Towards a Toolchain for Assertion-Driven Test Sequence Generation

Laurence PIERRE
TIMA Laboratory (Univ. Grenoble Alpes, CNRS)
46 Avenue Felix Viallet - 38031 Grenoble cedex - France
Laurence.Pierre@imag.fr

Abstract—Coverage is a major concern in simulation-based test and verification, but it usually addresses statements, conditions, or FSM transitions. The work reported here focuses on dynamic Assertion-Based Verification, which aims at checking that designs obey requirements formalized as temporal assertions. In that context, the selection of test sequences is related to coverage of the assertions activation conditions. This goal also differs from the one of usual ATPG methods (Automatic Test Pattern Generation), which target the production of test patterns designed to detect incorrect circuit behaviors and that are guided by fault models such as stuck-at faults.

This paper describes a toolchain for the automatic construction of test sequence generators directed by specifications expressed as temporal assertions. It also sketches some experimental results and discusses some issues related to the diversity of alternative solutions.

I. INTRODUCTION

The context of this paper is Assertion-Based Verification (ABV) [1], a technique that alleviates the characterization of the expected circuit behavior and eases verification [2]. Written in languages like the IEEE standards PSL [3] or SVA [4], logic and temporal assertions are used to capture the design intent and facilitate bug detection. Various solutions have been proposed for the construction of hardware property checkers (monitors) from temporal assertions [5], [6], [7], [8]. Such monitors are connected to the DUT (design under test) and inform about the satisfaction of the properties during simulation or emulation. In that context, it is essential to guarantee that the properties are not verified vacuously, or not checked too rarely. As a trivial example, consider the following PSL assertion which means “it is always true that, if req holds then, starting from 2 cycles after, grant occurs before end”:

always(req → next[2](grant before! end));

If the test suite is such that the precondition req never holds, then the property is always verified, but verified vacuously i.e., without having actually been checked. Beyond non-vacuity, test sequences must also be designed to ensure a good coverage of such monitor’s activation conditions.

Our goal is to complement the verification infrastructure (DUT equipped with hardware monitors) by test generators that will produce test sequences designed to reach a satisfying coverage of these monitor’s activation conditions (see Fig. 1). The specification of the generator components, like the monitors, will originate from the PSL formalization of the requirements.

The main contribution of this paper is a design flow to create those test generators (described in section III). Its core technique concentrates on the signals that play a role in the activation conditions for the property monitor: constraints are propagated “backward” in the cone of influence of those signals (up to primary inputs or to memory elements), to produce a set of solutions that fulfill these constraints [9]. Selected solutions lead to the behavioural specification of the test generator, under the form of a new PSL assertion that represents the sequence of input stimuli that can cause the satisfaction of the activation conditions. Finally this PSL specification serves as input for a tool that builds the corresponding test generator.

We consider first criteria for the selection of solutions, and alternative methods for the construction of test generators from these solutions. We illustrate and discuss the feasibility of the approach on simple and real-size examples.

II. STATE OF THE ART

A. Coverage

Coverage is a major concern in simulation-based test and verification, but it usually addresses statements, conditions, or FSM transitions [10]. Various methods have been proposed for the production of test suites that guarantee a satisfying level of coverage. Transition coverage in FSMs is considered in [11]; the algorithm that computes test sequences considers all the transitions and makes use of model-checking techniques to obtain counter-examples for the negations of the transitions. Approaches based on negating a formula and looking for one counter-example are widespread.

More recently, [12] also exploits model-checking methods
and proposes the tool CAT (Coverability Analysis Tool). The authors consider both the reachability of given variable values (out of a finite set of values), and the issue of statement coverage. The idea for dealing with the reachability of a value \( x \) for a variable \( v \) amounts to finding counter-examples for the CTL formula \( EF(v = x) \). A similar approach is also used for statement coverage.

The work of [13] targets the generation of test cases for digital circuits partially described in Esterel. It is based on the notion of state coverage, defined as the ratio between the number of states reached during simulation and the number of states theoretically reachable. The algorithm also addresses the negation of properties by model-checking.

The goal of [14] is to evaluate the quality of the testbench with regard to the amount of activated assertions. It is complementary to our target, which is to constrain test sequences that avoid vacuity during dynamic formula checking.

Similarly, the authors of [15] address the question: have we produced enough test vectors to exercise most of the possible behaviours of the circuit? They focus on assertions as automata, as they are interpreted by the technology of [7]. They propose coverage metrics for automata that represent assertions, and they relate them to their specific assertion coverage goals. The results are sequences of transition labels leading to the success or the violation of the assertion. However, only primary inputs can appear in the properties activation conditions. Our focus is based on the fact that any internal (or primary output) signal can also take part in assertions. Therefore, test generation typically requires to relate their constraints to constraints on primary inputs.

B. Vacuity

In [16], the problem of the identification of vacuous properties is considered. The authors use model-checking, and define vacuity as follows: a formula \( \varphi \) is satisfied vacuously in a model \( M \) if it is satisfied in \( M \) and there exists a sub-formula \( \psi \) of \( \varphi \) that can be changed arbitrarily without modifying the model-checking result. In other words, the idea is to check if there exists a sub-formula \( \psi \) of \( \varphi \) that does not affect \( \varphi \), considering syntactic perturbation.

The same principles are reused in [17], for PSL assertions. The idea is to analyze the effects of substituting a sub-formula by false (or true, depending on its polarity). The solution is based on a dynamic technique that makes use of “mutations”.

Considering the same definition of vacuity, [18] proposes a solution to consider a notion of semantic perturbation instead of a syntactic approach.

These methods focus on the detection of vacuity in temporal formulas, whereas our goal is the production of test sequences that avoid vacuity during dynamic formula checking.

C. Other approaches

The work of [19] is related to “triggerying” formulas. Models under consideration are Kripke structures. The problem amounts to determining scenarios that “trigger” a formula \( \varphi \) in a model \( M \). It is restricted to computing maximal regular expressions \( r \) such that \( M \models r \Rightarrow \varphi \). Since this problem has a polynomial complexity, it is still simplified into the computation of words \( w \) (of a bounded length) such that \( M \models w \Rightarrow \varphi \). Due to the intrinsic complexity of the problem, it is necessary to largely simplify it.

Our core technique focuses on the identification of conditions on primary inputs to fulfill constraints for the activation of assertions [9]. It analyzes assertions in logical implication
form, or in suffix implication form \( \Rightarrow \) which may be obtained using rewrite rules such as the ones of [20] or [7], for example:

\[
p \Rightarrow b \iff \{ (\neg b) (+) \Rightarrow p \} \\
(\neg p) \Rightarrow \neg b
\]

It concentrates on each atomic part of the left-hand side of the implication and produces solutions for the corresponding constraints. For instance, for the activation of the property \( (P_1) \) that will be given as example in section IV-A:

\[
\text{always}\{ \{ \text{gate/out-open} \}; \text{not gate/out-open} \} \Rightarrow \text{not gate/out-open} \quad \text{until} \quad \text{exitOK}
\]

the constraint is that the signal \text{gate/out-open} must have a falling edge i.e., a value 1 at a time \( t \) followed by value 0 at time \( t+1 \).

The goal of the algorithm recalled in Fig. 2 is as follows: for a signal \text{sig} of the design (internal signal or primary output) and a value \( x \), determine all the solutions to the constraint “\text{sig} at a time \( t \) must equal \( x \)” i.e., \( \text{sig}(t) = x \). The algorithm propagates this constraint “backward” in the cone of influence of \text{sig} i.e., going back up to the primary inputs if possible. When going through a structural loop, the algorithm detects the presence of a register that has already been visited, and stops iterating inside this loop. At each step of this backward traversal, the function \text{traverse_backward} applies the rules recalled in the table of Fig. 2 to traverse the structural component (gate or register) “behind” the current signal. Using these rules, the algorithm ultimately builds the solution tree for \text{sig}(t) = x. The nodes of this tree are and and or operators, and its leaves are of the form \text{signal(time) = value}, where \text{signal} is the identifier of a signal, and \text{time} can be \( t, t-1, \ldots \)

Let us briefly illustrate the meaning of these solutions with a simple example that will be developed in section IV-A. For this example, one of the solutions extracted from the tree is as follows, where \( S_0, S_1 \) and \( S_2 \) are state elements (registers) and \text{in\_req}, \text{ticket\_in} and \text{exitOK} are primary inputs:

\[
S0(t−1) = 0 \land S1(t−1) = 0 \land S2(t−1) = 0 \\
\land \text{in\_req}(t−1) = 0 \land \text{ticket\_in}(t−1) = 1 \land \text{exitOK}(t) = 1
\]

Like most of the solutions, this constraint contains a conjunction of conditions on registers and a conjunction of conditions on primary inputs. The resulting test generator should take as inputs \( S_0, S_1 \) and \( S_2 \), and produce the following stimuli for \text{in\_req}, \text{ticket\_in} and \text{exitOK}: as long as the condition \( S0(t−1) = 0 \land S1(t−1) = 0 \land S2(t−1) = 0 \) (which is the suitable condition for choosing ad hoc values for the primary inputs) does not hold, it can generate pseudo-random sequences; when this condition holds, it must force \text{in\_req} to the value 0 and \text{ticket\_in} to the value 1 in the current cycle, and \text{exitOK} to the value 1 at the next cycle. Remember that the tree is made of solutions for a constraint of the form \( \text{sig}(t) = x \), such a generator will provoke the arrival of the expected value \( x \) on the signal \text{sig} at time \( t \).

III. ASSERTION-DRIVEN TEST GENERATORS

A. Test Generator Construction Flow

On the basis of this latter approach, our automated toolchain is described on Fig. 3. It takes as input the synthesized circuit (VHDL netlist) and the assertion to be checked, and produces the dedicated test generator:

- the VHDL description is first converted into an internal XML format by the VSYML tool [21]
- solutions are then generated as described above, and selected as described in section III-B;
- these solutions are arranged and transformed into a new PSL assertion (see section III-C). To that goal, each selected solution is partitioned into a condition \( C_1 \) on DUT state elements and a condition \( C_2 \) on its primary inputs;
- the resulting test generator (see section III-C) will be designed to generate input sequences on the basis of a constraint that roughly reflects \( C_1 \rightarrow C_2 \). This test generator will have to produce pseudo-random inputs, except when the condition \( C_1 \) is observed; in that case it will have to force the inputs in order to respect condition \( C_2 \).

To build a synthesizable test generator, a tool like Synthorus [22] (or MYGEN [23]) can be used: given a PSL assertion, it can build a generator which produces pseudo-random sequences that fulfill this temporal constraint. Section III-C describes how the selected solutions can be transformed into a suitable PSL assertion. It also proposes an alternative technique, based on the construction of a simple non-synthesizable VHDL process.

In the general case, the interconnection between the DUT and the test generator will be as pictured by Fig. 4. The generator will receive as inputs values coming from the DUT, that give information about (a part of) its current state. The observation of condition \( C_1 \) on this state will trigger the generation of DUT input stimuli that fulfill \( C_2 \).
B. Extraction and Selection of Solutions

(1) Extraction of solutions. We recall that the algorithm of Fig. 2 produces a tree that contains all the solutions to $\text{sig}(t) = x$. Due to the formalization of the rules that are applied during this backward traversal, the nodes of this tree can only be and or operators, and its leaves are atomic solutions of the form $\text{signal}(time) = \text{value}$. Function extract_solutions given below implements a specific depth-first traversal of this tree to extract the list of solutions, which represents the disjunction of all the solutions. Each solution is a conjunction of atomic solutions (encoded here as a list of atomic solutions).

- if the root node of the current subtree is a disjunction operator, the solutions produced by the child nodes are simply concatenated into a new list of solutions, using the concat function. Indeed, we materialize here the following formula:

  $$(s_1 \lor s_2 \lor \ldots \lor s_m) \lor (s'_1 \lor s'_2 \lor \ldots \lor s'_n) = s_1 \lor s_2 \lor \ldots \lor s_m \lor s'_1 \lor s'_2 \lor \ldots \lor s'_n$$

- if the root node of the current subtree is a conjunction operator, the solutions produced by the child nodes are combined together using the merge function. It symbolizes the following formula (distributivity):

  $$(s_1 \land s_2 \land \ldots \land s_m) \land (s'_1 \land s'_2 \land \ldots \land s'_n) = (s_1 \land s'_1) \lor (s_1 \land s'_2) \lor \ldots \lor (s_1 \land s'_n) \lor (s_2 \land s'_1) \lor (s_2 \land s'_2) \lor \ldots \lor (s_2 \land s'_n) \lor \ldots \lor (s_m \land s'_1) \lor (s_m \land s'_2) \lor \ldots \lor (s_m \land s'_n)$$

The auxiliary function merge avoids duplicating the same atom in a solution. Moreover, it eliminates some trivially infeasible solutions (it detects incompatible constraints such as $X(t) = 1 \land X(t) = 0$, and eliminates the solutions that include such conjunction of conditions).

The function extract_solutions is called with a tree node as input and returns a list of solutions. The function is structured as follows:

```c
function extract_solutions(solution_tree tree) {
  solution_list l, lc;
  if (is_leaf(tree)) { // Atomic solution
    insert_solution(new_solution(tree), l);
  }
  else if (is_disjunction(tree)) {
    // The root node is a "or", the subsolutions
    // are concatenated in l:
    l = extract_solutions(tree->child[0]);
    for(int i=1; i < tree->n_children ; i++) {
      lc = extract_solutions(tree->child[i]);
      l = concat(l, lc);
    }
  }
  else if (is_conjunction(tree)) {
    // The root node is a "and", the
    // solutions are combined together:
    l = extract_solutions(tree->child[0]);
    for(int i=1; i < tree->n_children ; i++) {
      lc = extract_solutions(tree->child[i]);
      l = merge(l, lc);
    }
  }
  return l;
}
```

Remark. Another way of extracting the solutions can also be used. It consists in building the BDD (Binary Decision Diagram) associated with the Boolean function expressed by the solution tree, and then in extracting its prime implicants.

This alternative has also been implemented, with the CUDD package [24]. Owing to the compactness of the data structure, this solution can be more memory-efficient for limited-size problems. It sometimes provides less accurate results (because it maximizes the presence of don’t care values). The BDD-based algorithms are typically very efficient, provided that the BDD can be built. However for large problems, this solution implemented with the CUDD package ran out of time in this BDD construction phase.

(2) Selection of solutions. The reason for producing all the solutions lies in the observation that some solutions are more adequate than other ones to derive the final test generator.

The final selection is done interactively by the user, but we previously automatically filter them according to some criteria discussed here.

(1) A first straightforward criterion for eliminating a solution is the absence of terms related to primary inputs. If a solution is only a conjunction of expressions on internal signals, it cannot be useful to produce input stimuli.

(2) We must also take into account that we have to generate input stimuli depending on prior values of internal signals. In the toy example used in [9] for instance, the resulting solutions are the following, where $S_3$ and $S_4$ are state elements and $I$ is a primary input:

$$S_3(t-1) = 0 \land S_4(t-1) = 1 \land I(t-1) = 1 \land I(t) = 1$$

The first solution leads to the construction of a generator that detects when $S_3 = 0 \land S_4 = 1$ and then produces the sequence $\{I(t); I(t+1)\}$. Indeed the condition on $S_3$ and $S_4$, which is $S_3(t-1) = 0 \land S_4(t-1) = 1$, depends on values at time $t-1$, whereas the expression to be produced on the input is $I(t-1) = 1 \land I(t) = 1$ i.e., it constrains its value at times greater than $(t-1)$. The generator can react according to the condition and produce future values. The second solution is irrelevant because, once detecting the condition, the generator should have to fulfill $I(t-2) = 1 \land I(t) = 1$, but $I(t-2)$ is a past value.

(3) It is also crucial to force inputs while keeping simulations realistic. An analysis of the DUT can induce other criteria, based on the respect of its conditions of use and protocols. As a naive example, in case of an active low reset, it is better to avoid the selection of solutions that entail $\text{reset} = 0$ (thus making the DUT return to its initial state too often).

The filtering tool gives the possibility to choose among these criteria, through options on the command line.
C. Construction of the Test Generators

The test generator must force the primary inputs to respect condition $C_2$ when condition $C_1$ on the state elements is observed, otherwise it can simply produce pseudo-random inputs (and the inputs that are not involved in $C_2$ can also be generated pseudo-randomly).

(1) **Synthesizable generator.** A synthesizable component, usable for simulation but also for FPGA emulation, can be obtained using the SyntHorus tool. To that goal, the solution must previously be transformed into an appropriate PSL specification.

Conditions $C_1$ and $C_2$ may spread over several time steps $k$ to $k+j$ as follows:

\[
\begin{align*}
& cs_0(k) \land ci_0(k) \land cs_1(k+1) \land ci_1(k+1) \land \cdots \land \\
& cs_j(k+j) \land ci_j(k+j)
\end{align*}
\]

where each $cs_n$ is a condition on state elements (part of $C_1$) and each $ci_n$ is a condition on primary inputs (part of $C_2$).

The corresponding assertion for SyntHorus has the form below, that makes explicit the logical and temporal relationship between the $cs_n$ and $ci_n$ conditions.

\[
\begin{align*}
\text{always}(cs_0 \text{ and not busy} & \rightarrow \\
& (ci_0 \text{ and} \\
& \text{next}![cs_1 \rightarrow ci_1] \text{ and busy} \text{ and} \\
& \text{next}![2](cs_2 \rightarrow ci_2) \text{ and busy} \text{ and} \\
& \cdots \text{ and } \text{next}![j+1](\text{not busy}))
\end{align*}
\]

The hardware generator will involve a specific signal $busy$ (which it will both generate and observe); as long as $cs_0$ does not occur, this signal will be ‘0’ (and pseudo-random inputs will be produced); when $cs_0$ occurs, the first premise will hold and the generator will start producing the expected input sequence (provided the other conditions $cs_n$ hold) while maintaining $busy$ high; it will set it again to ‘0’ at the end of the sequence. The role of this signal is to prevent from starting a new sequence generation while a sequence is already being generated (thus causing contradictions), if $cs_0$ and another $cs_n$ hold simultaneously.

This solution will be illustrated on the examples of sections IV-A and IV-B.

(2) **Simple VHDL process.** If only simulation is targeted, another solution is to build a simple non synthesizable VHDL process. It has the following form, where each $at_n$ is the set of assignments that realize condition $ci_n$ (e.g., $i \leq \leq '1';$ for condition $i = 1$), and pseudo-random values can be generated for example as described in [25], using the math_real IEEE package.

```vhdl
process
variable seed1, seed2: positive;
-- ...
begin
    wait until ck'event and ck='0';
    if cs0 then
        ai0;
        wait until ck'event and ck='0';
        if cs1 then
            ai1;
            wait until ck'event and ck='0';
            if cs2 then
                ai2;
                wait until ck'event and ck='0';
                ...
            end if;
        end if;
        elsif (counter mod K = 0) then
            -- classical pseudo random generation
            UNIFORM(seed1, seed2, rand);
            int_rand := INTEGER(TRUNC(rand*32.0));
            stim := std_logic_vector
                (to_unsigned(int_rand, stim'LENGTH));
        end if;
    end process;
```

Here, if the expected condition is not met, pseudo-random values are generated every $K$ cycles ($K$ is a generic parameter).

IV. First Experiments and Discussion

Some experiments are reported here in order to discuss the impact on coverage of the choice among the solutions.

Note that the CPU times for steps 1 (VSYML), 3 (automatic filtering to help solutions selection), 4 (assertion construction) and 5 (test generator construction by SyntHorus) of the toolchain are negligible. The construction of the solution tree by the algorithm of Fig. 2 is also performed in at most few seconds, even for large designs.

Scalability is only uncertain for the extraction of the solutions from this tree. Extracting the prime implicants from the associated BDD becomes infeasible as soon as the construction of this BDD experiences a timeout. In contrast, the specific algorithm proposed in section III-B has the advantage of being time efficient, but it runs out of memory when hundreds of thousands of solutions must be built. Therefore to improve the method, it will be necessary to find: either an extremely compact data structure to store the solutions, or a strategy to construct enough relevant solutions while avoiding to extract all of them.

A. Toy Example: Parking Gate Controller

This is a simplified version of a parking gate controller that controls cars entrance and exit. The primary inputs are: request for entrance (in_req), signal from the entrance sensor (a car is entered, sensor_in), insertion of an exit ticket (ticket_in), the exit ticket is valid, the car is allowed to exit (exitOK), signal from the exit sensor (a car has exited, sensor_out). The primary outputs are: control of entrance gate opening (gate_in_open), command to increment the number of cars (car_in), command to the mechanism that checks if the ticket is valid (ticket_inserted), control of exit gate opening (gate_out_open), command to decrement the number of cars (car_out). When a driver presses the entrance button, if the car park is not full, the entrance gate must open. It remains open until the entrance sensor detects that the car is entered. A signal is then sent to the counting mechanism to increment the car number. When the driver wants to exit, he must first go to a self-checkout machine where he pays and gets a valid exit ticket. Once at
the exit gate, he inserts his exit ticket and the exit gate must open if the ticket is valid. It remains open until the exit sensor detects that the car has exited. A signal is then sent to the counting mechanism to decrement the car number.

For this simple parking gate controller, the coverage of all typical assertions is already satisfying even when purely pseudo-random test sequences are produced. However, we will use it to discuss the possible impact on coverage of the choice among the solutions.

Let us consider property ($P_1$) that asserts that it is always true that, if the exit gate closes (falling edge of the $\text{gate\_out\_open}$ signal), then it will remain closed until a valid exit ticket is inserted:

$$(P_1) \text{ always}(\text{gate\_out\_open}; \text{not gate\_out\_open}) \rightarrow (\text{not gate\_out\_open} \text{ until exitOK})$$

This property is already in suffix implication form, the activation condition is its premise, the falling edge of the signal $\text{gate\_out\_open}$. It is a primary output, which means that the constraints $\text{gate\_out\_open}(t) = 1$ and $\text{gate\_out\_open}(t + 1) = 0$ have to be transformed into constraints on the registers and primary inputs. After running the algorithm, we get 301 solutions for the former, and 220 solutions for the latter.

(1) Let us first consider one of the simplest solutions for the condition $\text{gate\_out\_open}(t) = 1$, which is:

$$S0(t) = 0 \land S1(t) = 0 \land S2(t) = 1 \land \text{sensor\_out}(t) = 0$$

On the other hand, a very simple solution to the constraint $\text{gate\_out\_open}(t + 1) = 0$ is:

$$\text{sensor\_out}(t + 1) = 1 \land \text{exitOK}(t + 1) = 0$$

Thus the following PSL activation assertion is generated for property ($P_1$):

$$\text{always}(\text{not S0 and not S1 and S2 and not busy} \rightarrow (\text{not sensor\_out} \land \text{next!}[(\text{sensor\_out and not exitOK and busy}) \land \text{next!}[2](\text{not busy}])))$$

To evaluate the quality of this solution, we compared simulation results with two different test sequence generators: the first version produces purely pseudo-random sequences, whereas the second version implements the specification above. We ran simulations over 5000 clock cycles. Surprisingly, assertion ($P_1$) was activated 264 times in the first case, and 235 times in the second case. Assertion coverage was more or less the same with the two versions.

(2) Let us now consider another, less simple, solution for $\text{gate\_out\_open}(t) = 1$, which is:

$$S0(t - 1) = 0 \land S1(t - 1) = 0 \land S2(t - 1) = 0 \land \text{in\_req}(t-1) = 0 \land \text{ticket\_in}(t-1) = 1 \land \text{exitOK}(t) = 1$$

The activation constraint for ($P_1$) now amounts to:

$$\text{always}(\text{not S0 and not S1 and not S2 and not busy} \rightarrow (\text{not in\_req and ticket\_in} \land \text{next!}[\text{exitOK and busy}] \land \text{next!}[2](\text{sensor\_out and not exitOK and busy}) \land \text{next!}[3](\text{not busy}])))$$

Here too, we ran simulations on 5000 clock cycles with the resulting test sequence generator. Assertion ($P_1$) was activated 1249 times in that case.

This simple example demonstrates that considering complexity criteria may not be sufficient when selecting a solution. The second choice produces a more complex generator, but which is such that forcing starts from a more highly probable state, hence more often.

B. Transmission from Dual ADCs to Serial Output

The example considered here comes from a tutorial for an ABV solution integrated in industrial EDA tools [26]. This circuit controls two external dual ADCs and transmits the four resulting 12-bit words through a serial output. It has two main requirements:

- the four 12-bit data are retrieved using SPI interfaces. Thus, a first set of assertions checks the SPI protocol requested by the external dual ADCs.
- the transfer protocol must provide enough time between two acquisitions in order to complete the serial transmission. Other assertions can be used to check this internal protocol.

One of the assertions related to the SPI protocol is as follows, where $CS_N$ is an output of the CTRL sub-component, that becomes low when data are coming from the first channel on $DOUTA$:

$$(P_2) \text{ always}(\{CS_N; not CS_N\} \rightarrow \text{assertion} \{\{DOUTA[+2]\} \mid \{\text{not DOUTA[+2]}\}\}[*12])$$

The left hand side of the suffix implication operator aims at detecting the falling edges of the signal $CS_N$. Hence we first determine the solutions for having $CS_N(t) = 1$, we get 600 solutions without obvious incompatibles. Then we compute 93 solutions for satisfying $CS_N(t + 1) = 0$. From the first set of solutions, we keep the one below (in which $EN\_CS\_N$ is a primary input of the transmitter core):

$$S0(t - 3) = 0 \land S1(t - 3) = 0 \land S2(t - 3) = 0 \land \text{EN\_CS\_N}(t - 3) = 0$$

As far as $CS_N(t+1) = 0$ is concerned, we keep the solution:

$$S1(t - 1) = 0 \land S2(t - 1) = 0 \land \text{EN\_CS\_N}(t - 1) = 1$$

The combined solution is then:

$$S0(t - 3) = 0 \land S1(t - 3) = 0 \land S2(t - 3) = 0 \land \text{EN\_CS\_N}(t - 3) = 0 \land S1(t - 1) = 0 \land S2(t - 1) = 0 \land \text{EN\_CS\_N}(t - 1) = 1$$

which becomes the following PSL constraint:

$$\text{always}\{(\text{not S0 and not S1 and not S2 and not busy} \rightarrow (\text{not EN\_CS\_N and next!}[\text{busy}] \land \text{next!}[2](\text{not S1 and not S2} \rightarrow \text{EN\_CS\_N} \land \text{bus}y) \land \text{next!}[3](\text{not busy}])))$$

For a pure pseudo-random simulation over 20000 clock cycles, assertion ($P_2$) is activated 709 times, and it is activated 722 times when forcing falling edges on $CS_N$ as described.
above. However the conditions of use of this device are such that the signal $CS_N$ is typically mostly high. Therefore, better results are obtained with a test generator that only forces $CS_N$ to 0. In that case the assertion is activated 769 times.

### C. HDLC Controller IP

This HDLC (High-level Data Link Control) controller IP (Intellectual Property component)\(^1\) includes independent transmitter and receiver modules that are able to handle transmission and reception with a specific frame format, a synchronous serial protocol called HDLC (ISO/IEC 13239:2002). The IP, pictured on Fig. 5 performs the physical level of the HDLC protocol (serialization/de-serialization, CRC generation, transparency and abort generation/detection).

Each frame is made of an Open Flag (the flag value is 0x7E), the Information (payload), the CRC, and the Closing Flag, that may be shared with open flag of the next frame.

![HDLC Controller IP](image)

**Fig. 5.** HDLC Controller IP

Here we concentrate on the receiver part. It transmits the received data to its external processor, by its $RxDout$ output bus, while signaling data availability by the $RxDataAvail$ signal. Its outputs $StartOfFrame$ and $EndOfFrame$ indicate the beginning and the end of each frame. When it receives an abort sequence ($7$ consecutive ones), it sets its $AbortFound$ output.

Let us consider the following assertion that states that $RxDataAvail$ cannot be high between two frames:

$$(P_3) \quad \text{always} \{ \text{not } EndOfFrame; EndOfFrame \} \rightarrow \text{next!(not } RxDataAvail \text{ until! StartOfFrame) }$$

The premise of the suffix implication refers to the rising edges of the output signal $EndOfFrame$. The method we use is similar to the one explained for the previous case studies. The constraint $EndOfFrame(t) = 0$ produces 2322 solutions, and $EndOfFrame(t + 1) = 1$ gives rise to 277 solutions. We select a combination of solutions and build the test generator. Then we evaluate assertion coverage, for a simulation with 15000 transmitted data.

With a pseudo-random generation of well-formed frames (a fixed well-formed part of a minimal length, followed by pseudo-randomly generated bits), $(P_3)$ is activated $47$ times. Then, if we introduce our test generator, surprisingly the expected rising edge never occurs. This is due to the fact that we produce abort sequences instead, because the closing flag and abort sequence patterns are very close (0x7E and 0x7F).

Here too, the conditions of use of this IP must be taken into account, they are such that $EndOfFrame$ is typically mostly low. Hence, if the test generator only tries to force it to 1, assertion coverage is now significantly improved: $(P_3)$ is activated $114$ times.

### V. Conclusion

We have described our approach for the automatic construction of test sequences generators driven by PSL assertions and proposed a discussion about its proper use, thanks to first experimental results.

The extraction and selection process still has to be improved for a better scalability. In particular, the correlation that may exist between assertions coverage and the complexity of the selected solution has to be investigated further.

Moreover, the method described here only builds a test generator for one PSL assertion. Future work includes producing a test generator for a set of assertions. In that context, it will be necessary to guarantee that constraints are correctly correlated.

### REFERENCES


\(^1\)Provided by Thales Communications & Security


ACKNOWLEDGMENTS

This author is grateful to Aurélien Guerre-Chaley, Mehdi Makhlof and Nicolas Bouscarle for their contribution to the automation of the toolchain, and to Pierre Ferris and Laila Damri for their help with the experiments.