Towards Coverage Closure: Using GoldMine Assertions for Generating Design Validation Stimulus

Lingyi Liu, David Sheridan, William Tuohy and Shobha Vasudevan
Department of ECE, University of Illinois at Urbana-Champaign, Urbana, IL, USA
{lui187, dsherid3, tuohy2, shobhav}@illinois.edu

Abstract—We present a methodology to generate input stimulus for design validation using GoldMine, an automatic assertion generation engine that uses data mining and formal verification. GoldMine mines the simulation traces of a behavioral Register Transfer Level (RTL) design using a decision tree based learning algorithm to produce candidate assertions. These candidate assertions are passed to a formal verification engine. If a candidate assertion is false, a counterexample trace is generated. In this work, we feed these counterexample traces to iteratively refine the original simulation trace data. We introduce an incremental decision tree to mine the new traces in each iteration. The algorithm converges when all the candidate assertions are true. We prove that our algorithm will always converge and capture the complete functionality of an output on convergence. We show that our method always results in a monotonic increase in simulation coverage. We also present an output-centric notion of coverage, and argue that we can attain coverage closure with respect to this notion of coverage. Experimental results to validate our arguments are presented on several designs from Rigel, OpenRisc and SpaceWire.

I. INTRODUCTION

Design verification is a primary source of bottlenecks in the system design cycle. Although directed tests capture much of the desired system behavior, they don’t suffice in checking for unintentional erroneous behavior. A phase of random input vector generation is employed with an intention to capture infrequent or unexpected design behavior. Due to the practical infeasibility of exhaustive simulation, the termination point of random simulation is very nebulous. Contemporary industries often use a numeric value like a few million simulation cycles before concluding the random simulation phase. Evidently, such a methodology is unsystematic and inconclusive. Despite multiple coverage metrics, there is no assurance that there are no gaping holes in the design behavior. Coverage closure, or the process of determining the completeness of functional coverage of input vectors is then, one of the most daunting challenges of the present day validation environment.

We propose a methodology for attaining coverage closure of design validation. The methodology is based on GoldMine, an automatic assertion generation tool that was introduced in [15]. In GoldMine, a counterexample is generated for each false assertion using formal verification engine. In this work, we incorporate feedback from the counterexamples generated by GoldMine to refine the simulation data that was used to generate assertions. The test data refined by counterexamples is now used to run another pass of GoldMine. The counterexamples from assertions that fail formal verification are again fed back into the input test suite. This iterative refinement continues until a pass of GoldMine where all the assertions pass the formal check. We introduce a variation on the original decision tree data structure that is built incrementally with every iteration. An incremental decision tree per output adds information from a counterexample for every failed assertion on its leaf nodes. An assertion can be false due to two reasons—either some behavior has not been observed by the decision tree due to insufficient data, or some inference has been made erroneously due to selecting a correlated, but not causal splitting variable. A counterexample prevents the generation of the same spurious assertion and guides the decision tree to navigate regions of input space that have not been considered/observed so far.

Let us now look at how this counterexample guided automatic assertion/test generation process attains coverage closure. Firstly, in GoldMine, we stipulate that only 100 per cent confidence candidate assertions need to be considered for formal verification. This ensures that a failing assertion that produced a counterexample is never reproduced by GoldMine in successive iterations. Since every counterexample provides a trace through the system and the addition of new variables, the corresponding input vector tests for as yet uncovered behavior. Every iteration, therefore, increases coverage of the test suite. This results in a monotonic decrease in the design space uncovered by the tests with successive iterations. In stark contrast to random or directed testing, where arbitrary long phases of coverage stagnation can occur, our method always makes forward progress with respect to test coverage. Secondly, the limiting condition for this algorithm to converge is when there are no failing assertions. At this point, for every decision tree corresponding to a design output, all the assertions in the leaf nodes are true. This provides a deterministic metric of progress for test development. Until all the assertions for a given output pass, the test suite can be improved upon. Thirdly, our method also provides an alternative notion of coverage— one that is output-directed. If all the leaves of a decision tree have true assertions, it implies that the (incremental) decision tree now captures the complete functionality of that output. The decision
our counterexample guided iterative refinement approach to mine tests using GoldMine.

2. Our method provides a deterministic metric of progress in the test development process through incremental decision trees.

3. Our method ensures a monotonic increase in the coverage of the test suite with every iteration. We do not plateau or stagnate with respect to test coverage.

4. We also introduce an output-directed notion of coverage. The tests from our technique will always stimulate only the reachable states of the design.

An outline of the paper is as follows. In Section II, we present the GoldMine background. In Section III, we describe our counterexample guided iterative refinement approach to mine tests. In Section IV, we argue for coverage closure and forward progress of coverage using our technique. In Section V, we demonstrate experimental evidence of the merit of our approach.

II. GoldMine Background

The data generator, decision tree building and formal verification blocks in Figure 2 comprise the GoldMine. The data generator provides the data for the data mining algorithm. The example in Figure 1 for an output $z$ shows the simulation trace data for inputs $a$, $b$ and $c$. GoldMine uses a decision tree based supervised learning algorithm to map the simulation trace data into conclusions or inferences about the design. An error function picks the best splitting variable by computing the variance between target output values and the values predicted by decision variables. The predicted value on each node is the mean of output values, denoted by $M$ while the error at a node is denoted by $E$ in the example. When the error value become zero, it means all output values are identical to the predicted value and the decision tree exits after reaching such a leaf node. A candidate assertion is a Boolean propositional logic statement computed by following the path from the root to the leaf of the tree. In the example, the splitting of input space into two groups after decision on variable $a$ leads to $E = 0$, corresponding to assertion $A1$. Along the $a = 1$ branch, another split occurs on $b$. Assertions $A2$ and $A3$ are obtained at the leaf nodes. Although all simulation data are from the design, GoldMine may still infer candidate assertions that are true of the simulation data, but are not true of all possible inputs. The SMV [12] model checking engine is a part of GoldMine, along with a commercial model checker, to check the correctness of each assertions. In the example in Figure 1, $A1$ is declared false, while $A2$ and $A3$ are declared true.

III. Counterexample-Based Incremental Decision Trees

In order to disprove an assertion, the new data instance consists of all antecedent variables of the assertion and some new additional variables. The antecedent variables’ values are also identical to that in the false assertion and the implied variable’s value is different from that in the false assertion. This characteristic of a counterexample enables a natural way to add it as new data instance to incrementally build a decision tree instead of rebuilding a decision tree from scratch every iteration.

In order to keep track of the improvisation of the decision tree for a given output, we devised an incremental version of the decision tree. The iterative algorithm using GoldMine (depicted in Figure 2) incrementally builds a decision tree for an output until it reaches the goal of generating only true assertions (no counterexamples).

In the recursive incremental decision tree algorithm described in Figure 3, the parts different from GoldMine (lines 4, 7, 8) are outlined. Figure 4 shows a regular decision tree and an incremental version of it.

A decision tree corresponds to a design output. The formal verification in line 4 is employed to check the correctness of assertion whenever a leaf node is reached during the incremental building of decision tree. If a candidate assertion is
true on design, the algorithm returns as in the regular decision tree. In the example, assertions $A1$ and $A2$ generated from original simulation traces are true on the design. If the checked assertion is false/spurious, a counterexample is reported by formal verification. A counterexample: $a = 0$, $b = 1$, $c = 0$ and $z = 1$ is generated to contradict the assertion $A0$ on the decision tree on the left.

The Ctx_simulation() function simulates the input pattern created by the counterexample. This lends concrete values to all the splitting variables in previous iterations of the decision tree in the new simulation run. Since the counterexample follows the same path as the failed assertion, the decision tree continues splitting when it reaches the leaf node corresponding to that false assertion. All other paths of the decision tree are kept unchanged. Due to the new data instance, the mean and error values for each node need to be recomputed using the Recompute_error() function. The error value of the leaf node will no longer be equal to zero. In the example, the incremental decision tree continues to split on the leaf node corresponding to false assertion $A0$ in the regular decision tree. The mean and error value are recomputed in this iteration on the path from the root to the leaf. The algorithm exits when all the assertions at the leaf nodes of an incremental tree are true.

**A. Stimulus Generation for Sequential Behavior**

The sequential behavior of a design is usually expressed in the form of temporal assertions. GoldMine is capable of generating combinational as well as sequential assertions. We need to provide a mining window length, or the duration of time cycles for which we want to capture temporal behavior. For instance, if we want to consider the following behavior: once $a$ is valid, $a$ will be valid two cycles later, the mining window length can be set to 2. All generated assertions will span up to 2 cycles, including $a \Rightarrow XXa$, which is the assertion of interest. We use LTL [14] notation for expressing GoldMine assertions. We can also produce SVA as well as PSL assertions. After unrolling the design for the mining window length, the simulation trace used for assertion mining may have internal register state visible. It may be desirable to have assertions form a single-cycle flat picture of the design, where assertions on the outputs are functions of internal state values and primary inputs. Assertions can also be formed for the internal state variables themselves, as functions of other state registers and inputs. Such a view of the design gives a “next cycle” model, where the assertions describe internal registers and primary outputs in a similar manner. On the other hand, it may be desirable to have temporal assertions on the design that capture only input-output behavior over some number of cycles.

We can generate assertions of both types with this algorithm, based on the mining window length and visible state provided. Although the assertion span sequential behavior over a given length, the generated counterexample may be longer than the mining window length. This may be to expose sequential behavior where an intermediate state variable can be driven to a specific value over several cycles starting from the primary input. In this case, the incremental decision tree algorithm considers only the variables until the farthest back temporal stage, i.e. unrolled until the mining window length. The concrete values of these variables can be acquired through simulation of the counterexample by the data generator. The result is a temporal assertion that spans the mining window length, bolstered by single-cycle assertions using internal state registers to describe the behavior.

**B. Final Decision Tree and Unreachable States**

Our counterexample based incremental decision tree building algorithm is a process of approximation and refinement of an output function. If the complete functionality of an output was available to the decision tree in the form of simulation data, it would completely represent the output function. Such a truth table (or state transition relation for sequential designs) would result in a complete decision tree. However, such an exhaustive enumeration of input patterns is not feasible to obtain as test data. Therefore, the decision tree tries to approximately predict the logic function of an output with available data. Faulty predictions are exposed and used for corrective purposes through counterexamples. This makes future predictions more accurate. At the point where all the predictions are accurate is where all the assertions of the decision tree are true. At this point of convergence, this final decision tree represents the complete functionality of an output in the design. The input patterns required to generate
such a final decision tree are sufficient for completely covering the functionality of that output.

It is important to note that final decision trees include only the legal, reachable states of the design. This is a subset of the state space that is obtained by statically enumerating input combinational or sequential patterns. Static traversal of states does not account for illegal inputs or dynamically unreachable state. However, since the decision trees are constructed out of dynamic simulation data, it only observes the behavior that is executable, thereby eliminating unreachable states. For sequential logic, the algorithm captures the behavior in the assertions for a given length. The constraints on register variables from previous cycles are also captured by the decision tree. Although the assertions are captured for only a bounded number of cycles, the formal verification ensures that the temporal assertions will exclude unreachable or illegal states in the design. This means that the input test patterns that generate a final decision tree comprise exactly the necessary stimulus to capture the output logic. There are no superfluous patterns that reach illegal state in our methodology.

When all the assertions generated from the decision tree are true, either all expressions for an output are completely covered or the uncovered logic in the design will be redundant logic.

C. Algorithm completeness and convergence analysis

We can formally prove the completeness and convergence of the algorithm. Due to space limitation, we just give the proof intuition. The entire proof can be found in technical report [2].

Theorem 1: It takes finite iterations to reach final decision tree for any given initial patterns.

Proof intuition: Provided that there are \( n \) variables in the logic cone, a maximum of \( 2^k \) nodes will be added to the decision tree in \( k^{th} \) iteration. As a result, \( k \) is bounded by \( 2^{k+1} - 1 \leq 2^n + 1 \).

Theorem 2: The final decision tree of the output corresponds to the entire functionality of that output.

Proof intuition: Let it not be, there must be a pattern to reach a state of output function from initial states that does not correspond to a path in final decision tree. Then, this pattern will cause assertion to fail in the final decision tree. But by definition, this final decision tree can have only true assertions. We have a contradiction.

IV. COVERAGE ANALYSIS

In the simplest terms, what we want from a coverage effort is expose the entire legal, reachable design behavioral space to examination so that this space can be validated against a statement of desired behavior. We posit that our algorithm using GoldMine and iterative refinement of the decision tree achieves exactly that property: when the final decision tree for an output has been constructed, the entire reachable design space for that output is captured by the tree. The combination of input patterns and assertions generated by the tree are artifacts that represent the complete functionality of that output. Our notion of coverage, then is output-space directed, as opposed to traditional input space directed notions of coverage. With respect to this notion of coverage, we can achieve functional coverage closure with respect to every output in the design.

Our test generation strategy automatically computes and explores only the reachable state space since it is dynamically derived from simulation data. This is distinct from traditional functional coverage notions that are input-space directed, like expression coverage or conditional coverage. These are not constrained by reachable state space or legal states. So, frequently, we can achieve complete coverage in our methodology, but not complete expression coverage.

GoldMine’s counterexample based approach for test generation ensures a monotonic decrease of the uncovered design space with each iterative refinement. In each iteration, the generated counterexample is able to cover a new design function which has not been covered before by previous patterns. The newly activated function can be in the form of conditional expression, branch or assignment statements in the RTL design. Moreover, the existence of a final decision tree as a goal provides a deterministic metric of progress through the refinement process. This is a significant improvement over random testing, whose coverage graph can be arbitrarily shaped, often resulting in plateaus where no progress is being made. In fact, due to the frequent lack of feedback in the random test generation process, it is difficult to acquire a satisfactory functional coverage picture in this process.

![Fig. 5. Figure showing the coverage of input patterns in the functional design space for an output.](image)

A pictorial example of this process is shown in Figure 5. The state space for a single output can be visualized as a discrete 2D plot, where the functional points covered by the starting input test patterns are marked. Each GoldMine assertion generated includes a set of variable-value pairs according to their statistical support in the patterns.

Every assertion is therefore shown to span a group of points in the output state space by rectangular boxes. This grouping by assertions into “regions” in the output space is similar to a Karnaugh map notation, but this includes sequential behavior as well. For the assertions that are true, the design region has been covered by the input test patterns in that iteration. For the ones that are false, there is always at least one additional design point that was uncovered by the input test pattern. This design point is exposed by a violation of the assertion. Each counterexample (Ctx) acts a bridge between an uncovered design point in (a) and a covered design point as in (b). However, the covered design point in (b) forms a part of the region covered by an assertion, that generates a
counterexample again. All previously true assertions do not perturb the coverage process and are retained in every phase. As a side effect, the original, general assertion is divided into multiple, more precise and subtle assertions.

We notice here that the GoldMine test generation strategy goes from uncovered regions in one iteration to covered regions in another, until it converges at all assertions passing as in (c). This is distinct from a traditional validation flow, where all the known regions are covered first, and an advancement is attempted toward uncovered regions.

V. EXPERIMENTAL RESULTS

We show the practical effectiveness of our method on modules from the Rigel RTL design [9], SpaceWire Codec [1] designs and several modules in OpenRisc [1]. These modules are used for the following experiments: (1) study the coverage increasing with the number of counterexample iterations; (2) limit studies of the counterexample method which starts with zero test patterns and 100% coverage patterns; (3) bug finding; (4) comparison to standard coverage.

The runtime for this algorithm is proportional to number of tests generated. The size of the design, number of initial samples, and maximum number of iterations all affect the number of counterexamples. All tests completed within 24 hours on an Intel Core 2 Quad Q6600 with 4GB of memory. Memory usage is proportional to the number of examples. However, we can dynamically prune all the subtree that generate all true assertions on the decision trees. All tests used well below the 4GB of the machine.

A. Coverage increase by counterexample iteration

The first experiment demonstrates the increase in coverage as the counterexample algorithm progresses, showing a monotonic increase in coverage. The experiment is performed on SpaceWire codec state machine circuit [1]. The original test suite in this experiment is completely random input patterns. We have summarized these results in Figure 6 and Figure 7.

In Figure 6, the input space coverage referring to each output was chosen to measure the validation process. Since each assertion compactly covers multiple concrete patterns of input space, we calculate the input space coverage referring to an output by accumulating the coverage of all generated assertions on that output. The results show a consistent increase in the input space covered by the assertions in each iteration.

B. Zero initial patterns

The second experiment is a limit study showing that the counterexample algorithm works even when no original directed or random test suite exists. The lack of any patterns would begin the procedure with a simple assertion of the form "output always 0". Figure 8 shows the increase in coverage for each design as the algorithm progresses. Even without initial test patterns, the counterexample method is able to create a test suite that achieves good coverage with few iterations. This indicates that this method may be a useful methodology to jump start a module design environment by creating many tests that can then be run on the testbench to check against the design specification.
at least some of the industrial standard coverage metric. The goal is to see whether our method can still find any of the uncovered part in the design by generating counterexamples. If a block already has full or high coverage on some metrics, it is often difficult to get to higher coverage or to know if there is still uncovered part. We evaluate such a condition and were able to derive counterexample tests that indeed improve some coverage that was already quite high on the initial test suites. Table I shows that the data cache controller of OpenRisc [1] with pretty high line and branch coverage achieved higher FSM coverage based on our counterexample method. This experiment also indicates that even the reporting of full coverage cannot guarantee that there are no uncovered part.

### D. Bug Detection by our generated assertions/stimuli

This experiment uses assertions to detect bugs in the design. We implement a systematic mutation-based method to test the assertions’ ability to detect bugs. The RTL code is selected to mutate and all generated assertions are then formally check on the mutated design model. The failed assertions detect the corresponding bug on the mutated design. We inject four types of errors [10]: operator replacement, variable to constant replacement, constant replacement and relational operator replacement. For each output, we inject the errors into its logic cone and then formally check all generated assertions on mutated design. The experiment is conducted on the data cache control module of OpenRisc [1]. For each injected error, there are many assertions that can detect the error. Table II shows the number of injected errors and average percentage of generated assertions that can detect these injected errors.

### E. Comparison to standard coverage

In this experiment, we show the output of several standard coverage analysis comparing standard directed with tests generated by our counterexample algorithm. The circuits three modules are from Rigel [9] cpu pipeline. The directed test suite is written by the designers. From this comparison, we see that counterexample method can help directed test to continually improve coverage. For example, the condition coverage in fetch_stage module is improved from 63.33% to 95.53%.

### VI. RELATED WORK AND CONCLUSION

Counterexample-based refinement of abstractions for verification has been studied widely [4]. The idea of generating tests from counterexamples using model checking has been explored in software testing and hardware validation [3]. These methods require a predefine set of properties and then formally verify these properties. Many techniques automatically generate validation patterns by incorporating coverage feedback dynamically [5] [13]. However, they do not use a flow similar to GoldMine for generating feedback. Statistical methods have been adopted in software validation for likely assertion generation [7] [8] [11] and test generation [6].

In conclusion, we have presented a completely automated stimulus generation methodology for systematic coverage closure based on GoldMine.

### REFERENCES