is named after its corresponding unit. Because we deal with embedded assertions, the name of the generated observer will be “*embedded_in_module_name.v*”(where *module_name* stands for the name of the module in which the embedded assertions were found). When you generate observers for multiple modules, an observer will be generated for each module.*
- 5. The tool generates an instantiations file (*instantiations.v*) that holds instantiation code for the generated observers. Since each observer is a Verilog module, we want to instantiate it prior to simulation. The instantiations file contains instantiation code for each instance of a selected module (relative to the selected top module). This code can be copied as-is and pasted in the selected top module to embed the observer code in the simulated design (a further discussion of the instantiation code can be found in the following section).*

Observers Instantiation Code

The instantiation code generated by the tool enables you to instantiate the observers as part of the design. The instantiation code is automatically generated, according to several user preferences. These preferences provide flexibility regarding the location of the generated code.

*To change the way FoCs generates the instantiation code, use the **Checker instantiation level** list box, which is part of the Verilog flavour tab in the Settings dialog. This list box has three options:*

1. *Design Top Module: This option instructs FoCs to generate instantiation code that will be put in the top module of the design. This means that signals connected to the observer ports will have hierarchical names starting with the top module.*
2. *Displayed Top Module: This option means the generated code will be relative to the top module selected in the tree view display (the tree view root can be altered with the **Start from module** list box). The generated code will use hierarchical names relative to that module.*
3. *MRCA of selected instances: Most Recent Common Ancestor (MRSA) means that FoCs will choose the lowest node it can find that is the ancestor if all instances selected for a generation of an observer in the tree view. The instantiation code will be generated relative to this calculated module.*

3 Tool Properties and Development

Features List

Table 1 List of features

	Present	Ref.
Mandatory features		
• Pointers to algorithms used	Yes	Chapter 2
• List of target operating systems	Yes	See below
• Explanation of coding standards	Yes	See below
• Discussion of license issues	Yes	See below
• User documentation, including documentation of user interface (command line switches) and imported/exported file formats	Yes	See below
• Test suite	Yes	See below
• Standard input language – PSL	Yes	See [1]
○ Support for Verilog flavour	Yes	
• Support for embedding PSL properties in hardware design	Yes	Chapter 2
• Improved GUI to support new property style	Yes	Chapter 2
Nice-to-have features		
• Support for other flavours	No	
• Mechanism for simulator-independent connection of observers	Yes	Chapter 2

Target operating systems: Linux, Solaris, AIX

Coding standards: The tool is implemented in C and C++, according to the coding standards of the IBM Haifa Research Laboratory.

License issues: A FlexLM license mechanism is included for the tool. The license can be purchased from IBM under a license agreement. Personal licenses, based on the user ID are also available.

User documentation: The FoCs user guide is available for free, inside the distribution package of the tool. This deliverable includes relevant excerpts from the user guide.

Test suite: Information about the test suite is delivered, but not the test suite itself.

Number of new tests written for PROSYD: FoCs Verilog flavour tests (about 900 tests) were converted to use embedded assertions instead of bounded vunits.

Work Summary

Starting Point

At the starting point of this deliverable, the FoCs tool supported the PSL Verilog flavour only for the standalone verification units. Verification units had to be provided in a separate file.

The module that supports PSL Verilog flavour is the VFE (Verilog Front-End) module, developed in the framework of the PROSYD deliverable 3.3/2 [4].

Work Performed

VFE Module

VFE was adapted to support all GUI requirements for displaying embedded assertions and support the flow of creating vunits from these embedded assertions. This includes the following items:

1. Supporting additional command line arguments.
2. Generating instantiation code based on the instantiation level, the selected top module, the checker name template, and most important – the information from the VFE symbol table about modules and their instances.
3. Change the way VFE works in auxiliary Verilog module creation. Instead of generating a single auxiliary module for all the vunits, it creates a module for each vunit.

GUI Module

A new GUI widget was developed to support the design tree view display (explained earlier). This display is composed of a flat vunit list (for regular vunits), a hierarchical list box for the design tree view, and a selection box for the top module.

Results

FoCs supports PSL assertions embedded in Verilog code.

Supported PSL Subset

The subset of PSL supported by FoCs is described in Section 5 in [4]. For the embedded PSL assertions, FoCs supports the same subsets of Boolean, temporal and modeling layers of PSL, with the following restrictions:

- Boolean and modeling layers are written in Verilog
- All the PSL constructs that are marked "RuleBase PE only" in [4] are unsupported

From the verification layer, the constructs "assert", "cover" and "default clock" are supported in embedded PSL assertions.

4 References

- [1] Accellera. Accellera Property Language Reference Manual. In <http://www.eda.org/vfv/docs/PSL-v1.1.pdf>, June 2004
- [2] The Institute of Electrical and Electronics Engineers, Inc. IEEE Standard Verilog Hardware Description Language, 2001
- [3] FoCs User Guide. Available as a part of the FoCs product
- [4] Porting of IBM tools to support Accelera-Standard version of PSL. PROSYD deliverable 3.3/2