Improving the Digital Design with Semi-formal Specification

Torres-Roman, Deni1,2  Cortez-Gonzalez, Joaquin2  Gonzalez-Torres Raúl Ernesto2
1Department of Electronics, Systems and Computer Science
ITESO, Guadalajara, México
2Department of Electrical Engineering
CINVESTAV-IPN, Guadalajara, México
Phone (33) 31345570  Fax (33) 31345579
dtorres@iteso.mx  jcortez@gdl.cinvestav.mx  regonzal@gdl.cinvestav.mx

Abstract

In this work, an improvement of the traditional design methodology is proposed. The major change is the use of semi-formal specification for the implementation and the establishment of properties for the formal verification. From semi-formal specification, assertions were written using Property Specification Language (PSL) for an alignment circuit. Finally, a set of properties for the verification of this module were established and proved using two verification tools. Our statistics proved that, considerable design time was saved, and the number of versions was low.

Index Terms.- Assertion based verification, semi-formal specification, model checking tools, properties, assertions

1. Introduction

Ensuring functional correctness on RTL designs continues being one of the greatest challenges for ASIC’s design teams. With ever increasing design sizes, verification becomes the bottleneck in modern design flows. Up to 80% of the overall costs are due to the verification task [8]. Generally, the verification engineers uses the simulation to demonstrate that the design’s implementation satisfies its specification using a black-box testing approach. They create a model of the design written in a Hardware Description Language (HDL) like Verilog or VHSIC Hardware Description Language (VHDL). After they creates a testbench, which includes the model for Device Under Verification (DUV), then input patterns, so-called simulation stimuli, are created to reflect typical or critical execution traces and apply to the DUV. Also, the functional verification process will never guarantee that a design is error-free [10], i.e., it is, until now, a time consuming task, which can only demonstrate the presence of errors but not their absence. One way to improve significantly the performance and saving time of this traditional verification is to increase the observability and the controllability of the design during the verification process. We can do this using assertions and formal verification techniques in a process called Assertion Based Verification (ABV). If the implemented code of a design, is represented by \( \psi_{IMP} \) and the set of the requirement specifications by \( \rho_{SPEC} \), then with ABV we will prove that the following logical relationship is true:

\[ \psi_{IMP} \rightarrow \rho_{SPEC} \]  \hspace{1cm} (1)

A formal verification tool should be able to prove mathematically this equation. The main objectives of this paper are:

- To establish a set of properties written as assertions using PSL, which allows the development of a formal specification for the aligner module for SONET/SDH
- To use a design cycle which involves the use of a formal verification tool (Safelogic Verifier) for design validation
- To apply the methodology ABV for the verification of aligner module showing major statistics

2. Assertion Based Verification (ABV)

2.1 Definition of ABV

The purpose of the ABV [8] is to convert functional features of a specification into explicit logical properties. The intended behavior of a design can be
defined as a set of logical and timing relationship called properties. Properties describe logical behaviors of the system over the time. An assertion can be considered as an implementation of a property. Assertions can be utilized as an input to both simulation and static verification tools. There is some languages and tools for implementation available including SystemVerilog [9], but only a few are accepted as standards. The current available static verification solutions are capable of verifying functional equivalence of different implementations (e.g. Register Transfer Level and Gate Level Descriptions). However, these tools have not been designed to verify an implementation against its specification. Each assertion verifies a certain functional feature of the design. Assertions can be embedded to HDL source code or they can be located in a separate property file. In both cases they are compiled as ordinary HDL descriptions. The target is to improve both verification quality and efficiency. Assertions can be used in methodologies based on simulation as hardware monitors to complement the existing verification environment. In static methods, assertions provide also an alternative verification path to verify all computational paths of the design.

2.2 Property Specification Language (PSL)

PSL developed by Accellera [4], is a language for the formal specification of hardware. The PSL description was taken of [4]. It is used to describe properties that are required to hold on the design under verification. PSL can be used to capture requirements regarding the overall behavior of design, as well as assumptions about the environment in which the design is expected to operate. A PSL specification consists of a set of assertions. PSL provides a standard means for hardware designers and verification engineers to document the design specification rigorously. PSL is easy to learn, write, read and it has a concise syntax and rigorously well-defined formal semantics. Both VHDL and Verilog languages are provided. PSL has been divided into four different layers: boolean, temporal, verification, and modelling layers. The boolean layer is used for basic digital expressions and the temporal layer is used to implement expressions in time domain.

2.2 Safelogic Verifier

Safelogic Verifier is a model checking tool developed by Safelogic [5] and it has already been used for circuit verification. It performs an automatic formal verification without user intervention of hardware designs implemented with VHDL language, using PSL assertions.

3. The Design Cycle

Our objective with the following proposal, see Fig. 1, is to introduce the use of assertions within the design cycle, using PSL properties for synchronous circuits.

Fig 1. Design Cycle

1. Stage of specification
Requirements Specification (or informal problem description). It defines a set of general characteristics \( \rho = \{ \rho_1, \rho_2, \ldots, \rho_n \} \) for the design of the specific circuit. Each requirement must capture only a particular behavior of the design. Each behavior will be expressed by a property. PSL assertions were used to implement the properties.

Architecture. Block diagrams, function tables, and timing diagrams were constructed to define the behavior of each system block, which must satisfy the requirements specification. Timing diagrams are particularly useful to define PSL assertions, because they show the expected temporal behavior of each system block.

Semi-Formal Specification. Making a refinement of the original requirements \( \rho \) jointly with the architecture document, a semi-formal specification is obtained \( \varphi = \{ \varphi_1, \varphi_2, \ldots, \varphi_m \} \). A semi-formal specification is a form to express a requirement using key words that represent the functionality of logical, temporal operators and specific statements associated to the design. From the semi-formal specification it is possible:

- To obtain a set of PSL assertions \( \xi = \{ \xi_1, \xi_2, \ldots, \xi_n \} \)
- To implement the code in VHDL

This way can ensure that each requirement \( \rho_i \in \rho \) is described by its semi-formal specification.

2. Stage of implementation
VHDL Implementation & Formal Verification. From the semi-formal specification, the VHDL
implementation $\Psi_{\text{VHDL}}$ and a set of PSL assertions $\xi = \{\xi_1, \xi_2, \ldots, \xi_p\}$ for each block of the circuit are obtained.

3. Stage of verification

Then, each block was verified using the model checking tool Safelogic Verifier. It allowed to assure that its implementation in VHDL, satisfies the initial requirements of the specification, i.e

$$\Psi_{\text{VHDL}} \rightarrow \rho \quad (2)$$

When an some error is found within of $\Psi_{\text{VHDL}}$, a counterexample is generated and it is used to correct $\Psi_{\text{VHDL}}$. Also to validate the design of the circuit, a functional verification is executed to assure the global correctness of the implementation. The functional verification is necessary to detect errors in $\varphi$ generated during the refinement of $\rho$.

4. Stage of results

In this stage are generated reports containing statistics about formal and functional verification. Therefore, conclusions about verification process will be given.

3.1 Generation of PSL assertions and semi-formal specification from requirements specification of the design

An assertion is built from Boolean Expressions, which describe behavior over one cycle, sequential expressions, which describe multi-cycle behavior, and temporal operators, which describe relations over time between Boolean expressions and sequences. Assertions are able to capture complex intended behaviors of the design in a formal way. Below is an example at low-level of how a PSL assertion can be extracted from one design requirement.

**Reset requirement (requirement specification)**

[\text{R\textunderscore SM\textcolon} 6] The design must allows the initialization of the circuit

**Semi-formal specification**

[\text{P\textunderscore SM\textcolon} 1] ResetOp

Always happens that:

If an operation of reset is required, then in the following edge clock FPG\textunderscore FPind output signal will be loaded with the initial value (LOW)

**PSL assertion**

property ResetOp is always(FPG\textunderscore Reset\textunderscore in $\rightarrow$ next (FPG\textunderscore FPind\textunderscore out = LOW));

4. Aligner Module for SONET/SDH

The alignment process intention is to identify the alignment pattern, or a part of it, contained in each one of the input frames. This is an important function because delimits bytes and input frames, providing synchronization for the correct operation of the remaining modules as much processing as generation in SONET/SDH [11] devices.

The aligner [11] blocks, see Fig. 2, are: (A) Frame Pattern Detector, (B) Sync Process, (C) Data Output Array, (D) Active Channel Selector and (E) Frame Position Detector. Then, the main task, during the alignment process, is to find this pattern within each frame, and to keep the data transmission for all times. This process is controlled by a Finite State Machine located in the Controller block, called Sync Process.

![Fig. 2 Aligner’s Block Diagram](image)

The verification of the aligner described above must be done by validating each one of their blocks. The verification process of the components of the aligner can be carried out generating a set of PSL assertions for each component and applying formal verification using the model checking tool safelogic verifier.

A. Requirement Specification for the Aligner Module

$\rho_0$. Reset Operation. An input port “Reset” allows the initialization of the circuit.

$\rho_1$. Transition to synchrony state. The module will be declared in “synchrony state”, activating a “synchronous system” signal after the “detected frames counter” reaches a specific value.

$\rho_2$. Permanence in synchrony state. If the module reaches the “synchrony state”, it must verify that the “frame pattern detect” signal appears each 125µS. Otherwise a “loss frames counter” must be incremented. If the “frame pattern detect” signal appears correctly then the “loss frames counter” must be set to zero.

$\rho_3$. Transition to out of frame state. If the module reaches the “synchrony state” and the “loss frames counter” reaches a “four” value, the module will change to the “non-synchronous state”, deactivating the “synchronous system” signal.

$\rho_4$. Out of frame state indication. If four consecutive times the “frame pattern detect” signal does not appear each 125uS, the module will change to the “out of frame state” and indicate it with an active signal.
ρ5. Maximum time to detect an out of frame state. The maximum detection time of an “out of frame state” must be 625µS, 5 frames, for a random SONET/SDH input frame.

ρ6. Loss of the “out of frame” state. The module will leave the “out of frame state” (OOF) when two "frame pattern detect" signal appear each 125 µS.

B. Architecture of Aligner

Fig. 3 shows the input and output signals of the aligner module. After a reset operation, the circuit starts in the state “out of frame” indicated by “OOF_out” output. Whenever the alignment pattern is detected, some internals counters are incremented. When “syncCounter” reaches the established configuration by “sync_config_in”, the “synchrony state” is declared, and it is indicated by the “sync_out” and “sync_aux_out” outputs. Once the synchrony state is declared, the module will return to the state “out of frame” when four patterns of alignment consecutive are lost, which is controlled by “lossCounter”. Also, whenever the frame last byte is detected it will be indicated by the port “last_byte”. 

C. A set of assertions for the Aligner Module

Making a refinement of the original requirements ρ jointly with the architecture document, a semi-formal specification ϕ = {ϕ1, ϕ2, …, ϕn} for each block of the aligner module is obtained. From the Semi-formal specification it is possible obtain a set of PSL assertions ξ = {ξ1, ξ2, …, ξp} and to implement the code in VHDL for each block of the aligner. We will use the following notation to make reference to the implementations in PSL and VHDL for each block of the aligner circuit in the rest of the document.

ξRFILL PSL Specification of Register Fill
ξFPDet PSL Specification of FP Detector
ξSyncP PSL Specification of Sync Process
ξDOA PSL Specification of Data Output Array
ξALIGNER PSL Specification of Aligner Complete
ψRFILL VHDL Implementation of Register Fill
ψFPDet VHDL Implementation of FP Detector
ψSyncP VHDL Implementation of Sync Process

ψDOA VHDL Implementation of Data Output Array
ψALIGNER VHDL Implementation of Aligner Complete

The formal verification process tries to prove that:

ψRFILL → ξRFILL (3)
ψFPDet → ξFPDet (4)
ψSyncP → ξSyncP (5)
ψDOA → ξDOA (6)
ψALIGNER → ξALIGNER (7)

The properties classification for each block of the aligner, see Table 1 is based on the proposal of F. Torres [3].

TABLE 1. Properties Classification for Aligner Module

<table>
<thead>
<tr>
<th>Category</th>
<th>ξRFILL</th>
<th>ξFPDet</th>
<th>ξDOA</th>
<th>ξSyncP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init State Reset</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>State Reachability</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>State Connectivity</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>State Transitions</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>State Loops</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>State non-connectivity</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Counters</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>Outputs Behavior</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>TOTAL PSL ASSERTIONS</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>37</td>
</tr>
</tbody>
</table>

Besides, for the control of the verification process of each block we used the next set of metrics: Number of assertions of PSL against the number of lines of code in VHDL, writing time of properties PSL, number of syntax errors for each version of the PSL specification, number of developed versions for each specification in PSL, errors detected in the semi-formal specification and the implementation RTL and verification time used by Safelogic Verifier.

5. Results

For this work we have three different implementations of the aligner for SONET/SDH. The first ψimpO designed by J. R. Verdín [11], and previously verified with functional tests, was developed from the requirements specification, i.e. this implementation was not intended for formal verification. The second implementation ψimpF by J. Moreno, was elaborated using our semi-formal specification and the obtained code is an optimized version with a minimum of code lines, so we called it
Free Designer Implementation. And the last one \( \psi_{imp}^{PSL} \), also by J. Moreno, was elaborated directly from our semi-formal specification, which previously was verified with functional tests, so we called it PSL-Assertion-Based. We choose \( \psi_{imp}^{PSL} \) as study case for our formal verification, it was applied to each one of the modules of the circuit aligner in individual form, later formal verification was carried out to the complete aligner using for both cases, the model checking tool safelogic verifier. From the learning process for writing the specification and formal verification of the aligner (see Table 2, Table 3, Fig. 4, Fig. 5, Fig. 6, Fig. 7) we concluded that:

**TABLE 2. Number of PSL Assertions and Number of VHDL Code Lines for the Aligner Module**

<table>
<thead>
<tr>
<th>BLOCK</th>
<th>NUMBER OF ASSERTIONS</th>
<th>NUMBER OF VHDL CODE LINES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Fill</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>FP Detector</td>
<td>5</td>
<td>104</td>
</tr>
<tr>
<td>Sync Process</td>
<td>37</td>
<td>140</td>
</tr>
<tr>
<td>Data Output Array</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>Complete Aligner</td>
<td>46</td>
<td>429</td>
</tr>
</tbody>
</table>

- All blocks of the aligner had different VHDL code lines. The number of assertions for each block was variable and it depends directly of abstraction level or the functionality of the design caught for PSL assertions that we will use to verify our design.
- All PSL specifications for each one of the blocks of the aligner had few errors (syntax and semi-formal specification errors) and was corrected in a very short time because it was written directly from the semi-formal specification.
- In the VHDL implementation only in the block FP Detector were find eight errors of VHDL code.
- We found six errors in some properties (see Table 6). This errors was due to the lack of precision in the semi-formal specification obtained. Ambiguities in the semi-formal specification led to an interpretation error.

Finally, the employed total time to obtain the final versions of each one of the specification in PSL for each block is shown in the Fig. 7. Components of this time are: *writing time* of all the assertions, *correction time of syntax errors in the specification*, *correction time of errors detected in the semi-formal specification*, and *correction time of errors detected in the implementation of VHDL code*. Let \( \xi^{SyncP}(i) \) be the PSL specification in the version \( i \). The set of assertions related with \( \xi^{SyncP}(i) \) doesn’t generate counterexamples, for \( i = 1,2,3 \) Therefore, some assertions, for generation of counterexamples, were included in \( \xi^{SyncP}(i) \) to find all possible errors in \( \psi^{SyncP} \).

**TABLE 3. Errors Found in \( \xi_{ALIGNER} \) and \( \psi_{ALIGNER} \)**

<table>
<thead>
<tr>
<th>BLOCK</th>
<th>ASSERTIONS ERRORS</th>
<th>VHDL CODE ERRORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Fill</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FP Detector</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Sync Process</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Data Output Array</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Complete Aligner</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

**Fig. 4 Syntax errors to write the PSL specification for aligner module**

**Fig. 5. Detected errors in the VHDL implementation of each block for aligner module**

**Fig. 6. Detected errors in the semi-formal specification of each block for aligner module**
6. Verification Time Used for Safelogic Verifier for Each Block of the Aligner Module

From the developed PSL assertions for each module, see Table 1, we selected all the assertions to make a comparison of the verification time versus the number of assertions for each module. The tests were executed in a computer with a Pentium IV processor at 2.8 GHz, 256MB of RAM Memory, and Windows Xp Professional. Furthermore, we specifically considered the obtained results for all assertions with frame size of 10 bytes. The obtained results are shown in Table 4.

<table>
<thead>
<tr>
<th>BLOCK</th>
<th>VERIFICATION TIME WITHOUT COUNTER-EXAMPLES</th>
<th>VERIFICATION TIME WITH COUNTER-EXAMPLES</th>
<th>NUMBER OF COUNTER-EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Fill</td>
<td>0.20 Sec</td>
<td>0.20 Sec</td>
<td>0</td>
</tr>
<tr>
<td>FP Detector</td>
<td>0.20 Sec</td>
<td>0.20 Sec</td>
<td>0</td>
</tr>
<tr>
<td>Sync Process</td>
<td>0.30 Sec</td>
<td>2.40 Sec</td>
<td>6</td>
</tr>
<tr>
<td>Data Output Array</td>
<td>0.20 Sec</td>
<td>0.20 Sec</td>
<td>0</td>
</tr>
<tr>
<td>Complete Aligner</td>
<td>1.20 Sec</td>
<td>3.90 Sec</td>
<td>0</td>
</tr>
</tbody>
</table>

7. Conclusions

The majors contributions of this work are:
- A set of PSL properties was established to carry out the formal specification and verification of a aligner circuit
- The structure of these properties can be used, in general, for different synchronous circuits
- From semi-formal specification, assertions using PSL were written and the code in VHDL for an alignment circuit was implemented
- In [3] we showed statistics that establish the impact provided by the use of assertions in the design cycle of digital circuits; specifically in the implementation of the code using HDL.

Our statistics in the verification process of aligner module show that:
- The number of versions for the PSL implementation of each block of the aligner module was relatively low
- The writing time decreased from version to version except in $\xi_{\text{SyncP}}$. This exception occurred due to some limitations in the tool
- These metrics should be used, joint to classical ones, to control the whole design process
- Formal specification can not discover possible errors at the refinement of general requirements to semi-formal specification. To find these errors it is useful to maintain functional verification
- Errors, that did not find the functional verification, were found with formal verification
- Semi-formal specification helps the designer to decrease the time dedicated to some simulations
- Therefore, ABV is useful not only for the verification process but also for the implementation of digital designs in VHDL.

Acknowledgements

The authors gratefully acknowledge the contribution of J. Moreno, Research Assistant at CINVESTAV-IPN, Guadalajara. Besides, they acknowledge, too, safelogic inc. to allow us the use safelogic verifier for the formal verification process of our code.

8. References