On-Line Test Vector Generation from Temporal Constraints Written in PSL

Yann Oddos, Katell Morin-Allory, Dominique Borrione
Tima Laboratory, 46 avenue Félix Viallet 38031 Grenoble Cedex, France
yann.oddos@imag.fr, katell.morin@imag.fr, dominique.borrione@imag.fr

Abstract—We propose an efficient solution to automatically generate test vectors that satisfy an assumed property written in PSL. From a “foundation language” formula, we build a synthesizable generator that produces random temporal test vectors compliant with the formula. Generators are space and speed efficient when synthesized on FPGA, and their connection to the device under test is a portable solution across verification platforms for simulation and emulation.

I. INTRODUCTION

The verification of complex designs requires a stepwise approach by which each component is first individually checked before performing the verification of the full interconnected system. In this context, it is of utmost importance to generate, for each module, the test vectors that faithfully represent the communication signals coming from their environment. The complexity of the test vector generation should be much less than the model of the environment it stands for. Moreover, it may be the case that only part of the environment is significant for a particular module. Characterizing the most important test sequences, or constraints on its inputs to ensure proper usage of the module, can be done in declarative form using a property specification language. In this paper PSL is considered. Applying the “assume-guarantee” paradigm, we specify the environment behavior by one or more assumed properties written in the “foundation language” of PSL, and we generate test vectors that satisfy the assumed properties on the module inputs. Our original solution relies on synthesizable generators that may be connected to the design under test in a simulator or an emulator. This solution is portable across verification platforms and is particularly efficient when implemented on FPGA.

The sequel of this paper is organized as follows: in the next section, we present a state of the art and contrast our method to those generally found in the literature and in commercial tools. Section III presents the overall principles for the construction of test vector generators. Sections IV and V give the heart of our solution, built around a library of primitive components and a systematic method to interconnect them. Section VI overviews our implementation, its correctness verification and gives performance results on the synthesized generators. We finally present our conclusions and on-going works.

II. STATE OF THE ART

The most common technique used in industry is the generation of random test vectors. This method is very easy, but many vectors do not comply to the specification and must be eliminated. This has two main disadvantages: the excessive time required to test the design sufficiently and the difficulty to reach corner test cases. This method is thus complemented by various approaches. We briefly present a few of them:

- Algorithmic description of the environment. The context of the module is described at a more abstract level, either in a conventional hardware description language, or using a special purpose algorithmic test generation and analysis language such as “e” [1].
- The generation under constraints. This method consists in writing specifications under the form of Boolean constraints. A software tool takes these formulas as inputs and provides test vectors satisfying the constraints as outputs [2]. The set of properties that can be submitted to the tool is very restricted and the expressive power is not sufficient to generate interesting test vectors for a real design.
- Extracting constraints by slicing the design [3], [4]. The design is cut in several parts, between the registers. Constraints are extracted from each part and recomposed to make the global constraint for the circuit. This technique allows to reduce the set of test vectors by eliminating those which do not satisfy the specification and the redundancies.
- Within the framework of the Prosysd project, a waveform generator was made to help designers understand PSL properties [5]. This tool takes as inputs a PSL property P and builds an illustrative trace for P. The user can define constraints in order to explore the submitted property. It relies on two possible representations of P: Büchi automaton or SNF (Separated Normal form). Then the tool is combined with a model checker to produce the trace according to the input property. The advantages are the possibility to constrain the trace by fixing values at several cycles and to show the traces for each subproperty of P. But the limits of this tool are due to the limits of the model checker (state explosion).

Our method takes a radically different approach to avoid facing the state explosion problem. Rather than analyzing pieces of a design to extract constraints for other sequentially connected parts, or taking an automata theoretic approach, we view the constrained test generation as a design problem in its own right, and the property specification language as a high-level
declarative design description language. We translate the PSL formula as the VHDL model of a hardware module which directly generates interesting test vectors and which can be synthesized in order to test the design in real conditions. As a possibly infinite number of test vectors may satisfy the formula, our solution is partial. It embeds a random time interval generation mechanism, but does not need to eliminate generated vectors.

III. TRACE GENERATOR CONSTRUCTION

A. The PSL subset covered by the method

PSL stands for Property Specification Language. It is an IEEE standard [6] developed to complement design descriptions with logic and temporal constraints, coverage requirements, and asserted properties to be checked by static or dynamic verification. In this paper, since we are interested in generating linear test sequences, we shall focus on the Boolean layer, sequence regular expressions (SERE’s) and foundation language (FL) expressions. The optional branching extension (OBE) that serves the purpose of property checking by symbolic state traversal of the circuit model, is irrelevant to our application and will not be discussed further.

In the following, we adopt the VHDL syntax for PSL. We shall use the following formula F as a running example.

\[ \text{property } F \text{ is always } A \rightarrow \text{next}[3] (B \text{ until } C); \]

**Meaning:** each time signal \( A \) takes value ’1’, it must be the case that three clock cycles later \( B \) takes value ’1’ and keeps value ’1’ until \( C \) is ’1’.

Figure 1 illustrates a possible trace satisfying \( F \). The vertical dotted lines represent the successive clock cycles. Signal \( A \) takes value ’1’ at clock edge 2. Starting from clock edge 5, \( B \) must have value ’1’ and remain ’1’ until \( C \) takes value ’1’ (at clock edge 9). At cycle 9, the sub-property \( A \rightarrow \text{next}[3] (B \text{ until } C) \) holds.

Holds

\[
\begin{align*}
A & \\
B & \\
C &
\end{align*}
\]

\[
\begin{align*}
0 & 1 & 2 & 3 & 4 & 5 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{align*}
\]

Fig. 1. A trace satisfying \( F \)

The IEEE standard identifies a “simple subset” of PSL for which properties can be evaluated on the fly, during simulation or execution. In this subset, time advances from left to right through the property. We limited our generators to formulas written in the subset, which is most widely understood and de facto recommended in the guidelines for writing assertions [7]. We first totally eliminated the negation from our formulas. The effect was to “restrict” the test generation to the coverage of the “assumed” properties. Then we introduced the negation, currently on Boolean operators only, and generated more general patterns, in particular patterns that would also satisfy an implication by non satisfying its left operand.

**Example:**

\[
\begin{align*}
\text{property } F_1 & \text{ is } A \rightarrow \text{next}[3] (B \text{ until } C); \\
& \text{ assume } F_1; \\
\end{align*}
\]

The assumption is vacuously satisfied by all test patterns in which \( A \) keeps value ’0’. In contrast,

\[
\begin{align*}
\text{restrict } \{A\}; \\
\end{align*}
\]

forces \( A \) to take value ’1’ at time 0, and therefore its conjunction with the above assumption forces the generation of patterns that satisfy \( (B \text{ until } C) \) starting from time 3. Thus the “restrict” statement forces the “coverage” of property \( F_1 \) at time 0.

In this paper, we present a method to automatically transform a PSL formula into a VHDL component that generates test vectors such that the formula holds. Our generated signals cover and satisfy the property, in a non vacuous way, without imposing that the property be covered at all times. In the remainder of this document, we take as granted the fact that an assumed property is not inconsistent. For instance, \( (B \text{ before! } B) \) cannot hold for any test sequence. Verifying property consistency is the topic of an independent research, and will not be discussed further here. For the sake of simplicity, all the generated signals are considered distinct.

B. Principles of the generators

Our method is based on two principles:

- A library of primitive generators: one primitive generator for each elementary operator of PSL.
- An interconnection method: the primitive generators are interconnected following the syntax tree of the PSL formula.

As an illustration, the syntax tree of the formula \( F \), defined in our example, is displayed on Figure 2. The generator synthesized from \( F \) is illustrated on Figure 3.

A generator takes as input the two global signals \( \text{Clk} \) and \( \text{Reset} \) also connected to the Design Under Test (DUT). These signals are used to synchronize and initialize the generator. There are two kinds of outputs:

- **Generated signals:** These outputs represent the signals involved in the PSL formula, which must hold on their resulting trace. In the example of formula \( F \), the generated signals are \( a, b \) and \( c \).
- **Finish:** This is a control signal that takes value ’0’ when the trace is being generated: the trace, considered incomplete, neither contradicts nor satisfies strongly its constraints. When \( \text{Finish} \) is ’1’, the trace holds strongly.

Figure 4 illustrates the interface of a generator.
always

→

a

next[3]

until

b

c

Fig. 2. Tree Structure of PSL Property F

interface for a binary operator.

The inputs are control signals. The outputs for a binary operator are the operands $OP_1$ and $OP_2$ and their associated control $GEN_{OP_1}$ and $GEN_{OP_2}$ (only the first of each output signal is present in the case of a unary operator). More precisely:

- Signals $Clk$ and $Reset$ are for synchronization and initialization, as explained in the previous section.
- $Start$ is used to initiate the constrained generation of the outputs for the current operator. Depending on the semantics of the operator, the generation runs for one or more clock cycles.
- When $Start$ is ‘1’, the $Holds$ signal is used to tell if the output trace must fulfill or not the semantics of the operator (only for Boolean primitive generators).
- Output $Finish$ takes value ‘0’ when the generator has been started, and as long as the outputs are produced in accordance to the operator semantics. $Finish$ takes value ‘1’ as soon as the generation of signals relative to the property is over. From that time until a subsequent restart of the generator, the generated output operands are either fixed to ‘0’, or take a random value (the choice is left to the user).
- $OP_i$ is a generated signal.
- $GEN_{OP_i}$ takes value ‘1’ to indicate when the $OP_i$ signal is constrained. When $GEN_{OP_i}$ is ‘0’, the value of $OP_i$ is a don’t care, and is either randomized or ‘0’.

IV. LIBRARY OF PRIMITIVE GENERATORS

A. Interface of the primitive generators

A generic component has been designed in VHDL for each elementary operator of PSL. Each primitive generator produces sequences of values on its output signals according to the semantics of its corresponding operator.

All primitive generators are built on a common general scheme, and have a uniform interface. Figure 5 displays the

![Generator Interface](image)

Fig. 5. Generator of the PSL property F

Generated Signals

Fig. 4. Interface of a generator

CLK  RESET  Holds  START

Primitive Generator  Finish

OP_1  Gen_OP1  Gen_OP1  OP_2

Fig. 5. Generator interface
All primitive generators except Not and Always use a random number generator. We use a LFSR (Linear Feedback Shift Register). A parameter defines the interval $[1 \ldots N]$ for the numbers that are generated. This component is completely modular, and it can be replaced in future versions of the tool without affecting the rest of the library components (e.g. to allow more flexibility or efficiency in the selection of the probability function built from the basic random number generation).

Another common feature of primitive generators for the PSL foundation language operators is the inclusion of a Shift Register. It is used to store the predictions made for the generation of the operands. The length of the shift register is a generic parameter, its value is user-defined at compile time. Predicting an output operand is done by setting the value '1' at the index $i$ of its associated shift register, where $i$ is the random integer generated with the LFSR.

**Example:** Consider the PSL formula $A \text{ Until } B$. When the signal Start of its generator takes value '1', the LFSR is used to compute a random integer $i$ less than $N$, corresponding to the distance to the future cycle at which signal $B$ will be set. Then the generator sets its first operand output ($A$) to '1' and its second operand ($B$) to '0' for $i$ cycles. After one more cycle, $B$ is set to '1' and $A$ takes the specified default don’t care value.

**Generic Parameters:** Two categories of generic parameters are associated with the primitive generators. Some parameters correspond to the existence of a single library component for closely related PSL operators; their value is automatically extracted by the parser of the PSL property, and fixed by the compiler. Other parameters serve the purpose of tuning the generator to the particular needs of the designer, and become a user-selected option of the elaboration tool. We briefly discuss the essential ones:

- One or two natural parameters are needed for the operators with a time value or a time range specification, such as $\text{next}[N], \text{next}_e[N \text{ to } M], \text{next}_a[N \text{ to } M]$.
- One parameter is used to distinguish between the overlapping and non-overlapping versions of binary operators, which differ only on whether the left operand should hold up to and including (or not necessarily including) the cycle when the second operand holds.
- One parameter is used to distinguish between the strong and weak versions of some operators: some corner case traces are acceptable for a weak operator, and not for its corresponding strong one.
- The first user-defined generic parameter is the length of the shift register, corresponding to the distance to the future cycle at which signal $B$ is to be set.
- Finally, the behavior of don’t care signals, corresponding to the outputs of an inactive (i.e. Start='0') or fully executed generator, can be selected: the primitive generator may produce random signals or signals fixed to the value '0'.

V. **INTERCONNECTION METHOD**

The generators for a PSL formula is built by following the formula syntax tree, where each node serves to instantiate a primitive generator and each leaf is an operand of the DUT. The primitive generators are interconnected in this way:

1) All the $\text{Clk}$ ports of the generators are connected to the DUT master clock.
2) All the $\text{Reset}$ ports are connected to a unique external signal used to initialize the primitive generators. By default, it is the global $\text{Reset}$ of the DUT.
3) For each generator instance that stands for a binary operator:
   a) The left and right operands $\text{Gen}_{\text{OP1}}$ and $\text{Gen}_{\text{OP2}}$ are connected to the port $\text{Start}$ of its children. If one of the child is a leaf, the corresponding $\text{Gen}_{\text{OP}}$ is left unconnected.
   b) Ports $\text{OP1}$ and $\text{OP2}$ (only for a unary operator) are connected:
      - directly to the primary output of the global generator if the corresponding child is a leaf,
      - to the $\text{Holds}$ input port if its child if the node child is a Boolean operator (such as $\rightarrow$),
      - left unconnected if the node child is a temporal operator (such as $\text{next}$).
4) The input $\text{Start}$ of the root node generator is set by a special purpose $\text{Gen}_{\text{INIT}}$ module, controlled by the primary $\text{Reset}$.
5) The output $\text{Finish}$ of the right most node is connected to the primary output $\text{Finish}$ of the global generator. Thus, the user is able to monitor when a generation has ended.

It is worth noting that the interconnection scheme is independent from the special behavior of an individual primitive generator, but only depends on the position of each node in the syntax tree.

Let us again consider our running example property $F$, and its syntactic structure (Figure 2). Four primitive generators, one for each operator (always, $\rightarrow$ next, until), plus a $\text{Gen}_{\text{INIT}}$ component are connected, as shown on Figure 3, to compose the generator for $F$. We can see that each primitive generator controls its subformula child through the signal $\text{Gen}_{\text{OP1}}$. Intermediate ports $\text{Finish}$ are not connected. As we decided to provide a uniform interface for all primitive operators, we connect only the useful ports during the fabrication of the generator. This method does not affect the efficiency of the resulting generator.

**Negation of a Boolean formula:** The simple subset recognizes formulas such as $\text{A} \rightarrow \text{B}$ where $\text{A}$ is a Boolean expression and $\text{B}$ is a FL expression. To cover the case where $\text{A}$, a complex Boolean expression, takes value '0', all primitive generators for Boolean operators have the added input port $\text{Holds}$. When the $\text{Start}$ signal is '1', the value of $\text{Holds}$ indicates the expected value of the Boolean expression rooted at this operator, for the generated sub-expressions. Example:
When its inputs are Start=‘1’ and Holds=‘0’, an AND will generate one of the combinations (‘0’, ‘0’), (‘0’, ‘1’), (‘1’, ‘0’) on its (OP.jpg, OP.jpg) ports.

VI. IMPLEMENTATION AND EXPERIMENTAL RESULTS

A. Verification of the Generators

As a first debugging step, the VHDL design for each primitive generator was simulated with an industrial tool. Then we simulated complex generators relative to PSL formulas.

In a second step, we performed a more extensive validation of our generators by connecting them to their corresponding monitors. A monitor is a synthesizable design which checks that the signals of a DUT satisfy a property. Our group has developed a method for automatically synthesizing monitors for PSL properties [8]. These monitors have been proved correct with respect to the formal semantics of the PSL operators, by theorem proving techniques [9]. The idea was thus to validate each primitive generator by interconnecting it to its corresponding monitor for the same operator, as shown in Figure 6.

The verification of a generator is then greatly simplified: instead of verifying the timing diagrams of the generated signals, it is sufficient to check that the output signal Valid never takes value ‘0’, which can easily be done by model checking for small sizes of the generator LFSR and shift register.

This method has been applied to verify the primitive generators, and then extended to more complex properties, such as:

- always (A -> next[3] (B until C))
- always ((A and B) -> next_a[2..5]C)
- always (A or (B and C))
  -> next_e[2 to 5] (D before E)

B. PSL to VHDL Elaboration Flow

We provide a tool to automatically turn a PSL assumption into VHDL description of a generator. This generator submits the test vectors generated according to the PSL formulas to the DUT. To do that, our tool takes a PSL assumption as input and the corresponding syntax tree is extracted. Then, the program selects in the library the primitive generators involved in the PSL assumption and connect them using the algorithm presented in the section V. Finally we obtain a VHDL description for the global generator. This last one could be directly connected to the DUT in order to proceed to a simulation or an emulation.

C. Experimental results

All the primitive generators have been synthesized with the industrial tool Leonardo on a Stratix1S40F780 FPGA platform and optimized for area. The results are given in the table below. The first column lists the primitive generator corresponding to the named PSL operators, the second column gives the maximum frequencies obtained for the generator, the third and fourth column indicate the number of logical cells and interconnection nets for each generator.

Since the LFSR size is parameterized, this element has been synthesized in isolation in order to exhibit the speed and space impact of this component.

The first part of the table list the FL operators; the maximum delay time has been fixed to 32 (for those that have this parameter). The second part of the table is for boolean operators. The bottom lines give the results for various size LFSR's. It is worth noting that the LFSR area is roughly proportional to the number of bits.

The primitive boolean generators are described as one combinational process for the semantics of the corresponding PSL operator, and one sequential process for the LFSR, synchronized on the CLK signal. The primitive generator Not does not contain a LFSR, so it is totally combinational and its frequency is not constrained by a CLK signal.

As a first application, the primitive generators were prototyped on the FPGA platform in order to generate test vectors for a simple counter which has two inputs: incrementation and decrementation. This experience allowed us to check the online correct behaviour of the primitive generators associated with a design.
### VII. Conclusions and Perspectives

In this paper, we have presented a new solution to generate test vectors that satisfy, in a non vacuous way, an assumed property written in PSL. Our proposal is based on a library of primitive generators and a uniform syntax-directed method to interconnect them. Our current prototype implementation takes as input the VHDL flavor of PSL and produces a synthesizable VHDL generator that can be connected to the design under test for simulation or emulation purposes. The generator can also be synthesized together with the design, and provide the kernel of a built-in self test mechanism, for safety-critical embedded circuits.

Seen as a “test IP”, each generator should be proven correct. So far, the proof has been performed for particular properties, involving particular time values: we interconnect the generator to a proven correct monitor for the same property, and show that their sequential composition always outputs a ‘1’ on the “Valid” signal of the monitor. As a future research, we intend to formally prove the correctness of the parameterized generator library, independently of the time parameter value, using mechanized theorem proving technology.

Much remains to be done to turn our prototype implementation into production quality CAD software. The first obvious extension is to accommodate other input syntax (Verilog, SystemC, SystemVerilog), both for the assumed properties and for the primitive generator library. Another extension concerns the inclusion of a preprocessing step that would check the individual and global consistency of the assumed properties. Last but not least, a user-friendly interface is needed.

Finally, the random number generation needs to be elaborated upon. We would like to not only “assume” logical and temporal properties, but also to generate test vectors that correspond to some “real time” or performance constraints that model the environment of the design under test. This would extend the application of our method towards performance estimation studies.

### REFERENCES


<table>
<thead>
<tr>
<th>Generator</th>
<th>Freq Max (Mhz)</th>
<th>Logical Cells</th>
<th>Nets</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td>492</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>before</td>
<td>133</td>
<td>203</td>
<td>241</td>
</tr>
<tr>
<td>until</td>
<td>176</td>
<td>73</td>
<td>93</td>
</tr>
<tr>
<td>eventually!</td>
<td>234</td>
<td>56</td>
<td>72</td>
</tr>
<tr>
<td>next</td>
<td>282</td>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>next_a</td>
<td>332</td>
<td>12</td>
<td>27</td>
</tr>
<tr>
<td>next_e</td>
<td>145</td>
<td>108</td>
<td>142</td>
</tr>
<tr>
<td>next_event</td>
<td>188</td>
<td>86</td>
<td>112</td>
</tr>
<tr>
<td>next_event_a</td>
<td>209</td>
<td>95</td>
<td>121</td>
</tr>
<tr>
<td>next_event_e</td>
<td>175</td>
<td>97</td>
<td>123</td>
</tr>
<tr>
<td>And</td>
<td>314</td>
<td>17</td>
<td>36</td>
</tr>
<tr>
<td>Or</td>
<td>339</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>Not</td>
<td>unconstrained</td>
<td>11</td>
<td>28</td>
</tr>
<tr>
<td>implication</td>
<td>458</td>
<td>11</td>
<td>28</td>
</tr>
<tr>
<td>LFSR (4)</td>
<td>337</td>
<td>12</td>
<td>27</td>
</tr>
<tr>
<td>LFSR (5)</td>
<td>330</td>
<td>15</td>
<td>32</td>
</tr>
<tr>
<td>LFSR (8)</td>
<td>312</td>
<td>24</td>
<td>47</td>
</tr>
<tr>
<td>LFSR (16)</td>
<td>301</td>
<td>48</td>
<td>87</td>
</tr>
</tbody>
</table>