International Journal of Latest Engineering Research and Applications (IJLERA) ISSN: 2455-7137

Volume – 09, Issue – 12, December 2024, PP – 06-15

Targeted Test Sequence Generation Using the BDD Method to Enhance Assertion-Based Verification Coverage in Simulation

Laila Damri

ENSA of Berrchid, First Hassan University, Morocco

Abstract: Assertion-Based Verification (ABV) is a powerful methodology used to ensure that designs adhere to specified properties, typically expressed as logical and temporal formulas. Widely regarded as an effective approach to functional verification, ABV dynamically evaluates these properties during simulation. Test sequences are crafted to validate the properties and achieve comprehensive coverage of activation conditions.

However, random test generation often leads to insufficient coverage, while exhaustive approaches quickly become impractical as system complexity grows. This article presents an innovative method for automatically generating test sequences using Binary Decision Diagrams (BDDs), ensuring rigorous and efficient verification for critical embedded systems.

Keywords: Assertion-Based Verification, Dynamic vérification, Embedded systems, Process automation, PSL, VHDL.

I. INTRODUCTION

Writing assertions concurrently with the design process is a proven approach that offers significant benefits, not only to the verification process but also to the design phase itself. Assertions, which are logical conditions or properties embedded within the design, serve as self-checking mechanisms that can be used to validate the correctness of the system as it evolves. When integrated early in the design flow, assertions enable the detection of functional bugs much earlier in the process [1] and [2], often when the design is still in its initial stages. This early detection allows for the identification of issues at their source, which reduces the likelihood of bugs remaining undetected after production and accelerates the debugging process. As a result, the design cycle is shortened, and the cost of fixing errors is significantly reduced. Moreover, assertions play a crucial role in improving the overall reliability and robustness of the final system by providing continuous verification of system properties throughout the development and testing phases.

This work focuses specifically on the assertion-based verification of designs at the system structural level, with an emphasis on verifying Property Specification Language (PSL) properties. PSL is a formal language used to specify temporal properties that a system should satisfy, and it has become an essential tool in modern digital design verification. The approach presented in this paper assumes that the designs being verified are synthesizable, meaning that they can be represented as netlists of gates and memory elements that are suitable for actual implementation in hardware. This assumption is critical, as it enables the verification process to bridge the gap between abstract design models and actual hardware realizations, which is essential for ensuring that the final product meets its functional requirements. Furthermore, the method described is flexible and can be applied to various types of PSL checkers, making it adaptable to different verification environments and tools.

In the first part of the article, we provide a detailed review of existing solutions in the field, including Automatic Test Pattern Generation (ATPG) methods, code coverage analysis techniques, and previous results related to the generation of test sequences for temporal properties. These methods, while valuable, fall short in addressing some of the specific challenges encountered when verifying complex temporal properties in embedded systems. For example, existing ATPG methods may not provide the level of coverage needed for certain types of properties, and traditional code coverage analysis does not necessarily correlate with effective temporal property validation. Furthermore, while previous work on test sequence generation has made significant strides, it often does not account for specific issues such as vacuous assertion satisfaction, where assertions are satisfied without genuinely testing the system's functionality.

To address these gaps, we introduce an improvement to our specific method for generating automatic test sequences. The focus of this enhancement is to ensure that generated test sequences avoid vacuous assertion satisfaction. Vacuous assertion satisfaction occurs when assertions pass under trivial or unintended conditions, rendering the test ineffective. This is a critical issue in assertion-based verification, as it undermines the reliability of the verification process and can lead to the false assumption that the system is functioning correctly. Our method ensures that the generated test sequences are meaningful and effective in verifying the functional properties of the system, thereby improving both the quality and efficiency of the verification process. By focusing on this issue, we aim to create a more rigorous approach to assertion-based verification, ensuring that test sequences provide substantial coverage of all relevant activation conditions, while avoiding unnecessary or misleading results.

In summary, this paper introduces a novel approach to automatic test sequence generation, leveraging the power of BDDs (Binary Decision Diagrams) to enhance assertion-based verification for embedded systems. The proposed method not only addresses the critical issue of vacuous assertion satisfaction but also ensures that verification efforts are both comprehensive and efficient. By integrating this approach into the design and verification flow, we aim to significantly improve the overall quality and reliability of embedded systems, making them more robust and ready for deployment in real-world applications.

II. RELATED WORK

The verification of embedded systems has long been a critical challenge in the development of reliable and efficient digital circuits. Over the years, a number of methods have emerged to address these challenges, each with its strengths and limitations. We propose some of this research:

2.1 Vacuity's Studies

The first works that align with the objectives of our research focus on the concept of vacuity in assertion-based verification. Vacuity can be explained through the following example, which illustrates how a property can be vacuously satisfied in a given system. Consider the property:
$P=AG(req \rightarrow AFgrant)P=AG(req \rightarrow AFgrant)$

This property states that any request (req) must eventually be followed by a grant (grant). In a system where req is never activated, this property is vacuously satisfied, as the condition for its activation is never triggered, making the property trivially true.

For example, the work presented in [3] addresses the identification of vacuous properties specifically for assertions expressed in a "simple subset" of Property Specification Language (PSL). In [4], this issue is also tackled using model checking, where vacuity is defined as follows: a formula φ is satisfied vacuously in a model M if φ holds true in M and there exists a sub-formula ψ of φ that can be modified arbitrarily without altering the result of the model checking. These approaches primarily focus on detecting vacuous satisfaction, aiming to identify when properties are trivially satisfied by a system.

However, the goal of these works differs from ours, as we focus on the generation of test sequences that actively avoid vacuity. While the previous methods are designed to detect when vacuity occurs, our approach aims to prevent it by ensuring that the generated test sequences trigger meaningful assertions, thus guaranteeing that the properties are not trivially satisfied. Our methodology is designed to enhance the effectiveness of assertion-based verification by ensuring that all properties are tested under conditions where they can be genuinely validated, avoiding the pitfalls of vacuous assertion satisfaction.

2.2 The ATPG

Automatic Test Pattern Generation (ATPG) is a widely used process that automatically generates test sequences to simulate and analyze the behavior of a circuit, with the goal of detecting faults (Fig.1).

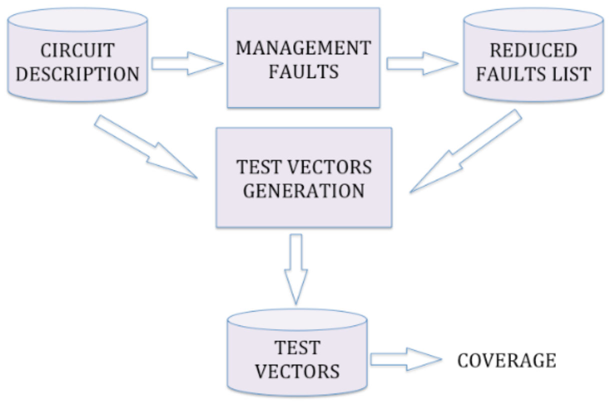


Fig. 1 Architecture of an ATPG system

ATPG methods, including the D-algorithm, PODEM [5], and FAN [6], are designed to generate test vectors that can detect specific types of faults, such as stuck-at faults. These methods typically focus on structural faults, applying predefined patterns to the circuit to observe its responses and identify discrepancies that suggest a fault in the design.

With the increasing adoption of SAT-based (Satisfiability) methods, ATPG tools have evolved to incorporate SAT solvers into their processes [7], [8]. For example, in [7], the Strategate sequential ATPG tool integrates SAT-based solutions to enhance the generation of test sequences. This integration serves two key objectives: first, to improve the generation of test sequences by converting the problem into the satisfaction of outputs from a "miter" circuit, which compares the actual outputs of the circuit under test with expected ones; second, to identify untestable faults that may not be detectable using traditional ATPG methods. While these SAT-based techniques provide improvements in fault detection and test coverage, they are primarily aimed at identifying structural faults and do not focus on verifying temporal properties.

However, these ATPG-based approaches are not suited to our specific needs. The main limitation of these methods is their emphasis on structural faults, such as stuck-at faults, which do not align with the goals of assertion-based verification. Furthermore, even constrained ATPG techniques, which apply specific test constraints, are inadequate in our case, as the constraints they use do not relate to the temporal conditions that are essential for validating assertion-based properties. As a result, while these ATPG methods are highly effective for structural fault detection, they are not applicable for the verification of temporal behaviors and properties in embedded systems, which is the focus of our work.

III. COVERAGE ANALYSIS

In both the software and hardware domains, numerous methods have been proposed for generating test sequences that ensure an adequate level of coverage, such as transition coverage, statement coverage, and branch coverage. For example, the authors in [9] address the issue of transition coverage in Finite State Machines (FSMs). Their algorithm computes test sequences by considering all transitions and leveraging model-checking techniques to generate counterexamples for the negations of these transitions. This approach, which focuses on negating a formula and searching for counterexamples, is a common technique in verification and could be adapted to our context as well.

However, unlike these approaches, which typically aim to find a single counterexample or solution, our work differs in that it aims to find all possible solutions, from which the most appropriate one can be selected. This distinction is critical because, in our approach, we focus on identifying a complete set of valid test sequences that meet the necessary coverage criteria, ensuring a more thorough verification process.

Most existing methods, including those based on model checking, tend to focus on negating properties and presenting only one solution. These approaches are typically designed to identify counterexamples to demonstrate when a property fails, but they do not necessarily ensure comprehensive coverage of all possible activation conditions for assertion-based verification. While these methods are valuable for certain types of verification, they are not well-suited for our needs, which involve guaranteeing optimal coverage for the activation of PSL (Property Specification Language) assertions. Our goal is not just to find one counterexample, but to generate all possible sequences that thoroughly exercise the system's assertions and ensure that the properties are properly verified.

IV. AUTOMATIC GENERATION OF TEST SEQUENCES USING BDD-BASED METHOD

Automatic generation of test sequences is a crucial process in the verification of hardware and software systems. It involves the creation of input stimuli or patterns that drive the system under test (SUT) [13] through different states, transitions, or functionalities, ensuring comprehensive testing. The main objective is to identify potential faults or inconsistencies by simulating real-world scenarios and edge cases.

The primary challenge addressed is the issue of vacuity in assertion-based verification. The work undertaken aims to resolve this issue to improve the quality of verification using assertions during dynamic verification (simulation) and to ensure better assertion coverage. For instance, in a simple example like *P : always (a-->b)*, it would be necessary to guarantee that "*a*" is true often enough. Since our goal is to cover every property in the simple subset, the solution is generally not trivial.

But, before delving further into this article, it is essential to introduce a few definitions to provide better context and enhance understanding of the concepts discussed in this work:

4.1. Assertion Based Verification (ABV)

It remains a challenging and evolving field within functional verification [12]. It involves the use of formal logics and specialized languages, such as PSL (Property Specification Language) and SVA (System

Verilog Assertions), to specify design properties in a concise and precise manner. These assertions capture the intended behavior of the system, ranging from simple combinational checks to complex temporal sequences.

One of the key challenges in ABV lies in the variety of logics and the extensive range of temporal operators available for defining properties. Temporal operators allow designers to express sophisticated time-based relationships between signals, such as causality, precedence, and timing constraints. For instance, properties may specify that "signal A must eventually trigger signal B within N cycles," or that "a reset condition must always precede a state transition." The flexibility and expressiveness of these constructs are powerful but can also introduce significant complexity in writing, understanding, and verifying assertions.

Moreover, the diversity of assertion languages and their underlying semantics can add to the difficulty. Each language offers unique capabilities, but this also means that verification engineers must navigate different syntax, toolsets, and methodologies to effectively implement ABV. Ensuring these assertions are comprehensive and free from vacuity—where a property is trivially satisfied without meaningful activation—is another layer of complexity in ABV workflows.

Despite these challenges, ABV has become an indispensable part of modern verification. It not only helps in identifying functional bugs earlier in the design cycle but also contributes to higher coverage and more robust verification outcomes. Continuous advancements in tools and methodologies aim to simplify ABV, making it more accessible while addressing the intricate needs of increasingly complex systems.

Key Concepts of ABV:

Assertions: Assertions are formal statements, typically written in languages like PSL (Property Specification Language) or SVA (SystemVerilog Assertions) [11], which capture the design's intended behavior. They specify properties such as timing constraints, state transitions, and data integrity.

Types of Assertions:

- ◦ Immediate Assertions: Checked instantaneously during simulation to ensure specific conditions hold true at a given time.
- ◦ Temporal Assertions: Define properties over time, such as "signal A must be followed by signal B within N cycles."

Verification Techniques:

- ◦ *Static Verification:* Uses formal methods like model checking to exhaustively verify that the design satisfies its assertions under all possible conditions.
- ◦ *Dynamic Verification:* Involves simulation-based testing, where assertions are evaluated during test execution to check if they hold true under specific scenarios.

Advantages of ABV:

Early Bug Detection: Assertions allow for quick identification of design flaws, often locating the root cause of the issue early in the verification process.

Improved Coverage: By systematically monitoring key properties, ABV ensures higher coverage of design functionality.

Simplified Debugging: When an assertion fails, it provides clear and localized feedback, making debugging more efficient.

Reusability: Assertions can be reused across different verification environments and adapted for regression testing.

4.2. The PSL

PSL (Property Specification Language) [10] is a formal, specification-oriented language specifically designed to describe both logical and temporal properties of hardware and software systems. It enables designers and verification engineers to precisely capture the intended behavior, constraints, and functional requirements of a system, ensuring that the system operates correctly under all conditions. By providing a unified way to express these properties, PSL aids in detecting design flaws early in the development process.

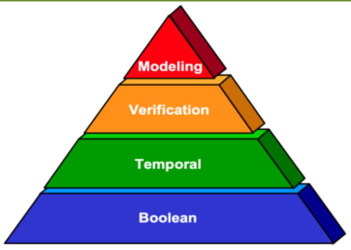


Fig. 2 General structure of the PSL (Property Specification Language)

PSL is structured into four distinct, yet complementary layers (Fig. 2), each addressing a specific aspect of the specification and verification process:

Boolean Layer: At its core, the Boolean layer allows for the expression of fundamental logical operations. It supports basic operations such as AND, OR, NOT, and implications, enabling the specification of simple properties regarding the truth values of signals and states. This layer serves as the foundation for more complex expressions and is essential for defining conditions that must hold true in specific states of the system.

Temporal Layer: The temporal layer is where PSL truly shines, enabling the specification of properties that span over time. It allows the definition of relationships between events that occur at different points in time, such as ordering, precedence, and eventualities. Through temporal operators, designers can express properties like "event A must always be followed by event B" or "if condition C holds, event D must eventually occur." These temporal constraints are crucial for capturing dynamic behavior and ensuring that the system operates according to time-sensitive requirements.

Verification Layer: The verification layer serves as the bridge between the PSL specifications and the actual verification environment. It connects the logical and temporal properties to tools and methodologies used for formal verification and simulation-based analysis. This layer ensures that the properties are not just theoretical, but are actively checked within the verification process, whether through model checking, simulation, or other verification techniques. The goal is to detect violations or bugs related to the specifications during the design and testing phases, enhancing the robustness of the system.

Modeling Layer: The modeling layer is concerned with the translation of a system's design or architecture into a formal representation that can be used by the PSL specification. It takes the high-level design descriptions, often in hardware description languages like VHDL or Verilog, and models them in a way that PSL can apply its properties and checks. This layer ensures that the system is accurately represented and that the PSL specifications can be seamlessly integrated into the design process, enabling a comprehensive verification cycle.

4.3. Automatic generation of test sequences to enhance Assertion-Based Verification (ABV)

In this section, we describe the stages of test sequence selection and demonstrate their application using a practical example: a Parking Management System (PMS). This process aims to ensure thorough verification of the system's functionality by activating relevant assertions and achieving high test coverage.

4.3.1. Case study : Parking Management System (PMS)

The Parking Management System (PMS) is a critical component in modern infrastructure, designed to automate and optimize parking operations. It manages vehicle entry and exit, tracks parking availability, and ensures efficient use of space while maintaining a smooth workflow. This case study focuses on the functional verification of a PMS, with an emphasis on assertion-based verification (ABV) techniques and test sequence generation.

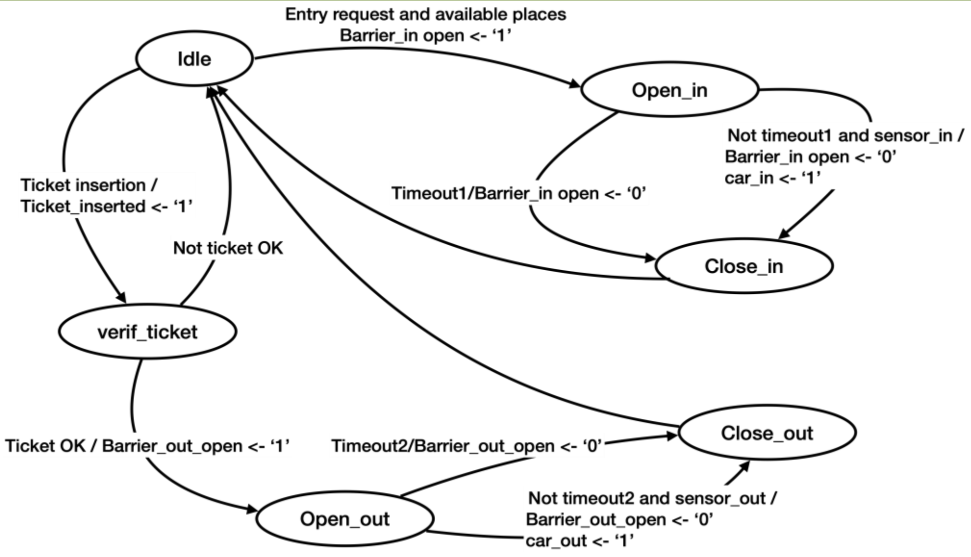


Fig. 3 FSM of the Parking Barrier Controller Circuit

As depicted in Fig. 3, this example operates across six distinct symbolic states. The right branch of the FSM is responsible for processing car entry requests, while the left branch handles exit requests, starting from the idle state.

From the idle state:

- When a vehicle requests entry, if parking spaces are available, the system transitions to the *open_in* state, triggering the barrier to open. If the vehicle does not enter within the allocated time, the system moves to the *close_in* state, commanding the barrier to close. If the vehicle does enter, the system also transitions to the *close_in* state, closing the barrier and updating the vehicle count to indicate a car has entered.
- When a driver inserts an exit ticket, the system transitions to the *verify_ticket* state, registering that a ticket has been inserted. If the ticket is valid, the system moves to the *open_out* state, triggering the barrier to open. If the vehicle does not exit within the allocated time, the system transitions to the *close_out* state, commanding the barrier to close. If the vehicle does exit, the system also transitions to the *close_out* state, closing the barrier and updating the count to reflect that a car has exited.

In this case study, we will examine the assertion P:

(P) always (ticket_inserted → next ! (not ticketOK → not barrier_out_open))

4.3.2. Identification of activation conditions

For assertion P, the structure includes an implication, requiring the identification of operands on the left-hand side. To activate this assertion, the conditions are as follows: the *ticket_inserted* signal must be true, and the *ticketOK* input must transition to false in the following cycle. While the input condition does not require the algorithm to generate a value of '0', the algorithm will be applied to satisfy the constraint t*icket_inserted(t) = true*, initiating its operation based on this signal.

4.3.3. Building a solution tree

The algorithm starts with the t*icket_inserted* output, tracing the constraint associated with this signal back to the circuit's primary inputs and internal states. A solution tree is then generated to capture all potential scenarios that satisfy the condition *ticket_inserted = true*.

4.3.3.1. Constraint transformation

The following step in the algorithm involves converting the conditions on internal signals or primary outputs into conditions on primary inputs whenever possible. However, if the circuit includes structural loops, memory elements will be part of the constraints. Therefore, the constraints are expressed in terms of both the current and, when required, the past values of primary inputs and memory elements.

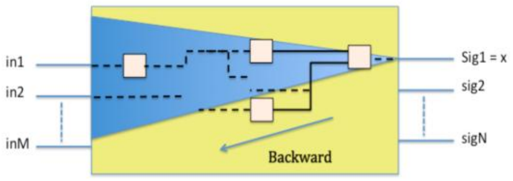
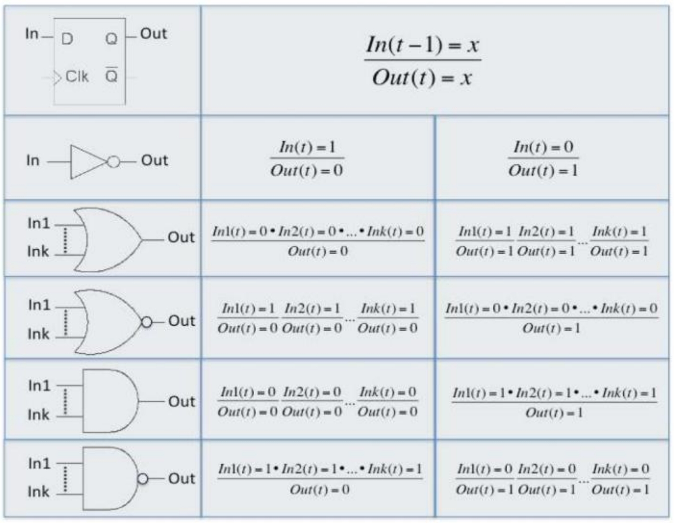


Fig. 4 Backward Traversal in the Cone of Influence

The proposed algorithm is designed to find all possible solutions for a specific signal, sig1, in the design and a given value, x, under the condition sig(t) = x. It achieves this by tracing the constraint "backward" through the cone of influence of sig (Fig. 4), aiming to connect it to the primary inputs whenever feasible.

To manage structural loops within the circuit, the algorithm is programmed to avoid infinite repetition. This is ensured by assuming the absence of combinational loops, with the stipulation that every structural loop must include at least one memory element. When traversing such loops, the algorithm detects revisited registers and halts further processing in those loops.

Using the principles outlined in Fig. 5, the algorithm constructs a solution tree for the condition sig(t) = x. The tree's structure includes AND and OR nodes, with its leaves represented by expressions of the form *signal (time) = value*. Here, signal refers to the signal identifier, and time represents specific time points such as t, t-1, and others.



In D Q Out Clk $\overline{Q}$	$\frac{In(t-1)=x}{Out(t)=x}$	
In Out	$\frac{In(t)=1}{Out(t)=0}$	$\frac{In(t)=0}{Out(t)=1}$
In1 Ink Out	$\frac{In1(t)=0 \bullet In2(t)=0 \bullet \ldots \bullet Ink(t)=0}{Out(t)=0}$	$\frac{In1(t)=1}{Out(t)=1} \frac{In2(t)=1}{Out(t)=1} \cdots \frac{Ink(t)=1}{Out(t)=1}$
In1 Ink Out	$\frac{In1(t)=1}{Out(t)=0} \frac{In2(t)=1}{Out(t)=0} \cdots \frac{Ink(t)=1}{Out(t)=0}$	$\frac{In1(t)=0 \bullet In2(t)=0 \bullet \ldots \bullet Ink(t)=0}{Out(t)=1}$
In1 Ink Out	$\frac{In1(t)=0}{Out(t)=0} \frac{In2(t)=0}{Out(t)=0} \cdots \frac{Ink(t)=0}{Out(t)=0}$	$\frac{In1(t)=1 \bullet In2(t)=1 \bullet \ldots \bullet Ink(t)=1}{Out(t)=1}$
In1 Ink Out	$\frac{In1(t)=1 \bullet In2(t)=1 \bullet \ldots \bullet Ink(t)=1}{Out(t)=0}$	$\frac{In1(t)=0}{Out(t)=1} \frac{In2(t)=0}{Out(t)=1} \cdots \frac{Ink(t)=0}{Out(t)=1}$

Fig. 5 Rules for the Backward Traversal

Once the solution tree is generated, it becomes crucial to extract the solutions it offers. The idea behind creating multiple solutions that fulfill the given constraints is to identify those that are most appropriate for developing targeted test sequence generators.

4.3.4. Extraction of solutions using Binary Decision Diagram (BDD)

4.3.4.1. Binary Decision Diagram (BDD)

The extraction of solutions from the solution tree is carried out through the use of Binary Decision Diagrams (BDD) [14] , a powerful tool for representing and analyzing Boolean functions. BDDs facilitate the identification of all possible solutions that satisfy the given constraints by efficiently representing logical relationships.

A binary decision tree is formed by recursively applying the first part of Shannon's theorem to each variable in a logical function. This process involves iterating through the variables, splitting the function based on each variable's value, and constructing a tree structure where each internal node corresponds to a decision based on a variable, and each leaf node represents the outcome of the logical function. By recursively applying this decomposition, complex logical expressions are broken down into simpler parts, making it easier to analyze and manipulate Boolean functions effectively.

Example: The Shannon theorem is applied to the logical function *f(a,b,c) = /a· b· c + a· c* (Fig. 6). This approach involves systematically breaking the function down by considering the possible values of the variables *a*, *b*, and *c*. The process creates a structured representation, such as a binary decision tree or diagram, which simplifies the interpretation and evaluation of the function for various input scenarios.

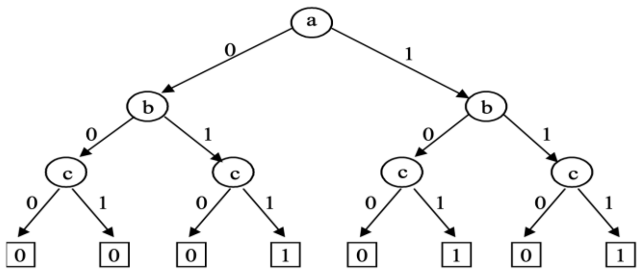


Fig. 6 Binary decision tree corresponding to the function *f(a,b,c) = /a· b· c + a· c*

Once the solution tree is translated into a BDD, the diagram allows for the systematic identification of valid solutions, with each solution corresponding to a distinct path in the diagram. These paths represent different combinations of input values and internal states that fulfill the required conditions.

A Binary Decision Diagram (BDD) is a structured representation where each variable appears only once along any path from the root to a leaf. This characteristic enhances the efficiency of the approach and avoids unnecessary redundancy. The size and complexity of the BDD, however, are greatly influenced by the sequence in which the variables are processed. Consequently, different variable orders can result in BDDs with vastly different sizes.

This approach not only enhances the efficiency of the extraction process but also enables it to handle larger and more complex circuits with ease. By employing BDDs, the method ensures that all potential solutions are captured in a concise format, which is highly beneficial for automating test generation and verification procedures.

4.3.4.2. The CUDD (CU Decision Diagram) package

The CUDD (CU Decision Diagram) package [15] is a powerful and efficient software library designed for the manipulation of decision diagrams, particularly Binary Decision Diagrams (BDDs), along with other types such as Algebraic Decision Diagrams (ADDs) and Multi-Terminal Binary Decision Diagrams (MTBDDs). It offers a comprehensive suite of operations for symbolic manipulation of Boolean functions, including logical operations like conjunction, disjunction, negation, and quantification.

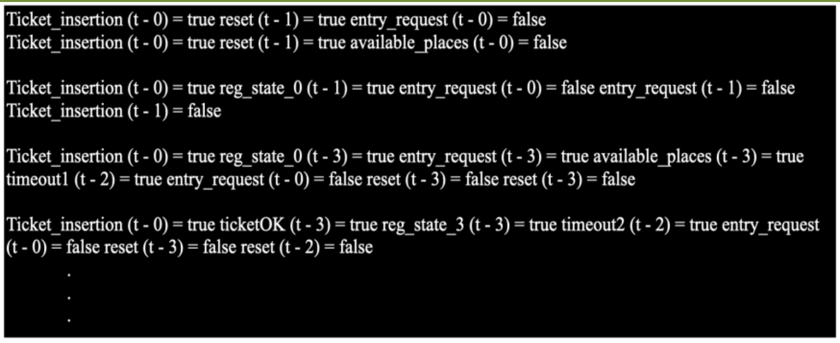
CUDD is specifically optimized to handle large and complex Boolean systems, reducing both memory usage and computational requirements. By employing compact representations of decision diagrams, it facilitates the efficient handling of logical operations even in large-scale systems. This makes CUDD highly valuable in areas such as formal verification, symbolic model checking, logic synthesis, and test generation.

The package provides a range of low-level functions and high-level abstractions for creating, modifying, and optimizing decision diagrams. Its efficiency and versatility make it ideal for tasks requiring symbolic representation and manipulation of Boolean logic, such as verifying hardware designs, performing symbolic model checking, and generating test sequences for verification purposes.

CUDD's scalability ensures that it can manage even the most complex systems, making it an essential tool in both research and industrial applications. Its ability to efficiently represent and manipulate Boolean functions, coupled with its performance, has made it a key tool for tasks like logic synthesis, formal verification, and automated testing.

4.3.4.3. Application

Once the tree is generated, the solutions are extracted by converting it into a Binary Decision Diagram (BDD). This transformation allows for an optimized and structured representation of all possible solutions. The extraction functions then generate a comprehensive list of 24 potential solutions (Fig. 7), each tailored to satisfy the conditions necessary to activate assertion P. This process ensures that all feasible paths are considered for effective assertion coverage.



```
Ticket_insertion (t - 0) = true reset (t - 1) = true entry_request (t - 0) = false
Ticket_insertion (t - 0) = true reset (t - 1) = true available_places (t - 0) = false

Ticket_insertion (t - 0) = true reg_state_0 (t - 1) = true entry_request (t - 0) = false entry_request (t - 1) = false
Ticket_insertion (t - 1) = false

Ticket_insertion (t - 0) = true reg_state_0 (t - 3) = true entry_request (t - 3) = true available_places (t - 3) = true
timeout1 (t - 2) = true entry_request (t - 0) = false reset (t - 3) = false reset (t - 3) = false

Ticket_insertion (t - 0) = true ticketOK (t - 3) = true reg_state_3 (t - 3) = true timeout2 (t - 2) = true entry_request
(t - 0) = false reset (t - 3) = false reset (t - 2) = false
        .
        .
        .
```

Fig. 7 Examples of solutions generated for the constraint *ticket_insere(t)=true* using the BDD

Let's assume we choose this solution:
state_0 (t - 1) = true ; available_places (t - 1) = false ; ticket_insertion (t - 1) = false ; available_palces (t) = false ; ticket_insertion (t) = true ; ticketOK (t + 1) = false ;

The constraint *ticket_insere (t) = true* can be decomposed into conditions that involve the registers. By focusing on the register values, we can determine how the internal states of the system contribute to fulfilling the constraint.

For instance, if the solution requires a specific value in a register at time t, this directly affects the system's response, aiding in the satisfaction of the constraint. Moreover, the values of other registers can influence how the constraint propagates, ensuring *ticket_insereted (t)* holds true.

By breaking the constraint into these components, we can better understand the relationship between the input conditions, register states, and the system's behavior. This method allows for a more focused analysis and efficient test sequence generation, emphasizing the critical elements needed to meet the constraint.

V. CONCLUSION

In the field of Assertion-Based Verification (ABV), this work seeks to provide a comprehensive solution for the characterization of targeted test sequence generators that fulfill the critical requirement of avoiding vacuous satisfaction of properties during runtime. This issue arises when assertions are satisfied without actually being tested in relevant scenarios, undermining the integrity of verification. The study presents an approach based on Binary Decision Diagrams (BDD) to effectively tackle the vacuity problem in assertion verification during simulation. The key contribution of this paper is the development of an algorithm that not only generates multiple candidate solutions but also ensures that the assertions are actively engaged during simulation, thereby preventing vacuity. This method ensures that both functional and temporal properties defined by the design specifications are rigorously verified, providing a robust and thorough validation process for complex systems.

REFERENCES

[1]. L Damri, Y, Generation of test sequences for accelerating assertions, TIMA Laboratory, (2012).

[2]. L.Pierre, L. Damri, J. Book, Improvement of Assertion-Based Verification through the generation of proper test sequences. , In : Proc. FDL (2011)

[3]. Di Guglielmo, L., Fummi, F., Pravadelli, G. J. Vacuity analysis for property qualification by mutation of checkers. In: Proc. DATE (2010)

[4]. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient Detection of Vacuity in Temporal Model Checking. Formal Methods in System Design 18(2) (2001).

[5]. Goel, P.: An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. IEEE Transactions on Computers 30(3) (1981).

[6]. Fujiwara, H., Shimono, T.: On the Acceleration of Test Generation Algorithms. IEEE Transactions on Computers C-32 (1983).

[7]. Prasad, M., Hsiao, M., Jain, J.: Can SAT be used to improve sequential ATPG methods? In: Proc. International Conference on VLSI Design (2004).

[8]. Chen, M., Qin, X., Mishra, P.: Ecient decision ordering techniques for SAT-based test generation. In: Proc. DATE (2010).

[9]. Geist, D., Farkas, M., Landver, A., Lichtenstein, Y., Ur, S., Wolfsthal, Y.: Coverage- Directed Test Generation Using Symbolic Techniques. In: Proc. International Conference on Formal Methods in Computer- Aided Design (FMCAD) (1996).

[10]. IEEE Std 1850-2005, IEEE Standard for Property Specification Language (PSL). IEEE (2005).
[11]. IEEE Std 1800-2005, IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language. IEEE (2005).
[12]. Foster, H., Krolnik, A., Lacey, D.: Assertion-Based Design. Kluwer Academic Pub. (2003).
[13]. Dietmeyer, D.: Logic Design of Digital Systems. Allyn and Bacon (1978).
[14]. Sheldon B. Akers, Binary Decision Diagrams, In : IEEE Transactions on Computers, Vol. c-27, No. 6, June (1978).
[15]. http://vlsi.colorado.edu/~fabio/CUDD/