Improving formal verification for embedded systems by Assertion-Based Verification

Laila DAMRI
Mundiapolis University
Mundiapolis Engineering School
20180 Nouacer, Casablanca, Morocco
l.damri@mundiapolis.ma

Abdellfattah BA-RAZZOUK
Faculté des Sciences et Techniques de Settat
Lab. d’Analyse des Systèmes et Traitement de l’Information
B.P. 577, Km. 3, Route de Casablanca, Morocco
barazzou@yahoo.ca

Abstract—Assertion-Based Verification (ABV) aims at guaranteeing that designs obey properties, usually expressed under the form of logic and temporal formulae. In dynamic ABV, those properties are checked at runtime (e.g., during simulation). In the context of simulation-based verification, the significance of the selected test sequences is well known. Moreover, if the validity of properties is also to be checked, test generation is of utmost importance because properties should not be considered as satisfied if they are satisfied vacuously i.e., without having actually been checked. Test sequences must be designed to ensure a good coverage of the property checker’s activation conditions. This article presents and explains a method for automatic test sequences generation to assure a good verification for critical embedded systems.

Keywords—Embedded systems; Assertion-Based Verification; PSL; VHDL.

I. INTRODUCTION

The continuous improvement of integrated circuit technology led industry to develop more complex and more efficient embedded systems, which exponentially increases the difficulty of verification. This is a problem that is growing day by day. In this context, assertion-based design and assertion-based verification can be viewed as unifying methods across design abstraction levels [2]. Writing assertions concurrently with the design has been found to bring significant benefits to both the design and verification processes for digital circuits. Assertions help detect more functional bugs, earlier and closer to their original cause which leads to fewer bugs remaining undetected after production, and faster debugging.

This work is placed at system structural level and focuses on the assertion-based verification of PSL properties and assumes that the designs under verification are synthesizable (i.e., can be considered as netlists of gates and memory elements). The method described in this paper may be applicable to any kind of PSL checkers.

In this article, we first recall various existing solutions like ATPG\(^2\) methods, techniques for code coverage analysis and some results about the production of test sequences related to temporal properties. These methods are not actually adapted to our needs. We present an improvement to our specific method for automatic test sequences generation directed by the necessity to avoid vacuous assertion satisfaction.

II. RELATED WORK

A. Vacuity

The first works that converge to the purpose as our work is the study of the case of vacuity. The vacuity can be explaining with the following example that shows the vacuity in a given property. It verifies that a system meets the following specification:

\[ P = AG (\text{req} \rightarrow AF \text{grant}) \]

This property means that any \text{req} is inevitably followed by \text{grant}. This property is vacuously satisfied in a system for which \text{req} is never activated.

For example, the work presented in [3] targets the identification of vacuous properties, for assertions of the “simple subset” of PSL. In [4], this problem is already considered using model-checking, and vacuity is defined as follows: a formula \( \phi \) is satisfied vacuously in a model \( M \) if it is satisfied in \( M \) and there exists a sub-formula \( \psi \) of \( \phi \) that can be changed arbitrarily without modifying the model-checking result. Those approaches focus on the detection of vacuity, their goal is different from our which is the production of test sequences that avoid vacuity.

B. ATPG

The ATPG (Automatic Test Pattern Generation) is a process that can generate test sequences in an automatic way (Figure 1) to simulate and analyze the responses of a circuit to detect faults.

---

1 Property Specification Language

2 Automatic Test Pattern Generation or Automatic Test Pattern Generator
ATPG methods (D-algorithm, PODEM [5], FAN [6],...) are designed to produce sequences of test vectors, but aim at detecting given faults, such as stuck-at faults. With the growing interest in SAT-based methods, such methods have recently been integrated into ATPG tools [7], [8]. In [7] for instance, SAT solutions are integrated in the sequential ATPG tool Strategate. The goal is twofold: to improve test sequence generation (the problem is converted into the satisfaction of the outputs of a “miter” circuit that compares the circuit outputs), and to help identify untestable faults. Such tools do not fit our needs. Even constrained ATPG is not appropriate, because constraints are not related to temporal conditions.

C. Coverage analysis

Coverage. In the software and hardware domains, a variety of methods have been proposed for the production of test suites that guarantee a satisfying level of coverage (transition coverage, statement coverage...). For example, the authors of [9] consider the issue of transition coverage in FSMs (Finite State Machines); 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. As discussed thereafter, such an approach based on negating a formula and looking for a counter-example is widespread and could also be used in our case. However, we do not focus on finding one solution, but on finding all the solutions, to select one of the most appropriate ones among them.

The most of the proposed methods are based on Model-checking that always propose the negation of properties, and present the only solution. All these categories of solutions are still not adapted to our needs (guaranteeing an optimum coverage rate for the activation of PSL assertions).

III. VERIFICATION BASED ON AUTOMATIC TEST SEQUENCES

A. Context

Before going any further in this article some definitions are necessary to mention to be able to understand better the context of this work.

• PSL. PSL [10] is a formal language for writing a specification using logical and temporal properties. Based on simple language syntax, it can be combined with various description languages such as VHDL (VHSIC3 Hardware Description Language) and Verilog. It is composed of four layers: Boolean layer, temporal layer, verification Layer and modeling layer.

• Assertions. Logic and temporal assertions written in languages like PSL or SVA [11], used to capture the design intent. It can be checked using static (model-checking) or dynamic (simulation-based) techniques.

Here is a simple example of assertion:

\[ \text{always}(A \rightarrow \text{next}[2](B \text{ before! } C)) \]

The meaning of this property is: it is always true that, if A holds then, starting from two cycles after, B happens before C.

B. Illustrative example: BCD Code Recognizer

The circuit of Figure 2 is a simple implementation of a BCD code recognizer [13]. It has only one primary input I and includes four registers S1 to S4.

![Figure 2: BCD Code Recognizer](image)

This small example will be useful to give a deep illustration of the proposed algorithm.

C. Proper Test Generation

The proposed algorithm is composed on several steps. We will start by:

• Identification of the Activation Conditions. It is necessary to characterize the activation conditions for the property monitors. If the property is already

\[ \text{next}[2](B \text{ before! } C) \]

3 Very High Speed Integrated Circuit
in the (logical or suffix) implication form, the activation condition is trivially the left-hand side of this implication. If the left-hand side of the implication is a combination of conditions on signals (for example a and not b), the activation condition will be deduced by combining accordingly the constraints on each signal.

• Transformation of Constraints. The next step of the algorithm is the transformation of conditions on internal signals or primary outputs into, ideally, conditions on primary inputs. In fact, as soon as the circuit contains structural loops, memory elements will be involved in the resulting constraints. The result is thus expressed in terms of current and possibly past values of primary inputs and memory elements.

![Figure 3: backward in the cone of influence](image)

The goal of the proposed algorithm is as follows: for a signal sig1 of the design and a value x, determine all the solutions for the constraint sig(t) = x. The algorithm propagates this constraint “backward” in the cone of influence of sig (Figure 3) (i.e., going back up to the primary inputs if possible). If the circuit contains structural loops, the algorithm must clearly avoid iterating infinitely in a loop. We assume here that combinational loops are forbidden (i.e., that every structural loop contains at least one memory element). When going through a loop, the algorithm is able to detect the presence of a register that has already been visited, and stops iterating inside this loop.

Using the rules in Figure 4, the algorithm builds the solution tree for sig(t) = x. The nodes of this tree are and and or operators, and its leaves are of the form signal (time) = value, where signal is the identifier of a signal, and time can be t, t-1, ...

• Extraction of solutions. After obtaining the solution tree, it remains to extract the possible solutions from this tree, under the form of conjunctions of atomic solutions. Two ways are proposed in [1] to extract solutions from the tree and build efficient test sequences to activate a given assertion during simulation.

![Figure 4: Rules for the Backward Traversal](image)

![Figure 5: Result for O(t) = false](image)

Example. Let us illustrate the result of this algorithm on the example of BCD code recognizer (Figure 2). If we run the algorithm for the constraint O(t) = false, we get the tree depicted on Figure 5.

Extracting the solutions from this tree, we get 2 solutions:

\[
S3(t) = false \land S4(t) = true \land I(t) = true \land I(t-1) = true
\]  

(1)

\[
S3(t) = false \land S4(t) = true \land I(t) = true \land I(t-2) = true
\]  

(2)

Let us consider that we select solution (1). This constraint can be partitioned into the condition on registers S3(t - 1) = false and S4(t - 1) = true and the condition on the primary input I(t) = true and I(t - 1) = true. When the conditions on S3 and S4 (S3 at false and S4 at true) hold, it forces I to the value true at the current cycle and at the next cycle.

IV. CONCLUSION

In the context of Assertion-Based Verification, we are working towards a solution for characterizing test sequence generators that can fit the requirements imposed by the necessity to avoid vacuous runtime satisfaction of properties.
associated with functional and temporal design specifications. This work is presented to resolve the problem of the vacuity of assertion verification during simulation. Also, to answer to the question that makes difference with others methods presented in section 2: why do we choose to produce all the solutions instead of just generating the first one? The reason lies in the observation that some solutions are more adequate than others to derive a test generator from them.

This work is more detailed in [1] with more examples of critical embedded systems that explain the strongest points of this algorithm over others described in the related works. The main idea is to offer more than one possibility (set of vectors) that can verify a given property with temporal conditions in a real way (without vacuity).

REFERENCES