# Airwolf-TG: A Test Generator for Assertion-Based Dynamic Verification

Jason G. Tong, Marc Boulé and Zeljko Zilic
Integrated Microsystems Laboratory – McGill University
Montréal, Québec, Canada
{jason.tong, marc.boule, zeljko.zilic}@mcgill.ca

***Abstract*—With the emerging predominance of assertion-based dynamic verification, test generation is a key area where assertions can play a bigger role. We consider the generation of test sequences from properties defined by assertions. Such tests are aimed at finding failures in corner-case scenarios of the design specification that test generation alone, without assertions, may not be able to achieve. As such, we take advantage of the information present in the assertions to help build more effective test scenarios – a much needed endeavor given the increasing challenges in verification. We present algorithms in *Airwolf-TG* that generate test sequences from efficient and compact automata produced by *MBAC* tool.**

## I. Introduction

Dynamic, or simulation-based verification is still the most widely used approach for verifying and validating Integrated Circuits (ICs). Its purpose is to compute the output of the Design Under Test (DUT) and to compare the response with its expected behaviour from a specification or a golden model. With the continuing improvements in circuit technology, ICs now contain more functional blocks, processor cores, memory, Digital Signal Processors (DSP), buses and many other components. This has led to an increase in the time spent in the verification stage; however, even with the extra effort, it is increasingly hard to guarantee that the circuit will be entirely bug-free.

Assertion-Based Verification (ABV) has swept into both dynamic and formal verification. Assertions help by increasing the observability within the circuit, and can also help to create behavioural scenarios that can be seen as potential coverage criteria [1], [2]. Properties are defined in modern assertion languages that are based on linear time temporal logic and extended regular expressions. They also define the expected behaviour to which the circuit must abide, i.e. they serve as a specification mechanism. Any deviation causes an assertion to fail, which can be captured by either the simulation environment or by formal methods.

Test generation is an essential step in the dynamic verification process, with the main goal often expressed in some form of a coverage metric. In general, the coverage goal can be thought of as the exploration of the specification space. The effectiveness of dynamic ABV depends entirely on the ability of the generated test sequences to exert the functional coverage from the defined assertions. Hence, in ABV, the test generation process should result in test vectors that try to fully exercise the defined properties.

Two questions are raised in conjunction with dynamic ABV methodology:

- Have we written enough assertions to properly specify the functionality of the circuit?
- Have we produced enough test vectors to exercise most of the possible behaviours of the circuit?

The work presented in this paper addresses the second question, assuming that the first question is answered satisfactorily, i.e., a sufficient set of assertions has been added to the DUT to clearly and unambiguously specify the design intent. We would want to achieve the minimal number of test vectors in order to thoroughly exercise the circuit behaviour based on the defined properties. Most commonly, verification engineers are asked to perform the functional verification, i.e., exercise the expected "good behaviour" by generating test sequences that should produce the anticipated response from the circuit. To more thoroughly verify the circuit, one should attempt to generate test sequences that "break" the DUT as well. By attempting the test sequences that could lead to a faulty behaviour, we can assert much more than just the straightforward simulation of the expected cases.

Within ABV, the test generation could actually use the readily available assertions to generate test sequences. If assertions thoroughly define the properties of the design, then they also provide a blueprint for exploring the relevant common cases as well as the corner-case scenarios. Assertions, then, form the model of the DUT behaviours and provide the means to undertake a model-based test generation. However, if the amount of assertions is insufficient, this can lead to insufficient test vectors which in turn lead to low coverage.

This work falls within the framework of the model-based test generation. Most commonly, generating test sequences in the model-based approach amounts to the generation of a witness trace or a counter-example in Model Checking (MC). MC uses a finite state machine description of the circuit and the temporal logic properties that it must satisfy. From these two inputs, it generates a *product automaton* that describes acceptable behaviours for infinite input sequences. While the product automaton structure, typically a Büchi automaton, allows us (in principle) to verify whether the properties hold for an infinite duration of time, it is does not scale very well and is often not manageable by practical tools, except for reasonably small circuits. This principle is illustrated in Figure 1(a). Generating witness traces or counter examples in

978-1-4244-4823-4/09/$25.00 ©2009 IEEE

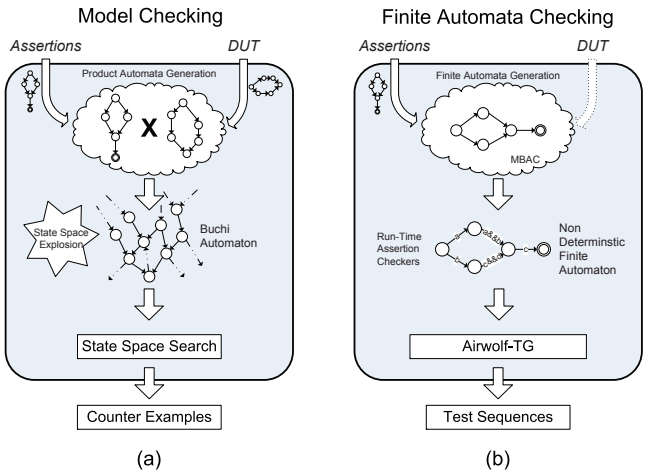


Fig. 1. Model Checker vs. Finite Automata Checker in Model-based Test Generation

MC involves traversing the search space to find a set of input signals that causes the property to pass or fail. This method often suffers from the state space explosion and is prohibitive for realistic circuits.

We propose instead to use finite automata describing *properties alone*. For that, we rely on the tool *MBAC* [3], which produces assertion checkers, which are directly suitable for dynamic verification. The tool explicitly optimizes the finite automata for subsequently producing checker circuits. In our approach, we gain by relying on finite (as opposed to infinite trace) automata, and on the single automaton principle, as opposed to the product automata in MC, shown in Figure 1 (a). The automata that *MBAC* produces [4] are either for failure mode (the typical sequences of inputs leading to assertion failures) or acceptance mode (sequences of successes), and are independent of the two industry-leading assertion languages, namely Property Specification Language (PSL) and SystemVerilog Assertion (SVA). The *MBAC* automata are suitable for run-time verification [3], as opposed to infinite-trace Büchi automata used in formal verification [5].

*Airwolf-TG*, a test generator based on assertion checker automata, takes the automata generated by *MBAC* and efficiently performs a state space search for generating test sequences for failures or acceptances. Table I shows the differences between MC and our proposed Finite Automata checking approach. *Airwolf-TG* has the ability to find test sequences in negligible computation time as opposed to the MC counterpart. This difference is due to the fact that the checker automata are used in isolation, whereas in MC a product automaton construction is required, which leads to a larger search space.

TABLE I
MODEL CHECKING VS. FINITE AUTOMATA (FA) CHECKING

| Category | Model Checking | FA Checking |
|---|---|---|
| Automata Type | Büchi | NDA |
| Represented Traces | Infinite | Finite |
| Use of Product Automata | Yes | No |
| Computation Time | Potentially huge | Low |
| Output Type | Counter Examples | Test Sequences |
| Input HDL | Yes | No |

Our approach is to strategically search the entire automata state space while generating compact test sequences using either failure or acceptance automata. This differs from traditional graph traversing algorithms in several ways. First, a loop detection algorithm is implemented as the automata can contain cycles. These cycles can create an infinite-length test sequence and traversing them should be kept at a minimum. Second, the automata can experience non-deterministic behaviour such that it can enter in more than one new state. Our algorithm has the capability to detect which outgoing edges have similar Boolean expressions (referred as NDA edges), monitor the multiple active states and generates the appropriate test sequences that causes non-determinism. Lastly, *Airwolf-TG* strategically chooses the next subsequent set of edges to include as part of the test vector. This is done by determining the number of times it was traversed (referred as "edge weights") and analyzing its directional properties (ie. cycle, new node, old node, final node, direct-to-finish, NDA edges). This ensures that the test generation uses all of the edges as evenly as possible while not creating any additional, redundant test sequences. The proposed strategy should enable it to generate test sequences that satisfy or fail a property non-vacuously.

The organization of this paper is as follows: Section II presents a survey of previous work related to test generation in ABV, followed by a brief overview of finite automata. In additon, we also show the importance of vacuity in test coverage and end with a discussion on automata coverage metrics. Section III presents *Airwolf-TG* and describes the major algorithms that were used. Section IV shows the experimental results obtained from *Airwolf-TG*, followed by conclusions in Section V.

## II. BACKGROUND AND PREVIOUS WORK

In this section, we begin by providing a short summary of the related work and contributions in the area of test generation based on property specifications. In addition to the literature survey, we also present a brief discussion on Finite Automata (FA) assertion checking, followed by vacuity and test coverage. Lastly, we present some automata coverage metrics that are also used in software testing.

### A. Related Work on Test Generation from Properties

*Oddos et al.* [6] recently developed a tool, namely *MyGen*, which generates hardware circuits that produce test vectors based on automata created by *MBAC*. Their approach aims to produce the test sequences pseudo-randomly with a Linear Feedback Shift Register (LFSR) and Cellular Automata that satisfy a given property based on the *acceptance automaton*. The hardware generators were synthesized for FPGAs; however the size and complexity depends entirely on the transition conditions (Boolean expressions) of each edge.

*Koo et al.* [7] proposed a bounded model-checking method for validating pipelined processors. They generated a model of the MIPS processor in a graph structure. Each node represents

either a pipeline unit (fetch, decode, execute, writeback) or a storage unit (memory or registers). An edge can be either a pipeline or data storage link. Their objective is to generate a counter-example program that violates the user defined property. This is done by traversing the graph model of the microprocessor by finding a set of instructions that causes the property to fail.

*Shimizu et al.* [8] presented a method for generating test sequences based on constraints written as Boolean formulas. These formulas abide to a set of syntactic rules and are independent of each other. They only rely on the state variables from previous states in which they are used for biasing during subsequent test generations. These constraints are then converted into a Binary Decision Diagram (BDD) representation which is used for generating test sequences that lead to a "true" value. The generated sequences are the inputs to the circuit which exerts the expected (good) behaviour of the design. This state space search is repeated for every dynamically created BDD at each clock cycle.

*Calamé et al.* [9] also demonstrated automata techniques for finding test sequences that lead to a property failure. This method requires the specification of the design and the test purpose. The specification is described in the form of a finite state machine which contains a finite set of states, transitions between different states and an initial state. The test purpose is described in automaton form, which represents the constraints that guide the generation of test sequences. Performing the product of the specification and the test purpose automata creates the search space for test generation. State space traversal algorithms are used in creating the test sequences for property failure.

*Pal et al.* [10] proposed a "black-box" approach for performing test generation from user-defined assertions. The objective is to increase and accelerate the production of satisfying test sequences without vacuity. They have varied the controllability and the observability of the signals that were used for assertions by using three vacuity models: direct vacuity, indirect and context-driven vacuity. They have demonstrated that their algorithms are independent from the size and implementation of the DUT.

Previous research efforts have shown different techniques and strategies for generating test sequences from properties. In our approach, *Airwolf-TG* solely uses properties translated into finite automata by *MBAC*, in order to generate test sequences either for failures or acceptances. The next section presents a brief overview of assertion checking with finite automata.

### B. Finite Automata (FA) Assertion Checking

Two modern assertions languages commonly used are PSL [11] and SVA [12]. Some assertions may not be well-suited for circuit simulation but rather for formal methods. For instance, in PSL, there are certain properties that are only suited for model checking over the infinite signal traces, and don't have easily interpretable simulation semantics. In the simple subset, for example, when creating any property, the sequence of events (or Boolean expressions) should occur only in one direction in the expression, just as time flows during simulation. In that way there can be no references to the DUT signal values in the future.

*Boulé et al* created a tool called *MBAC*, that generates hardware checkers from properties written either in SVA or PSL under the simple subset guidelines. *MBAC* first generates a finite automaton representation of each property either in acceptance or failure mode. Following this process, a Verilog representation of the checker circuit is produced, which can then be used as a run-time assertion checker.

An automaton can be represented as a directed graph, where the finite set of states is connected by directed edges. Each edge represents a transition labeled with a Boolean equation that needs to be satisfied prior to the automaton activating a new state. Automata can contain any number of intial and final (or accepting) states. Whenever the automaton enters into any of the final states, the assertion has failed (in failure mode) or passed (in acceptance mode).

The generated checker automata may have edges that are not entirely mutually exclusive symbols. Hence, they are nondeterministic, as opposed to the classical deterministic automata. A more precise way to describe two different types of automata is as follows. *Deterministic Finite Automata* (DFA) contain distinct, mutually exclusive Boolean conditions in which the automata can enter only one active state at a time. *Non-Deterministic Finite Automata* (NDA) contain edges with partially or completely equivalent Boolean conditions that can cause it to enter *a set* of new states at each clock cycle.

The automata that are produced are used in the generation of the hardware checkers, to be used in verification, on-line monitoring and pre- and post-silicon debug. The process of generating the actual hardware for the assertion checkers is beyond the scope of this paper. For further details, the book [3] includes the variety of techniques for producing and using checkers.

### C. Vacuity and Test Coverage in Assertion-Based Verification

We now outline the coverage metrics considered in conjunction with model-based test generation derived from assertions. The goal is to relate the high-level, verification goals to the automata coverage goals, such that we only need to worry about the automata coverage during the test generation

*1) Vacuity in ABV:* Vacuity is a common issue that the ABV paradigm needs to address. Upon simulating a test sequence, an assertion should pass by having the design perform the expected behaviour as specified in the property. An example would be in the SVA statement:

```
assert property (@(posedge clk) req |-> ##1 grant)
```

This property is satisfied when for every assertion of signal `req`, the signal `grant` is asserted on the next clock cycle. On the other hand, the assertion will be also satisfied if the signal `req` was not asserted throughout the entire simulation. As the antecedent `req` was not set, the implication holds true, so the assertion will hold true, even though the circuit could be faulty. The property is said to be satisfied *vacuously* if it passes without being effectively exercised. A vacuous pass of a certain property means that it was not thoroughly

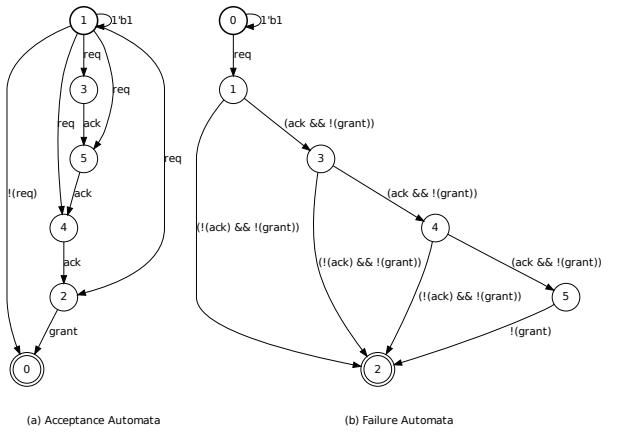


Fig. 2. Acceptance and Failure Automata for the Example Assertion : "assert property (@(posedge clk) req |→ ##1 (ack[*0:3]) ##1 grant);"

executed, such as when the antecedent of the property was not fulfilled [13].

*2) Test Coverage in ABV:* Test coverage is an important metric in verification. It is the measurement of how well a test suite can satisfy a set of requirements or specifications of the design. In the context of ATPG in ABV, the goal of test coverage is to generate test sequences that can satisfy the expected behaviour of the design and exercise corner-case scenarios that can potentially cause a fault.

An example of a property would be the following:
*When the request signal (req) goes from low to high, then at the next clock cycle, the acknowledge (ack) signal must be asserted in at most three consecutive cycles followed by a grant.*
The above specification is translated into an SVA assertion as shown in the caption of Figure 2 and the accompanying automata generated by MBAC.

It is evident in the figures that the automata structures are unique for the acceptance and failure modes in *MBAC*. This can lead to an increase in the number of generated test vectors which can potentially increase the test coverage of the property. For instance, the first automaton in Figure 2(a) shows the paths (or pattern sequences) in order to exert the proper behaviour of the circuit. By efficiently traversing through all of the available paths in this automaton and covering all of the edges in order to generate non-vacuous test patterns, a test set of 5 vectors was derived. Table II lists all the node traversals of each vector for the acceptance automaton. The traversed edges between each node are included in each test vector.

However, when taking into consideration the failure automaton shown in Figure 2(b), an additional test set of 4 vectors was obtained. The corresponding columns in Table II shows each test vectors' node traversal. This implies that when solely using acceptance automata for generating test sequences, there is a possibility of achieving low test coverage. With additional test vectors generated using the failure automata, we can potentially increase the test coverage of the entire test suite. It would be favourable to test the design for all of the correct and incorrect behaviour.

TABLE II
NODE TRAVERSALS FOR FIGURE 2

| Acceptance Automaton | | Failure Automaton | |
|---|---|---|---|
| **Vector #** | **Node Traversals** | **Vector #** | **Node Traversals** |
| 1 | 1,0 | 1 | 0,1,2 |
| 2 | 1,4,2,0 | 2 | 0,1,3,2 |
| 3 | 1,3,5,4,2,0 | 3 | 0,1,3,4,2 |
| 4 | 1,5,4,2,0 | 4 | 0,1,3,4,5,2 |
| 5 | 1,2,0 | | |

TABLE III
TYPES OF AUTOMATA COVERAGE

| Coverage Metric | Coverage Category |
|---|---|
| Node Coverage | Statement or Block Code Coverages |
| Edge Coverage | Branch Coverage |
| Complete Round-Trip Coverage | Deadlocks or Livelocks coverage |
| Complete Path Coverage | Covering executable paths |

### D. Automata Coverage Metrics

In this section, we outline the types of automata (or graph) coverage metrics. These coverage metrics are presented in software testing approaches for code coverage [14], [15] which can also be applied to finite automata for test generation. Shown in Table III are the type of coverages that can be applied to any graph.

*Node Coverage* is the most common type of metric which covers all the nodes of the entire graph. Usually node coverage refers to "statement" or "block coverage" in a source code where each block contains a set of code statements that gets executed. This metric ensures all the nodes in the graph were visited at least once during traversal.

*Edge Coverage* is one of the most widely accepted metric in which its intent is to traverse all the edges of a graph at least once. This is usually referred to as branch coverage where at each node, all the Boolean conditions of outgoing the edges have been evaluated to either true or false at least once.

*Complete Round-Trip Coverage* is a path that contains all the cycles (or loops) that are reachable from a node. A round trip path is defined as a path having identical initial and final nodes of non-zero length. This type of coverage is usually used for detecting dead or live locks. In some cases, round trips can represent an instruction that causes a system to remain in the same state.

*Complete Path Coverage* involves covering all possible independent paths that exist in a graph (or automaton). A path starts from an initial node, then traverses through edges of the graph until it reaches the final node. Obtaining complete path coverage is infeasible if the graph contains any cycles (or loops) that can lead to an infinite path length.

These coverage metrics are applied in our approach in order to generate test sequences without vacuity. The following section describes our proposed algorithms that are implemented in *Airwolf-TG*.

## III. THE AIRWOLF TEST GENERATOR

In the previous section, we presented simple assertions with their automaton representations. Based on the automaton

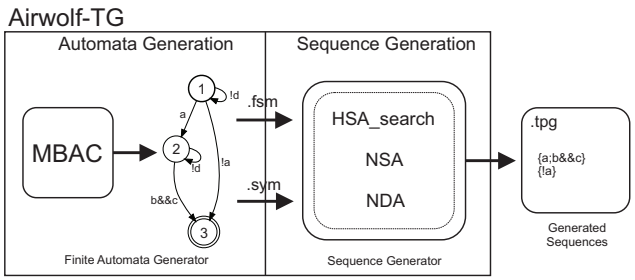


Fig. 3. Test Generation Overview with MBAC and Airwolf-TG

structure, a set of test sequences were generated either to satisfy or fail an assertion. Exploration of the automaton's state space can be a complex task manually, which is best to be automated. We now present our test generation tool: *Airwolf-TG*.

### A. Test Generation Overview

Figure 3 shows *MBAC* and *Airwolf-TG* combined together in generating the test sequences. Test generation occurs in two phases. In the first phase, *MBAC* receives two types of inputs: an HDL model of the hardware design and a list of assertions (or properties) written either in SVA or PSL. *MBAC* analyzes the assertions and produces an efficient automaton represention of the property which in turn will be used to model a hardware checker. In addition, *MBAC* will generate a finite state machine-like description of the automaton along with a list of its symbols, both of which will be used by *Airwolf-TG*. Finally, with *Airwolf-TG*'s core functions, the tool has the ability to generate efficent test sequences.

### B. Objectives

The objective of *Airwolf-TG* is to generate efficient test sequences that can either exercise the expected behaviour of a property or its failure. The goal is to attain 100% automata coverage by applying the test coverage metrics as shown in Table III.

Here are some of the constraints imposed on *Airwolf-TG*:

- Generate test sequences non-vacuously: This requires 100% *Edge Coverage* of an automaton where all edges are used at least once. Our generated test sequences can satisfy or fail a specification non-vacuously. No edge will be left out of the test set.
- Minimizing the reuse of edges that were traversed previously: This helps by reducing the amount of redundant edges that were already covered from a previous recursive search.
- Cycle traversals should be kept at minimum: This constraint implies the use of the *Complete Round-Trip Coverage* criteria by including all the cycles in the automaton; however, in order to reduce the test pattern length, the assumption is to traverse a cycle only once when possible.

The above constraints are then followed and implemented into the algorithms which will be described in the subsequent sections.

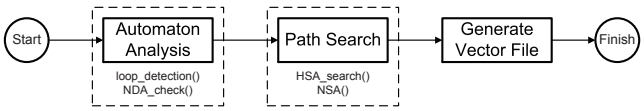


Fig. 4. Airwolf Test Generation Flow

### C. The Flow of Airwolf-TG's Test Generation

Figure 4 shows the test generation flow of *Airwolf-TG*. Initially, the program starts by performing an *Automaton Analysis*. This involves analyzing each node's outgoing edges if there exists an edge that can cause the automaton to enter into a set of new states. Non-determinism can occur when two or more edges can have equal or partially similar Boolean expressions. In addition, Tarjan's algorithm [16] is used to detect cycles in the automaton and to label the appropriate outgoing edge as a "cycle" edge. The directional properties of each edge in the automaton will be analyzed prior to selecting a set of edges for traversal during the state space search.

Following this process is the *Path Search* stage in which the algorithm uses a combination of depth-first and breadth-first graph searching methods. Each active node's outgoing edges is subject to non-determinstic evaluation. A node, or a set of new nodes, is selected for the next recursive call and this process continues until the final node of the automaton has been reached.

The core functions implemented in *Airwolf-TG* are the *Hybrid Search Algorithm* (`HSA_search()`) and the *Node Selection Algorithm* (`NSA()`) which will be described in the subsequent sections.

### D. A Hybrid DFS/BFS Automata Search Algorithm

Algorithm 1 shows the pseudocode of the proposed "*Hybrid DFS/BFS Automata Search Algorithm*" (HSA) which is the automata search-space strategy *Airwolf-TG* employs in order to generate test sequences.

**Algorithm 1** HSA_Search (DAG, CNL, In_paths)

```
1:  NCNL ← initialize_to_empty
2:  while CN != -1 do
3:    for all CN in CNL do
4:      if CN = Finish_Node then
5:        print_vector (In_paths);
6:      else
7:        //Execute Node Selection Algorithm
8:        NSA(DAG, NCNL, CN, IN_paths, back_track);
9:      end if
10:   end for
11:   if NCNL is empty then
12:     break;
13:   else
14:     HFS_search (DAG, NCNL, IN_PATHS);
15:     back_track ← 1
16:   end if
17: end while
```

The HSA function receives three parameters which are: (*DAG*), a directed graph structure of the automaton; Current Node List (*CNL*), a list of nodes that are currently active in the present function call; and *In_paths*, an array where its size is the number of nodes in the automaton that stores all the incoming active edge paths of each node. The latter variable is relevant since it will be used to extract the test sequences.

The HSA starts by analyzing the initial node of the automaton. A *while*-loop (lines 2-17) controls the DFS portion of the algorithm and ends when the Current Node (*CN*) reaches the ID value of the initial node, which is -1. At each recursive call, a *for*-loop (lines 3-10) cycles through a set of active nodes in the Current Node List (*CNL*) that were sent from the previous call. Each node is evaluated if any of them is a final node of the automaton. If this condition is true (lines 4-5), the HSA algorithm will start printing the test pattern which is a set of Boolean expressions obtained for each traversed edge starting from the intial node to the final node. Otherwise (lines 6-9), the outgoing edges of the current node will be passed to the NSA function which will determine the new set of active nodes that will be stored in the New Current Node List (*NCNL*).

After the *for*-loop, the HSA determines if there are any new nodes in the *NCNL*. If this list is empty, the HSA algorithm returns to the previous set of nodes from the previous recursive call of the function. Otherwise, if there is a set of new nodes to be analyzed, the HSA algorithm will call itself again by passing the new node list *NCNL* as the current node list *CNL* parameter for the next recursive function call.

If the HSA does return to the previous recursive call, a flag called "*back_track*" is set to 1 which allows the algorithm to go back to preceding set of nodes to determine if there are any remaining unused outgoing edges. This flag is used in the Node Selection Algorithm (NSA) as described in the next section.

### *E. Node Selection Algorithm*

The pseudocode of the Node Selection Algorithm (NSA) is shown in Algorithm 2. The NSA receives five parameters which are: *DAG*, the contents of which were previously passed by the HSA function; New Current Node List (*NCNL*), which stores the new set of states that an automaton can enter for the subsequent HSA recursive call; Current Node (*CN*) which is an integer node ID that was passed-on by the HSA; *In_paths* passed by the HSA; and (*back_track*) which is a flag that indicates the HSA algorithm is backtracking to a previous set of nodes.

The purpose of the algorithm is to select which set of nodes and edges to traverse and be used as part of the test vector generation. Each node stored in the current list, *CNL*, gets examined by analyzing the type of node and its outgoing edges' directional properties. When the NSA identifies a new set of edges to traverse, it will store the edges into the incoming paths array *In_paths* at each of the edges' Destination Node *DNode*. The stored edges of each *DNode* in the array depicts the active incoming edges at that particular node. Some nodes may have more than one active incoming path. Additionally, the *DNodes* of the selected outgoing edges are added to a new list called the New Current Node List *NCNL*. This new list will be used as the next current node list for subsequent HSA recursive function calls.

**Algorithm 2** NSA (DAG, NCNL, CN, In_paths, back_track)

```
1:  for all paths in In_Paths[CN] do
2:    if CN was not visited then
3:      if DAG[CN] = F-NDA then
4:        Add all NDA edges and DNodes to NCNL
5:      else if DAG[CN] = P-NDA then
6:        Add P-NDA edges and DNodes to NCNL
7:      else if DAG[CN] = N-NDA then
8:        Add one DNode to NCNL
9:      end if
10:     Add selected edges to each DNode's In_paths array
11:     Set CN as visited
12:   else if CN was visited then
13:     if CN has unused edges then
14:       if DAG[CN] = P-NDA then
15:         Add the remaining P-NDA edges
16:       else if DAG[CN] = N-NDA then
17:         Add the remaining N-NDA edges
18:       end if
19:     else if CN has no unused edges, backtrack = 0 then
20:       if DAG[CN] has any DTF then
21:         Add DTF edge to In_Paths[DNode]
22:       else if DAG[CN] has no DTF then
23:         Sort edges by least weight
24:         Add least weight edge to DNode's In_paths
25:       end if
26:     end if
27:   end if
28: end for
```

There are two conditions that determine the set of selected edges for test generation and *DNodes* for subsequent searches. First, the algorithm must determine if the node was previously visited, and secondly, if it contains any non-deterministic edges that can cause an activation of more the one state. If the current node was not previously visited (lines 2-11), the NSA algorithm will evaluate it either as a Full-NDA (*F-NDA*), Partial-NDA (*P-NDA*) or Normal (*N-NDA*) node. For a F-NDA node, all outgoing edges are included. A P-NDA node causes the NSA to select the edges that cause the non-deterministic behaviour while the remaining edges will be incorporated in subsequent recursive calls. For a Normal node, the NSA will select one unused edge.

However, if the current node was previously visited (lines 16-26), the NSA algorithm will determine if that node has any remaining unused P-NDA or normal edges to include in the test generation. If the former condition is true, then the algorithm determines which set of edges will be selected. Otherwise, the NSA function will determine if it can locate an immediate Direct-To-Finish edge (DTF) that leads directly to the final node of the automaton. If none exist, it will sort the outgoing edges in ascending order according to "edge weights" and selects the edge with the lowest weight value. This value depicts the number of times the edge was used from previous

TABLE IV
ACCEPTANCE VERSUS FAILING SEQUENCES

| | | Acceptance Automata | | | | | Failure Automata | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Property ID** | **Total Vectors** | **Airwolf-TG Generated** | **# of States** | **# of Edges** | **Min. Length** | **Max. Length** | **Airwolf-TG Generated** | **# of States** | **# of Edges** | **Min. Length** | **Max. Length** |
| CPX_0 | 5 | 3 | 6 | 12 | 1 | 5 | 2 | 6 | 11 | 4 | 5 |
| CPX_1 | 6 | 4 | 3 | 7 | 1 | 2 | 2 | 3 | 5 | 1 | 2 |
| CPX_3 | 5 | 3 | 3 | 6 | 1 | 2 | 2 | 3 | 5 | 1 | 2 |
| CPX_4 | 8 | **2** | 4 | 5 | 2 | 3 | **6** | 5 | 10 | 1 | 3 |
| CPX_6 | 4 | **1** | 10 | 10 | 9 | 9 | **3** | 10 | 12 | 1 | 9 |
| CPX_7 | 6 | **1** | 43 | 44 | 42 | 42 | **5** | 44 | 50 | 1 | 44 |
| CPX_9 | 12 | 11 | 12 | 22 | 1 | 11 | 1 | 12 | 12 | 11 | 11 |
| CPX_10 | 8 | **3** | 5 | 8 | 1 | 4 | **5** | 5 | 10 | 1 | 4 |
| CPX_12 | 19 | **10** | 12 | 31 | 1 | 11 | **9** | 12 | 30 | 3 | 11 |
| CPX_13 | 12 | 10 | 12 | 31 | 1 | 11 | 2 | 12 | 23 | 10 | 11 |
| CPX_15 | 2 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 2 | 1 | 1 |
| CPX_17 | 33 | **12** | 14 | 26 | 5 | 5 | **21** | 19 | 39 | 1 | 7 |
| CPX_19 | 22 | **6** | 7 | 14 | 1 | 7 | **16** | 9 | 27 | 1 | 8 |

recursive calls which ensures all the edges in the automaton are used as equally as possible.

The algorithms presented in this section allow *Airwolf-TG* to traverse through the finite automata efficiently. It is capable of handling non-determinstic traversals, detection of cycles, and minimize the use of previously traversed edges. In the next section, we present results obtained from *Airwolf-TG* on a set of a selected benchmark properties.

## IV. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present the benchmarks that were used for evaluating the generated test sequences from *Airwolf-TG*. Discussion and analysis are shown in the subsequent sections.

### A. Automata Generation with MBAC

Generation of the two types of automata is performed by the *MBAC* tool which offers two operating modes that include *failure* and *accepting* modes. In the *failure* mode, *MBAC* generates an automaton (or hardware assertion checker) that detects a series of Boolean expressions that should not occur during run-time, with any occurrence leading to a property failure. However, *acceptance* mode generates an automaton that detects a series of expected Boolean expressions that must occur in order to satisfy the property.

### B. Benchmarks Setup

In order to observe the difference in the number of generated test vectors between acceptance and failure automata, we used a set of defined properties from [6], to which *MyGen* tool was applied initially. A set of properties called "Complex" (CPX) represents assertions that are used in industry. The PSL properties are translated into SVA by using the rewrite rules from [3].

Each translated SVA property was given to *MBAC* in order to generate the acceptance and failure automata. The benchmarks listings are shown in [17]. The set of automata and Boolean expression files are then passed to *Airwolf-TG* to generate test sequences. Results are shown and analyzed in the next section.

### C. Results and Analysis

Table IV shows the number of generated vectors for each property when using acceptance and failure automata. The set goal was to have 100% edge coverage, which avoids generating test sequences vacuously. The column "Total Vectors" gives the number of vectors produced by our tool for acceptance and failiure automata combined. The "Min. Length" and "Max. Length" columns gives the length of the smallest and largest vector in the test set respectively. This can be viewed as the number of clock cycles required to either pass or fail the property.

In properties CPX_SVA_0, 1, 3, 9, 13 and 15 there is a subtle difference in terms of the number of generated test vectors when using either acceptance or failure automata; however, for properties CPX_SVA_4, 6, 7, 10, 12, 17, and 19, the difference is more obvious. For instance, in properties CPX_4, 6, 7, and 10, the number of generated vectors for acceptance automata range between 1-3 which implies that the DUT can pass that particular property based on those sequences. By observing the number of sequences generated using failure automata, we see an additional 3-6 vectors required to cause a property to fail. Additionally, some properties are observed to have a significant difference. In particular, properties CPX_SVA_12, 17, and 19 have shown an additional 9, 21 and 16 vectors generated respectively and added to their total test suite. This implies that when considering only test patterns for verifying the proper behaviour of a design, there is a possibility that an unexpected behaviour can occur during its execution or simulation which may be perceived as a correct behaviour, when in fact it may be an incorrect response. The additional vectors generated when using failure automata can potentially increase the test coverage as those patterns should "intentionally break" the circuit and the improper response should manifest itself during simulation.

*MyGen*'s test vectors using acceptance automata are similarly matched to those produced by *Airwolf-TG*. Some properties, however, were not able to generate test sequences to cover all the edges of the acceptance automaton. This is due to the fact that some of the complex Boolean expressions rely on the bit values produced by an LFSR that has never appeared. The authors are currently working towards a better solution to

improve the quality of their pseudo-random vector generation.

Since the *MyGen* tool [6] produces the vectors only for the acceptance automata, in Table V we quantify the extra coverage obtained by our tool. For that, we present the ratio of the number of vectors generated between acceptance and failure automata for each property. The data is obtained by taking the acceptance and failure vector sets and is normalized such that 100% denotes when both cases are taken into account. As shown in the table, a significant increase is observed over using only acceptance automata for generating test sequences, which is the case with the *MyGen* tool. For instance, the inclusion of failing states in examples such as CPX_SVA_12, 17 and 19, can contribute 47.3% to 72.7% of the test suite. Two properties such as CPX_SVA_9 and 13 are the only properties whose acceptance vectors contribute 80% or more of the test suite. These results show that there is a possibility that generating acceptance test sequences alone, as is the case for *MyGen*, may not contribute to a large portion of the coverage. When adding the failure test sequences, as we do in *Airwolf-TG*, additional vectors are available to try exploring the improper behavior. In the case of the design errors violating the properties, the incorrect response would manifest itself at its outputs upon applying these sequences. In contrast, tools like *MyGen* are not aimed at detecting such failures.

TABLE V
ADDITIONAL COVERAGE RELATIVE TO [6]

| CPX ID | Total Vectors | Acceptance Vectors | Failure Vectors | Additional Coverage |
|---|---|---|---|---|
| 0 | 5 | 60% | 40% | 66.7% |
| 1 | 6 | 66.6% | 33.3% | 50% |
| 3 | 5 | 60% | 40% | 66.7% |
| 4 | 8 | 25% | 75% | 300% |
| 6 | 4 | 25% | 75% | 300% |
| 7 | 6 | 16.6% | 83.3% | 500% |
| 9 | 11 | 91.6% | 8.3% | 9.1% |
| 10 | 8 | 37.5% | 62.5% | 166.7% |
| 12 | 19 | 52.6% | 47.3% | 90% |
| 13 | 12 | 83.3% | 16.6% | 20% |
| 15 | 2 | 50% | 50% | 100% |
| 17 | 33 | 36.3% | 63.6% | 175% |
| 19 | 22 | 27.2% | 72.7% | 267% |

## V. CONCLUSIONS AND CONTINUING WORK

In this paper, we have presented *Airwolf-TG*, a tool that generates test sequences by covering the state space of the assertion automata created by *MBAC*. We have introduced methods that effectively traverse the state space of assertions in order to generate efficient test sequences either from failure or acceptance automata. Our proposal is based on a hybrid search algorithm that is catered towards automata for runtime assertion checkers. The automata generated by *MBAC* are not always deterministic, which means that more than one active path can be present in each clock cycle. Our algorithm improvements can exploit the nondeterminism towards more efficient traversal. In addition, as automata may contain cycles that can create longer test sequences or activate more paths, the proposed algorithm minimizes the amount of looping trails during the state space search.

With our *Airwolf-TG* tool being able to traverse efficiently through the automata state-space, we investigated and compared the generated test vectors when using acceptance or failure automata on the same sets of properties. From our experimental results, we have shown that some properties have generated additional test vectors while having a modest increase in the test vector length. This implies that the failing automata can generate in a relatively economical way the additional test cases that can be included into the test suite, which can then potentially increase the overall test coverage.

Since the edges in the automata are Boolean variables that were defined in the property and used during test generation with *Airwolf-TG*, the test generation by automata traversal alone does not result in the sequences of primary inputs, which will be ultimately needed. We will incorporate the methods for justifying primary input signals from the generated test cases obtained by our test generator, using a method as in [7]. With our tool, assertion based test generation is improved—a much needed endeavor given the complex verification tasks both present and future.

## REFERENCES


[1] H. Foster, D. Lacey, and A. Krolnik, *Assertion-Based Design*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[2] D. A. Mathaikutty, S. Ahuja, A. Dingankar, and S. Shukla, "Model-driven test generation for system level validation," *High-Level Design, Validation, and Test Workshop, IEEE International*, pp. 83–90, 2007.

[3] M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer Publishing Company, Incorporated, 2008.

[4] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of psl properties," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 1, pp. 1–21, 2008.

[5] R. Armoni, D. Korchemny, A. Tiemeyer, M. Y. Vardi, and Y. Zbar, "Deterministic Dynamic Monitors for Linear-Time Assertions," in *Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification (FATES/RV'06), volume 4262 of Lecture Notes in Computer Science, Springer*, 2006, pp. 163–177.

[6] Y. Oddos, K. Morin-Allory, D. Borrione, M. Boulé, and Z. Zilic, "Mygen: automata-based on-line test generator for assertion-based verification," in *GLSVLSI '09: Proceedings of the 19th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2009, pp. 75–80.

[7] H.-M. Koo and P. Mishra, "Test generation using sat-based bounded model checking for validation of pipelined processors," in *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2006, pp. 362–365.

[8] K. Shimizu and D. L. Dill, "Deriving a simulation input generator and a coverage metric from a formal specification," in *DAC '02: Proceedings of the 39th annual Design Automation Conference*. New York, NY, USA: ACM, June 2002, pp. 801–806.

[9] J. Calamé, *Specification-based Test Generation with TGV*, Centrum voor Wiskunde en Informatica, May 2006.

[10] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta, "Accelerating assertion coverage with adaptive testbenches," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 27, no. 5, pp. 967–972, 2008.

[11] C. Eisner and D. Fisman, *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[12] S. Vijayaraghavan and M. Ramanathan, *A Pratical Guide for SystemVerilog Assertions*. Springer, 2005.

[13] T. Ball and O. Kupferman, "Vacuity in testing," in *Test and Proofs*, ser. LNCS, vol. 4966. Springer, 2008, pp. 4–17.

[14] P. Ammann and J. Offutt, *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, 2008.

[15] P. Jorgensen, *Software Testing: A Craftman's Approach*. Boca Raton, FL, USA: CRC Press, Inc., 2001.

[16] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[17] [Online]. Available: http://iml.ece.mcgill.ca/~jaytong