Prototyping Generators for on-line test vector generation based on PSL properties

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—From an assumed property, which constrains the inputs of a design under test, we produce a RTL synthesizable design that generates compliant sequences of values for all the signals named in the property. Such generator can be connected to the design under test for verification by simulation or emulation. Experiments on our prototype tool show that the technique is efficient, and allows to test the design at its full speed when implemented on an FPGA platform.

I. INTRODUCTION

The design of complex systems on a chip increasingly relies on the concept of platform, where pre-designed components are assembled to constitute the overall system. Needless to say, the correctness of each component should be as fully checked as possible. In many cases, components communicate with their environment according to specific protocols, and can only be considered correct relatively to a proper usage. The "assume-guarantee" reasoning technique was initially introduced in the early eighties to infer the behavior of software built as interacting processes from the behavior of the individual parts [1]. It was then made popular by T. Henzinger [2] and K. McMillan as a decomposition strategy for modular model checking of hardware, before being adopted as a test generation method for hardware and software systems [4].

The "assume-guarantee" verification method relies on the specification of logical and temporal properties, both on the inputs and on the outputs of the interconnected modules. Consider two modules M1 and M2, where the inputs of M1 are connected to the outputs of M2. A property G, shown to hold on the outputs of M2 (guaranteed by M2), can be "assumed" on the inputs of M1 in the process of proving or testing M1 correct. This principle is extended to all the components that provide inputs to M1, and thus constitute its environment.

The advantage of this method, based on the systematic writing of properties under the form of logic and temporal assertions, is that the same properties that are "assumed" in one direction to constrain the inputs and characterize a module environment, are also to be "guaranteed" in the other direction by the module(s) that constitute the environment. In other words, the properties have to be written only once [5]. They can be used across verification software, for documentation, simulation, emulation and formal verification purposes. Moreover, with the adoption of standard and user-readable property specification languages such as PSL [6] and SystemVerilog Assertions [7], libraries of properties can be developed, validated and carried across designs, together with the components they apply to.

In this paper, we place ourselves in the context of hardware prototyping on FPGA, with the objective of running extensive tests at large speed for functional verification purposes. To this aim, we write properties that characterize the input constraints, in order to avoid the full description and synthesis of the environment of the module(s) under test, while avoiding the detection of "false errors", which would inevitably happen in the case of brute execution under random test inputs.

II. STATE OF THE ART

The generation of random test vectors goes back to the early days of simulation. Keeping only the test vectors that satisfy the required properties constitutes an easy, but inefficient solution: large size sequences might be produced, evaluated, and eliminated until corner cases can finally be reached. We are thus interested in techniques that directly produce input vectors that satisfy the assumed properties. To this aim, the literature reports a variety of technologies.

With the algorithmic method, largely employed in simulation, the designer uses a hardware description language, or a special purpose algorithmic environment language such as "E" [8], to create an abstract virtual environment that feeds its inputs to the module. The drawback of this method is its lack of proven correctness: there is no guarantee that the handwritten algorithm generates property compliant inputs unless the environment description is also proved, which adds one more verification step. Our objective, on the contrary, is to automatically synthesize a generator of satisfying input vectors from the assumed properties in declarative form.

A first step in this direction was published by Shimizu and Dill [9]. However, their specifications are restricted to Boolean formulas, resulting in an insufficient expressive power.

The "slicing" approach [10], [11] starts from the design itself, cutting it between the registers to automatically extract the constraints from each part; their composition builds global constraints for the design. In the context of the structural test of VHDL designs, Paoli extracts the control flow graph of the design, and applies software test techniques to generate input sequences that will enumerate all the paths [12]. An enhancement of this method is applied to behavioral VHDL designs in the context of automatic pattern generation for manufacturing test [13]. Taking a different viewpoint, genetic algorithms [14] have been applied to a test vector "population"
to improve its cover rate and limit its size. While providing useful insights on the test generation technology, none of the above works actually tackles the satisfaction of temporal properties.

In contrast, a diversion of model checking [15] has been used to generate test vectors for an automaton, by negating the assumed property: the result is a “counter example”, i.e. an input sequence that drives the automaton into an execution that exhibits the property. Calam systematizes the adoption of so called IOLTS automata [16]; the design, the specification and the behavior aspect to be tested are all modeled as automata; test cases are automatically produced for the property, by a combination of elaborate state space pruning and reachability techniques. The complexity of the automata is however a limiting factor for these techniques. A conspicuous advantage of our technique, compared to the above ones, is that the complexity of our generator is totally independent of the complexity of the design to be tested.

The last series of works reviewed herein directly aim at generating test vectors that are compliant to a PSL property. In the Prosys project, a waveform generator was made to help designers understand PSL [17]. This tool takes as inputs a PSL property P and builds an illustrative trace for P. The objective was mainly pedagogical, and the traces are as small as possible. In this tool, the internal representation of properties is in terms of Büchi automata or Separated Normal form, which limits its applicability to a small number of nested temporal operators in writing the property. Instead of taking an automata theoretic approach, and enumerate the reachable states, we adopt a pragmatic designer’s approach, and build a VHDL synthesizable design which produces satisfying test vectors in a modular way.

A partial solution, limited to the generation of temporal properties written in the foundation language of PSL was described in [18]. The method has been extended to properties written as regular sequential expressions, the preferred writing form, which limits its applicability to a small number of nested temporal operators in writing the property. Instead of taking an automata theoretic approach, and enumerate the reachable states, we adopt a pragmatic designer’s approach, and build a VHDL synthesizable design which produces satisfying test vectors in a modular way.

A partial solution, limited to the generation of temporal properties written in the foundation language of PSL was described in [18]. The method has been extended to properties written as regular sequential expressions, the preferred writing form, which limits its applicability to a small number of nested temporal operators in writing the property. Instead of taking an automata theoretic approach, and enumerate the reachable states, we adopt a pragmatic designer’s approach, and build a VHDL synthesizable design which produces satisfying test vectors in a modular way.

III. PRINCIPLES OF GENERATOR SYNTHESIS

A. The method

Consider the following three properties S, P1 and P1_s, where P1_s and S are sub-properties of P1.

PSL property S is \{B;C[*];D\};
PSL property P1_s is A \rightarrow \text{next\{B;C[*];D\}};
PSL property P1 is
Always A \rightarrow \text{next\{B;C[*];D\}};

S is a SERE which means: “B followed in the next cycle by C repeated zero or several cycles, followed by D”. The implication operator on temporal sequences ‘\rightarrow’ means: if the left hand side temporal sequence holds, then starting from the same cycle the right hand side sequence must hold.

Property P1_s holds on a trace if either \{A\} does not hold on the first cycle, or next S holds (i.e. S holds on the second cycle).

Property P1 is defined as P1_s preceded by the Always operator. While P1_s is evaluated only with respect to the first cycle of the trace, the evaluation of P1 is restarted each cycle: P1 holds at cycle T if and only if sub property P1_s holds at cycle T and at each subsequent cycle T’ > T.

The waveform given on Figure 1 illustrates a trace that satisfies P1. The signal A takes value ‘1’ at clock edges \$1, \$5\ and \$9\: starting at \$1, S is satisfied since B takes value ‘1’ at \$2, then C is ‘1’ until \$4, and on the next clock edge, \$5, D is ‘1’. For the second and third occurrence of A, both evaluations of S are satisfied simultaneously at clock edge \$11, there are 4 occurrences of C for the subtrace starting at \$5 and zero for the one starting at \$9; the occurrence of D at \$11 satisfies both evaluations.

Our generators randomly produce this kind of waveforms. Their synthesis 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.

a) The library: As visible on Figure 2, all primitive generators are built on a common general scheme, and have a uniform interface.

The inputs are control signals. The outputs for a binary operator are the operands OP_1 and OP_2 (only the first output signal is present in the case of a unary operator).

- Signals Clk and Reset are used for synchronization and initialization.
- 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.
- Output Finish takes value ‘1’ as soon as the generation of the left operand is done or, in other words, as soon as it is time to generate a constrained value for the right operand. For example, the Finish signal of the sequence operator ‘\rightarrow’ takes the value ‘1’ one clock cycle after the generation of signals relative to the left operand is over. For unary operators, the operand may be considered either as right
or left according to the operator position in the formula: the operand of 'next' is a right one, the operand of '*' is a left one.

In order to illustrate the library, a brief insight of the unbounded repetition operator [*] VHDL code, is given:

```vhdl
entity star is begin
  generic(
    HIGH:integer:=32);
  port(
    clk,reset,start:in bit;
    A,Finish:out bit);
end entity;

architecture arch of star is
  signal As:bit;
begin
  process (clk,reset)
  begin
    if clk'event and clk='1' then
      if Reset='1' then
        REG:=(OTHERS => '0');
        Finish<='0';
        As<='0';
      else
        REG:=sll(REG);
        REG(timer):=Start;
        Finish<=REG(0);
        if REG(HIGH downto 1)=0 then
          As<='0';
        else
          As<='1';
        end if;
      end if;
    end if;
  end process;

  A<='1' when start and timer!=0 else As;
end arch;
```

Its architecture is based on
- a random variable timer computed by a linear feedback shift register (LFSR) that predicts the number of repetitions when start is enabled
- a shift register REG that counts the number of repetitions that remains: if REG(i) is ‘1’, A is ‘1’ and remains ‘1’ for i – 1 cycles. The signal Finish is ‘1’ as soon as a generation is finished (REG(i)=’0’) even if there is another generation in progress.

This two blocks architecture can be generalized to all primitive generators: one block implements a LFSR, the other one implements the semantics. Furthermore it can be parametrized either to tune the generator to the particular needs of the designer, as a user-selected option of the elaboration tool (e.g. the size of the LFSR) or to distinguish some closely related PSL operators, the parameter value is automatically extracted by the parser of the PSL property, and fixed by the compiler.

b) Interconnection scheme: The generator for a PSL expression is built from the formula syntax tree:
- All the Clk ports of the generators are connected to the Design Under Test (DUT) master clock.
- All the Reset ports are connected to a unique external signal used to initialize the primitive generators. By default, it is the global Reset of the DUT.
- For each operator, the output signal Finish of the current primitive generator is connected to the input Start of its child generator. If the child does not exist, the signal Finish of the generator is connected to the output Finish of the global generator.

Figure 2 illustrates the interconnection for the property P_1.

B. Validation and Implementation

We provide a tool to automatically turn a PSL assumption into the 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 from the library the primitive generators involved in the PSL assumption and connect them using the interconnection scheme described below. Finally, we obtain a VHDL description for the global generator, which can be directly connected to the DUT in order to proceed to a simulation or an emulation.

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 [19] (a proven correct synthetizable design checking that the signals of a DUT satisfy a property). The idea was thus to validate each primitive generator by interconnecting it to its corresponding monitor for the same operator. This method has been applied to verify the primitive generators, and then extended to more complex properties.

IV. APPLICATION AND RESULTS

A. The prototyping platform

A bus based platform: Many solutions can be considered to emulate a design for verification purposes: a direct connection between the DUT and the generator on the same FPGA circuit, an interconnection of two distinct chips (one for the
generator and one for the DUT) on a board, etc. We have chosen to implement the generator and the DUT on the same chip, and connect them to an on-chip bus. The advantages are many:

- This architecture is very flexible, and a variety of components can be interconnected.
- The generator can easily be controlled, and the behavior of the generator-DUT couple is equally easy to observe.
- The generator may produce input vectors for several modules that are connected to the bus.

An evaluation platform: In our current project, we use the prototyping platform shown on Figure 4. This is a NIOS [20] development architecture, in which several components are connected to an Avalon bus [21]. The platform exchanges data with an external PC through the RS232 port. The on-chip memory stores the program that is executed on the CPU; in particular, a dedicated routine observes the data that are emitted on the bus by the generator, and sends them to the PC for display.

We have written a generic interface to connect any module to the Avalon bus. In particular, this interface serves the purpose of connecting the generator to the bus. This interface is of course reusable, and automates our method for the verification engineer: for all DUT’s that are intended to be connected to a different bus, the extra effort to connect it to the Avalon bus is negligible.

B. Results

The NIOS architecture on which our experiments have been performed contains 2126 logical cells and 1030 registers. Its maximal clock frequency is 50MHz. Our generators have been synthesized with the Quartus II 5.0. software, on a Cyclone II EP2C35 FPGA platform.

<table>
<thead>
<tr>
<th>Generator</th>
<th>LC</th>
<th>REG</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>12</td>
<td>6</td>
<td>79.37</td>
</tr>
<tr>
<td>OR</td>
<td>12</td>
<td>6</td>
<td>73.92</td>
</tr>
<tr>
<td>&amp; &gt;</td>
<td>10</td>
<td>6</td>
<td>72.45</td>
</tr>
<tr>
<td>NOT</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

The primitive binary Boolean generators are described as one combinational process for the semantics of the corresponding Boolean operator, and one sequential process for the LFSR, synchronized on the CLK signal. In the case of the NOT, there is no LFSR; in the case of the binary operators, the size of their LFSR is 2. As a result, their operating frequency is always greater than 72 MHz, and their area is very small. The results are given in Table 1 (LC: number of logical cells, REG: number of registers, Freq: operating frequency).

For instance, we randomly generate the couples of values (0,1), (1,0) or (1,1) for the OR generator.

The primitive generators for the SERE and FL operators involve a LFSR, the size of which is a generic parameter to be set by the user. Table II shows the results obtained for the FL generators. The table is divided in two parts.

In the upper section of the table, we give the performances of the generators for an 8-bit LFSR, thus allowing temporal ranges up to 255 cycles. For this LFSR size, only three generators have a frequency less than the NIOS system frequency (50 MHz). The UNTIL generator is clearly the most costly one.

<table>
<thead>
<tr>
<th>Generator</th>
<th>LC</th>
<th>REG</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td>1</td>
<td>0</td>
<td>62.20</td>
</tr>
<tr>
<td>eventually</td>
<td>576</td>
<td>273</td>
<td>53.48</td>
</tr>
<tr>
<td>before</td>
<td>1343</td>
<td>327</td>
<td>62.57</td>
</tr>
<tr>
<td>next_e</td>
<td>670</td>
<td>283</td>
<td>43.86</td>
</tr>
<tr>
<td>next_a</td>
<td>260</td>
<td>256</td>
<td>49.37</td>
</tr>
<tr>
<td>next</td>
<td>2</td>
<td>256</td>
<td>49.37</td>
</tr>
<tr>
<td>until</td>
<td>2077</td>
<td>762</td>
<td>75.08</td>
</tr>
<tr>
<td>next_event</td>
<td>355</td>
<td>283</td>
<td>61.11</td>
</tr>
<tr>
<td>next_event_a</td>
<td>608</td>
<td>285</td>
<td>59</td>
</tr>
<tr>
<td>next_event_e</td>
<td>356</td>
<td>286</td>
<td>62.93</td>
</tr>
<tr>
<td>LFSR (6 bits)</td>
<td>33</td>
<td>6</td>
<td>259</td>
</tr>
<tr>
<td>LFSR (7 bits)</td>
<td>38</td>
<td>7</td>
<td>240</td>
</tr>
<tr>
<td>LFSR (8 bits)</td>
<td>45</td>
<td>8</td>
<td>209</td>
</tr>
</tbody>
</table>

The second section of the table shows the impact of the LFSR bit length on its area and frequency. This shows the impact of the LFSR on the overall generator (the figures given in the upper part of the table include the logical cells and registers for the 8-bit LFSR).

Table III shows the results obtained for the SERE operators, for a LFSR of size 8 (with the exception of ’;’ that has no LFSR).

Each repetition generator is defined with a generic parameter LEAF that specializes its instantiation as needed: LEAF is ’1’ for a repetition over a Boolean expression, ‘0’ for
a repetition over a SERE sequence. The table shows the synthesis results for the two values of \texttt{LEAF}. For both kinds of repetition, the operating frequencies and the number of registers are close.

\textbf{C. Considerations}

It is important to note that the LFSR size greatly impacts the area and the frequency of all generators. Extensive measurements, not reported here, show that this dependency becomes linear for larger than 9-bit LFSR’s.

In most examples that we found in the state of the art, as well as in the Prosys project [22], the PSL formulas expressing the properties are simple, and the time intervals between successive events are small. For most practical cases, a maximal size of 16 for the LFSR’s is sufficient. The size overhead of the generators is expected to be low.

The size of the generators should also be related to the size of the DUT. In most cases, the DUT is several orders of magnitude bigger. Thus, the speed limitations for the hardware prototype comes from the DUT, not the test generator. Recall also that the generator replaces the DUT environment, a possibly very large set of modules.

\textbf{V. CONCLUSIONS AND PERSPECTIVES}

A method to automatically turn a PSL property into a generator of satisfying test sequences has been worked out. This approach enables a designer to test the circuit under input constraints defined by the property. Combined with monitors which observe the circuit behavior, this method constitutes an efficient on-line debugging tool.

The test vector generation is totally independent of the design complexity, it only depends on the temporal complexity of the input constraints. From a PSL property, we automatically produce a RTL synthesizable VHDL model of the generator: the technology applies to testing the design either by simulation or by emulation.

In our example, a generic interface has been built and can be reused. Thus, any design connected to the Avalon bus can be tested with generators without effort during the test step. Thanks to its capacity to automatically turn a PSL formula into a generator, the generic interface provides a completely automatic flow if the DUT is connected to the Avalon bus.

\textbf{Problem with great LFSR’s}: The size of the generators is linear with the number of clock cycles involved in the PSL formula. Thus when a property reaches thousands of clock cycles, the size of its generator becomes important. Moreover, the number of cycles involved in the property has an impact on the operating frequency of its generator. In practice, however, the complexity of the DUT is always greater, and is the limiting factor on the prototype clock frequency. In all our experiments, and in all the cases reported in the literature, it is never the case that the generators would slow down the DUT and prevent an optimal on-line testing.

\textbf{Random generation}: In the current approach, we use a classical pseudo random generation for the uniform law. We foresee that the correct modeling of the DUT environment will require more elaborate probability laws for the generation of the individual signals involved in the PSL formula. This is the subject of future works.

\textbf{Generator validation}: As we develop tools for complex system verification and debug, it is essential to provide a trustworthy method. To this aim, we use the “theorem-proving” techniques to prove: (1) that the implementation of each primitive generator is correct with respect to the semantics of its corresponding operator, and (2) that the interconnect algorithm is also correct (i.e. the global generator conforms to the semantics of its corresponding PSL formula). This ongoing formal validation is performed with the use of the PVS proof system.

\textbf{Cover rate}: For the time being, we rely on the quality of the properties to correctly and fully describe the DUT environment. The generated tests cover the assumed property, by construction. Whether the properties generate input sequences that sufficiently cover all the logic, registers, execution paths, etc. of the DUT is an important topic not dealt with in this paper. Future works will tackle the problem of automatically producing properties for testing a design, with respect to expected behaviors (verification) or denied behaviors (debug).

\begin{table}[h]
\centering
\caption{Prototyping results for SERE generators}
\begin{tabular}{|c|c|c|c|}
\hline
Generator & LC & REG & Freq \\
\hline
\texttt{:} & 7 & 7 & 70.72 \\
\texttt{[*] (leaf=’0’)} & 146 & 310 & 74.00 \\
\texttt{[*] (leaf=’1’)} & 399 & 310 & 71.66 \\
\texttt{[*] (leaf=’0’) \& \&} & 311 & 310 & 71.77 \\
\texttt{[*] (leaf=’1’) \& \&} & 475 & 310 & 71.47 \\
\hline
\end{tabular}
\end{table}

\textbf{REFERENCES}


