GoldMine: Automatic Assertion Generation Using Data Mining and Static Analysis

Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy and Daniel Johnson
Electrical and Computer Engineering Department
University of Illinois at Urbana Champaign

***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. We present results of using GoldMine for assertion generation of the RTL of a 1000-core processor design that is still in an evolving stage. Our results show that GoldMine can generate complex, high coverage assertions in RTL, thereby minimizing human effort in this process.**

I. INTRODUCTION AND MOTIVATION

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], as well as post-Silicon debug and in-field diagnosis [1]], [[4]. Assertion based verification has emerged as the most popular candidate [5] solution for "pre-Silicon" design functionality checking.

The key question then is: How are these assertions generated? Assertion generation is an entirely manual effort in the hardware system design cycle. The trade-off point for crafting minimal, but effective (high coverage) assertions takes multiple iterations and man-months to achieve [2]. Another challenge with assertion generation is due to the modular nature of system development. 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. Temporal assertion generation is very difficult for the human mind to reason with.

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. 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 behavioral 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 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.

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 [6]. GoldMine can generate assertions that are *complex* or span multiple logic levels in the RTL.

We present the Rigel [7] 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 hardware, there have been no prior attempts to generate assertions through data mining and static analysis of RTL source code. Assertion generation in hardware by statically analyzing the hardware structure and topology has been explored before [8]], [[9]. IODINE [10] 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 [11] use dynamic simulation trace data for generating assertions, but

978-3-9810801-6-2/DATE10 © 2010 EDAA

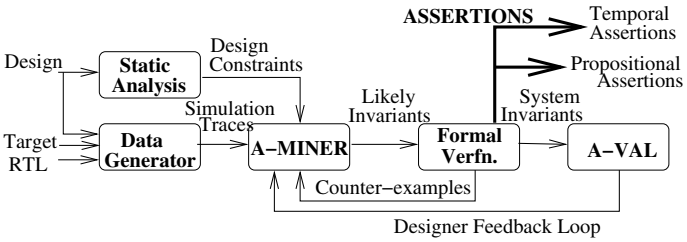

Fig. 1. Goldmine Tool Suite

```
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

their technology does not use data mining. Commercial tools [12] that try to generate assertions capture only simple, predefined invariants.

II. GoldMine: Assertion Generation Methodology

We propose **GoldMine**, a methodology to automatically generate assertions using data mining and static analysis. Figure 1 shows the main parts of GoldMine.

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.

Lightweight Static Analyzer: The static analyzer extracts domain-specific information about the design that can be passed to A-Miner. It includes cone-of-influence, localization reductions, topographical variable ordering and other behavioral analysis techniques.

A-Miner: The A-Miner phase derives knowledge and information from the simulation trace data. A-Miner provides hooks for incorporating domain specific information from the lightweight static analyzer into the mining algorithms. This phase of GoldMine has multiple feedback loops from different parts of the tool. A-Miner finally produces a set of candidate assertions.

Metrics that are entirely statistical, like support and confidence, are used to rank the candidate assertions generated by A-Miner. 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 is the 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. Support and confidence can jointly be used to rank the rules according to their utility.

Decision Tree Based Supervised Learning Algorithms

We primarily use decision tree based supervised learning algorithms [13] in A-Miner. 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.

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 variable. 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.

Formal Verifier: In order to check if the likely invariants are system invariants, the design as well as the candidate assertions are passed through a formal verification engine. If a candidate assertion fails formal verification, a counterexample is generated. The formal checking phase provides an excellent goodness metric for A-Miner's performance. We quantify this metric as the hit rate. The *hit rate* of a run in GoldMine is the ratio of true assertions to candidate assertions. We use SMV [14] as our formal verification engine.

A-Val: Evaluation and Ranking: Once the assertions have been generated through GoldMine, their evaluation is extremely important to the process. We would like to compare the machine generated results to those generated by a human and provide feedback to GoldMine to refine its results accordingly.

Human judgment is made a part of the GoldMine process. The designer ranks the true assertions according to some predefined ranks that can be used as feedback into A-Miner.

A. An example run through GoldMine

We illustrate the GoldMine process for the RTL in Figure 2. The Data Generator runs a few simulations and produces the simulation results shown in the table in Figure 2.

A-Miner now forms a decision tree for the data shown in Figure 3. The mean of `wb_valid0` is set to 0.25 (average of its values) and the error is set to absolute difference from the mean, 0.375. The decision tree now tries to split based on the maximum error reduction among all the input values. The values of error for the 0/1 values of `int.valid`, `int.L1_hit` and `int.has_dreg`are (0,0.5). Since all values (0/1) of all inputs produce equal error, and in the absence of any guidance from the static analyzer, the decision tree uses the simple heuristic of splitting on the first variable in the list, `int.valid`. On the `int.valid = 0` branch, error is reduced to 0, making it a leaf node. *A0: if (int.valid = 0) then (wb_valid0 = 0)* is the candidate assertion generated. Since the error value has not yet reached 0 on the `int.valid = 1` branch, the decision tree tries to split again. Although the value of `int.has_dreg` is the variable that affects the output of interest, the splitting variable is `int.L1_hit` since the error reduction for all variable values are equal, and it is first in the list. Since both branches of the tree at this level reach error=0, the leaves produce *A1: if (int.valid = 0) and (int.L1_hit = 0) then (wb_valid0 = 0)* and *A2: if (int.valid = 1) and (int.L1_hit = 1) then (wb_valid0 = 1)* as candidate assertions.

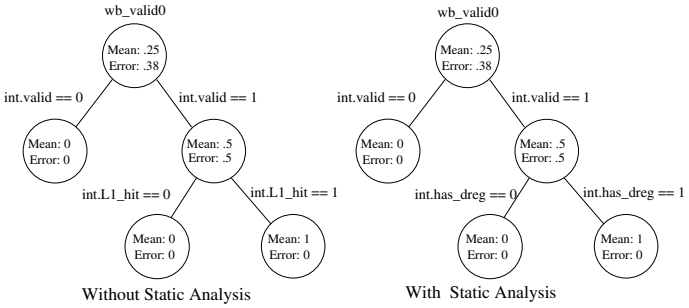

Fig. 3. Example Decision Tree Output with and without Static Analysis

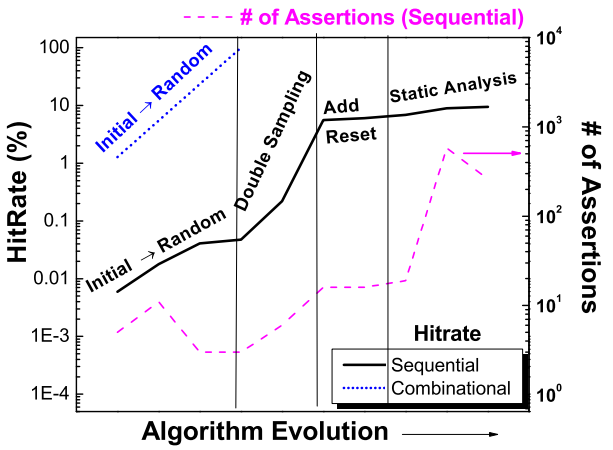

Fig. 4. GoldMine evolution: Aspects of the algorithm that increased yield wrt hit rate and number of true assertions

All candidate assertions *A0, A1, A2* are passed to a formal verification engine, that passes *A0* and *A1*, but fails *A2*. Hit rate is 2/3 in this case. *A*3 fails due to the false causality that is established by simulation data.

In the presence of the lightweight static analyzer, the logic cone establishes the part of the design that is causal to `int.valid`, providing a list of variables to the decision tree that excludes `int_in.L1_hit` (Figure 3). The candidate assertions produced now are *A0* (same as in previous case), *A1: if (int.valid = 0) and (int.has_dreg = 0) then (wb_valid = 0)* and *A2: if (int.valid = 1) and (int.has_dreg = 1) then (wb_valid = 1)*. All these candidate assertions are passed by the formal verifier, with a consequent hit rate of 1.

III. 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. 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 (on a 2.26GHz dual core processor with 4GB RAM). SMV takes about 30 minutes per 2000 candidate assertions.

Our experiments establish that combinational properties are very easy to generate in GoldMine. Temporal properties require more methodological fine-tuning. Figure 4 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.

The next set of experiments help evaluate GoldMine's assertions. 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 follows. Rank 1 is for a trivial assertion that the designer would not write. Rank 2 is for an assertion the designer would write. Rank 3 is for an assertion that the

designer would write and captures subtle design intent. Rank 4 is for a complex assertion that the designer would not be able to write.

For a sample of representative assertions over all modules, the designer ranking distribution is as follows. **17.65% at Rank 1, 62.75% at Rank 2, 1.96% at Rank 3, and 17.65% at Rank 4**. 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, is due to intra module behavior not being 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.

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.*

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. We assessed complexity by the number of levels (depth) of the design captured by assertions. Over all three modules, the complexity distribution of assertions was as follows. **70.59% of assertions at Level 1, 15.69% at Level 2, and 13.73% at Level 3 or more.** 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 and may present avenues for optimizing the RTL.

We analyzed the assertion coverage per output 46.76% of the assertions werDecode stage, 35.71% Fetch stage, and 87.5% Writeback stage. Although candidate assertions were generated for all the module outputs, the assertions that passed formal verification covered a percentage of them. In the probability distribution of true assertions per output (not shown for space reasons), at the 50% mark, *there will be approximately 4-5 unique assertions per output in each 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.

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.

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. These 90 assertions constitute the additional value provided by GoldMine, which provides significant coverage of the unexplored regions of the design at this early stage.

These "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. GoldMine, a mechanical assertion generator, can explore the design space much before the manual assertion generation process arrives at that point through multiple iterations. The designers of Rigel have evaluated GoldMine's contribution as "covering a wide design space much earlier in the design cycle than typically achievable".

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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