



FP6-IST-507219

# PROSYD

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

## **Property-based automatic generation of simulation monitors for digital designs**

**(public version)**

**(Deliverable 3.2/11)**

Due date of deliverable: 30.06.2006

Actual Delivery date: 02.07.2006

Start date of project: 01.01.2004

Duration: 3 years

Organisation name of lead contractor for this deliverable: IBM

Revision: 1.1

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](mailto:pidan1@il.ibm.com).

This document is intended to fulfil the obligations of the PROSYD project concerning deliverable 3.2/11, 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	11.05.2005	First draft	Dmitry Pidan	All
0.2	30.05.2005	Revised draft	Dmitry Pidan	All
0.3	25.06.2006	Revised after project manager's review	Dmitry Pidan, Sitvanit Ruah	All
1.1	10.12.2006	Public version	Dmitry Pidan	

## Authors

Dmitry Pidan  
Michael Shamis  
Sitvanit Ruah

## Executive Summary

This document provides a description of the simulation monitor optimization implemented inside IBM FoCs – tool for automatic generation of simulation monitors from PSL specifications. The materials covered in the document offer background information – a basic description of the FoCs tool, excerpts from the user guide and simulation monitor optimization implemented in the framework of this deliverable.

## Purpose

This document describes the optimizations implemented inside the IBM FoCs – tool for automatic generation of simulation monitors from PSL specifications.

## Intended Audience

This document is intended for anyone who uses PSL to generate HDL simulation monitors.

## Background

Dynamic verification (simulation) is a very important part of the full design verification process. The goal of dynamic verification is to find as many bugs in the design as possible. In order to reach this goal, a variety of techniques are used, and one of them is to equip simulation with simulation monitors.

Simulation monitors are programs that follow after the design behaviour and verify that design indeed behaves according to the specifications set out. The quality of the simulation is measured in the number of bugs found in the post-simulation stage of the design development, and is highly dependant on the quality and amount of monitors used in simulation. Generally, a large number of simulation monitors should be written in order to cover as many design features as possible.

Writing good simulation monitors is a very difficult and time-consuming task. A lot of effort is invested in developing techniques that can make this process easier. One possible technique is to use an assertion language (PSL) to automatically

generate of simulation monitors. This paper presents an optimization of the simulation monitors generated from PSL specifications



# Contents

Table of Revisions .....	iv
Authors .....	iv
Executive Summary .....	iv
Purpose .....	iv
Intended Audience .....	iv
Background.....	iv
List of Figures.....	vii
List of Tables .....	viii
Glossary.....	ix
1 Introduction.....	1
2 Background – FoCs Tool.....	2
Overview .....	2
Building a Simulation Monitor .....	3
Evaluation Process Builder.....	5
Interface Builder.....	5
HDL Code Writer.....	5
Excerpts from User Guide .....	7
Running FoCs .....	7
3 Optimizing the Simulation Monitor.....	9
Optimizing the Automata for Suffix Implication.....	9
Building a monitor for non-overlapping instances .....	9
Overlapping Instances .....	10
Optimizing the Property Evaluation Process .....	11
Excerpts from User Guide .....	12
4 Development Summary .....	13
Feature List .....	13
Work Summary .....	14
Starting Point.....	14
Work Performed.....	14
Supported PSL Subset.....	14
5 References.....	15

## List of Figures

Figure 1: Flow of simulation with monitors .....	2
Figure 2: Connection between design under test and monitor.....	3
Figure 3: Flow of simulation monitor building.....	4
Figure 4: FoCs main panel.....	7
Figure 5: Trace of "assert {[*];a;b;c} =>{d}" .....	11
Figure 6: Division of NFA representing "{[*];a;b;c} =>{d}" .....	12

## List of Tables

Table 1: Feature list .....	13
-----------------------------	----

# Glossary

**Assertion:**

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

**Checker:**

See Simulation Monitor

**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 that is 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 [7], [8].

**HDL event-based process**

An HDL block of statements that are executed sequentially. The block is activated every time the corresponding event is evaluated to "true". The HDL event-based process is itself an HDL concurrent statement. See also [7], [8].

**NFA**

A non-deterministic finite automaton. This is also referred as a *non-deterministic finite state machine*. See also [10].

**Simulation monitor**

An HDL code unit that runs with the design in the simulation, checks the design under test for the user-defined properties, and reports an error for any property that is violated. A simulation monitor is sometimes referred to as "checker".

**Vunit**

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



# 1 Introduction

Rapidly increasing complexity of digital designs requires verification techniques to become more and more sophisticated. Dynamic verification, and more specifically simulation, remains one of most useful methods for finding bugs in complex systems. The quality of simulation is measured by the number of bugs 'escaped' it – the smaller the number of escaped bugs, the more successful the simulation process.

One of the widely-used techniques in simulation is equipping the simulation with "bug hunters" – simulation monitors (also known as "checkers"). The objective of the simulation monitor is to observe the design behaviour during the simulation and catch the situations where design enters erroneous states and behaves in an unexpected manner. As the design becomes more complex and acquires a larger number of possible behaviours, a larger number of simulation monitors is required, and as the complexity of the design behaviour grows, the complexity of the simulation monitors grows as well.

Keeping up with design complexity has a significant impact on the development and maintenance of simulation monitors, rendering it a very time-consuming and inefficient process. This creates a number of challenges, and one of them is how to improve the quality of the simulation monitors while reducing the time required for their development and maintenance. One of the goals of PROSYD project is to address this problem by adapting the PSL language [5] for these needs.

PSL is an assertion language accepted by Accellera and, more recently, the IEEE organization as a standard. It enables formulation of the design specification in a precise and compact manner. This is done by writing assertions that describe the design behaviour in a high level declarative language. PSL suits the concept of assertion-based verification, a verification paradigm in which assertions play the role of bug hunters and verifiers, for both dynamic and static verification.

In this document we present the optimizations of simulation monitors generated by the IBM FoCs (Formal Checkers) tool that produces simulation monitors from specifications written in PSL. Simulation monitors produced by FoCs are programs written in an HDL language (VHDL [7] / Verilog [8]), so they can be adapted to any simulation environment that supports one of these languages. Moreover, FoCs simulation monitors produced by FoCs are synthesizable and can also be used in the post-simulation verification process (e.g. hardware emulation).

The rest of this document is organized as follows: Chapter 2 presents a high-level description of the FoCs tool. Chapter 3 contains a description of the implemented optimizations and Chapter 4 presents the main tool features and describes work that was done in the framework of this deliverable.

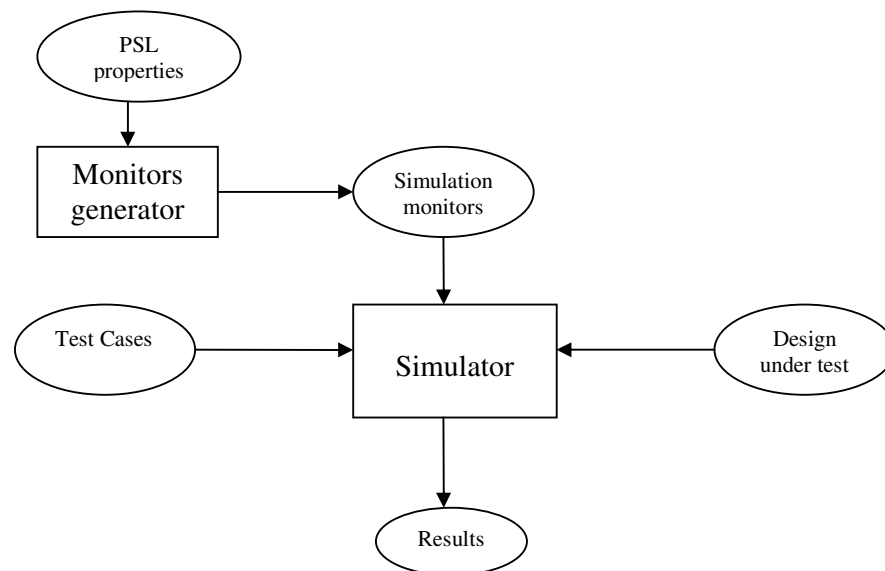
## 2 Background – FoCs Tool

This chapter describes IBM FoCs tool [2] for generation of simulation monitors. Generation of the simulation monitors is based on the algorithmic framework reported in [1], [3] and [4]. Optimization of the simulation monitors that was developed in the framework of PROSYD and was reported in deliverable 3.2/5 research report [5] was implemented inside this tool.

---

### Overview

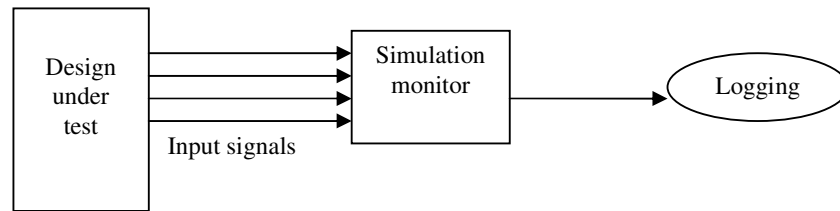
In dynamic verification, PSL properties are usually simulated together with the design under test, to verify the behaviour of the design. For simulation, each PSL property can be represented as a simulation monitor – execution module that observes the design behaviour cycle-by-cycle and evaluates the property. Figure 1 illustrates the flow of the simulation for a design under test with PSL properties:



**Figure 1: Flow of simulation with monitors**

In this flow, simulation monitors are connected to the design under test using an interface – design signals that influence the evaluation process of the property. During each cycle, the property evaluation process is performed. It calculates a

new state of the evaluation using the current values of the relevant design signals and then logs an error message if the evaluation process reaches the error state (Figure 2)



**Figure 2: Connection between design under test and monitor**

The simulation monitor interface contains the design signals that appear in the PSL property and/or modeling layer. Only these signals can influence the behavior of the property evaluation process.

---

## Building a Simulation Monitor

Simulation monitors are built from PSL Verification Units (vunits) [5]. The simulation monitor contains an evaluation process for each property that appears in the vunit and the modeling layer of the same vunit, including all the vunits/vmodes/vprops inherited by it (see "Verification units' inheritance" in [5]).

Figure 3 illustrates the flow of simulation monitor building process. This flow includes the following elements:

- **PSL Verification Unit:** Source PSL code.
- **PSL parser/elaborator:** Module that parses PSL code.
- **Properties:** Intermediate data structures that contain PSL properties that appears in the original PSL vunit (or inherited units).
- **Modeling layer code:** intermediate data structures that contain modeling layer code that appear in original PSL vunit (or inherited units).
- **Non-deterministic automata builder:** Module that converts PSL properties into a representation of non-deterministic finite automata [10]. This module uses algorithms described in [3], [4], whose details are beyond the scope of this document.
- **NFA's:** Intermediate data structures that contain PSL properties represented as non-deterministic finite automata.
- **Specs:** Intermediate data structures that contain the failure conditions of the properties. Failure conditions are based on the property NFA representation.
- **Vacuity specs:** Intermediate data structures that contain the vacuity conditions of the properties. Vacuity conditions are based on the property NFA representation.
- **Evaluation process builder:** Module that constructs simulation implantable code, which runs the property NFA representation and

evaluates failure and vacuity conditions. This module uses algorithms mentioned in [1], [2] (described briefly later in this document).

- **Interface builder:** Module that goes over all intermediate data structures that represent the original PSL vunit and extracts interface signals (described briefly later in this document).
- **HDL code writer:** Module that writes the simulation monitor as an HDL unit [7], [8] (described briefly later in this document).

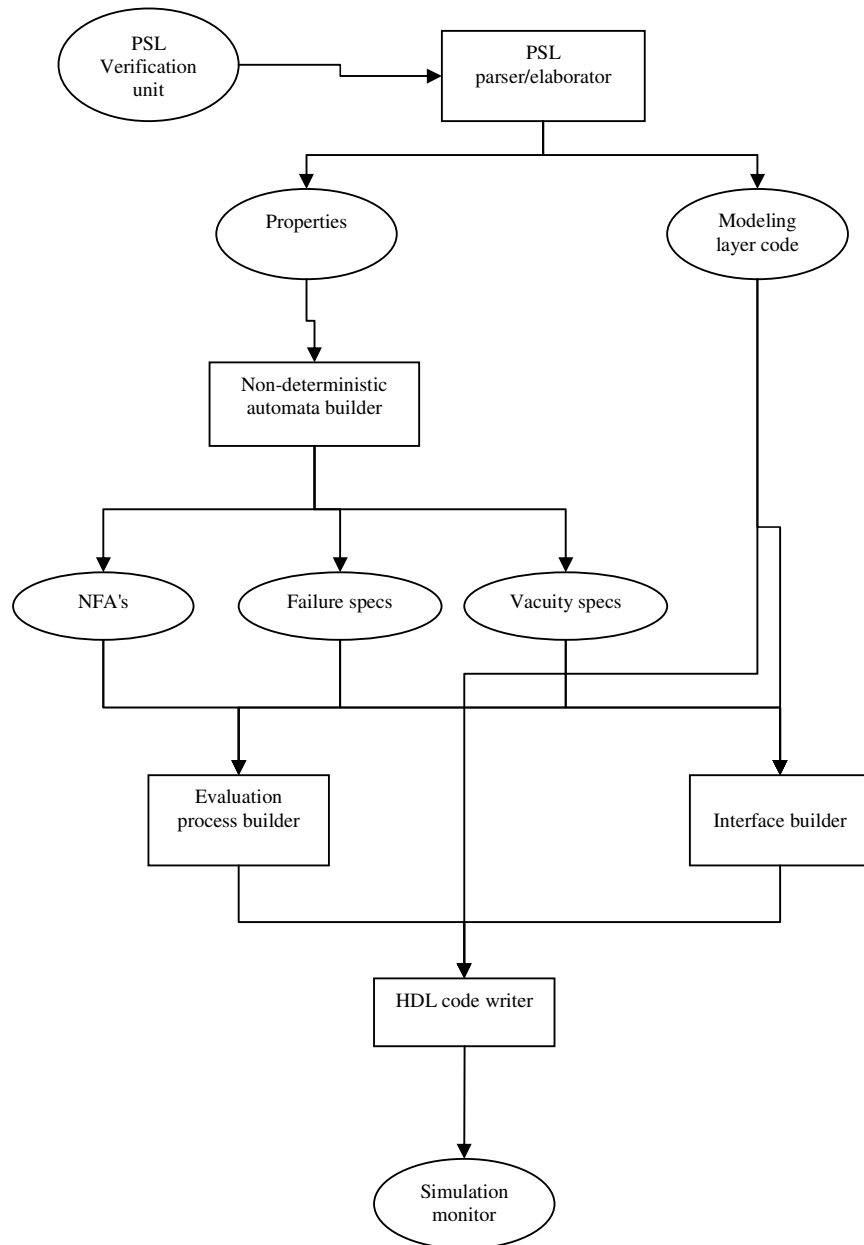


Figure 3: Flow of simulation monitor building

## Evaluation Process Builder

The evaluation process builder receives NFA representations, failure specs and vacuity specs of properties. It then builds the code, which on a cycle-by-cycle basis performs transactions on the NFA of each property. For each cycle, the process evaluates whether the failure spec and vacuity spec hold for the current property. This code together with the monitor interface comprises the simulation monitor. This section offers a brief description of how this evaluation code is built.

Every NFA is represented as a bit vector and every bit represents a different state in the NFA. On every cycle, the value of this bit vector represents an evaluation process state. For every state in the NFA, a condition that should hold true in order to reach this state from its predecessors is computed from NFA transitions. This condition represents the value of the corresponding bit in the next state of the evaluation process – if the condition holds, the value of the corresponding bit in the next evaluation state will be '1'; otherwise it will be '0'. Failure and vacuity specs undergo a simple transformation – instead of examining the current state of the NFA, they are converted to examine which bits in the bit vector are currently asserted.

To correctly evaluate the property, a sequence of reasonable time points on which interface signals will be sampled and evaluation process will be activated need to be defined. Because properties are evaluated on a cycle-by-cycle basis, the time points sequence chosen for the activation of the evaluation process is a sequence of design clock cycles, based on the clock signal that controls memory elements of the design. This clock signal is not determined automatically by the tool, rather it is provided explicitly by the user, either using PSL syntax for clocking properties [5] or via the tool settings [6].

The evaluation process also needs at least one reset cycle in order to set its state to the one corresponding to the initial NFA state. The design reset signal is generally used for this purpose (the reset signal is provided via the tool settings [6])

## Interface Builder

The interface of the simulation monitor is a set of the design signals that influence the evaluation process of the properties. The interface builder identifies this set of signals.

The process of identifying interface signals is iterative. It starts by identifying a set of signals that influence the NFA's, the failure specs and the vacuity specs of all properties. Next, it performs iteration. For every signal in this set, the interface builder examines the modeling layer code for the definition of the behavior of this signal. If the definition was found, the signal is removed from the set, and signals that influence its behavior are inserted in its place. Iterations are performed until the fix point is reached, when a set of signals remains unchanged after the iteration. This final set of signals is the interface of the simulation monitor.

## HDL Code Writer

The HDL code writer module outputs the simulation monitor as a VHDL [7] or Verilog [8] unit. The set of signals received from interface builder becomes the interface of the unit, i.e. the unit's ports. Modeling layer code is output as code in the target language. Output of the evaluation process builder is converted into an edge-triggered block of sequential code (process in VHDL, 'always' block in Verilog). This code is performed on every active edge of the clock signal that defines a time point sequence on which property may be evaluated.

Below is an example of the Verilog simulation monitor produced from a vunit (PSL/GDL flavor)

```
vunit test
{
    define my_a := a1&a2;
    assert always { my_a; b } | => {c};
}
```

Simulation monitor:

```
module test (
    clock,
    reset,
    b,
    a1,
    a2,
    c
);
    input clock;
    input reset;
    input b;
    input a1;
    input a2;
    input c;

    wire my_a;
    reg focs_ok_test_1;
    reg focs_cover_test_1;
    reg [4:0] focs_v_test_1;
    assign my_a = (a1 & a2);
    always @(posedge (clock))
    begin :assert1
        reg [4:0] focs_vout_test_1;
        if (reset) begin
            focs_ok_test_1 = 1'd1;
            focs_cover_test_1 = 1'd1;
            focs_v_test_1[4:0] <= 5'b1111;
        end
        else begin
            focs_ok_test_1 = !((focs_v_test_1[4] & !(c)));
            focs_cover_test_1 = (!(!((focs_v_test_1[4] & !(c)))) | !(
focs_v_test_1[4]));
            focs_vout_test_1[4:0] = {(focs_v_test_1[4] & !(c)), (
focs_v_test_1[3] & b), (focs_v_test_1[2] & my_a), (focs_v_test_1[1]
& 1'd1), 1'd0};
            focs_v_test_1[4:0] <= {focs_vout_test_1[3], focs_vout_test_1[2]
, (focs_vout_test_1[0] | focs_vout_test_1[1]), (focs_vout_test_1[0]
| focs_vout_test_1[1]), (focs_vout_test_1[0] | focs_vout_test_1[4])}
        ;
            if (!(focs_ok_test_1)) begin

                $display("Assertion Failed; vunit : test property 1");
            end
            if (!(focs_cover_test_1)) begin

                $display("Property covered: vunit : test property 1");
            end
        end
    end
end
endmodule
```

# Excerpts from User Guide

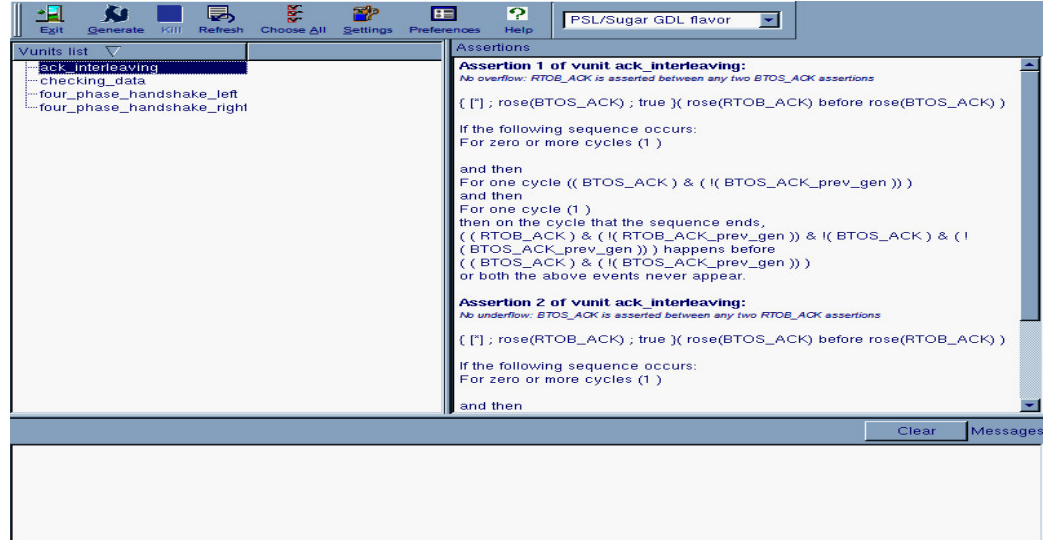


Figure 4: FoCs main panel

## Running FoCs

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

### Checker Generation

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

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.

- Select the target language (VHDL or Verilog), the clock signal name, and the reset signal name (unless you ask FoCs to generate an internal reset).
- Select the output unit (module for Verilog, entity for VHDL) name and output file name.
- Select the rules file. This is the file in which you write your rules.
- 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.
- When you are done, close the Settings dialog and return to the main window.
- 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 appropriate message containing the checker file name will appear in the Messages window.

If you wish to translate several rules into one checker file, select these rules (using control-click, shift-click or the **Choose All** button), and click **Generate**. You will be asked to provide a name for the checker file.

## Batch Mode

Checkers can also be generated from the command line without invoking the GUI.

- To generate a checker for a specific rule, type in the command line: `'focs -batch <rule name>'`.
- To generate a checker for all rules in the rules file, type in the command line: `'focs -batch all'`.
- To generate several rules, type in the command line: `'focs -batch <rules names>'`.

In all cases, 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:

- **-s <file name>** reads the setup file name from the command line instead of reading the default setup file `focs.setup` from the current directory.
- **-r <file name>** reads the PSL file name from the command line instead of reading the PSL file name from the setup file.
- **-o <file name>** gives a specific name for the output file (checker name).

# 3 Optimizing the Simulation Monitor

This chapter describes the optimizations of a simulation monitor that were implemented in the framework of this PROSYD deliverable. Optimizations are based on the algorithmic framework reported in PROSYD deliverables [1], [3]. The first one is a general optimization of automata constructed from the PSL property and it was implemented to run always. The second one is an alternative approach for building a simulation monitor under some runtime assumptions and the user control was added for enabling/disabling it.

---

## Optimizing the Automata for Suffix Implication

The syntax of the weak *suffix implication* operators is  $\{r1\}|->\{r2\}$  and  $\{r1\}|=>\{r2\}$  where  $r1$  and  $r2$  are SEREs.

The suffix implication family of operators, specify that the sequence  $\{r2\}$  holds if the pre-requisite sequence  $\{r1\}$  holds (See section 6.2.1.6.1 in [5] for more details).

At the starting point of the PROSYD project the NFA for  $\{r1\}|->\{r2\}$  was constructed as follows:

1. Separate NFAs were constructed for  $r1$  and  $r2$  using the algorithm in [4].
2. A DFA for  $\{r2\}$  was constructed by reserving a bit in the DFA for each NFA state.
3. The NFA for  $\{r1\}$  and the DFA for  $\{r2\}$  were concatenated to form an NFA for  $\{r1\}|->\{r2\}$ .

Step 2 results an inefficient DFA because even subsets of states that are not reachable from the initial state are represented in the DFA.

In this deliverable we changed step 2 by producing only DFA states that are reachable from the initial state as described in [3].

---

## Building a monitor for non-overlapping instances

This optimization is based on the algorithmic framework reported in [1], Chapter 3. The main goal of this optimization is to build simulation monitors that do not care about overlapping instances of the sub-property. Instead they allow the evaluation

process to perform at most one parallel evaluation of this sub-property at every time point.

Consider the PSL property "assert {[\*]; a; b; c} | => {d}". The NFA that is built for this property using algorithms from [3], [4] contains five states, i.e. five memory elements are required in order to code evaluation process for this property in HDL. However, HDL the coding (shown below) of the "naïve" evaluation process for this property requires only 2 memory elements, for the state representation, and thus seems to be more practical.

```
initial state = 0
if state = 0 and a
    state = 1
else
    state = 0
if state = 1 and b
    state = 2
else
    state = 0
if state = 2 and c
    state = 3
else
    state = 0
if state = 3 and !d report failure and state = 0
if state = 3 and d state = 0
```

Although this evaluation process seems to be correct, it does not handle concurrent evaluations of the property and thus can miss property failures. However, this simpler and more efficient way of building the property evaluation process can still be used under certain circumstances.

## Overlapping Instances

The intuitive meaning of a property having overlapping instances is that during the evaluation process of the property, before one evaluation is finished, another one is started. Let's demonstrate this using an example:

Consider the PSL property "assert {[\*]; a; b; c} | => {d}" and the simulation trace shown in **Error! Reference source not found.** As it can be seen from the trace, in Cycle 1 evaluation of the "{a; b; c} | => {d}" sub-property is started, and is continued in Cycles 2 to 4. In Cycle 3, another evaluation of this sub-property is started, before the first one is finished. This situation demonstrates overlapping instances on the sub-property "{a; b; c} | => {d}".

Applying the evaluation process described previously, in Cycle 1 the state will become 1, in Cycle 2 it will become 2, in Cycle 3 it will become 3 and in Cycle 4 it will become 0. Here the evaluation process will stop until the next time "a" is asserted. But "a" was asserted in Cycle 3! Moreover, this assertion of "a" leads to the property failure in Cycle 6, and this failure will be missed by the evaluation process.

Now, consider that this situation is impossible. Say the behaviour of the design under test does not allow transaction "{a; b; c}" to be started in the middle of another "{a; b; c}" transaction. In this case, overlapping instances are not allowed for the sub-property "{a; b; c} | => {d}" and the evaluation process for this sub-property can be implemented using the efficient implementation shown previously. Usually, the information about whether overlapping instances are allowed or disallowed for the sub-property cannot be derived from the property itself and can only be assumed while building the evaluation process for the property.

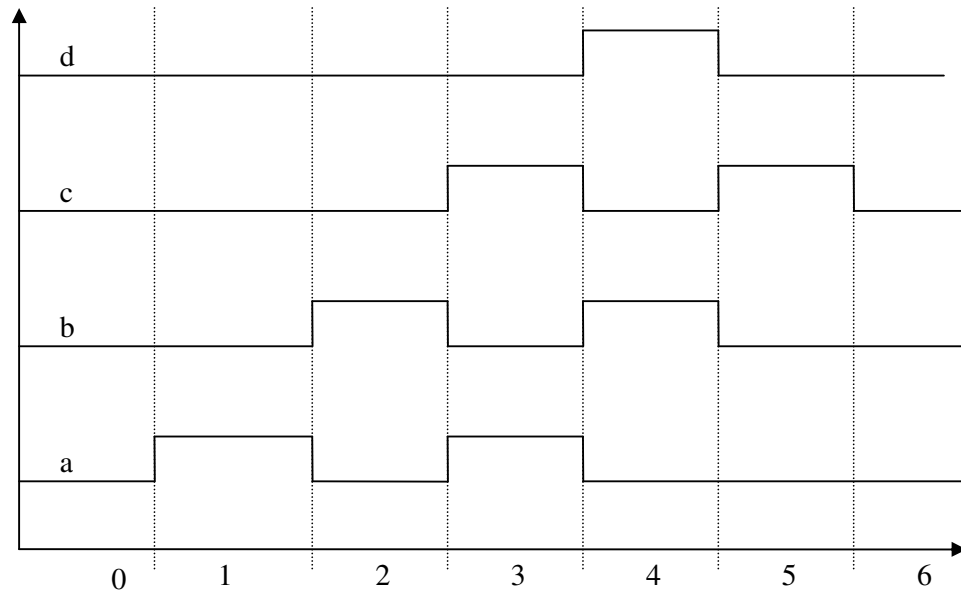


Figure 5: Trace of "assert {[\*];a;b;c}|=>{d}"

The question to be asked is 'why is this more efficient implementation applied only to the sub-property "{a; b; c} | => {d}" and not to the whole property?'. The answer to this question is that on the whole property, it is impossible for no overlapping instances to occur, because the property starts with "[\*]" ("true[\*]"), and no design policy can disallow "true" to happen.

## Optimizing the Property Evaluation Process

Optimization is a part of the evaluation process builder module. Depending on the user's choice, the evaluation process builder can produce either a regular evaluation process or an evaluation process with a no-overlapping assumption on a maximally possible sub-property.

First, the optimization process finds an optimization-appropriate barrier. This is a state that is used to divide the NFA representation of the property into a part that can not be optimized and a part that can be optimized [1]. There can be more than one such a state in the NFA, and the intent is to find a barrier that maximizes the part that can be optimized. **Error! Reference source not found.** demonstrates an example of such a division of the NFA, which represents the PSL property from the previous section ("assert {[\*]; a; b; c} | => {d}"). The double-lined state (State 3) is an optimization-appropriate barrier and the framed part of the NFA is the part for which an optimized evaluation process can not be built [1].

Next, the property evaluation process is built. For the NFA representation of the sub-property that cannot be evaluated using the optimized evaluation process, this process is built as described in "Evaluation Process Builder" section. For the NFA representation of the sub-property that can be evaluated using the optimized evaluation process, the construction of the evaluation process works as follows: every state is assigned an integer value (starting with '1', the '0' value is saved for "no transition") and the bit vector that can hold the maximal integer value of the state is produced [1]. Actually, this bit vector will consist of a  $\log_2 n$  number of bits, where "n" is the number of states in the NFA representation of the sub-property. Then, for every bit a condition is built that is required to hold in order to

have this bit asserted on the next state of the evaluation process. The condition is built from the transitions of the NFA as follows: for every integer value that represents the state, if the current bit is asserted in this value binary representation, then transition to this state will cause the bit to be asserted on the next state of the evaluation process.

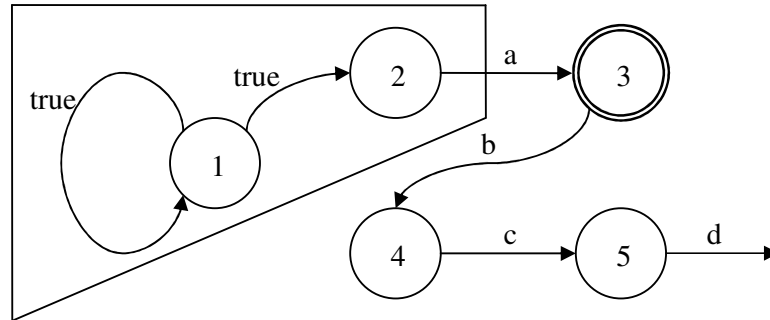


Figure 6: Division of NFA representing "[\*];a;b;c|=>{d}"

Finally, two bit vectors are concatenated together. The failure and vacuity specs of the appropriate property undergo the transformation required to make them examine the correct values of the slice of the bit vector that implements the state of the optimized evaluation process of the sub-property [1].

## Excerpts from User Guide

Below is a citation from FoCs User Guide [6], Section 5.4.12. Text in *italics* describes the setting added for the control of the discussed optimization.

### Optimization Level

When generating checkers, optimizations may take place, in order to generate an effective checker.

- **None** - no optimizations.
- **Renaming optimizations** (default) - these optimizations are effective when generating a checker from multiple vunits, which share parts of the environment. For other cases they are neither effective nor they affect the checker generating performance.
- **Full optimizations** - the full-blown FoCs optimizations set will take place. Please note that there are cases in which the full optimizations slow down the generation process so badly, that the user might prefer to avoid them.
- **Non-overlapping optimization** – *when this optimization is chosen, FoCs will produce checkers with smaller state machine, that does not care about overlapping instances of the property*

# 4 Development Summary

## Feature List

	Present	Ref.
<b>Mandatory</b>		
• 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 [9]
○ Support for GDL and Verilog flavour	yes	
• Standard output languages for design – Verilog, VHDL	yes	Chapter 2
• Graphical user interface	yes	Chapter 2, FoCs User Guide [6]
• Optimized translation of properties	yes	Chapter 3
<b>Desirable</b>		
• Mechanism for connection of monitors to industrial simulators	yes	FoCs User Guide [6]
<b>Nice-to-have</b>		
• Support for Other flavours	no	

**Table 1: Feature list**

**Target operating systems:** Linux, Solaris, AIX

**Coding standards:** The tools are implemented in C and C++, according to the coding standards of 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. The generation of simulation monitors (optimized and non-optimized) was tested on about 1000 PSL properties (monitor generation, compilation and simulation).

---

## Work Summary

### Starting Point

At the starting point, FoCs tool [2] has the capability to produce regular simulation monitors, as described in section "Building a Simulation Monitor". The FoCs tool has a graphic user interface and supports PSL GDL and Verilog flavors. FoCs also has the ability to produce connection mechanisms for simulation monitors that connect to the ModelSim and NCSim environments or a generic connection mechanism (pure Verilog/VHDL instantiation of unit) that can be used to port the simulation monitor to any simulation environment.

### Work Performed

#### Evaluation Process Builder Module

Optimization of the automata construction, as described in Chapter 3, Section "Optimizing the Automata for Suffix Implication" was implemented.

A generation of optimized evaluation process for the sub-property with no overlapping instances, as described in Chapter 3, Section "Building a monitor for non-overlapping instances", was implemented.

#### GUI Module

An option to switch between regular and optimized generation of the simulation monitor was added.

---

## Supported PSL Subset

The subset of PSL supported by FoCs is described in Section 5 in [9]. Constructs listed in this section are supported except for the constructs marked "RuleBase PE only".

# 5 References

1. D. Pidan, S. Keidar-Barner, D. Fisman, M. Moulin – Optimized algorithms for dynamic verification. PROSYD deliverable 3.2/5.
2. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, Y. Wolfsthal. FoCs – Automatic generation of simulation checkers from formal specifications. In International Conference on Computer Aided Verification, volume 1855 of Lecture notes in Computer Science. Springer-Verlag, 2000.
3. S. Ben-David, R. Bloem, D. Fisman, A. Griesmayer, I. Pill, S. Ruah – Automata Construction Algorithms Optimized for PSL. PROSYD deliverable 3.2/4
4. S. Ben-David, D. Fisman, and S. Ruah. Automata construction for regular expressions in model checking, June 2004. IBM research report H-0229. [http://domino.research.ibm.com/library/cyberdig.nsf/papers/A14E9FE3829B557785256EE6005006A5/\\$File/H-0229.pdf](http://domino.research.ibm.com/library/cyberdig.nsf/papers/A14E9FE3829B557785256EE6005006A5/$File/H-0229.pdf)
5. Accellera. Accellera Property Language Reference Manual. In <http://www.eda.org/vfv/docs/PSL-v1.1.pdf>, June 2004
6. FoCs User Guide. Available as a part of the FoCs product
7. The Institute of Electrical and Electronics Engineers, Inc. IEEE Standard VHDL Language Reference Manual, 2000.
8. The Institute of Electrical and Electronics Engineers, Inc. IEEE Standard Verilog Hardware Description Language, 2001.
9. Porting of IBM tools to support Accelera-Standard version of PSL. PROSYD deliverable 3.3/2
10. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1979.