Mining Hardware Assertions With Guidance From Static Analysis

Samuel Hertz, David Sheridan and Shobha Vasudevan
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

***Abstract*—We present GoldMine, a methodology for generating assertions automatically in hardware. Our method involves a combination of data mining and static analysis of the Register Transfer Level (RTL) design. The RTL design is first simulated to generate data about the design's dynamic behavior. The generated data is then mined for "candidate assertions" that are likely to be invariants. The data mining algorithm is a decision tree based supervised learning algorithm. These candidate assertions are then passed through a formal verification engine to filter out the spurious candidates. The assertions that are attested as true by the formal engine are system invariants. These are then evaluated by a process of designer ranking that is provided as feedback to the data mining engine. We demonstrate the scalability of GoldMine by showing assertion generation of the RTL of Sun's OpenSparc T2 many-threaded processor. Our results show that GoldMine can generate complex, high coverage assertions for sequential as well as combinational designs in RTL, thereby minimizing human effort in this process. GoldMine assertions distill the random input stimulus space and can be used for calibrating directed tests. They can be used in a regression test suite of an evolving RTL. They are also useful in providing differing perspectives from the designer, as well as hints to designers for manually writing assertions.**

***Index Terms*—Verification, Validation, Assertion, Data Mining, Static Analysis**

I. Introduction and Motivation

Whether it is hardware, software or embedded systems, it is hard to imagine their development free of bugs. Lack of satisfactory specifications makes the processes of bug detection and checking correctness more precarious, since these processes hinge on *knowing what one is looking for*.

Assertions or *invariants* [36] provide a mechanism to express desirable or required properties that should be true in the system. Assertions are expressed as predicates in Boolean logic. Assertions encode intended behavior in the form of logical expressions for a state transition system.

Assertions are used for validating hardware designs at different stages through their life-cycle, such as pre-silicon formal verification, dynamic validation, runtime monitoring, and emulation [11], [28], [7]. Assertions are also synthesized into hardware for post-Silicon debug and validation and in-field diagnosis [11], [12].

Among all the solutions for ensuring robustness of hardware systems, assertion based verification has emerged as the most popular candidate [31] solution for "pre-Silicon" design functionality checking. Assertions are used for static (formal) verification as well as dynamic verification of the Register Transfer Level (RTL) design in the pre-Silicon phase.

The key question then is: How are these assertions generated? Assertion generation is an entirely manual effort in the hardware system design cycle. Placing too many assertions can result in an unreasonable performance overhead. Placing too few assertions, on the other hand, results in insufficient coverage of behavior. The trade-off point for crafting minimal, but effective (high coverage) assertions takes multiple iterations and man-months to achieve [28], [65], [49]. Another challenge with assertion generation is due to the modular nature of system development. A module developer would write *local* assertions that pertain to his/her module. Maintaining consistency of inter-modular *global assertions* as the system evolves in this fragmented framework is very tedious. In sequential hardware, *temporal properties* that cut across time cycles are usually the source of subtle, but serious bugs. It is difficult for the human mind to express and reason with temporal relations, making temporal assertion generation very challenging.

We integrate two solution spaces, statistical, dynamic techniques (data mining) and deterministic, static techniques (lightweight static analysis and formal verification), to provide a solution to the assertion generation problem. Static analysis can make excellent generalizations and abstractions, but its algorithms are limited by computational capacity. Data mining, on the other hand, is computationally efficient with dynamic behavioral data, but lacks perspective and domain context.

We present GoldMine, a tool for automatically generating RTL assertions. An RTL design is simulated using random vectors to produce dynamic behavioral data for the system. This data is mined by data mining algorithms to produce rules that are *candidate assertions*, since they are inferred from the simulation data, but not for all possible inputs. These candidate assertions are then passed through a formal verification engine along with the RTL design to filter out spurious assertions and retain the system invariants. Static analysis techniques are employed to guide the data mining process. A designer evaluation and ranking process is facilitated in GoldMine to provide useful feedback to the iterative data mining process.

GoldMine proposes a validation paradigm, where the random pattern generation process can become more tractable. It can distill random input stimulus and then report its findings in a human digestible form (assertions) early on and with minimal manual effort. This technique is intended to replace the traditional flow where the engineer deduces all possible correct behaviors, captures them in assertions, tests assertions, creates directed tests to observe behavior and finally applies random stimulus.

GoldMine is entirely automatic. It is able to generate many

Copyright (c) 2012 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

assertions per output for a large percentage of module outputs in very reasonable runtimes (see case study). It has the ability to minimize human effort, time and resources in the long-drawn assertion generation process and increase validation productivity. Along with input/output or *propositional* assertions, GoldMine can also generate temporal assertions in Linear Temporal Logic [52].[1] GoldMine can generate assertions that are *complex* or span multiple logic levels in the RTL.

A. Use Cases of GoldMine

A legitimate concern about GoldMine's methodology is in the fact that the assertions are generated from the RTL implementation. Hence, the flaws in the RTL could be reflected in the assertions. We provide the context and use cases of GoldMine.

Hardware designers, as a community, find it very challenging to write assertions for checking their own design. With verification and reliability constraints being very stringent, each designer is typically mandated to write some number of assertions for the module developed by him/her. Since the most effective verification is provided by meaningful, non-trivial assertions that have a different perspective from the original design, the task becomes even more complicated.

In this scenario, GoldMine provides a mechanism to generate assertions with differing perspectives from the designer. Due to its mechanical nature, it is able to frequently relate variables across multiple clock cycles that result in complex temporal relationships that human beings cannot generate on their own. From designer feedback for GoldMine, we find that it can also provide excellent hints that can then be used by designers to write their own manual assertions.

In software, there are multiple techniques where invariants are generated from the source code itself, since programmers, like designers find it very challenging to write assertions [8], [61], [50], [27], [67], [29], [57]. Therefore, a similar use case is observed there as well.

Random stimulus is applied late in the validation phase, when the design and assertion-based verification environment are mature enough to withstand and interpret random behavior. GoldMine explores the random stimulus space and distills it into assertions that a human can review. GoldMine's data mining, then, gains knowledge about design spaces that are as yet unexplored by a human-directed validation phase. Eventually, the manual, iterative process of validation will arrive at a point of high coverage. Using GoldMine, however, this step can be done very early in the design, making a leap in the validation cycle. If an unintended invariant behavior is observed, a bug is detected. Otherwise, an assertion that can be used for all future versions of the design has been generated. GoldMine assertions can therefore be used to calibrate the directed test suite, where a metric of goodness for the tests can be to stimulate a majority of the assertions.

GoldMine can be applied in a variety of ways in the verification process. Generated assertions can be used as a regression test suite when generated from a stable or legacy RTL to ensure correctness of future evolutions of the RTL. In [45], the GoldMine assertion generation loop is taken to completion using iterative refinement. This has an application in generating validation input stimulus that is monotonically increasing in coverage. Eventually, this provides a complete test set for an output. The data mining algorithms of GoldMine can be applied at system level or golden RTL, and the candidate assertions can be used for verification against other implementations. We have applied GoldMine to SystemC [2], UML and other specification level descriptions of a design.

An outline of the GoldMine methodology and tool flow was presented in [63]. We originally presented a brief case study of GoldMine on the Rigel [40] 1000+ core architecture design. In this work, we provide a detailed treatment of our methodology. We include details of the decision tree implementation and temporal assertion generation.

In this work, we apply GoldMine to the Sun OpenSparc T2 processor [1] which demonstrates the scalability of GoldMine to an industrial size design. For all tested designs, including OpenSparc, execution times were less than an hour when 1 million cycles of simulation data were used. We present a detailed analysis of all the experimental results. We show that GoldMine is able to find true assertions for more than 80% of the output variables in the OpenSparc MMU. We show that assertions generated by GoldMine can be formally verified efficiently within 5 minutes. We demonstrate that assertions generated by GoldMine achieve high functional coverage. We also show that assertions generated by GoldMine are representative of the specification of the design under test. Finally, we show that GoldMine requires less than 3 hours and 1 gigabyte of memory to generate assertions for the OpenSparc MMU. The results show that GoldMine is suitable for designs of large size and complexity, making it a valuable addition to any verification environment in an industrial sized design.

Our contributions in this work are as follows.

- Our tool, GoldMine, produces complex, high coverage assertions. Our method abridges the validation phase by distilling random stimuli and achieves coverage of unexplored spaces earlier than typical in the design cycle.
- We demonstrate the scalability of GoldMine by applying it to the OpenSparc T2 processor, which is an industrial sized design.
- We introduce an algorithmic methodology to introduce designer subjectivity into the assertion generation process. Conceptually, the combination of statistical, dynamic methods with deterministic, static methods is novel in the context of assertion generation.

This paper is organized as follows. Section II provides insight into the scalability of GoldMine. Section III provides background terms. In Section IV, we provide an overview of the GoldMine assertion generation engine, the decision tree algorithm and a detailed example. In section V, we discuss work related to our methodology. In section VI, we analyze our methodology using Rigel, a 1000+ core CPU. In section VII, we show experimental results for our methodology using the OpenSparc T2 CPU. In section VIII, we analyze the experimental results for the OpenSparc processor. We conclude with Section IX.

[1] We express our assertions using LTL in the definitions. We express the tool output using the SVA language.

II. The GoldMine Principle: Statistics Meet Static

In this section we present the intuition behind the scalability and effectiveness of our solution. This section can be skipped without loss in continuity to the reader.

Data mining is the process of deciphering knowledge from data [60], [59]. Data mining uses dynamic behavior in the form of simulation data or training sets to find statistical correlations and make inferences about the system. In typical data mining applications like web-mining, online recommendation systems, health-care and bioinformatics, there are three striking features.

Firstly, the underlying systems, namely human psychology, human interest patterns, medical records, protein sequences or web data are highly unstructured and prone to wild fluctuations.

Secondly, it is not possible to claim that the knowledge generated by data mining is *true*, since it relies purely on statistical evidence and there is no oracle to attest it. For highly unstructured systems (or those whose structure has not yet been deciphered), behavioral traces are the only information that can be gleaned from the system.

Thirdly, data mining suffers from the problem of not being able to simulate judgement i.e., it will not be able to decide how "interesting" a piece of information (say, a rule) is to a domain expert [35], [47], [58], [59]. It relies on iterative learning from a human domain expert and tries to predict interestingness, or relevance of a rule to the domain where it is applied.

In contrast to typical applications of data mining, hardware designs are extremely structured and their design is a very regulated process. This means that data mining algorithms would have higher accuracy and predictability than usual.

Since these systems are designed by humans, there is a notion of "absolute truth" in these systems that can be attested by an oracle - either the specification document, a reference model, or the system architect. The data mining algorithms can get reinforcement for their correct statistical inferences as well as quick recovery from their wrong ones. Formal verification techniques are built to answer the questions of truth about a system automatically [30] as well as expose falsehood through a witness. A formal verification algorithm coupled with a data mining algorithm makes a statistical approach a rigorous method; it would also make an inherently intractable problem of state space exploration directed and restrained.

Another reason is that measures of interestingness [47], [58] for hardware systems can, to a large extent be identified and quantified. For example, in a digital circuit, a signal with a high fan-in may be considered important, and any assertion about its behavior interesting. As another example, if a certain output signal is interesting, its logic cone-of-influence [26] can be a means to direct the data mining algorithm to the relevant neighborhood. This type of steady and accurate high level guidance is atypical for data mining algorithms.

The existence of techniques that can extract domain knowledge from systems and guide the data mining is unique to computer systems. We have given examples of fan-in-based priority and cone-of-influence that can provide this knowledge. There are multiple such known techniques in hardware that can be used for analyzing the target system. We group all these techniques under the term *static analysis techniques*. Static analysis literally refers to techniques to reason with all possible behaviors of a system without executing the system. In hardware, we use the term static analysis to mean methods that analyze design structure/function (analogous to program syntax and semantics in software). Examples of structural methods include cone-of-influence [9], [23], [24] and localization reduction [41]. Formal verification can be considered a static analysis of the semantics of a model. Formal verification, therefore, is also a type of static analysis in our terminology. We distinguish between the two by calling one "lightweight" and the other "formal."

Data mining is very effective when localized, since it does not have to infer broad generalizations and has no capacity issues. Static analysis can make excellent generalizations, since it groups many concrete executions into abstractions of some kind, but has capacity issues when used by itself. This synergy between the two methods is used in GoldMine to automate the manual generation of assertions.

III. Preliminaries

In this section we provide some preliminary definitions necessary to understand our methodology.

Let M be our RTL design. A *target* is a variable in M for which we want to generate assertions. Variables in the *logic cone* of a target are those variables that can affect the value of the target [42], [9], [23], [24]. A *feature* is a variable that is used to predict the target's value. A GoldMine *candidate assertion* A_C is a linear temporal logic (LTL) [53] formula of the form $G(A \implies C)$, where both the antecedent A and the consequent B can be propositional or temporal. A proposition in this formula is a $(variable, value)$ pair. A propositional logic formula can have a conjunction, disjunction or negation. The temporal operators X, U and F can appear within a specified bound.

A *true assertion* A_T is a candidate assertion such that $M \vdash G(A_T)$, or the assertion holds globally on model M.

The *mining window length* is the duration of time cycles within which the generated assertions capture temporal behavior. It depends on the sequential depth of the target signal. The mining window length provides the bound for the generated GoldMine assertions.

Typical candidate assertions generated by GoldMine have an antecedent A of the form $A = a_0 \wedge X(a_1) \wedge XX(a_2) \wedge \ldots \wedge X^{=m}(a_m)$, and a consequent C of the form $C = X^{=n}(c)$, $n \geq m$. Here each a_i and c denote a conjunction of propositions, and $X^{=n}$ $(n > 0)$ is equivalent to a delay by n cycles ($XX \ldots X$ n-times).

In addition, assertions in some other forms can also be reduced to a set of equivalent assertions of this form. For example, the assertion $a : G((v_1 \vee \neg v_2) \wedge X(v_3) \Rightarrow XX(\neg v_4))$ can be reduced to a set of two assertions as follows.

$a_1 : G(v_1 \wedge X(v_3) \Rightarrow XX(\neg v_4))$

$a_2 : G(\neg v_2 \wedge X(v_3) \Rightarrow XX(\neg v_4))$

Our methodology is extensible to the U and F operators in LTL within a bounded time window. This is because bounded liveness properties are equivalent to safety properties. Our methodology cannot generate unbounded safety or liveness properties.

The GoldMine assertion format represents a fairly large subset of safety properties, which are of primary interest from a verification standpoint [10] details how the Power PC was verified using safety properties.

IV. GoldMine: Assertion Generation Methodology

We present **GoldMine**, a methodology to automatically generate assertions using data mining and static analysis. Figure 1 depicts the five main components of GoldMine.

A. Data Generator

The Data Generator uses a test bench to generate simulation traces for a given design. The Data Generator generates a random test bench if a directed or constrained random test bench is unavailable. The random test bench assigns a random value to each input in each simulation cycle. We have found that simulating the random test bench for 10000 cycles is sufficient for most designs.

B. Static Analyzer

The Static Analyzer extracts information from the design that can be used by A-Miner. Such information can improve both the algorithmic performance of A-Miner and the quality of the generated assertions.

Currently, the Static Analyzer uses the classic cone of influence [9], [23], [24] to select a set of feature variables for a given target variable. The classic cone of influence is a reduction technique employed by most unbounded model checking algorithms. The classic cone of influence is a special case of the localization reduction in [42]. The classic cone of influence uses an RTL dependency graph to transitively compute which variables can affect the consequent of an assertion. Since data mining relies on statistical methods to infer relationships, one variable may be correlated with another that does not affect its value. The classic cone of influence eliminates this problem by restricting the search space to variables that can affect the value of the target variable.

C. A-Miner

A-Miner uses a data mining algorithm to search the simulation traces for correlations between the feature variables and target variable. Statistics such as *support* and *confidence* can help determine the validity of a correlation. Consider the rule $R = A \implies B$. The support of R refers to the fraction of examples in the data space that satisfy A. The confidence of R refers to the fraction of examples in the data space that satisfy B given that A is satisfied. The confidence of R can be considered an estimate of the conditional probability $P(B|A)$. For example, the confidence of R is 100% if B is satisfied whenever A is satisfied in the data space. If a rule has 100 percent confidence, then A and B always occur simultaneously in the dataset. If this rule has high support, then it means A occurs frequently in the dataset.

A-Miner requires that candidate assertions have 100% confidence. If the confidence of a candidate assertion is less than 100%, then its antecedent correlates with conflicting target variable values. Such an assertion cannot be true since its antecedent implies that the target variable is both true and false.

By default, the Static Analyzer selects the output variables in a design as target variables. However, since A-Miner is unaware of each variable's type, any variable can be selected as a target variable. Consequently, A-Miner is not restricted to generating assertions that specify input/output relationships.

Although many data mining algorithms can be implemented in A-Miner, we present a decision tree based rule induction algorithm in this work.

D. A-Miner: Decision Tree Algorithm

A-Miner uses a decision tree algorithm [54], [55], [15] to generate assertions. Decision tree algorithms are rule induction data mining algorithms. These algorithms recursively partition the data space by assigning values to feature variables until the value of the target variable is consistent within a data subspace. Decision tree algorithms use decision trees to represent the dataset. A decision tree is a multary tree that consists of branch and leaf nodes. Each node represents a data subspace while each edge represents a partition in the data subspace of its parent.

The *gain* of a feature variable at a particular node is the reduction in error between that node and the child nodes produced when that feature variable is used to partition the data space. The *error* of a node is the mean absolute deviation of the value of the target variable from the mean of that node. The *mean* of a node is the mean value of the target variable within the data subspace represented by that node. Decision tree algorithms compute the gain of each feature variable to determine which will best partition the data space.

Since A-Miner generates bit-level assertions, each node in the decision tree will have at most two child nodes. A-Miner generates a candidate assertion by walking a path in the tree from a leaf node to the root node. In each such path, A-Miner uses each feature variable assignment at each edge to define a proposition in the assertion's antecedent. A-Miner uses the mean of the leaf node to predict the value of the target variable in the assertion's consequent.

Algorithm 1 Decision Tree Algorithm

1: **procedure** $decision_tree(\mathbf{V}, \mathbf{E}, \mathbf{P})$
2: $m \leftarrow mean(v_t, \mathbf{E})$
3: $e \leftarrow error(v_t, \mathbf{E})$
4: **if** $e = 0$ **then**
5: $\mathbf{A} \leftarrow \mathbf{A} \cup (\mathbf{P} \implies (v_t, m))$
6: **return**
7: **end if**
8: $v_{best} = \emptyset$
9: $g_{best} = -\infty$
10: **for all** $v \in \mathbf{V}$ **do**
11: $g \leftarrow e - error(v_t, E_{v=0}) - error(v_t, E_{v=1})$
12: **if** $g > g_{best}$ **then**
13: $v_{best} \leftarrow v$
14: $g_{best} \leftarrow g$
15: **end if**
16: **end for**
17: $decision_tree(V \setminus v_{best}, E_{v_{best}=0}, P \cup (v_{best}, 0))$
18: $decision_tree(V \setminus v_{best}, E_{v_{best}=1}, P \cup (v_{best}, 1))$
19: **end procedure**

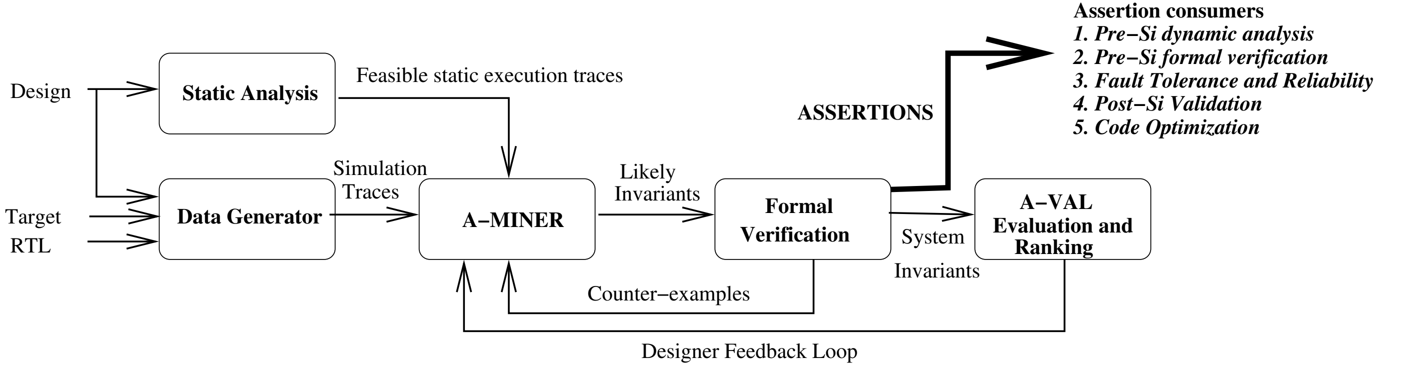

Fig. 1: The GoldMine methodology

A-Miner uses algorithm 1 to generate assertions. The Decision Tree algorithm expects the sets **V**, **E**, and **P** as inputs, which are defined as follows. Let $\mathbf{B} = \{0, 1\}$ denote the set of Boolean values and let $b \in \mathbf{B}$ denote an arbitrary Boolean value. Let **V** denote a set of variables and let $v_t \in \mathbf{V}$ denote the target variable. Let **E** denote a set of Boolean vectors, let $e_i = (v_0 = b, v_1 = b, \ldots, v_t = b) \in \mathbf{E}$ denote a vector that assigns a Boolean value to each variable in **V**, and let e_{ij} denote the value of variable j in vector i. Let **P** denote a set of propositions and let $p_i = (v_j, b) \in \mathbf{P}$ denote a variable-value pair that assigns a Boolean value to variable v_j. Finally, let **A** denote the set of assertions generated by the Decision Tree algorithm.

The Decision Tree algorithm requires the functions $mean$ and $error$, which are defined as follows. The $mean(v_i, \mathbf{E})$ function computes the mean value of v_i using the vectors in **E**. The $error(v_i, \mathbf{E})$ function computes the absolute deviation of v_i from its mean value.

The Decision Tree algorithm begins by computing the mean and error values of v_t in **E**, the current partition of the data space. If the error of $v_t = 0$, then the assertion $\mathbf{P} \implies (v_t, mean(v_t, \mathbf{E}))$ is added to **A** and the algorithm terminates. In such cases, the value of v_t does not change in **E**. Therefore, the value of $mean(v_t, \mathbf{E})$ will be equal to the value of v_t in **E**.

If $error(v_t, \mathbf{E}) \neq 0$, then the algorithm uses the variable $v_{best} \in \mathbf{V}$ with the highest gain to partition the data space. The algorithm makes two recursive calls. Both the first and second call remove v_{best} from **V**. The first call removes vectors from **E** where $v_i = 1$, and adds the proposition $(v_i, 0)$ to **P**, while the second call removes vectors from **E** where $v_i = 0$, and adds the proposition $(v_i, 1)$ to **P**. In other words, these calls partition the data space such that one partition includes vectors where $v_i = 0$, and the other partition includes vectors where $v_i = 1$.

E. Formal Verifier

The Formal Verifier uses Cadence Incisive Formal Verifier to verify the candidate assertions generated by A-Miner. The Formal Verifier generates Verilog code to check the candidate assertions at the appropriate edge of each clock cycle. During verification, we constrain the reset signal to prevent the design from resetting. Assertions that pass formal verification are reported as system invariants. In [46], we detail a methodology that enables A-Miner to refine assertions that fail formal verification using their counterexample simulation traces. Although GoldMine attempts to minimize human effort in the assertion generation process, we require human intervention to differentiate a spurious candidate assertion that fails formal verification from a genuine system invariant that fails formal verification because of an implementation bug.

F. Evaluation and Ranking

Assertion generation has been a completely manual process thus far in the system design cycle. Therefore, evaluation of generated assertions is a crucial aspect of GoldMine.

There are several ways to evaluate A-Miner's performance. One basic metric is *hit rate*. The hit rate of a set of assertions is the fraction of true assertions within the set. In addition, we consider the *output hit rate*. The output hit rate of a set of assertions is the fraction of outputs for which the set contains at least one true assertion.

The *coverage* of an assertion refers to the fraction of design functionality that is covered when the antecedent of the assertion is triggered. Though methodologies exist to compute the RTL statement coverage of an assertion [37], [39], [21], [19], [20], [33], [6], [64], none have been implemented practically. Consequently, we use *input space coverage* to evaluate the coverage of an assertion. Consider a truth table that computes the value of the target variable as a function of the feature variables. We say that an assertion covers an entry in such a truth table if the values of the feature variables in that entry satisfy the antecedent of the assertion. The input space coverage of an assertion refers to the percentage of truth table entries covered by that assertion.

Input space coverage is both intuitive and easy to compute. We can compute the fraction of input space that an assertion covers without having to know the entire input space. We can compute the input space coverage of an assertion using the formula $\frac{1}{2^{|P|}}$, where $|P|$ denotes the number of propositions in the antecedent of the assertion. If we consider the truth table for a specific output, each entry that satisfies the antecedent of an assertion is said to be covered by that assertion. For example, the input space coverage of the assertion $(a = 1 \& b = 1 \implies c = 1)$ is 25 percent because 25 percent of the truth table entries contain $a = 1, b = 1$. Intuitively, if a set of assertions covers each entry in the truth table of an output, then the output has high coverage with respect to those assertions. This metric is simple to calculate since we can determine the percentage of the input space that an assertion covers without knowing all input combinations. Consequently, an assertion that has many propositions in its antecedent will have lower

input space coverage than one with few propositions in its antecedent.

G. Temporal Assertions

Most interesting assertions span more than one clock cycle. Such assertions can be found without having to change the data mining algorithm. When the Data Generator generates a simulation trace, each sample in the trace defines the state of the system at the current time, t. The user specifies the maximum length of temporal assertions as l. In each example, we wish to represent the signals at the previous times $t-1, t-2, ..., t-l$. For a given sample at time t, the value of each signal at time $t-i$ is determined by checking its value in the sample at time $t-i$. The data mining algorithm proceeds normally, considering each temporal signal as a new addition to the simulation trace. For example, consider a protocol that asserts $ack = 1$ two cycles after $req = 1$. Table I shows the simulation data for such a module.

time	req	ack
0	0	0
1	1	0
2	0	0
3	0	1
4	0	0

TABLE I: The simulation data for a req/ack protocol

Table II shows the transformed simulation data when the maximum temporal length $l = 2$. We discard the data in cycle 0 and cycle 1 since there is no information for $[t-1]$ in cycle 0 or $[t-2]$ in cycle 0 or 1.

time	req[t]	ack[t]	req[t-1]	ack[t-1]	req[t-2]	ack[t-2]
0	0	0	x	x	x	x
1	1	0	0	0	x	x
2	0	0	1	0	0	0
3	0	1	0	0	1	0
4	0	0	0	1	0	0

TABLE II: The previous cycle information is added to enable temporal assertion mining

The new dataset can be used with the original data mining algorithm. In cycle 3, there is a clear relationship between $ack[t]$ and $req[t-2]$, which produces the assertion `req == 1 |-> ##2 ack == 1`. The generated assertion expresses the expected behavior for the protocol.

H. Example

```
always @ *
  if (int.valid &&
      int.has_dreg)
    wb_valid0 = 1;
  else
    wb_valid0 = 0;

always @ *
  int.L1_hit = int.has_dreg
```

int.valid	int.L1_hit	int.has_dreg	wb_valid0
0	0	0	0
0	1	1	0
1	0	0	0
1	1	1	1

Fig. 2: Example RTL and Data Generator traces

Figure 2 depicts a fragment of RTL source code from the Rigel processor. From the code it is apparent that if there is a valid integer writeback (variable $int.valid$) and a register is available (variable $int.has_dreg$), then there is a valid writeback on port 0 (variable wb_valid0). The described event always updates the L1 cache hit rate. We use this code to illustrate the GoldMine assertion generation process from algorithm 1. Figure 2 shows a simulation trace produced by the Data Generator.

The decision tree produced by algorithm 1 in this particular example is shown in figure 3. In line 1 of algorithm 1, $mean(wb_valid0, \mathbf{E}) = 0.25$. Line 2 determines the error of wb_valid0. To determine the error of wb_valid0, the mean of wb_valid0 is first subtracted from each its individual values. Next, the absolute value of these differences are summed. Finally, the sum is divided by the total number of samples. In this example, $error(wb_valid0, \mathbf{E}) = 0.375$. Because $error(wb_valid0, \mathbf{E}) \neq 0$, the algorithm proceeds to line 8.

Lines 8 through 16 select the feature variable that will yield the greatest reduction in error in the target variable. The algorithm computes the error reduction of $int.valid$ on line 11. When $int.valid = 0$, the mean and error of wb_valid0 are both 0. When $int.valid = 1$, the mean and error of wb_valid0 are both 0.5. Therefore the error reduction of partitioning the data space using $int.valid$ is -0.125. Note that partitioning using $int.valid$ will produce a child node that has error equal to 0. Such partitions are preferred because they will generate a candidate assertion. Consequently, the algorithm explicitly maximizes the error reduction of splitting on $int.valid$ to ∞.

On line 12, the computed error reduction is greater than the current best error reduction. Thus, line 13 assigns $int.valid$ to v_{best} and line 14 updates the best error reduction. Next, the algorithm computes the error reduction of partitioning using $int.L1_hit$ and $int.has_dreg$. Because both error reductions are equivalent to that of $int.valid$, the best error reduction is not updated on lines 12 through 15. Lines 17 and 18 partition the data space with respect to $int.valid$ and recurse the algorithm.

In the recursive call where $int.valid = 0$, $error(wb_valid0, \mathbf{E}) = 0$ as well. Therefore, the algorithm creates a leaf node on line 5 and generates the candidate assertion `A0: (int.valid == 0) |-> wb_valid == 0`. The recursive call terminates on line 6.

In the recursive call where $int.valid = 1$, $error(wb_valid0, \mathbf{E}) > 0$. Therefore, the algorithm must again determine the feature variable that will yield the greatest reduction in error in the target variable. The error reduction of partitioning using $int.L1_hit$ and $int.has_dreg$ are equivalent. As a result, the algorithm assigns $int.L1_hit$ to v_{best} since it occurs first in the dataset.

In both recursive calls after partitioning using $int.L1_hit$, $error(wb_valid0, \mathbf{E}) = 0$. Therefore, two leaf nodes and their respective candidate assertions are generated. The recursive call where $int.L1_hit = 0$ generates the candidate assertion `A1: (int.valid == 1 && int.L1_hit == 0) |-> wb_valid0 == 0`, while the recursive call where $int.L1_hit = 1$ generates the candidate assertion `A2: (int.valid == 1 && int.L1_hit == 1) |-> wb_valid0 == 1`.

The candidate assertions $A0$, $A1$, and $A2$ are passed to the Formal Verifier. $A0$ and $A1$ pass, but $A2$ fails due to the false causality established by the simulation data. Two out of the

three candidate assertions pass formal verification. Therefore, the hit rate of the assertion set is 2/3.

The Static Analyzer can eliminate the false causality established by the simulation data. The Static Analyzer determines the portion of the design that is causal to $int.valid$, and provides a list of feature variables to the decision tree that excludes $int.L1_hit$. The corresponding decision tree is shown in Figure 3. The algorithm generates the candidate assertions $A0$, $A1 : \{(int.valid, 1), (int.has_dreg, 0)\} \implies (wb_valid0, 0)$, and $A2 : \{(int.valid, 1), (int.has_dreg, 1)\} \implies (wb_valid0, 1)$. The Formal Verifier passes these candidate assertions. Consequently, the hit rate of the assertion set is 1.

Mining temporal assertions has three disadvantages. First, there must be a user-specified bound on the maximum number of cycles in an assertion, l. Second, as l increases, the runtime of the algorithm increases since its search space has also increased. Finally, as l increases, the quality of the generated assertions can decrease since the feature space can grow so large that choosing the optimal feature variable to partition the data space becomes difficult. These disadvantages can be mitigated by using background knowledge of the design to choose a good maximum cycle length, l, or by testing several different values for l to optimize results.

I. Limitations

The dynamic nature of GoldMine limits the types of assertions it can generate. GoldMine is limited by the quality of the data produced by the Data Generator. If the Data Generator does not simulate interesting design behavior, then A-Miner will not find meaningful assertions. However, since the Data Generator uses an unconstrained random test bench, A-Miner frequently discovers subtle assertions. In addition, if A-Miner uses feedback from the Formal Verifier, it can improve the quality of the generated assertions.

GoldMine cannot generate unbounded temporal assertions since the simulation traces cannot convey such information. However, we have found that GoldMine generates assertions with sufficiently high coverage. In addition, bounding the assertions in time allows the algorithm to scale, unlike many static and formal methodologies.

V. Related Work

In this section, we discuss work related to our methodology. Assertion generation though static analysis of source code or a model has been studied in the context of deductive program verification [16], [48], [8] since the seventies. The deduction of the "weakest liberal precondition" from the loop body can quickly become very complex. Static analysis techniques have been used to learn invariants for assisting software verification [61], [50]. Dynamic analysis [5], [49] as well as data mining [18] have been used in software to determine system invariants.

Previous work [34], [66], [51] has used static analysis for hardware assertion generation. IODINE [32] generates detailed, low-level dynamic invariants for hardware designs. Unlike GoldMine, IODINE does not use data mining techniques. Instead, IODINE analyzes dynamic program behavior with respect to standard property templates such as one-hot encoding or mutex. The methodology in [56] uses dynamic simulation trace data to generate assertions, but does not use data mining. Instead, they try to generalize design behavior based on a set of simulation traces. Commercial tools such as [38] can only capture simple, pre-defined invariants. To the best of our knowledge, ours [63] was the first methodology to generate assertions for hardware using a combination of data mining and static analysis methods.

Inferno [25] uses simulation traces to extract the semantic protocol from a communication interface RTL design. Inferno infers a set of transaction diagrams which can be used to generate a set of assertions. Since inferno only generates transactional assertions, it might not generate meaningful assertions for designs with high temporal depth. Consequently, inferno assertions might complicate the search bugs in such designs. In addition, Inferno is limited to designs that implement a communication protocol, which comprise small subset of the RTL design space.

In [17], Chang et al use sequential pattern mining to infer causal relationships between frequently occurring sequences of input and output events. Since their methodology only seeks relationships between inputs and outputs, it might not generate meaningful assertions for designs with high temporal depth. In addition, since their methodology enumerates each unique input and output event, it might not scale for large designs. Since Chang et al do not present any generated assertions or quantitative measures of functional coverage, we cannot properly compare our methodologies.

In [22], Chung et al use symbolic simulation to extract properties for an RTL design based on test bench constraints. The properties can be used to optimize the design under test. Since their methodology extracts properties from a constrained test bench, the properties might reflect a bug in the test bench or overlook untested behavior. In addition, the properties are extracted using simple templates and are verified using bounded formal verification, which might limit their value. Since Chung et al use the properties for optimization purposes, they do not provide results that we can use to compare our methodologies.

In [44], we apply the GoldMine methodology at the System C level. We use sequential pattern mining to infer sequential transaction-level assertions from frequently occurring event sequences. The purpose of these assertions is different from that of RTL assertions. Hence, a detailed discussion is beyond the scope of this paper.

Although commercial methodologies to automatically generate RTL assertions including NextOp's BugScope [3] and Jasper's ActiveProp [4] exist, they are not available for comparison.

VI. Case Study: Rigel RTL

We first present the results of applying GoldMine to the 1000+ core Rigel RTL design. Our intention is to use assertions from GoldMine to provide a regression test suite for the Rigel RTL that is in the later stages of its evolution. We generated assertions for three principal modules in Rigel: the writeback stage, the decode stage and the fetch stage. The writeback stage is a combinational module with interesting propositional properties. The decode and fetch stages are sequential modules with many interesting temporal properties.

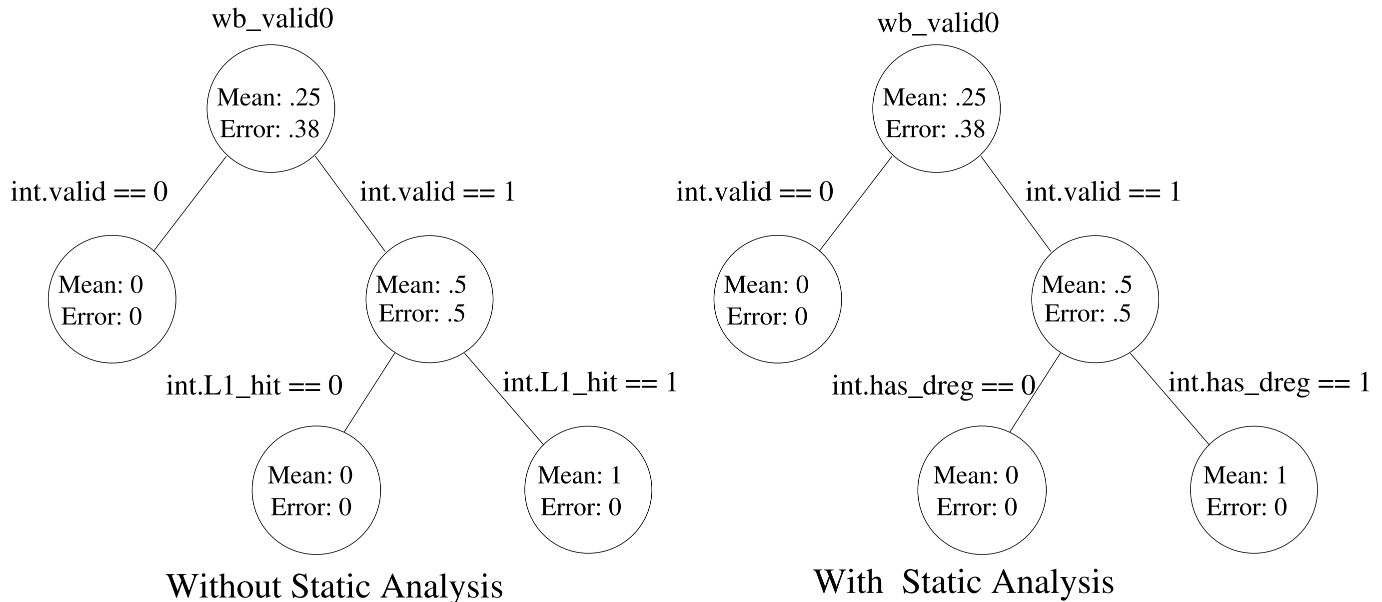

Fig. 3: Decision trees generated by Algorithm 1 with and without the Static Analyzer

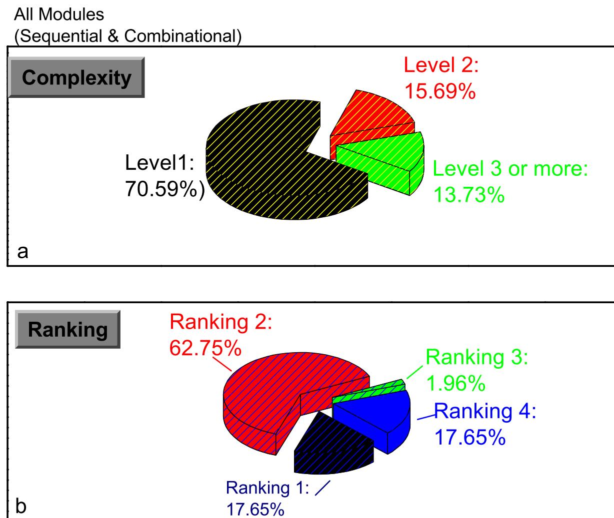

Fig. 4: GoldMine assertion complexity and designer rankings

A. Subjective Ranking of Assertions by a Designer

We performed experiments to help evaluate GoldMine assertions. We performed an extensive designer ranking session for every phase of assertion generation of each module. Also, since the Rigel RTL does not have manual target assertions to compare against, we performed a subjective, but intensive evaluation strategy. Rankings were from 1 to 4, defined as follows.

1) Trivial assertion that the designer would not write
2) Designer would write the assertion
3) Designer would write, captures subtle design intent
4) Complex assertion that designer would not write

The results presented in Figure 4 show the distribution of these ranks for a sample of representative assertions for all the modules. Most assertions in this analysis are ranked 2. The writeback module has some assertions ranked 3. The absence of assertions ranked 3 in the sequential modules, according to the designers, is due to the fact that intra module behavior is not complicated enough to have many subtle relationships.

An assertion ranked 1 expresses the property *if the halt signal in the integer, floating point, and memory units is low, then the halt signal is low.* In the RTL, the halt signal is a logical or between the integer, floating point, and memory units. GoldMine found a true, but over-constrained assertion. The designers would not have written such an assertion and hence ranked it 1.

Consider the following RTL code:

```
decode2mem.valid <=
    valid_mem &&
    !issue_halt &&
    !branch_mispredict &&
    fetch2decode.valid &&
    !follows_vld_branch;
```

An assertion ranked 2 expresses the property *if branch_mispredict is high, then one cycle later decode2memvalid will be high.*

An assertion ranked 3 expresses the property *if an integer unit does not request the first port, and the floating point unit does not request the second port, then the second port remains unused.*

B. Complex Assertions in GoldMine

Despite the small size of the modules, GoldMine generated rank 4 assertions. In other words, GoldMine generated assertions that capture complex relationships in the design. This is an advantage of mechanically derived assertions: they can capture unintentional, but true relationships that can be excellent counter checks and can be brought to the designer's attention. We assessed complexity by the number of levels (depth) of the design captured by assertions. In a few cases, the **assertions capture temporal relationships that are more than 6 logic levels deep in the design.** This provides a different perspective on the RTL, outside of the expectation, but may provide avenues for optimizing or analyzing the RTL.

For example, consider the following RTL code:

```
if (choice_mem)
    decode_packet <= decode_packet1;
```

An assertion ranked 4 expresses the property: *if reset is low, issue0 is low and decode_packet_dreg is low, and one cycle later instr0_issued is low, then decode_packet_dreg is low*. This assertion relates a single field in the *decode_packet* variable to *reset* and *instr0_issued*, both of which are related to *choice_mem* when the code is traversed beyond 6 levels of (sequential) logic. Such a relationship would have been extremely hard to decipher through static analysis and code traversal. To the best of our knowledge, there is no state-of-the-art tool/technique that can claim to decipher such complex assertions. Figure 4 shows the distribution of assertions with respect to complexity.

C. Outputs Covered by GoldMine

Module	Percentage of Outputs Covered
Decode Stage	46.76%
Fetch Stage	35.71%
Writeback Stage	87.50%

TABLE III: Percentage of Rigel outputs covered by GoldMine

Table III shows the percentage of outputs per module for which assertions were generated by GoldMine. Although candidate assertions were generated for all the module outputs, a subset of the assertions were passed by formal verification engine. Figure 5 shows the probability distribution of true assertions per output. At 50%, there are approximately 4 to 5 unique assertions per output in the decode module. Although we are not able to get a precise notion of path coverage per output signal, the number of unique assertions per output is indicative of high path coverage.

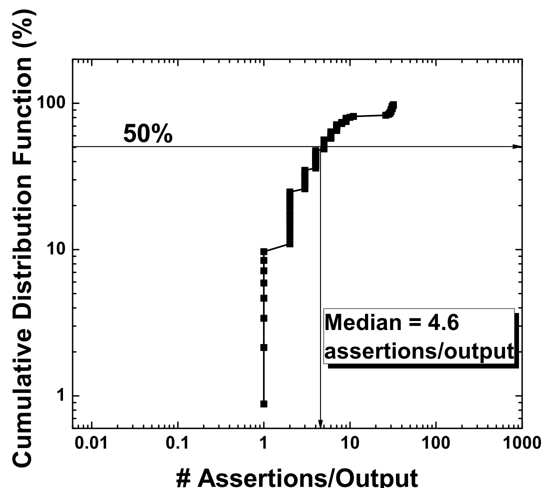

Fig. 5: Distribution of unique assertions per output in all modules

D. The Acid Test: Regression Test Experiments

As a final evaluation of the entire regression suite of GoldMine assertions, we appended them in the RTL and ran a new set of directed Rigel tests.

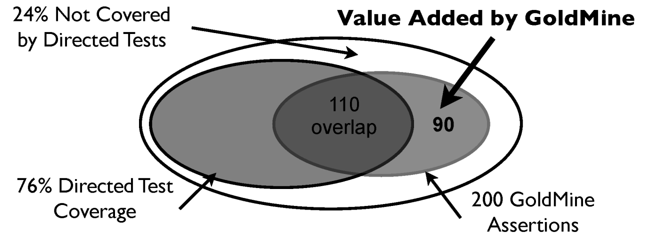

Fig. 6: The added coverage of design behavior through GoldMine assertions for the writeback module

We analyze the results for the writeback module, since the fetch and decode are very similar. We used Synopsys VCS with RTL conditional coverage to procure the coverage of the directed tests. We used the conditional coverage metric since unique assertions in GoldMine pertain to different paths. This metric is meaningful since it considers the individual path conditions required to generate an output.

The writeback module directed tests achieved 76% conditional coverage, while GoldMine's random tests achieved 100% conditional coverage and generated 200 unique assertions. When the GoldMine assertions were included in the directed test runs, 110 (55%) of the assertions were triggered[2] by the directed tests. Therefore, 90 assertions, or 45%, refer to untested design behavior with respect to the directed tests. Figure 6 shows the overlap of assertions with directed tests. GoldMine provides significant coverage of the unexplored regions of the design.

The assertions that overlap with the directed tests can be used for static checking, formal verification, etc. The remaining assertions can be used to improve the quality of the directed tests. They can be used as regression checks as the regression test suite evolves. It is probable that the manual assertion generation process would eventually get to this point after multiple iterations. In contrast, GoldMine, a mechanical assertion generator, could explore the design space far beyond the human generated tests. The designers of Rigel have deemed that GoldMine "covers a wide design space much earlier in the design cycle than typically achievable" [62].

VII. The Power of GoldMine: OpenSparc T2

In this section, we evaluate the performance of GoldMine using Sun Microsystem's OpenSparc T2 CPU [1]. For our experiments, we used GoldMine to generate assertions for the memory management unit (MMU) of the core. The MMU queries the translation lookahead buffer (TLB) for the data and instruction caches. If a TLB miss occurs, the MMU walks the page table. The MMU has 59 input variable vectors, 54 output variable vectors, and 313 internal variable vectors. We selected 16 outputs for which we could generate a significant number of samples using random input vectors. In the absence of the Static Analyzer, the search space of A-Miner is nearly 3000 bits. We generated assertions using both 10,000 and one million cycles of simulation data.

A. Evaluation of True Assertion Success Rate

The first experiment evaluates the percentage of outputs for which GoldMine generates at least one true assertion. We

[2]An assertion is triggered if the antecedent is satisfied.

present the results for several configurations of GoldMine. The first and second configurations use 10,000 cycles of simulation data while the third and fourth configurations use one million cycles of simulation data. The first and third configurations exclude the static analysis engine from GoldMine while the second and fourth do not. Figure 7 shows that the number of outputs with at least one true assertion increases with both the number of cycles of simulation data and the inclusion of the static analysis engine. It is worth noting that including the static analysis engine is significantly more effective than increasing the number of cycles of simulation data.

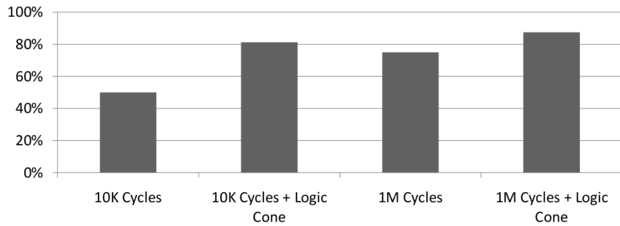

Fig. 7: Percentage of outputs for which at least one true assertion was generated

B. Evaluation of Formal Verification Effort

When the formal verification engine aborts proving an assertion, GoldMine will discard it. This experiment examines the percentage of assertions that were proven true, proven false, and aborted by the formal verification engine for several different configurations of GoldMine. All configurations use 10,000 cycles of simulation data, logic cone information, and a maximum decision tree depth of 10. The configurations vary the sequential length of the mining window from 1 to 3 respectively. Each configuration alternates the effort of the formal verification engine between low and high. Table IV shows that in general, increasing the sequential length of the mining window will increase the percentage of explored assertions. When effort is set to low, the formal verification engine will spend at most 10 seconds proving each assertion. When effort is set to high, the formal verification engine will spend at most 5 minutes proving each assertion. Consequently, Table IV shows that 5 minutes is a sufficient amount of time to prove most GoldMine assertions.

C. Evaluation of Input Space Coverage

This experiment evaluates the input space coverage of GoldMine assertions. Figure 8 shows that GoldMine can produce assertions with high input space coverage. In addition, the input space coverage increases with the number of cycles of simulation data. However, figure 9 shows that GoldMine must generate many additional assertions to achieve high input space coverage.

D. Comparing the Generated Assertions with the OpenSparc Specification

This section provides a qualitative analysis of GoldMine assertions for the OpenSparc L2 cache controller (L2T). We generate assertions for the L2 pipeline stall signal. We describe the behavior of the L2 pipeline stall signal as follows. The input queue (IQ) in L2T queues L2 cache requests. When IQ is full, L2T uses the signal $l2t_pcx_stall_pq$ to stall the

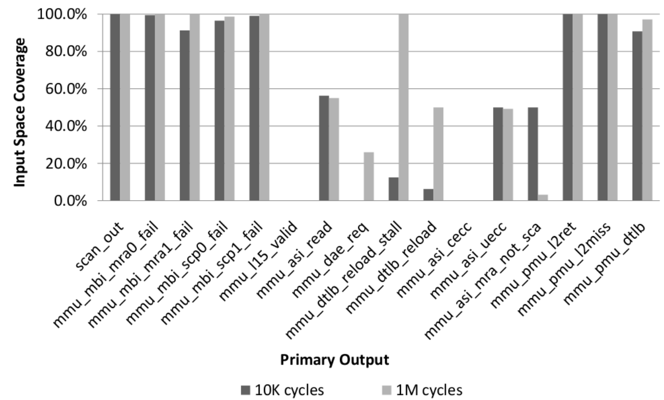

Fig. 8: Input space coverage for MMU

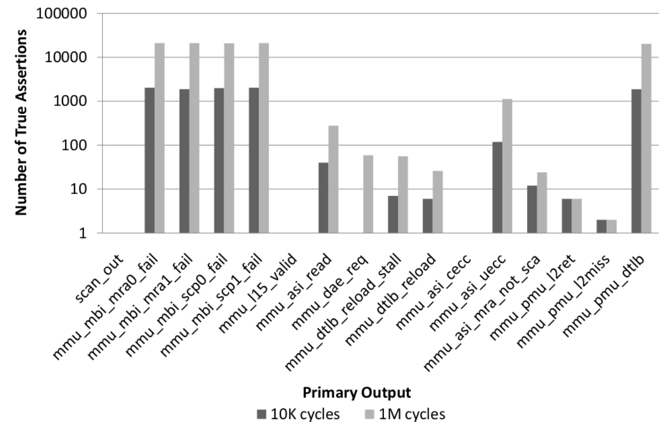

Fig. 9: Total number of true assertions generated for MMU - Logarithmic

pipeline. The data ready signal $pcx_l2t_data_rdy_px2_d1$ indicates when a request is being added to IQ and the input queue select signal $arb_iqsel_px2_d1$ indicates when a request can be removed from IQ. The input signals $pcx_l2t_data_rdy_px1$ and arb_iqsel_px2 generate the data ready and input queue select signals after several clock cycles. Table V shows the temporal relationship between $pcx_l2t_data_rdy_px1$ and $pcx_l2t_data_rdy_px2_d1$, and arb_iqsel_px2 and $arb_iqsel_px2_d1$.

GoldMine generates the following candidate assertions:

```
GM1: (l2t_pcx_stall_pq == 1) |=> ##1
      l2t_pcx_stall_pq == 1;

GM2: (pcx_l2t_data_rdy_px1 == 1 ##2
      arb_iqsel_px2 == 0 &&
      l2t_pcx_stall_pq == 0) |=> ##1
      l2t_pcx_stall_pq == 0;

GM3: (pcx_l2t_data_rdy_px1 == 1 ##2
      arb_iqsel_px2 == 1 &&
      l2t_pcx_stall_pq == 0) |=> ##1
      l2t_pcx_stall_pq == 0;

GM4: (pcx_l2t_data_rdy_px1 == 0 ##2
      l2t_pcx_stall_pq == 0) |=> ##1
      l2t_pcx_stall_pq == 0;
```

Assertion `GM1` states that if there is a stall, then one cycle later there will be a stall. If IQ is full and one cycle later L2T

Signal	Effort	1 Cycle			2 Cycles			3 Cycles		
		%P	%F	%E	%P	%F	%E	%P	%F	%E
l2ret	low	100	0	0	100	0	0	100	0	0
	high	100	0	0	100	0	0	100	0	0
l2miss	low	100	0	0	100	0	0	100	0	0
	high	100	0	0	100	0	0	100	0	0
dtlb	low	0	37	63	0	52	48	0	0	100
	high	0	100	0	0	100	0	0	100	0
scp0_fail	low	0	2	98	0	0	100	0	0	100
	high	0	100	0	0	100	0	0	100	0
scp1_fail	low	0	13	87	0	8	92	0	0	100
	high	0	100	0	0	100	0	0	100	0

TABLE IV: Percentage of assertions that were passed (%P), failed (%F), and explored (%E) by the formal verification engine.

Signal	Cycle t-3	Cycle t-2	Cycle t-1	Cycle t
IQ Select	–	–	arb_iqsel_px2	arb_iqsel_px2_d1
Data Ready	data_rdy_px1	data_rdy_px1_fn	data_rdy_px2_d1	data_rdy_px2_d1

TABLE V: This table shows the temporal relationship between signals.

serves a request, then IQ will no longer be full. Therefore, the simulation will contain many traces where a stall is followed by a stall. Consequently, `GM1` does not express a causal relationship and is false.

Assertion `GM2` states that if the pipeline is not stalling and a new request is added to IQ, then the pipeline will not stall. If the pipeline is not stalling, then IQ is not full. Therefore, the size of the IQ must increase since the data ready signal is true and the IQ select signal is false. Assertion `GM2` will be true most of the time. However, if IQ has one empty slot, then a new request will fill IQ and stall the pipeline. Therefore, `GM2` is false.

Assertion `GM3` states that if the pipeline is not stalling and the size of IQ does not change, then the pipeline will not stall. Since the data ready signal is true, the queue size remains the same. Therefore a request is added. However, since the IQ select signal is true, a request is also processed. Consequently, if IQ is not full and its size remains the same, then it will not stall. Therefore, assertion `GM3` is true.

Assertion `GM4` states that if the pipeline is not stalling and there are no pending IQ requests, then the pipeline will not stall. Since the data ready signal is false, the queue size must either decrease or remain the same. Therefore, IQ cannot be full and assertion `GM4` is true. GoldMine can generate assertions that describe complex design behavior. In addition, GoldMine can learn such behavior quickly and efficiently.

E. Evaluation of Runtime and Memory Requirements

This experiment evaluates the performance of GoldMine. We compare the runtime and memory use of GoldMine when it generates assertions for the Rigel and OpenSparc. Table VI shows various statistics for these modules.

Module	Inputs	Outputs	Area
Rigel - Decode Stage	2195	79	32735
Rigel - Fetch Stage	458	6	4165
Rigel - Writeback Stage	963	3	269
OpenSparc - MMU	3393	16	66395

TABLE VI: The characteristics of each tested module

We first discuss the runtime of GoldMine. We used a 2.66 gigahertz Intel Core 2 Quad CPU with 4 gigabytes of memory to measure the performance of GoldMine. Figure 10 shows the runtime of GoldMine without the formal verification engine. We can see that GoldMine generates candidate assertions quickly. GoldMine runs in just minutes for both 10,000 and one million cycles of simulation traces.

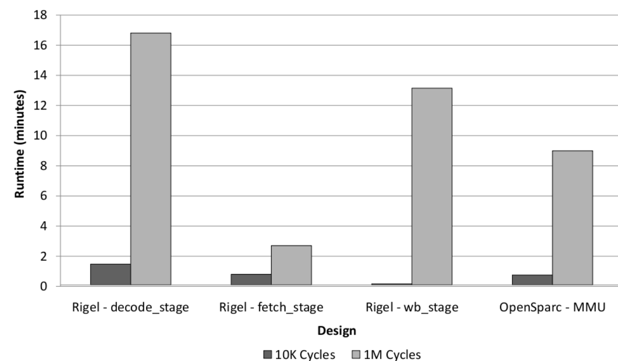

Fig. 10: GoldMine runtime without formal verification

We evaluate the runtime of GoldMine when it includes the formal verification engine. We used a cluster of four six-core AMD Opteron 8435 CPUs and multi-threaded the formal verification engine. Figure 11 shows the runtime of GoldMine when it includes the formal verification engine. As expected, the runtime of GoldMine is much higher. However, GoldMine still completes in a reasonable amount of time. Even for the complex OpenSparc MMU module, the 10,000 cycle test completes in only one hour and the 1,000,000 cycle test completes in just over 2 hours.

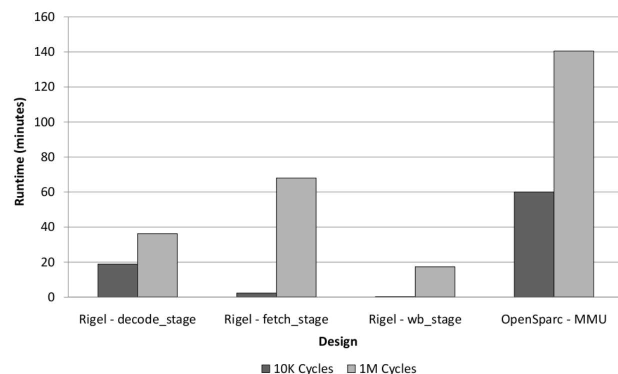

Fig. 11: GoldMine runtime; formal verification enabled

Finally, we evaluate the maximum memory use of GoldMine. For this experiment, we used a 2.66 gigahertz Intel Core 2 Quad CPU with 4 gigabytes of memory. We disabled

the formal verification engine since it does not affect the memory use of GoldMine. Figure 12 shows the memory use of GoldMine. We can see that in general, GoldMine uses a reasonable amount of memory. In the worst case, GoldMine uses less than 1 gigabyte of memory. The depth of the decision tree is a function of the number of signals in the design. Therefore, the memory use of GoldMine will be most affected by the number of signals in the design.

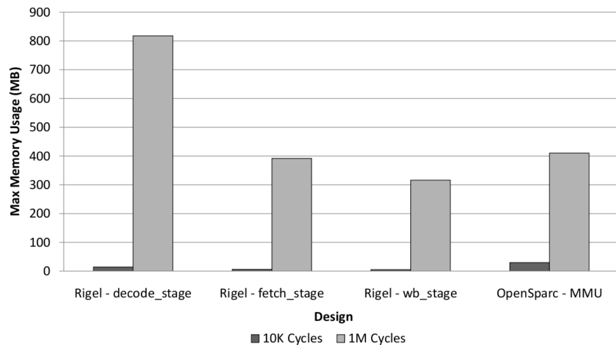

Fig. 12: GoldMine maximum memory usage

VIII. Discussion and Analysis of the Experimental Results

A. Reducing the Number of Generated Assertions

Since GoldMine is a mechanical process that generates assertions at the bit level, we encounter the issue of numerous assertions. For example, consider the RTL pictured in figure 13.

```
 1 module one_hot(a, b, state);
 2
 3 parameter active = 2'b01;
 4 parameter idle = 2'b10;
 5
 6 input [1:0] a, b;
 7 output reg [1:0] state;
 8
 9 always @ (a or b)
10     if (a > b)
11         state = active;
12     else
13         state = idle;
14 endmodule
```

Fig. 13: Example RTL

Let us assume that GoldMine generates the following assertions for the signal state:

```
GM1: (a[0] == 1 && b[0] == 0) |->
     state[0] == 1

GM2: (a[0] == 1 && b[0] == 0) |->
     state[1] == 0

GM3: (a[1] == 1 && b[1] == 0) |->
     state[0] == 1

GM4: (a[1] == 1 && b[1] == 0) |->
     state[1] == 0
```

Conceptually, these assertions capture the behavior of the statements in lines 10 and 11 in the example RTL. Although these assertions are correct and capture real behavior of the design, they are repetitive from a readability perspective. This would be acceptable when the assertions are used for regression testing. However, if a human is reading these assertions for immediate use or hints, it is better to generate an assertion like:

```
GM1: (a > b) |-> state == active
```

The current version of GoldMine addresses these issues with a number of techniques. We allow the user to specify the maximum number of propositions that generated assertions may contain. The number of propositions in a generated assertion is equivalent to the depth of its respective leaf node in the decision tree. Therefore, specifying a maximum number of propositions explicitly limits the depth of the decision tree. As a result, assertions become more readable and fewer in number.

Figure 14 shows how limiting the number of propositions per assertion can significantly reduce the number of true assertions generated by GoldMine. As expected the total number of true assertions increase with tree depth. However, when the maximum tree depth is 5, the total number of true assertions does not increase. This is likely because many of the design outputs are either simple enough to describe with fewer than 5 propositions, or too complex to describe with fewer than 10 propositions.

Fig. 14: Total number of true assertions generated for the UltraSparc MMU with respect to maximum tree depth.

GoldMine also allows the user to specify the maximum number of candidates to generate per output. Even if it is possible to continue, the decision tree algorithm will terminate after generating the specified maximum number of candidate assertions. Because GoldMine will produce fewer candidate assertions, GoldMine will benefit from a lower formal verification cost. In ongoing work, we have incorporated word level techniques for increasing the level of abstraction of our assertions. This can potentially spawn many directions of research.

In [43], we detail a methodology to generate assertions that utilize word-level features. The methodology discovers word-level features by computing the weakest precondition for word-level predicates in the RTL source code. To maintain scalability, the methodology uses simulation traces to guide the weakest precondition computation along feasible paths. GoldMine can use the word-level features to generate word-

level assertions. For example, the two candidate assertions `A0: (a == 1) |-> f[1] == 1` and `A1: (a == 1) |-> f[0] == 1` can be rewritten as the candidate assertion `A2: (a == 1) |-> f == 3`. In addition to reducing the total number of generated assertions, such reductions will yield assertions with greater expressivity.

GoldMine will be able to utilize additional quality metrics to filter generated assertions. For example, the assertion coverage technique presented in [6] can be used to keep only those assertions that have high RTL code coverage. Such techniques can be applied to unverified assertions to reduce both the cost of formal verification and the total number of generated assertions. In addition, filtering results according to coverage metrics can produce an assertion set with high quality.

B. The Cost of Formal Verification

From the experiments on formal verification effort and runtime as shown in section VII-B, it can be inferred that formal verification does not limit the usability of GoldMine in the multiple examples we have tried. We discuss the reasons in this section. Firstly, in the context of assertion generation, human readability of the assertions is a primary goal. Hence, we typically limit the scope to 2-3 modules at a time when we generate candidate assertions. A side effect of this process is that we are usually not trying to formally verify very large designs. In the cases where we might want to generate candidate assertions across multiple modules, formal verification can be potentially limiting.

Secondly, the type of candidate assertions generated by GoldMine are limited in scope. GoldMine generates bounded safety properties. The sequential depth of the generated assertions is limited by the user-specified temporal length of the mining window. Typically, the mining window is somewhere between one and five cycles. GoldMine does not generate unbounded liveness properties in its current version.

Thirdly, the process does not demand exhaustive formal verification of all candidate assertions. We are able to control the number of candidate assertions generated through the mining process, in order to limit the number of assertions that are formally verified. This may also put a bound on the number of true assertions generated, but this consequence is not a "show-stopper." We will still be able to produce some assertions in the module that are considered valuable according to our metrics.

Given that modern formal verification engines like IC3 [13] and ABC [14] are able to scale up to designs of 80000 registers, we do not anticipate that we will encounter issues that will prevent use of GoldMine due to formal verification.

IX. Conclusions

In this work we presented GoldMine, a methodology to automatically generate RTL assertions using data mining and static analysis. Our methodology uses a combination of data mining and static analysis to produce complex, high coverage assertions. We demonstrated the scalability of GoldMine by applying it to the OpenSparc T2 processor, which is an industrial sized design. Finally, we provided a qualitative analysis of our experimental results and discussed future research directions for our methodology.

Since we are deriving the assertions from the design itself, we may not be able to uncover bugs in the design that do not follow the specification. In future work, we plan to extend GoldMine to mine specifications for assertions and use them for checking the RTL design. We believe that GoldMine will be an important first step in increasing productivity and minimizing human resources/cost in the assertion generation process.

References

[1] Sun OpenSparc T2.
[2] Ieee standard system c language reference manual, 2006.
[3] Assertion synthesis, 2010.
[4] Activeprop assertion-based verification system, 2012.
[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.
[6] V. Athavale. Coverage analysis for assertions and emulation based verification. Master's thesis, University of Illinois at Urbana-Champaign, February 2012.
[7] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proc. of the 2005 IEEE/ACM Intl. Conf. on Computer-aided design*, pages 1052–1059, Washington, DC, USA, 2005.
[8] S. Bensalem and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Computer-Aided Verification*, pages 323–335. Springer-Verlag, 1996.
[9] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifiying safety properties of a power pc microprocessor using symbolic model checking without bdds. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 60–71, London, UK, UK, 1999. Springer-Verlag.
[10] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifiying safety properties of a power pc microprocessor using symbolic model checking without bdds. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 60–71, London, UK, UK, 1999. Springer-Verlag.
[11] M. Boule, J.-S. Chenard, and Z. Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *ISQED '07: Proc. of the 8th Intl. Symposium on Quality Electronic Design*, pages 613–620, 2007.
[12] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1):1–21, 2008.
[13] A. R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.
[14] R. Brayton and A. Mishchenko. Abc: an academic industrial-strength verification tool. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 24–40, Berlin, Heidelberg, 2010. Springer-Verlag.
[15] L. A. Breslow and D. W. Aha. Simplifying decision trees: A survey. In *Knowl. Eng. Rev.*, volume 12, pages 1–40, New York, NY, USA, January 1997. Cambridge University Press.
[16] M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the international conference on Reliable software*, pages 165–171, New York, NY, USA, 1975. ACM.
[17] P.-H. Chang and L.-C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 607–612, Piscataway, NJ, USA, 2010. IEEE Press.
[18] X. Cheng and M. S. Hsiao. Simulation-directed invariant mining for software verification. In *Proc.of the conf. on Design, Automation and Test in Europe*, pages 682–687, New York, NY, USA, 2008. ACM.
[19] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. A practical approach to coverage in model checking. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 66–78, London, UK, UK, 2001. Springer-Verlag.
[20] H. Chockler, O. Kupferman, and M. Vardi. Coverage metrics for formal verification. *Int. J. Softw. Tools Technol. Transf.*, 8(4):373–386, Aug. 2006.
[21] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 528–542, London, UK, UK, 2001. Springer-Verlag.
[22] C.-N. Chung, C.-W. Chang, K.-H. Chang, and S.-Y. Kuo. Applying verification intention for design customization via property mining under constrained testbenches. In *Proceedings of the 2011 IEEE 29th International Conference on Computer Design*, ICCD '11, pages 84–89, Washington, DC, USA, 2011. IEEE Computer Society.
[23] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.
[24] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Correct Hardware Design and Verification Methods*, pages 298–312, 1999.
[25] A. DeOrio, A. Bauserman, V. Bertacco, and B. Isaksen. Inferno: Streamlining verification with inferred semantics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(5):728 –741, may 2009.
[26] O. G. Edmund M. Clarke and D. A. Peled. *Model checking*. The MIT Press, 2000.

[27] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, S. Member, and I. C. Society. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27:213–224, 2001.
[28] H. Foster, D. Lacey, and A. Krolnik. *Assertion-Based Design*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.
[29] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.
[30] O. Grumberg and H. Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.
[31] A. Gupta. Assertion-based verification turns the corner. *IEEE Des. Test*, 19(4):131–132, 2002.
[32] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 775–778, New York, NY, USA, 2005. ACM.
[33] A. Hazra, A. Banerjee, S. Mitra, P. Dasgupta, P. P. Chakrabarti, and C. R. Mohan. Cohesive coverage management for simulation and formal property verification. In *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '08, pages 251–256, Washington, DC, USA, 2008. IEEE Computer Society.
[34] A. Hekmatpour and A. Salehi. Block-based schema-driven assertion generation for functional verification. In *ATS '05: Proceedings of the 14th Asian Test Symposium on Asian Test Symposium*, pages 34–39, Washington, DC, USA, 2005. IEEE Computer Society.
[35] R. Hilderman and H. Hamilton. Knowledge discovery and interestingness measures: A survey. Technical report, University of Regina, 1999.
[36] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
[37] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 300–305, New York, NY, USA, 1999. ACM.
[38] R. Intent. Real intent white paper.
[39] S. Katz, O. Grumberg, and D. Geist. "have i written enough properties?" - a method of comparison between specification and implementation. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 280–297, London, UK, UK, 1999. Springer-Verlag.
[40] 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 *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 140–151, 2009.
[41] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
[42] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA, 1994.
[43] L. Liu, C. Lin, and S. Vasudevan. Word level feature discovery to enhance the quality of assertion mining. In *Computer Aided Design, 2012. ICCAD 2012. IEEE/ACM International Conference on*, 2012.
[44] L. Liu, D. Sheridan, V. Athavale, and S. Vasudevan. Automatic generation of assertions from system level design using data mining. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 191 –200, july 2011.
[45] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan. Towards coverage closure: Using GoldMine assertions for generating design validation stimulus. Technical report, University of Illinois Urbana Champaign, 2011.
[46] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan. Towards coverage closure: Using goldmine assertions for generating design validation stimulus. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –6, march 2011.
[47] K. McGarry. A survey of interestingness measures for knowledge discovery. *Knowl. Eng. Rev.*, 20(1):39–61, 2005.
[48] J. Misra. Prospects and limitations of automatic assertion generation for loop programs. *SIAM Journal on Computing*, pages 718–729, 1977.
[49] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 232–242, 2002.
[50] C. S. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proc. of the SPIN Workshop on Model Checking and Software*, page 2989, Barcelona, Spain, 2004.
[51] G. Pinter and I. Majzik. Automatic generation of executable assertions for runtime checking temporal requirements. In *HASE '05: Proc. of the 9th IEEE Intl. Symposium on High-Assurance Systems Engineering*, pages 111–120, Washington, DC, USA, 2005. IEEE Computer Society.
[52] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Symp. on Foundations of Computer Science*, pages 46–57, 1977.
[53] A. Pnueli. The temporal logic of programs. In *In Proc. of FOCS*, pages 46–57, 1977.
[54] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986.
[55] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
[56] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke. Automatic generation of complex properties for hardware designs. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 545–548, New York, NY, USA, 2008. ACM.
[57] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 84 –88, 0-0 2006.
[58] A. Silberschatz and A. Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Transactions on Knowledge and Data Engineering*, 8:970–974, 1996.
[59] P.-N. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In *KDD '02: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 32–41, New York, NY, USA, 2002. ACM.
[60] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
[61] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 113–127, London, UK, 2001. Springer-Verlag.
[62] B. Touhy. private communication, 2009.
[63] S. Vasudevan, D. Sheridan, D. Tcheng, S. Patel, W. Tuohy, and D. Johnson. GoldMine: Automatic assertion generation using data mining and static analysis. In *Proc. of the Conf. on Design, automation and test in Europe*, pages 626–629, 2010.
[64] D. J. V. G. J. K. Viraj Athavale, Sam Hertz and S. Vasudevan. Using static analysis for coverage extraction from emulation/prototyping platforms. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2012 IEEE/ACM/IFIP International Conference on*, 2012.
[65] D. Wang and J. Levitt. Automatic assume guarantee analysis for assertion-based formal verification. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 561–566, New York, NY, USA, 2005. ACM.
[66] L.-C. Wang, M. S. Abadir, and N. Krishnamurthy. Automatic generation of assertions for formal verification of powerpc microprocessor arrays using symbolic trajectory evaluation. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 534–537, 1998.
[67] J. Yang and D. Evans. Dynamically inferring temporal properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 23–28, New York, NY, USA, 2004. ACM.

Samuel Hertz earned his B.S. and is pursuing his M.S. in electrical and computer engineering from the University of Illinois at Urbana-Champaign. His research interests include hardware verification, hardware validation, system generation, and assertion evaluation. He developed and maintains GoldMine: an automatic assertion generation engine.

David Sheridan earned his B.S. and M.S. in electrical and computer engineering from the University of Illinois at Urbana-Champaign. His research interests included hardware verification, hardware validation, and assertion generation. He developed algorithms for GoldMine: an automatic assertion generation engine.

Shobha Vasudevan is an assistant professor in the Electrical and Computer Engineering Department at the University of Ilinois at Urbana-Champaign. Her research interests are in hardware verification, validation, reliability, security, as well as software testing. She earned her M.S. and Ph.D. from the University of Texas at Austin. She is a recipient of the 2010 NSF CAREER award.