SEPTEMBER 2010

UILU-ENG-10-2206
CRHC-10-03

TOWARDS COVERAGE CLOSURE: USING GOLDMINE ASSERTIONS FOR GENERATING DESIGN VALIDATION STIMULUS

Lingyi Liu, David Sheridan, William Tuohy, Shobha Vasudevan

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September, 2010	

4. TITLE AND SUBTITLE
Towards Coverage Closure: Using GoldMine Assertions for Generating Design Validation Stimulus

5. FUNDING NUMBERS

6. AUTHOR(S)
Lingyi Liu, David Sheridan, William Tuohy, Shobha Vasudevan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Coordinated Science Laboratory
University of Illinois
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
UILU-ENG-10-2206
(CRHC-10-03)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

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 outputcentric notion of coverage, and argue that we can attain coverage closure with respect to this notion of coverage. Experimental resultsto validate our arguments are presented on Rigel, a 1000+ core processor design as well as several other benchmarks.

14. SUBJECT TERMS
Data mining, verification, test generation, Coverage

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

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

Lingyi Liu, David Sheridan, William Tuohy, Shobha Vasudevan
Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{liu187, 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 Rigel, a 1000+ core processor design as well as several other benchmarks.

1. 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 [16]. GoldMine uses data mining and formal verification to generate assertions. A Register Transfer Level (RTL) design is simulated and the simulation traces are passed as data to GoldMine. GoldMine uses decision tree based supervised learning algorithms to mine rules from the simulated test data. A decision tree is used to statistically infer rules from the data, such that the leaves of the decision tree correspond to candidate assertions for a given output. These candidate assertions are true of the simulated design inputs, but may not be true over all possible inputs. In order to determine if a candidate assertion is true for all inputs or not, the candidate assertions is passed with the design to a formal verification engine. If the formal verification passes a candidate assertion, it is a system invariant. If not, it generates a counterexample. In [16], we showed generation of complex and useful propositional (combinational) as well as sequential (temporal) assertions.

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. The test suite that remains at that point, along with the passing assertions, are the output of our method. 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.

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. Even a single contradicting example in the simulation data is enough to discard a candidate assertion. 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 tree that was predicting design behavior by observing dynamic data, has completely captured the output logic function at the convergence point. The design functionality with respect to

that output is completely covered in our method for both combinational and sequential designs. We provide a proof intuition for the correctness and convergence of our algorithm. Since the decision tree extracts information from dynamic, simulation data, it generates only the reachable state of an output. Unlike static analysis, it is not possible to reach illegal or unreachable states in our method. At the point of convergence, the input test patterns along with the GoldMine assertions represent the validation artifacts required for achieving coverage closure. We consider the validation task complete when the entire functionality of all outputs in the design have been captured, *i.e.* all the assertions for the outputs are true.

Our contributions in this paper are as follows.

- We present a counterexample guided iterative refinement approach to mine tests using GoldMine, an assertion generation tool.
- Our method provides a deterministic metric of progress in the test development process through incremental decision trees.
- 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.
- We prove that for a given output, the incremental decision tree will always converge. We also prove that the complete functionality of an output will be captured at the point of convergence.
- 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 2, we present the background information regarding the tool flow of GoldMine. In Section 3, we describe our counterexample guided iterative refinement approach to mine tests. We detail the creation of incremental decision trees, the principal artifact in mining the tests. Since the treatment of combinational tests is different from sequential tests, we present these discussions separately. We outline a proof that shows the convergence and completeness of our algorithm in Section 4. In Section 5, we argue for coverage closure and forward progress of coverage using our technique. Section 6 details an example that runs through the entire algorithm. In Section 7, we demonstrate experimental evidence of the merit of our approach, using RTL designs from Rigel, a 1000+ core processor, as well as standard ITC benchmarks.

2. INTRODUCTION TO GOLDMINE

We provide the background on GoldMine required to explain our counterexample guided input pattern generation algorithm. GoldMine combines two diverse technologies- data mining and static analysis, to generate assertions automatically. Data mining comprises dynamic, statistical techniques that are computationally efficient, but depend heavily on domain information for deriving relevant knowledge from a system. Static analysis of designs (including formal verification) captures domain (design) information, but suffers from computational complexity issues. Together, these two technologies offset each other's disadvantages. The static analysis when used to guide the data mining, gives rise to useful domain knowledge, *i.e.* assertions. 1 shows the architecture of GoldMine. It is composed of a Data generator, Static analyzer, A-Miner, Formal verifier and A-Val components.

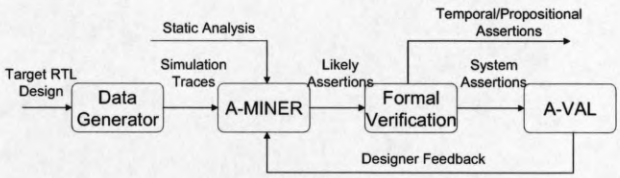

Figure 1: GoldMine architecture

2.1 Data Generator

The data generator phase of GoldMine is used to provide the data for the data mining algorithm. For a given RTL design, the data is obtained through dynamic simulation traces. The design is simulated for a fixed number of cycles (10000) using random input patterns. Regression tests, if available can also be applied to obtain traces.

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, d 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 $a \implies XXb$, which is the assertion of interest. [1]

2.2 Static analyzer

The static analyzer extracts domain-specific information about the design and passes it to the data mining algorithm. We extract the logic cone of influence for every output that we are interested in assertions for. The data mining phase (A-miner) is restricted to analyzing only the logic cone of any output. This limits the search space of the mining algorithm from all the inputs, to the relevant variables in the design.

2.3 Decision tree based learning and A-Miner

The data mining phase of GoldMine is called A-miner, for assertion mining. A-Miner uses a decision tree based supervised learning algorithm to map the simulation trace data into conclusions or inferences about the design.

In the decision tree, the data space is locally divided into a sequence of recursive splits on the input variables. A decision tree is composed of internal nodes and terminal leaves. Each decision node implements a "splitting function" with discrete outcomes labeling the branches. This hierarchical decision process that divides input data space into local regions continues recursively until it reaches a leaf. We require only Boolean splits at every decision node, since our domain of interest is digital hardware. The example in Figure 2 for an output z shows the simulation trace data for inputs a, b and c.

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. When the error value is not zero, the variable with minimum error value is chosen to form the next level of decision tree. A candidate assertion is a Boolean

[1]We use LTL [14] notation for expressing GoldMine assertions. We can produce SVA [1] as well as PSL assertions.

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.

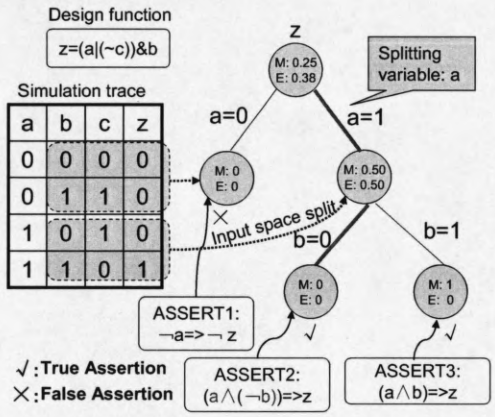

Figure 2: Decision Tree Building Process and Assertion Generation

2.4 Formal verifier and A-Val

The candidate assertions inferred by A-miner are based purely on statistical correlation metrics like mean and error. We restrict the candidate assertions we consider to those with 100% confidence. This means that even if a single example in the trace data does not subscribe to a rule generated by the tree, the rule will be discarded. Despite this strict restriction, A-miner may still infer candidate assertions that are true of the simulation data, but are not true of all possible inputs. To identify candidate assertions that are system invariants, the design as well as the candidate assertions are passed to a formal verification engine. If a candidate assertion passes the formal check, it is a system invariant. Otherwise, the formal verifier generates a counterexample trace that shows a violation of the candidate assertion. The SMV [12] model checking engine is a part of GoldMine, along with a commercial model checker. In the example in Figure2, $A1$ is declared false, while $A2$ and $A3$ are declared true. In GoldMine, A-val forms the evaluation phase for the assertions, to bridge the gap between the human and machine generated assertions.

GoldMine provides a radical, but powerful validation method. Through mining the simulation trace, it reports its findings in a human digestible form (assertion) early on and with minimal manual effort. However, in GoldMine, there is no concept of feedback from any phase to the data miner. Given that data mining performs very effectively when given feedback, in this work we have incorporated feedback from the formal verification phase for enhancing the simulation test data.

3. COUNTEREXAMPLE-BASED INCREMENTAL DECISION TREES

The decision tree is a structure that captures the design model from the perspective of observable behavior. 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,

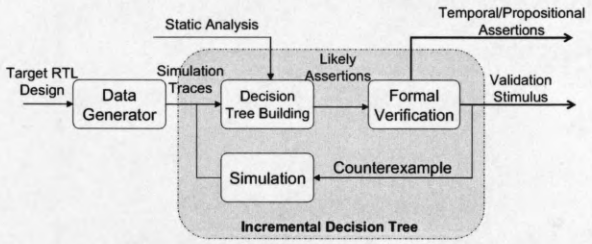

Figure 3: Flow of counterexample-based incremental decision tree algorithm for generating validation stimulus in GoldMine

but not causal splitting variable. A counterexample trace exposes both these situations by introducing scenarios that involve at least one new variable. If this new scenario is now included in the input pattern data observed by the decision tree, firstly it prevents the generation of the same spurious assertion. Secondly, it guides the decision tree to navigate regions of input space that have not been considered/observed so far. A beneficial side effect of this process is the increase in coverage of the input simulation data steadily with every iteration.

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 3) incrementally builds a decision tree for an output until it reaches the goal of generating only true assertions (no counterexamples). The full set of correct assertions, plus the new test patterns created from counterexamples during iterations comprise the tangible outputs of the algorithm.

In the recursive incremental decision tree algorithm described in Figure 4, the parts different from GoldMine (lines 4, 7, 8) are outlined. Figure 5 shows the 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

```
1.  Incr_Decision_Tree_Building(TreeNode Node)
2.  begin
3.    if(Error(Node)==0) begin
4.      if(Formal_verfn(Node.assertion)==true)
5.        return;
6.      else begin
7.        Ctx_simulation();
8.        Recompute_error(Node);
9.      end
10.   end
11.   Node.left=New_node();
12.   Node.right=New_node();
13.   Select_splitting_variable();
14.   Incr_Decision_Tree_Building(Node.left);
15.   Incr_Decision_Tree_Building(Node.right);
16. end
```

Figure 4: Incremental Decision Tree Building Algorithm. The dotted lines represent parts that are different from GoldMine's decision tree building approach.

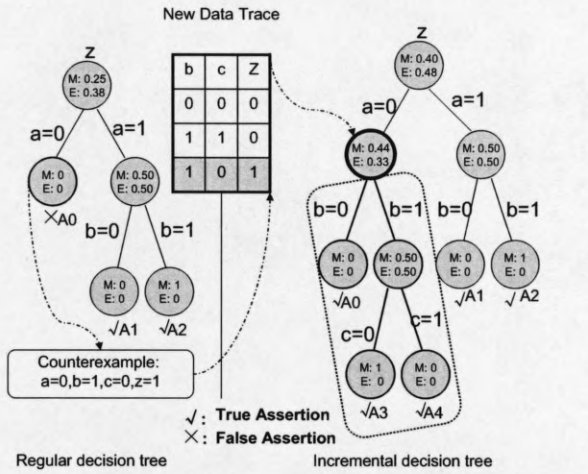

Figure 5: Difference between a regular decision tree and an incremental decision tree for an output z and Boolean inputs a, b and c. The counterexample trace is included in the bottom row of the trace data.

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. It can also be observed that 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.

3.1 Stimulus Generation for Sequential Behavior

During the building of a decision tree, the design should be unrolled until the mining window length, as defined in Section 2.1. 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. We discuss an example of sequential logic coverage in Section 6.

3.2 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.

4. ALGORITHM COMPLETENESS AND CONVERGENCE ANALYSIS

In this section, we prove that our counterexample based test generation algorithm converges and at the point of convergence for any output, the corresponding decision tree for that output represents the complete functionality of that output.

We present some definitions that are required for proving the mentioned claims. Let us consider an RTL design whose state transition graph (Kripke structure [12]) model is depicted by M. We will use M synonymously for the design as well its model. Let there be N inputs in M. An *input pattern* is a unique assignment of values to inputs of M. Input patterns can be combinational (single cycle) or sequential (across multiple cycles). An *input pattern set* is a set of all such input patterns in use for a design validation effort.

The input pattern set for M forms the data for the decision tree algorithm. We define decision trees as used in our context.

DEFINITION 1. *A decision tree D^z for an output z is a binary tree where each node corresponds to a unique splitting variable that is statistically correlated to z. A path for a decision tree is a sequence of nodes from the root node to a leaf node.*

In general, decision trees need not be binary trees, but since our variables are in the Boolean domain, there are only two possible values of each (one-bit) variable. A decision tree is a data structure used in predictive modeling to map observations about a variable of interest to inferences about the variable's target value. In our case, every output of M is a variable of interest. Every output has a corresponding decision tree that makes inferences about the output's target value (true and false). These inferences are made at the leaves of the decision tree, where the branches leading from the root to the leaf represent conjunctions of splitting (correlated) variables. These inferences are also considered likely or candidate assertions for the concerned output.

DEFINITION 2. *A candidate assertion A_C of D^z is a Boolean conjunction of propositions $(variable, value\ pairs)$ along a path in D^z.*

In the next phase of GoldMine, model checking [12] [2] is used to compute the truth or falsehood of a candidate assertion. In case a candidate assertion is false, a *counterexample* or simulation trace through the design is generated, that exemplifies the violation of the assertion.

DEFINITION 3. *A true assertion A_T is a candidate assertion such that $M \models A_T$.*

DEFINITION 4. *The support of a Boolean conjunction y, which is denoted as $support(y)$, is the set of variables in y.*

DEFINITION 5. *If $M \not\models A_C$, the conjunction of variable value pairs in the counterexample is represented by χ_{A_C} such that $support(\chi_{A_C}) \supset support(A_C)$.*

Since the counterexample represents a valid simulation trace through the design that is not yet a part of the current input pattern set, it is added to the input pattern set. An incremental version of the decision tree is used in order to keep track of the coverage. The incremental decision tree maintains the ordering of variables as the decision tree from a previous iteration for all the variables until the leaf nodes. If the counterexample in the current iteration coincides with a path in the incremental decision tree, the variable(s) added by the counterexample will now be used as the splitting variable(s) at the leaf nodes of the incremental decision tree.

DEFINITION 6. *An incremental decision tree I^z for an output z and a previous decision tree D^z, is a decision tree such that the variable ordering of all variables in D^z is preserved until a leaf node. Every variable v in $support(\chi_{A_C}) - support(A_C)$ becomes a splitting variable at the leaf node of I^z along the path of A_C.*

DEFINITION 7. *The final decision tree F^z is an incremental decision tree such that for all assertions A_C of F^z, $M \models A_C$.*

DEFINITION 8. *The logic cone of an output z in M is the set of variables that affect z.*

The logic cone is deciphered by computing the transitive closure of all variables pertaining to an output. In GoldMine, we do a logic cone analysis for every output. The decision tree for an output is therefore restricted to the variables in its logic cone, or the relevant variables with respect to that output.

THEOREM 1. *It takes finite iterations to reach F^z for any given I^z.*

Proof: Let us run the incremental algorithm for k iterations, then the minimum number of new nodes added to I^z is $2k$. The minimum total number of nodes in I^z after k iterations is $2k+1$. Let $n \subseteq N$ be the number of variables in the logic cone of z. The maximum size for D^z by construction and by definition of binary trees is 2^{n+1}-1. Therefore, $2k+1 \leqslant 2^{n+1}$-1. This bounds the size of the incremental decision tree.

It may be noted that since we are restricting the decision tree for an output to focus only on the relevant variables, the maximum size of the decision tree is not exponential in the size of the entire set of inputs N, but in n. In practice, we observe that $n << N$.

THEOREM 2. *The final decision tree F^z corresponds to the entire functionality of z.*

Proof: Assuming a final decision tree F^z does not correspond to the entire functionality of z, then there is at least one input pattern to reach a state of z that does not correspond to a path in F^z, so at least one A_C of F^z should be such that $M \not\models A_C$. But this is false by definition of F^z. Therefore, the assumption is contradicted.

The above theorem makes a powerful statement about the coverage of our method. When all the assertions are true, the complete functionality of an output is captured.

In practice, the learning-based data mining algorithm is able to generate compact assertions, each of which represents several satisfiable input patterns for the corresponding output. The incremental decision tree algorithm can converge quickly to cover all the logic function of corresponding output.

5. 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

[2] We categorize the formal verification algorithms in SMV and Cadence IFV under the umbrella of model checking for this discussion.

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.

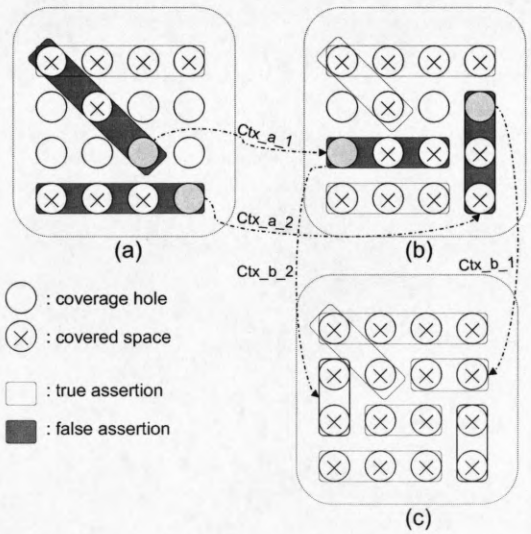

Figure 6: Figure showing the coverage of input patterns in the functional design space for an output.

A pictorial example of this process is shown in Figure 6. 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.

6. EXAMPLE: TWO PORT ARBITER

In this section, we will demonstrate GoldMine's incremental counterexample refinement using a 2 port arbiter. This arbiter uses round robin logic with priority on port 0. In our example, we will unroll the circuit 1 cycle in GoldMine to capture some temporal properties of the of the port 0 access signal, gnt0. The simulation data below represents a directed test that a validation engineer might write. We will show how the A-Miner makes inferences about the design and is aided by the counterexample refinement to improve assertion and directed test quality.

```
always @ (posedge clk)
if (rst) begin
   gnt0 <= 0;
   gnt1 <= 0;
end else begin
   gnt0 <= (~gnt0 & req0) |
   (gnt0 & req0 & ~req1);
   gnt1 <= (gnt0 & req1) |
   (~gnt0 & ~req0 & req1);
end
```

req0 (t-1)	req1 (t-1)	req0 (t)	req1 (t)	gnt0 (t)	gnt0 (t+1)
0	0	1	0	0	1
1	0	1	1	1	0
1	1	0	1	0	0
0	1	1	1	0	1

Figure 7: Arbiter: RTL and simulation trace

The goal of the A-Miner is to partition our simulation data, also known as our example set, into subsets that all display some common behavior based on statistics. Our decision tree data structure starts with a root node which contains all examples and the examples are partitioned into likely behavior by the time they reach the leaf nodes. The initial structure of the decision tree is represented below.

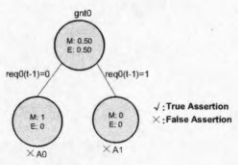

Figure 8: Initial Decision Tree

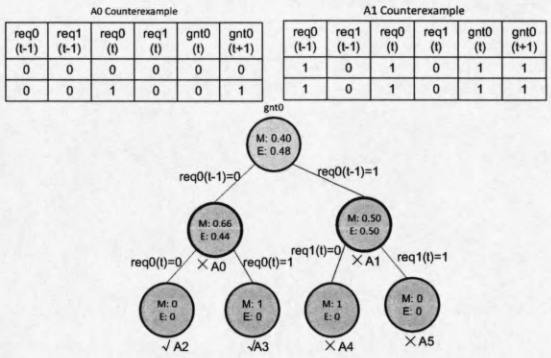

Figure 9: First Iteration: Counterexamples and Refined Tree

A0: $\neg$req0$\Rightarrow X$gnt0
A1: req0$\Rightarrow X\neg$gnt0
The two candidate assertions generated above are proven false by formal verification. A counterexample is produced for each failed assertion containing the series of states that will contradict this assertion. We simulate these counterexamples and add the results to our example set as show below. The decision tree continues to grow since the error is greater than 0 for each node. This means that the confidence is no longer 100% for A0 and A1. The A-Miner finds four more candidate assertions based on the new data.
A2: $\neg$req0$\wedge(X\neg$req0$)\Rightarrow XX\neg$gnt0
A3: $\neg$req0$\wedge(X$req0$)\Rightarrow XX$gnt0
A4: req0$\wedge(X\neg$req1$)\Rightarrow XX$gnt0
A5: req0$\wedge(X$req1$)\Rightarrow XX\neg$gnt0
After one iteration, A2 and A3 are verified to be true. However, A4 and A5 both fail formal verification and a counterexample is produced for each. We again simulate the counterexamples and add them to our data set. The refined tree is shown below.

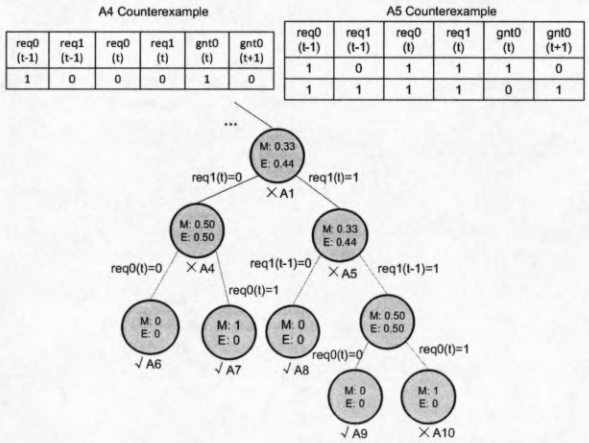

Figure 10: Second Iteration: Counterexamples and Refined Tree

A6: req0$\wedge(X\neg$req0$)\wedge(X\neg$req1$)\Rightarrow XX\neg$gnt0
A7: req0$\wedge(X$req0$)\wedge(X\neg$req1$)\Rightarrow XX$gnt0
A8: req0$\wedge(\neg$req1$)\wedge(X$req1$)\Rightarrow XX\neg$gnt0
A9: req0$\wedge$req1$\wedge(X\neg$req0$)\wedge(X$req1$)\Rightarrow XX\neg$gnt0
A10:req0$\wedge$req1$\wedge(X$req0$)\wedge(X$req1$)\Rightarrow XX$gnt0
A6, A7, A8, and A9 are verified as true. However, A10 is shown to be false even though all primary inputs have been assigned. This is because there is a state outside of the window that affects the output, gnt0. At this point, we allow the A-Miner to search the registers and primary outputs in the farthest back temporal state for a suitable split. In our example, we add the signal gnt0(t-1) to the search. The A-Miner makes this split and produces the full tree below and A11 and A12 are newly generated true assertions.

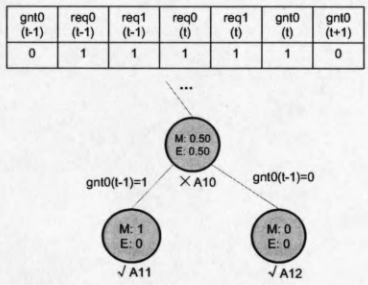

Figure 11: Third Iteration: Full Tree

A11: req0$\wedge$req1$\wedge(X$req0$)\wedge(X$req1$)\wedge$gnt0$\Rightarrow XX$gnt0
A12: req0$\wedge$req1$\wedge(X$req0$)\wedge(X$req1$)\wedge(\neg$gnt0$)\Rightarrow XX(\neg$gnt0$)$
After the assertions are generated by incremental counterexample refinement, the counterexamples can be added to the the original directed test to improve coverage of the test. The series of inputs for each counterexample are simply added to the current input stimulation in the directed test. The improvement in expression coverage of each counterexample iteration is shown below.

Counterexample Iteration	Input Space Coverage (%)	Expression Coverage (%)
0	0	70
1	50	80
2	93.75	90
3	100	90

Figure 12: Coverage of Arbiter Design

7. EXPERIMENTAL RESULTS

To evaluate the quality of our method, we implement the incremental decision tree building algorithm and generate validation stimulus and assertions for several design modules. These include some simple synthetic blocks we created to test various features, and some blocks from the Rigel RTL design [11] and the ITC Benchmark Suite of designs . The simple blocks include a small combinatorial example block (cex_small), a 2-input arbiter (arbiter2), and a 4-input arbiter with more internal state (arbiter4). Specific signals are sometimes indicated in the results or figures, such as `arbiter2.gnt0` for the output signal *gnt0* of the abiters2 module. From the Rigel design three key modules in the processor are chosen: Instruction Fetch (fetch), Instruction Decode (decode), and Instruction Writeback (wbstage). These modules are used for the following experiments:

1. Plot expression coverage of simple modules as GoldMine test generation proceeds
2. Limit studies of the counterexample method
   (a) Zero-pattern seed, start with no test patterns and iterate
   (b) Full-coverage seed, start with patterns that provide

3. Bug finding, inject random faults and use the previously derived assertions as a regression suite to find the bugs

4. Compare and contrast to standard coverage metrics

The runtime for this algorithm is proportional to number of GoldMine tests generated. The size of the design, number of initial samples, and maximum number of iterations all affect the number of counterexamples. A more complex design will require more counterexamples to be generated because in the worst case, the assertion will only be refined by one variable per iteration. The number of initial samples affects how complete the decision tree is before counterexamples are added because a larger number of examples gives a better indication of the design and therefore a more accurate decision tree. As the completeness of the initial tree increases, the number of counterexamples decreases because fewer candidate assertions will need to be refined. The maximum counterexample depth determines how many iterations before the decision tree stops trying to refine an assertion. As the counterexample depth decreases, the runtime decreases as does the accuracy of the decision tree. Experimental results show the average time per formal verification of an assertion to be 1.5 seconds. If a counterexample is produced, the time to produce that counterexample is an additional 1.5 seconds. Most 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, which increases as the number of counterexamples increases. All tests used well below the 4GB of the machine.

Our implementation for this algorithm is a very naTve approach and there are many potential performance optimizations. Every time a candidate assertion is produced, that assertion is formally verified. This means that the formal verifier must compile the full design every time a candidate assertion is produced. An improved approach would collect all candidate assertions and formally verify all assertions at once, drastically decreasing the time spent during formal verification. Counterexamples are also produced at the same time that an assertion is run through formal verification. Combining the counterexamples into one large test bench could drastically cut down on simulation time as well. In addition, only the current node in the decision tree and its descendants can benefit from a counterexample. Using the batched approach, all nodes with failing assertions would benefit from all counterexamples produced, rather than just their own counterexample. Given these enhancements, it would be very reasonable to expect at least a 100% speed increase.

7.1 Coverage Increase

The first experiment demonstrates the increase in expression coverage as the counterexample algorithm progresses, showing a monotonic increase in coverage. We have summarized these results in Figure 13 and FIgure 14. Expression coverage was chosen as a representative example of an industry standard metric, though as explained in Section 5 this metric is often unable to achieve 100% coverage on its own. Redundant statements, unreachable states, and other RTL characteristics often limit expression coverage effectiveness. A steady increase in such coverage is however an indicator of progress in the quality of the assertions and tests created by this algorithm.

This experiment was performed on several typical circuit designs. The first step of the test is to simulate the original test suite. The original test suite can be in the form of a directed test or a completely random input stimulus test. Any spurious assertions are refined using counterexamples until the A-Miner has generated a true assertion or until the counterexample exceeds the length of the unrolled circuit. We can easily calculate the input space covered by an assertion as $1/(2^{depth\ of\ node})$. We accumulate the coverage of all system invariants to determine the input space coverage of our set of assertions. This experiment can be split into several groups.

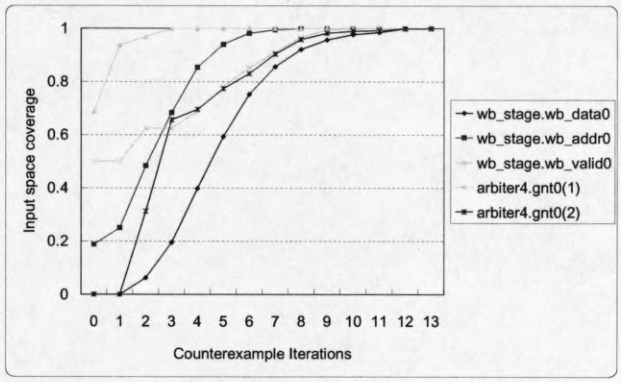

Figure 13: Design Space Coverage by Iteration

Iterations	cex_small	arbiter2	arbiter4
0	66.67%	70%	39%
1	83.33%	80%	82%
2	83.33%	90%	87%
3	83.33%	90%	88%

Figure 14: Expression Coverage Increase by Iteration

- Combinational, directed test: cex_small
- Combinational, random stimulus test: wb_stage
- Sequential, directed test: arbiter2, arbiter4
- Sequential, random stimulus test: fetch_stage

The coverage of the true assertions is evaluated by considering the percentage of the truth table that is covered by that assertion. We consider the possible input space as
$2^{(number\ of\ inputs\ *\ length\ of\ unrolling)}$
We can consider the inputs specified in the assertion as concrete inputs and the rest as don't care inputs. This means that the number of input combinations covered by an assertions is equal to
$2^{(number\ of\ don't\ care\ inputs)}$
In our decision tree, the number of concrete inputs in an assertion is equal to the depth of the node containing it. Based on this information, we can see that the input space covered by a potential assertion is cut in half every time the depth increases. This shows that the input space covered is $1/(2^{depth})$.

The results show a consistent increase in the input space covered by the assertions in each iteration. For the wb_stage and cex_small modules, we note that incremental refinement converges to 100% input space covered. We also note that since the cex_small module is a simple design and the wb_stage module is a complex design, there is a direct correlation between the complexity of the module and the number of counterexample iterations required to converge.

We also notice that there is an exponential increase in the input space covered in the early iterations but only a logarithmic increase in input space covered in the later iterations. This shows that even if incremental refinement is only applied up to a certain depth, there will still be a relatively large coverage gain.

7.2 Zero Initial Patterns

The second experiment is a limit study showing that the counterexample algorithm works even when no directed tests exist. The lack of any patterns would begin the procedure with a simple assertion of the form "output always 0", for example, which the formal verification would show false and provide a counterexample, which would be the first functional pattern. Table 1 shows the increase in coverage for each tested design as the algorithm progresses. Even with no initial test patterns, the counterexample method is able to create a test suite that achieves good coverage with few iterations. This indicates that counterexamples may be a useful methodology to jump start a module design environment by creating many tests that can then be run on the testbench model checkers.

Iterations	0	1	2	5	12	15	17
arbiter2.gnt0	0%	50	75	100	100	100	100
arbiter4.gnt0	0%	0	31.25	69.53	97.29	99.97	100
fetchstage.valid	0%	0	25	100	100	100	100

Table 1: Coverage Percentage by Iteration Starting From Zero Patterns

7.3 Complete Coverage Initial Patterns

The third experiment explores GoldMIne test development on a module that already has full coverage by at least some of the common coverage metrics. The goal is to see if GoldMine tests can find any of the uncovered state in the design by finding counterexamples. If a block already has full coverage on some metrics, and very high coverage on others, it is often difficult to get to higher coverage or to know if higher coverage is even possible. We evaluated such a condition and were able to derive counterexample tests that did indeed improve expression coverage that was already quite high. Figure 15 shows that a block with 100% line and branch coverage, and high condition coverage, achieved higher condition coverage after GoldMine test generation.

Test	line	branch	cond
50 Random Cycles	100	100	93.02
50 Random Cycles + GoldMine	100	100	95.35

Figure 15: Increasing Coverage on High Coverage Block

7.4 Fault Detection by Assertions

The fourth experiment is an example of using the provided assertions in a regression testing environment by injecting and finding bugs in a design that has previously had assertions built on the correct version. We implement a systematic mutation-based method to test the assertions' ability to detect bugs. The internal design signal is selected to mutate and all generated assertions are then formally check on the mutated design model. The failed assertions are considered able to cover the corresponding bug on the mutated design. Since we do not have actual block-level testbench code with monitors and checkers, we have used assertions as the regression vehicle in this experiment, but since the generated test vector suite also has very high coverage it would also be an effective regression suite.

Table 2 shows the results for several randomly chosen signals in the Rigel RTL modules. For a randomly chosen signal in a design, the figure shows the number of assertions that detected the fault. In each case, the assertion suite is able to detect the faults.

Signal	stuck at 0	stuck at 1
stall_in	269	94
branch_pc	35	35
branch_mispredict	8	66
icache_rdvl_i	1	2

Table 2: Faults Covered by Assertions

7.5 Comparison to Standard Coverages

In the second experiment, we show the output of several standard coverage analyses comparing standard directed and random tests with tests generated by the counterexample algorithm. Final coverage values for both Rigel designs and ITC Benchmark designs are included, showing the coverage achieved by the various methods.

	Number of	Random					GoldMine				
Module	Cycles	line	cond	toggle	fsm	branch	line	cond	toggle	fsm	branch
b01	85	98.42	84.38	87.5	71.43	88.89	100	93.75	94.44	76.19	94.44
b02	50	100	X	92.86	66.67	91.67	100	X	92.86	66.67	91.67
b09	28000	100	100	96.77	57.14	90	100	100	96.77	57.14	90
b12	12000	39.42	40.7	58.59	10.47	30.67	40.88	40.7	58.59	10.47	33.33
b17	23000	40.23	17.19	21.85	29.86	34.64	40.23	17.19	21.85	29.86	34.64
b18	10000	33.81	10.53	16.17	25.69	21.61	33.81	10.53	16.17	25.69	21.61

Figure 16: Coverage Comparison Between Random Tests and GoldMine Tests

Table 3 and Figure 16 show comparisons to the GoldMine generated test method and both directed and random test pattern methods, applied to different designs. The **Cycles** column of each table lists the number of simulation cycles run to achieve the given test coverage for the Directed and Random examples, and the number of final test pattern cycles created for the Counterexamples section. From these tables we see that some of the coverage metrics remain low even after 1.5M cycles. In all cases, the final coverage results for tests created by the counterexample method are very high.

8. RELATED WORK

We discuss work that is related to different aspects of our approach. Our counterexample based stimulus generation approach distinguishes itself from all the existing coverage guided test generation approaches in that the generated counterexample is able to automatically explore logic not covered by previous stimulus. 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 [2] [3] [7]. These methods require a predefine set of properties and then formally verify these properties. In our work, the set of properties are generated automatically, minimizing human intervention in the loop. Many techniques in prior art automatically generate validation patterns by incorporating coverage feedback [6] [13] dynamically. However, they do not use a flow similar to GoldMine for generating feedback.

Statistical methods have been adopted in hardware validation for assertion generation [9] [10] [15] and test generation [17] [8] [3]. IODINE [9] tries to automatically infer likely invariants by hypothesizing a set of predefined invariant pattern across one or more variables in the design and then analyzing the deisgn's dynamic behavior during simulation. The generated assertions need not be sound,

Design-	Directed test					GoldMine				
Module	Cycles	Line	Cond	Toggle	Branch	Cycles	Line	Cond	Toggle	Branch
wbstage	1.5M	100%	63.33%	33.96%	100%	9182	100%	95.53%	96.75%	100%
fetch	1.5M	95.92%	87.5%	55.22%	95%	13466	100%	92%	94.46%	100%
decode	1.5M	47.82%	55.04%	81.89%	57.82%	14649	99.87%	76.96%	91.42%	88.17%

Table 3: Coverage Comparison Between Directed Tests and GoldMine Tests

as well as they are usually simple assertions like one-hot encoding. In the field of data mining, incremental decision tree algorithms like VFDT(Very Fast Decision Trees) [5] are explored to allow an existing tree to be updated or revised using new data instances. These are typically applied to handling stream data whose characteristics change over time.

9. CONCLUSIONS

In conclusion, we have presented a completely automated stimulus generation methodology for systematic coverage closure based on GoldMine. The forward progress and termination properties of the algorithm make it a sound and practically attractive solution.

Our methodology starts from uncovered design space and covers it systematically. With every iteration, a new uncovered region is converted to a covered region. This is different from covering all known state space and inching forwards toward the uncovered space.

Although we achieve coverage closure within an implementation, this does not imply adherence to a higher level specification or design intent. We believe that the enhanced test suite output from our methodology can be applied in a validation environment that includes traditional monitors and checkers. We also believe that together, the GoldMine assertions and test vectors have significant value to a design validation effort. but the key contribution comes when a completed Final Decision Tree has been constructed. The existence of this structure itself implies a fully explored and validated design.

Acknowledgements. The authors wish to thank Professor Steven S. Lumetta of the University of Illinois Electrical and Computer Engineering Department for his valuable insights and generous help with computing resources for experiments.

10. REFERENCES

[1] Ieee standard for system verilog: Unified hardware design, specification, and verification language. *IEEE Std 1800-2005*, pages 0–648, 2005.

[2] D. Beyer, A. J. Chlipala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, 2004.

[3] M. Chen and P. Mishra. Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):396 –404, march 2010.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[5] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, 2000.

[6] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of the 40th annual Design Automation Conference*, pages 286–291, 2003.

[7] S. Gurumurthy, R. Vemu, J. A. Abraham, and S. Natarajan. On efficient generation of instruction sequences to test for delay defects in a processor. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 279–284, 2008.

[8] O. Guzey, L.-C. Wang, J. R. Levitt, and H. Foster. Functional test selection based on unsupervised support vector analysis. In *Proceedings of the 44nd annual Design Automation Conference*, pages 262–267, 2008.

[9] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42nd annual Design Automation Conference*, pages 775–778, 2005.

[10] B. Isaksen and V. Bertacco. Verification through the principle of least astonishment. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 860–867, 2006.

[11] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the International Symposium on Computer Architecture*, June 2009.

[12] K. L. Mcmillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[13] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 678–683, 2005.

[14] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[15] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke. Automatic generation of complex properties for hardware designs. In *Proceedings of the conference on Design, automation and test in Europe*, pages 545–548, 2008.

[16] S. Vasudevan, D. Sheridan, D. Tcheng, S. Patel, W. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the conference on Design, automation and test in Europe*, 2010.

[17] C. H.-P. Wen, O. Guzey, and L.-C. Wang. Simulation-based functional test justification using a decision-digram-based boolean data miner. In *International Conference on Computer Design*, 2006.