GoldMine: Automatic Assertion Generation and Coverage Closure in Design Validation

David Sheridan, Lingyi Liu, and Shobha Vasudevan

Electrical and Computer Engineering Department, University of Illinois at Urbana Champaign
1308 West Main St. Urbana, IL 61801, USA
dsherid3@illinois.edu
liu187@illinois.edu
shobhav@illinois.edu

***Abstract*—We present GoldMine, a methodology for generating assertions automatically. 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. 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. The counter-examples generated by the formal verification can then we used as feedback to the decision tree algorithm to increase the design coverage. These assertions are evaluated by a process of designer ranking that can be provided as feedback to the data mining engine. We present results of using GoldMine for assertion generation of the Rigel 1000+ core processor. Our results show that GoldMine can generate complex, high coverage assertions in RTL, thereby minimizing human effort in this process.**

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* provide a mechanism to express desirable properties that should be true in the system. Assertions are used for validating hardware designs at different stages through its life-cycle like pre-Silicon formal verification, dynamic validation, runtime monitoring and emulation [1]]–[[3]. Assertions are also synthesized into hardware for post-Silicon debug and validation and in-field diagnosis [1]], [[4].

Among all the solutions for ensuring robustness of hardware systems, assertion based verification has emerged as the most popular candidate [5] 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 [2]], [[6]], [[7]. 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 advanced data mining algorithms to produce rules that are *candidate assertions*, since they are inferred from the simulation data, but not for all possible inputs. Static behavioral analysis techniques are employed to guide the data mining process. 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. The formal verification provides counter-examples which the decision tree uses to increase its knowledge of the design. This increases the number of true assertions, thereby increasing the assertion coverage of the design. A designer evaluation and ranking process is facilitated in GoldMine to provide useful feedback to the iterative data mining process.

GoldMine proposes a radical, but powerful validation paradigm. It uses two high impact technologies- data mining and static analysis symbiotically to assimilate the design space. It then reports its findings in a human digestible form (assertions) early on and with minimal manual effort. This is intended to replace the traditional method of the engineer deducing all possible correct behaviors, capturing them in assertions, testing assertions, creating directed tests to observe behavior and finally applying random stimulus.

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 quantum 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 is best utilized in the regression test suite of an RTL design.

GoldMine is completely automatic. It is able to generate many 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 [8]. [1] GoldMine can generate assertions that are *complex* or span multiple logic levels in the RTL.

We present the Rigel [9] 1000+ core architecture design as a detailed case study for GoldMine. The Rigel RTL has been developed recently and is in a stage of functional verification. The evolving Rigel RTL provides a fertile ground for investigating our methodology. The assertions generated through GoldMine can be used as a *regression test suite for Rigel*. In addition, we explore the SpaceWire [10], OpenRisc [11], and ITC benchmark [12] designs to further demonstrate the versatility of GoldMine.

Our contributions in this work are as follows.

- We introduce a methodology and tool flow to generate system invariants in hardware automatically using data mining and static analysis.
- Our tool can produce temporal and complex assertions for sequential and combinational modules. To the best of our knowledge, such a result has not been achieved in the state-of-the-art.
- Our method abridges the validation phase by distilling random stimuli and achieves coverage of unexplored spaces earlier than typical in the design cycle.
- With GoldMine, we propose a validation paradigm that would significantly reduce time and resource consumption, increasing validation productivity.
  We demonstrate that GoldMine produces excellent results on a real RTL design in the form of complex, high coverage assertions attested by Rigel designers as very interesting.

[1] At this time, we can generate assertions with the X operator.

.nosparc

II. GoldMine: Assertion Generation Methodology

We propose **GoldMine**, a methodology to automatically generate assertions using data mining and static analysis. There

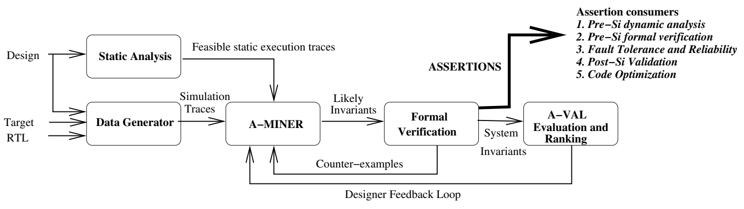

Fig. 1. Goldmine Tool Suite

are five main parts in GoldMine.

A. Data Generator

The Data Generator simulates a given design (or a "module" of the design). If regression tests or workloads for the design are available, they can be used to obtain the simulation traces. GoldMine also generates its own set of simulation traces using random input vectors.

Typically, simulating with randomized inputs produces the largest number of true assertions. We used a script to generate a testbench for each verilog design that we wanted to test. In the testbench, each input bit is assigned with completely random value each cycle by using the verilog*$random* function. We have the ability to expand this method in the future by constraining the random input values using background information where certain input combinations may not be allowed. For most of our tests, we simulate for 10,000 cycles, though we can increase this number for extremely large or complex designs.

B. Lightweight Static Analyzer

The static analyzer extracts domain-specific information about the design that can be passed to A-Miner. It can include cone-of-influence, localization reductions [13], topographical variable ordering and other behavioral analysis techniques.

The current version of the tool only uses static analysis for logic cone information. The logic cone of a signal consists of all of the inputs which can influence the value of a given output. Since data mining methods can only use statistical methods to infer relationships between signals, it is possible that an unrelated input may be correlated to an output. The logic cone prevents this problem by restricting the searched inputs to only those which are related to the output. This static analysis is also advantageous in that it decreases the runtime in many data mining algorithms since there are fewer inputs to consider.

C. A-Miner

The A-Miner phase derives knowledge and information from the simulation trace data. This is done by searching for correlations between the inputs and a target output. For example, in a simulation trace, when ever inputs A and B

are both 1, the output C is also 1. A data mining algorithm can quickly and efficiently recognize this pattern. Data mining algorithms use statistics such as support and confidence to determine whether there is actually a relationship between the inputs and target output. Given a rule $A \implies B$ (henceforth of the form *if a then b*), *support(A)* is the proportion of instances in the data that contain A. Confidence can be interpreted as an estimate of the conditional probability $P(B|A)$. If a rule has 100 per cent confidence, it means that within the data set, there is complete coincidence between A and B. A high support for this rule means that A occurs frequently in the data set. In GoldMine, we must guarantee that the confidence is 100% if we want to generate an assertion that is likely to be true. The reason for this is that if a given antecedent is correlated with an output that has multiple different values, then that can not be an assertion since the antecedent does not imply a single value.

A-Miner also provides hooks for incorporating domain specific information from the lightweight static analyzer into the mining algorithms. The data mining algorithm allows specification of which inputs have relationship with the target output as determined by the logic cone. In addition, this phase of GoldMine can have multiple feedback loops from different parts of the tool. Using the information provided to it, the A-Miner produces a set of candidate assertions which are likely to be true. Objective measures of interestingness [14] can be used to rank this set of candidate assertions like the support as specified above.

D. Decision Tree Based Supervised Learning Algorithms

Association rule based data mining algorithms like FP Growth [15] find all possible associations between sets of predicates and rank them according to support/confidence. For sequential blocks that might have temporal properties, exhaustive search is an inefficient option in our experience (see case study).

We primarily use decision tree based supervised learning algorithms [16] in A-Miner. In a decision tree, the data space is locally divided into a sequence of recursive splits in a small number of steps. 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 the input data space into local regions continues recursively until it reaches a leaf.

We require only Boolean splits (for Boolean variables) at every decision node. The error function implemented to select the best splitting variable at each node is the variance between the target output values and the values predicted by a candidate antecedent. The winner is the one whose error is minimum which then forms the next level of the decision tree. Each leaf in the decision tree becomes a candidate assertion where the variable and value at each split represents a proposition in the antecedent and the mean of the output represents its predicted value in the consequent.

E. Formal Verifier

In order to check if the likely invariants generated by A-Miner are system invariants, the design and candidate assertions are passed through a formal verification engine. If a candidate assertion fails formal verification, a counterexample can be generated for feedback to the A-Miner. We use SMV [17] and Incisive Formal Verifier as our formal verification engines. The candidate assertions are attached to the design for verification and checked at the positive edge of the clock cycle. The reset signal of the design is constrained to off as to prevent spurious counterexamples. Although the attempt in GoldMine is to minimize the human effort in the assertion generation process, we need human intervention to differentiate between a spurious candidate assertion that fails the formal verification and a genuine system invariant whose failure reports the existence of a bug.

F. Counter-example Refinement

When a candidate assertion is proven false using a formal verifier, a counter-example is produced that shows the reason why that rule is not true. We can use this counter-example feature of the formal verifier to increase the number of true assertions generated using a technique referred to as counter-example refinement. This method turns the decision tree algorithm into an iterative algorithm which can infer previously unknown information using these counter-examples to increase the total number of true assertions for a design.

When a counter-example is produced by the formal-verifier, it indicates that the simulation trace data provided to the decision tree was not sufficient to generate a true correlation. Because the decision tree only produces candidate assertions based on rules with 100% confidence, this indicates that the rule is true with respect to the simulation data, but untrue with respect to the actual design since the simulation data does not necessarily capture all of the design. A counter-example can be viewed as data sample which contradicts the data samples which were used to create the untrue rule. Thus, when you combine samples from the original data set and the counter-example, the confidence of the untrue rule must be less than 100%. Since the confidence is less than 100%, the decision tree can continue to split until more rules with 100% confidence are created.

G. A-Val: Evaluation and Ranking

Once the assertions have been generated through GoldMine, their evaluation is extremely important to the process. This is because assertion generation has been a completely manual process thus far in the system design cycle.

There are several ways for us to evaluate A-Miner's performance. One basic metric is the hit rate of true assertions. The *hit rate* of a run in GoldMine is the ratio of true assertions to candidate assertions. This provides a very crude indicator of performance. In addition, we can consider *output hit rate*, which is the number of outputs for which GoldMine could generate a true assertion over the total number of inputs.

Since there are no commercially used metrics for evaluating the coverage of an assertion, we have devised a method to evaluate assertion coverage. We can evaluate the coverage of an assertion by considering the input space that is covered by the antecedent of the assertion. If we consider the truth table with respect to some output, each entry that corresponds to the propositions in the antecedent of an assertion is defined as covered by that assertion. For example, if there is an assertion $(a = 1 \& b = 1 \implies c = 1)$, we can consider the input space coverage to be 25% since we know that 25% of the truth table entries contain $a = 1, b = 1$. The reasoning behind this thinking is that if there is a set of assertions that covers each entry in the truth table of an output, that output is well covered by the set of assertions. This metric is simple to calculate since we can determine the percentage of the input space that an antecedent of an assertion covers without knowing every single input combination. The input space coverage is defined as $1/2^{|P|}$ where $|P|$ is the number of propositions in the antecedent. Based on this definition, it can be seen that the input space coverage is relative to the number of propositions in the antecedent.

In order to bridge the gap between the human and the machine generated assertions, human judgment can also be made a part of the GoldMine process where the designer ranks the true assertions according to some pre-defined ranks. This provides an objectification of an inherently subjective decision and can be used as feedback into A-Miner, with a view to predict the ranking of a generated assertion and optimize the process for achieving higher ranks.

III. Related Work

We address the related work in the broad spectrum of topics allied to our work. Assertion generation though static analysis of source code or a model has been studied in the context of deductive program verification [18]]–[[20] since the seventies. The deduction of "weakest liberal precondition" from the loop body can quickly get very complex. Static analysis techniques have been used to learn invariants for assisting software verification [21]], [[22] Dynamic analysis [7]], [[23] as well as data mining [24] have been used in software to determine system invariants.

In hardware, to the best of our knowledge, there have been no prior attempts to generate assertions through data mining and static analysis of RTL source code. Some work has been done in assertion generation for hardware by statically analyzing the hardware structure and topology as in [25]]–[[27]. IODINE [28] infers detailed, low-level dynamic invariants for hardware designs. This is different from our work, since it does not use data mining techniques to infer invariants, but a dynamic analysis framework that analyzes the program behavior with respect to standard property templates like one-hot encoding, mutex etc. The work in [29] use dynamic simulation trace data for generating assertions, but their technology does not use data mining. Instead, they try to generalize trace behavior. Commercial tools [30] that try to generate assertions capture simple, pre-defined invariants.

IV. Case Study: Rigel RTL

We present results of applying GoldMine for 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. A-Miner takes about 45 minutes to run on 121 outputs, 700 inputs and 20000 data samples. SMV takes about 30 minutes per 2000 candidate assertions. We have not faced any state space explosion issues with SMV so far. All experiments were run on a 2.26GHz dual core processor with 4GB RAM.

The decision tree algorithm is very quick, but the formal verification in GoldMine can take a long time when there are many assertions to verify. By using a commercial tool for formal verification instead of SMV, we were able to produce a significant speedup. We have also used parallelism to increase the speed of the formal verification step. Since each assertion can be verified concurrently, several formal verification threads can be used for a significant speedup. Because there is some overhead in creating the model in formal verification, a small batch of assertions in verified in each thread. This gives a speedup that is nearly linear with respect to the number of processor cores.

Since memory conservation is important for large problems, we have ported our code from Java to C++. Since Java has dynamic memory management, it is difficult to control the memory usage and can make it difficult to debug memory leaks. Since C++ requires manual memory management, it is easier to keep the memory usage low and controlled.

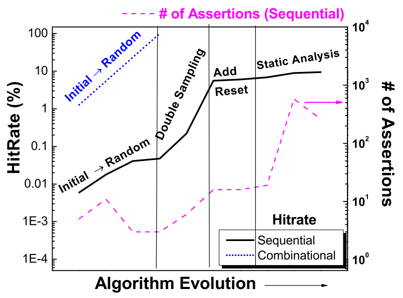

Fig. 2. GoldMine evolution: Aspects of the algorithm that increased yield wrt hit rate and number of true assertions

Our experiments establish that combinational properties are very easy to generate in GoldMine. Temporal properties require more methodological fine-tuning. Figure 2 shows the evolution of the GoldMine process with respect to the hit rate of assertions and the number of true assertions generated for the combinational as well as sequential modules. The x-axis shows some of the "knobs" that we used for fine-tuning our methodology.

In the initial phase of the Data Generator, we used simulation workloads from the Rigel test suite (denoted by Initial). This data was insufficient, (approximately 15 tests of 1000 samples each) producing a very low hit rate for both propositional as well as temporal assertions (combinational and sequential modules). We then used **random input vector generation on the RTL** for the target modules. These tests had about 10000 samples each. This drastically increased the hit rate as well as number of true assertions, demonstrating that the amount of simulation data can significantly affect the performance of GoldMine. For the writeback module, we achieved a 100 percent hit rate with this step alone.

In the next phase, we made a change to the way we sampled the simulation data. Initially, we collected data only at the positive edge of the clock signal, making it appear that blocking and non-blocking statements were happening at the same time. As a result, we were generating many candidate temporal assertions, with very few of them being true. We then changed this sampling process such that data was captured at positive as well as negative edges of the clock signal, (when the input changes) so that there is a distinction in when the output is assigned. This is shown by **double sampling** in the graph. These two modifications were added at the same time, increasing our hit rate for temporal assertions in sequential modules significantly.

An interesting spike in the hit rate was caused by **increasing the frequency of the reset** event in the simulation data. Initially, our simulation data was collected with the reset signal being high in the first few cycles and low thereafter. We noticed that our true assertions did not include the reset signal. We then forced reset to be high once every 500 cycles. This expanded the scope of our assertions to those that had the reset signal in them.

In the next phase of GoldMine, we added **lightweight static analyzer information** that was specific to the domain, like logic cone-of-influence generation and static topographical variable ordering. Although this increased the hit rate only marginally, it increased the number of true assertions significantly. This shows that the static analysis information was very useful in helping A-Miner focus on the relevant neighborhood of variables to generate candidate assertions.

We describe some algorithmic details of the GoldMine process not shown in Figure 2. In the initial phase, we used the FP-Growth option. This took unreasonable time (>10 hours) for reaching rules with just 3 predicates for the decode module. We therefore resorted to the decision tree algorithm for our purposes. Another aspect is the stopping criterion of the decision tree splitting. Our initial experiments continued the splitting process beyond the point where the minimum error reduction was reached. This process gave us an extremely high number of candidate assertions (>80000) with many duplicates (289 out of 300). In the later stages, we elected to end the decision tree splitting when error was numerically equal to "0", *i.e.* at the point of 100 % confidence.

GoldMine has evolved continuously since the original concept for the tool was developed. The tool has been molded to better match expected usage. Originally, GoldMine only worked with combinational circuits which interesting, but not very useful to the average verification engineer. The reason for this is that pattern recognition algorithms used in data mining look for correlations that hold true in all samples, which is consistent with combinational behavior since outputs change immediately. However, in sequential circuits, outputs do not change until the positive edge of the clock. This means that if a sample is taken before the clock edge, the output will contain the value determined by the inputs at the previous clock edge, and not the inputs at the current time. This means that no relationship can be found since the current inputs have not influenced the output yet. If a sample is taken after the clock edge, the inputs have already changed from the values that determined the current output, meaning that there is still no relationship that can be inferred by this samples.

This problem can be solved without having to change the data mining algorithm. The data is only sampled once per positive clock edge since that is when the interesting behavior happens. The exact time at which the signal is sampled depends on the type of signal. If the signal is an input, it is sampled right before the positive clock edge and if the signal is an output, it is sampled right after the clock edge. To the data miner, it seems as if the inputs and outputs have changed at the same time and is able to find any relationships between the inputs and outputs.

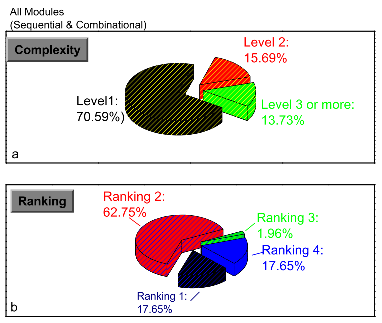

Fig. 3. GoldMine assertion complexity; A-Val ranking by designers

The next set of experiments help evaluate GoldMine's assertions. Since designer evaluation and ranking forms an important part of A-Val, 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 1-4, calibrated as below:

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 3 show the distribution of these ranks for a sample of representative assertions over all the modules. The algorithmic knobs that produced the highest hit rate as well as the highest number of assertions were turned on for this experiment. The maximum number of assertions in this analysis rank at 2. The writeback module has some assertions ranked 3. The absence of 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. For example, an assertion ranked 1 is: *If the halt signal in the integer, floating point and memory unit is set to 0, the halt signal is 0.* In the RTL, the halt signal is a logical OR between the integer, floating and memory units. GoldMine found a true, but over-constraining rule. The designers ranked it 1, since they would not have written this rule. Now, consider this RTL code:

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

An assertion ranked 2: *if branch_mispredict is high, decode2memvalid will be high in the next cycle.* An assertion ranked 3: *If an integer unit does not want to use the first port, and the floating point unit does not want to use the second port, then the second port remains unused.*

a) Complex assertions in GoldMine: Despite the small size of the modules, GoldMine achieved rank 4, *i.e.* it produced assertions that capture complex relationships in the design. This is an advantage of mechanically derived assertions: they are able to 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, the RTL has the following relationship:

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

An assertion ranked 4 is: *if (reset=0) and (issue0=0) and (decode_packet_dreg=0), and in the next cycle if (instr0_issued = 0), then decode_packet_dreg = 0.* 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 3 shows the distribution of assertions with respect to complexity.

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

Fig. 4. GoldMine output coverage

Figure 4 shows the number of outputs per module for which assertions were generated by GoldMine. Although candidate assertions were generated for all the module outputs, the assertions that passed formal verification covered a percentage of them. Figure 5 shows the probability distribution of true assertions per output. At the 50% mark, there will be approximately 4-5 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 unique assertions per output are indicative of high path coverage.

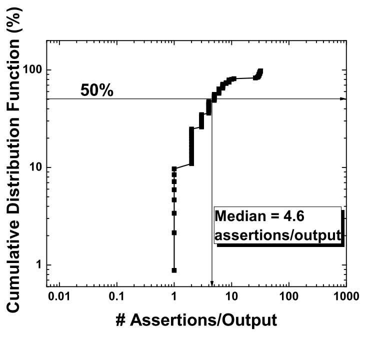

Fig. 5. Distribution of unique assertions per output in all modules

b) 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.

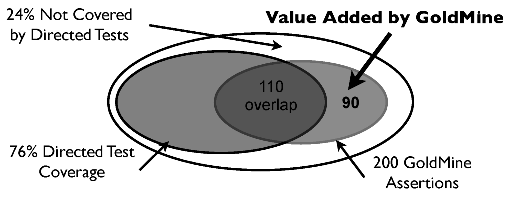

Fig. 6. Figure showing the added coverage of design behavior through GoldMine assertions for the writeback module

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

The writeback module directed tests achieved 76% conditional coverage, while the random tests used to generate the GoldMine assertions 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 stimulated by the directed tests. Therefore 90 assertions, or 45%, refer to design behavior as yet untested by the directed tests. Figure 6 shows the overlap of assertions with directed tests, and highlights the value of GoldMine providing significant coverage of the unexplored regions of the design at this early stage.

The overlapping assertions that coincide with the designer-crafted directed tests can be used for static checking, formal verification etc. However, the untouched assertions can be used to improve the quality of the directed tests. They can be used as regression checks as the test patterns mature and 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 evaluated GoldMine's contribution as "covering a wide design space much earlier in the design cycle than typically achievable".

A. Counter-example Refinement Experiments

We also want to evaluate the results of the counterexample refinement method. The first experiment demonstrates the increase in coverage as the counterexample algorithm progresses, showing a monotonic increase in coverage. The experiment is performed on SpaceWire codec state machine circuit and Rigel write back stage design. The original test suite can be in the form of a directed test or a completely random input stimulus test. In this experiment, we simply use the initial random input patterns. In each iteration, any spurious assertions are refined using counterexamples until the A-Miner has generated a true assertion. The input space coverage and industrial standard coverage metric are used in this experiment. The input space coverage of each true assertion referring corresponding output is calculated by considering the percentage of the truth table entries that is covered that assertion. We have summarized these coverage results in Figure 7 and Figure 8.

In Figure 7, the input space coverage referring to each output was chosen to measure the validation process. Since each assertion compactly covers multiple concrete patterns of input space, we calculate the input space coverage referring to an output by accumulating the input space coverage of all generated assertions on that output. The results show a consistent increase in the input space covered by the assertions in each iteration.

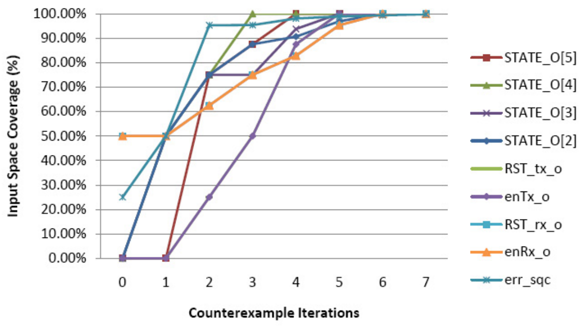

Fig. 7. Input space coverage of each output increasing over the number of counterexample iteration on SpaceWire-FSM design and Rigel-wb stage design

In Figure 8, we choose the line, conditional, branch, toggle and FSM coverage from industrial standard metric. Redundant statements, unreachable states and other RTL characteristic often limit some kind of coverage to achieve 100%, but a steady increase in such coverage is an indicator of monotonic progress in the quality of the assertion/tests generated by our algorithm.

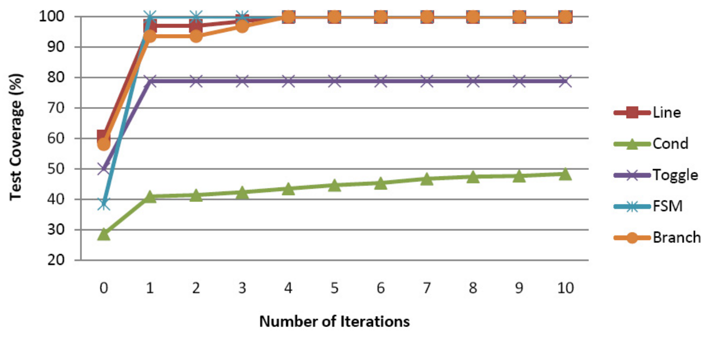

Fig. 8. Standard coverage increasing over the number of counterexample iteration on SpaceWire-FSM design and Rigel-Wb stage design

We also notice that the coverage increases quickly in the early iteration and slowly in the later iteration. However, different from the tradition industrial flow, our method can guarantee coverage gain in each step and finally reach full coverage. In the worst case, the maximum number of iteration required to reach full coverage is equal to the number of input variables in the logic cone of corresponding output since at least one variable is added to the original assertion as counterexample to disprove the spurious assertion.

The second experiment is a limit study showing that the counterexample algorithm works even when no original directed or random test suite exists. The lack of any patterns would begin the procedure with a simple assertion of the form *"output always 0"*. Figure 9 shows the increase in coverage for each design as the algorithm progresses. Even without initial test patterns, the counterexample method is able to create a test suite that achieves good coverage with few iterations. This indicates that this method may be a useful methodology to jump start a module design environment by creating many tests that can then be run on the testbench to check against the design specification.

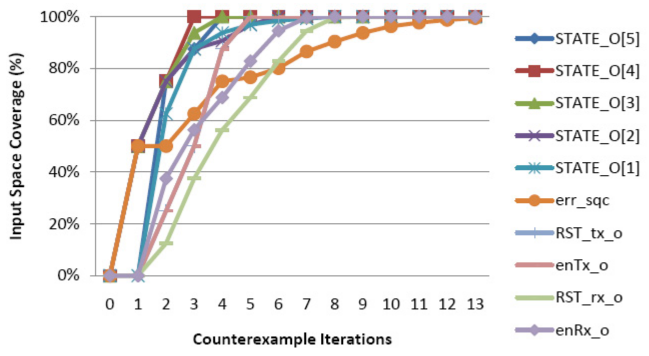

Fig. 9. Coverage increasing by iteration starting from zero pattern on SpaceWire-FSM design and OpenRisc cache controller design

V. GoldMine Applications

Though GoldMine is an interesting tool, it can be difficult to see how it can be used is a realistic verification environment. Since GoldMine produces assertions based on RTL which are then verified using formal verification, it is trivial that generated assertions will pass on the given RTL. The beauty of this tool is that it can actually be applied in a number of ways, included applications that have not even been developed yet.

One way to use GoldMine effectively is to use the assertions as a regression test throughout development. The assertions that are true in one revision may fail in a later revision. This can indicated that the assertions are no longer relevant, which indicates that those assertions must be updated. However, it can also indicate that a revision of the design introduced a bug which the assertion can help to locate. For example, GoldMine is used on an ALU unit and produces a set of assertions. The ALU is then revised to make a certain function faster. If there are any assertions that fail, it likely indicates that there is a bug in the revised code.

When using random testing to verify a design, it can be difficult to determine the number of cycles to simulate before declaring a unit fully verified. One way to measure testing completeness is to use standard coverage metrics but this method only gives a very general idea of the coverage. GoldMine can also be used in addition to standard coverage metrics to increase confidence of a design. The trace from the random test simulation can be mined for assertions using GoldMine. Any assertion mined from this trace indicates behavior that is covered in the simulation trace. This means that if the assertions generated in GoldMine have a high coverage, it is likely that a high percentage of design behavior has been covered in the random test. If the assertions generated do not have high coverage, the simulation likely needs to run for more cycles.

VI. Conclusions

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] 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*, 2007, pp. 613–620.

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

[3] 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*, Washington, DC, USA, 2005, pp. 1052–1059.

[4] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of psl properties," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 1, pp. 1–21, 2008.

[5] A. Gupta, "Assertion-based verification turns the corner," *IEEE Des. Test*, vol. 19, no. 4, pp. 131–132, 2002.

[6] 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*. New York, NY, USA: ACM, 2005, pp. 561–566.

[7] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," in *In ISSTA*, 2002, pp. 232–242.

[8] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *POPL*, 1981, pp. 164–176.

[9] 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*, 2009, pp. 140–151.

[10] "European Space Agency SpaceWire: *http://spacewire.esa.int*."

[11] "Openrisc web page: *http://www.opencores.org*."

[12] "International Test Conference Benchmarks: *http://itc02socbenchm.pratt.duke.edu*."

[13] R. P. Kurshan, *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[14] 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*. New York, NY, USA: ACM, 2002, pp. 32–41.

[15] C. Borgelt, "An implementation of the fp-growth algorithm," in *OSDM '05: Proceedings of the 1st international workshop on open source data mining*. New York, NY, USA: ACM, 2005, pp. 1–5.

[16] L. A. Breslow and D. W. Aha, "Simplifying decision trees: A survey," 1996.

[17] K. L. Mcmillan, "The smv system," 1992.

[18] M. Caplain, "Finding invariant assertions for proving programs," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 165–171.

[19] J. Misra, "Prospects and limitations of automatic assertion generation for loop programs," *SIAM Journal on Computing*, 1977.

[20] S. Bensalem and H. Saidi, "Powerful techniques for the automatic generation of invariants," in *CAV*. Springer-Verlag, 1996, pp. 323–335.

[21] 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*. London, UK: Springer-Verlag, 2001, pp. 113–127.

[22] C. S. Pasareanu and W. Visser, "Verification of java programs using symbolic execution and invariant generation," 2004.

[23] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *SIGPLAN Not.*, vol. 37, no. 1, pp. 4–16, 2002.

[24] 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*. New York, NY, USA: ACM, 2008, pp. 682–687.

[25] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 34–39.

[26] 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*, 1998, pp. 534–537.

[27] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 111–120.

[28] 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*. New York, NY, USA: ACM, 2005, pp. 775–778.

[29] 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*. New York, NY, USA: ACM, 2008, pp. 545–548.

[30] "Real intent white paper," http://www.winian.us/7w/insidetemplate.htm.