GoldMine: A tool for enhancing verification productivity

Debjit Pal, Vibhor Dodeja, Anjana S. Kumar, Shobha Vasudevan
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, USA
Email: {dpal2, vdodeja2, anjanas3, shobhav}@illinois.edu

***Abstract*—We present GoldMine, an efficient, modular, scalable, open-source tool with a set of rich features targeting a wide spectrum of hardware design analysis. GoldMine uses multiple technologies as part of its design analysis at the Register Transfer Level (RTL). In this work, we showcase different features of GoldMine for various hardware verification use cases such as design understanding, assertion generation, debugging and bug localization, assertion ranking, and coverage analysis.**

I. Introduction

With ever increasing design complexity, hardware verification has become one of the most critical tasks of the modern hardware design cycle that is performed under aggressive time-to-market schedule, takes up to 2 years per design, and accounts for more than 70% of design costs [2]. Hardware design verification comprises multiple components including design understanding, summarizing design behavior, debugging and bug localization, test generation etc.

GoldMine [6], [7] was originally developed in 2010 [13] as an assertion generation tool in published literature to effectively automate a ubiquitous, but manual assertion writing process. Since then many other assertion generation tools have been developed in industry [10] and in academia [3], [8], [9]. In recent years, we have repurposed GoldMine by integrating a plethora of new features that have enabled new use-cases of GoldMine across a wide spectrum of verification tasks, thereby enhancing GoldMine's viability beyond its primary goal of assertion generation. Figure 1 shows the different components of GoldMine. GoldMine encapsulates design information in various data structures and analyzes them with complex algorithms. These data structures can be used independently for various verification applications beyond assertion generation. In its current implementation, we have modularized key components of GoldMine in different ways to make it useful for a versatile set of verification tasks.

Verification, in general, is a *battle of scale* and *intractable*. Over the past decades many techniques have been proposed for various verification tasks, but all of them suffers from the scalability issue. Through GoldMine, we have established that *data-driven statistical reasoning* like machine learning (ML) can tame the scalability issue and make many verification problems *tractable*. GoldMine integrates two solution spaces, statistical, dynamic techniques (such as ML) and deterministic, static techniques (such as static analysis and formal verification) to provide a solution to the different verification problem. Static analysis makes generalizations and abstractions but suffers from computational capacity. On the other hand, data-driven technique such as ML is computationally efficient but lacks perspective and domain knowledge. In [6], [7] powerful algorithms for the GoldMine were developed that could surpass human generated assertions in Register Transfer Level (RTL).

In GoldMine, the *static analyzer* comprises algorithms i) to identify *important* design variables with respect to a given output, ii) to calculate transitive symbolic functional dependencies of a variable in the context of an output, iii) to calculate sequential dependencies over design variables across multiple clock cycles per design output, and iv) to summarize relevant design information in succinct data structures. Using the information gathered by the static analyzer, the *dynamic analyzer* can generate testbenches automatically to explore random stimulus space for the design. The *assertion analyzer* can systematically identify the design statements that are in the scope of an assertion.

We outline our motivation for the versatile set of verification tasks that GoldMine can now be applied to. Each of the verification tasks can be invoked from GoldMine command-line via appropriate command-line flags.[1]

1) **Design understanding**: The *static analyzer* of GoldMine can be repurposed for *design understanding*, *design interpretation*, *identifying opportunities for design optimization*, and *removing design redundancies*. The outputs of *static analyzer* can also identify important design components with respect to a given variable. From Figure 1, it can be seen that multiple data structures that generate alternate perspectives of the same design can be generated as a part of static analysis.
2) **Summarizing design behavior**: Summarizing design behavior refers to assertion generation from simulation traces. Assertions generated from the simulation traces can generate succinct explanations of the simulation "data" and can capture the summaries of a design succinctly. Summarized design behaviors can be used as "golden" design specifications, assertions in a system-level reference model, or as guidelines for future design enhancements and optimizations.
3) **Figure of merit for assertions**: In order to enhance verification quality, it is important to use high-quality

[1]For more details, please see Section II.

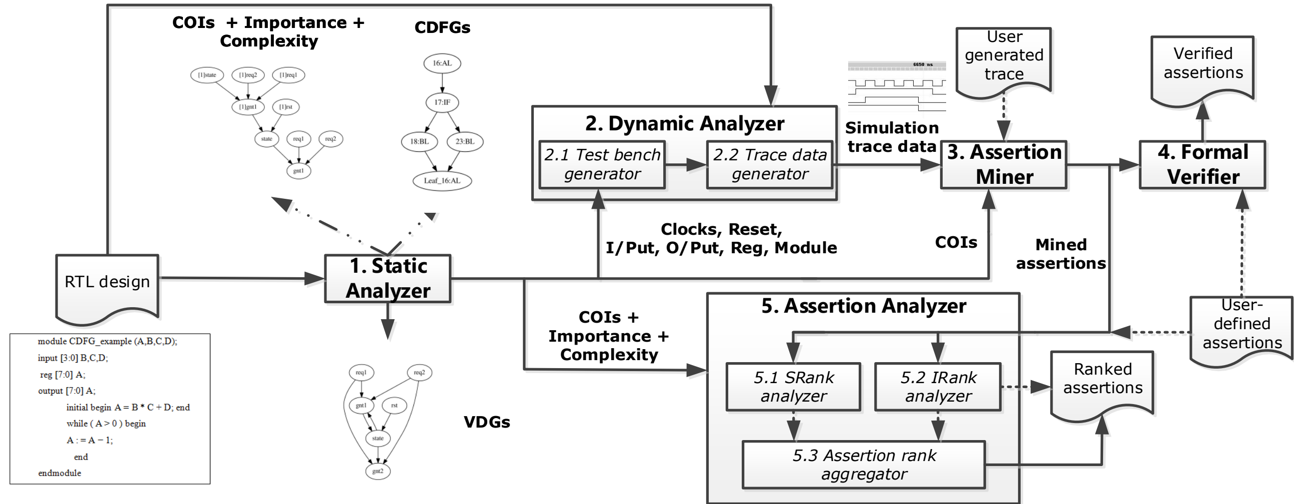


Fig. 1: GOLDMINE **architecture** – Each edge in the flow is marked with the output of the source component and the input of the destination component. A source component can generate multiple outputs to be used by different destination components. **VDG**: Variable dependency graph, **CDFG**: Control data flow graph, and **COI**: Cone of influence.

assertions that can capture subtle and important design behaviors. To automatically identify such high-quality assertions, constructing a quantitative figure of merit is a critically important problem. We have described a ranking strategy for assertions based on several goodness metrics [4], [5], [11]. We have implemented this ranking scheme in the *assertion analyzer*. The ranked list of assertions accelerates verification closure by identifying *the most valuable assertions* for verification tasks.

4) **Ranking user-generated/manual assertions**: While GOLDMINE generated assertions are ranked using the assertion analyzer, user generated assertions can also be ranked using the current implementation of GOLDMINE. We ensure this by combining the *static analyzer* and the *assertion analyzer* modules, such that they can function independently as a ranking engine.
5) **Coverage analysis**: While it is important to summarize design behavior in terms of assertions, it is equally important to know the design statements that are covered by an assertion. In [1] we have developed a notion of *correctness-based statement coverage* in RTL for an assertion. This is one of the first attempts to rigorously define assertion coverage in RTL. We have incorporated correctness-based statement coverage in GOLDMINE as part of *SRank Analyzer*. Intuitively, SRank Analyzer identifies the statements that are in the scope of an assertion *i.e.*, the design statements that needs to be executed for an assertion to be true.
6) **Debugging and bug localization**: Debugging and bug localization is an important verification task to ensure that an implementation of a hardware design follows the specification. Much like summarizing design behaviors in terms of assertions, one can summarize failing simulation traces by mining bug symptoms in the form of assertions. Just as assertions cover statements in the design, statements covered by the bug symptom can be viewed as being in the scope of the symptom. The collective set of symptoms for a design output will then correspond to localized buggy code zones [12] that are the most suspicious for debugging a bug in that design output.

Apart from the above mentioned use cases, the different technologies in GOLDMINE can aid in writing directed and constrained random testbenches, to identify missing design-intent, and design hooks for efficient debug and diagnosis.[2]

II. USE CASES OF GOLDMINE

The complete GOLDMINE tool flow is shown in Figure 1. In the same figure, we have annotated the different new capabilities that are integrated into GOLDMINE to broaden its use cases. In this section, we explain different verification use cases of GOLDMINE as referred in Section I. We use Verilog code of a 2-port arbiter of Figure 2a as a running example. We have used a `__BUG__` pragma directive to introduce a bug in L19 of the same design to explain debugging and bug localization.[3]

A. Design understanding

For design understanding, the static analyzer can be invoked from command-line via `-S`/`--static_dump` argument. The static analyzer captures various design information in multiple

[2]The GOLDMINE implementation with all the above mentioned technologies has been made available at https://goldmine.csl.illinois.edu.

[3]For brevity, we have skipped details of tool usage. The details can be found at https://bitbucket.org/debjitp/goldminer/src/master/README.

data structures *e.g.*, control-data flow graph (CDFG), variable dependency graph (VDG), cone-of-influence (COI), variable definition chain, and variable use chain.

A **control-data flow graph** (CDFG) is a directed acyclic graph (DAG) for each procedural block in the Verilog design. Each node in the DAG represents a Verilog construct. A directed edge represents control/data flow from source node to destination node. Multiple such CDFGs can be fused to construct the complete CDFG of a Verilog design. Figure 2b shows the complete CDFG of the 2-port arbiter in which `9:AL`, `11:IF`, `12:NS`, `14:NS`, and `Leaf_9:AL` forms the CDFG for the procedural block at L9 of the 2-port arbiter.

A **variable dependency graph** (VDG) is a weighted directed graph summarizing dependencies among design variables. Each node represents a design variable and each edge represents a control/data dependency among a pair of design variables. The edge weight refers to the number of times a particular variable dependency observed in a design. Figure 2c and Figure 2d show the VDG for the non-buggy and buggy 2-port arbiter design respectively.

A **cone of influence** (COI) is a DAG per output variable capturing its dependencies on other design variables across multiple design cycles. Each node of a COI is a cycle annotated design variable where the cycle number is prefixed with the variable name. The COI encompasses both data and control dependencies of an output on other variables. Figure 2e and Figure 2f show the COI for the primary output `gnt1` of the non-buggy and buggy 2-port arbiter design respectively.

A **variable definition chain** (VDC) is a data structure to record all definitions of a design variable along with its control and data dependencies. A **variable use chain** (VUC) is a data structure to record all usages of a design variable. Figure 3 shows VDC and VUC for the `state` variable.

B. Summarizing design behavior

To summarize design behaviors, assertion generator can be invoked using `-m`, `-c`, `-r`, `-I`, `-u`, `-F`, and `-V` command-line options. Figure 4a shows two assertions for the primary output `gnt1` of the 2-port arbiter.

C. Figure of merit for assertions

To identify high-quality assertions, we define two metrics importance and complexity [5], [11].

The **importance** of an assertion estimates its ability to cover important execution paths between satisfaction of its antecedent and consequent while the **complexity** of an assertion estimates its ability to cover complex behaviors that would require reasoning across multiple clock cycles. We combine importance and complexity to calculate an assertion's final rank score such that *an assertion that captures important design behavior in least complex way is ranked higher*.

The IRank analyzer of Figure 1 computes the importance and complexity of each of the assertions. IRank analyzer is invoked automatically whenever the assertions are generated as demonstrated in Section II-B. Figure 4a shows assertion importance , assertion complexity, and assertion rank score for two assertions for the 2-port arbiter.

D. Coverage analysis

For coverage analysis, we use SRank analyzer of Figure 1. The SRank analyzer can be invoked via `-a` command-line option. When SRank analyzer is invoked, GoldMine automatically computes an assertion's importance and complexity score (c.f. Section II-C) and an assertion's rank based on assertion's statement coverage. Finally, the assertion rank aggregator aggregates importance/complexity-based rank and the statement coverage-based rank to generate a final ranking for a set of assertions.

E. Debugging and bug localization

The combined outputs of *static analyzer*, *assertion miner*, and *assertion analyzer* of GoldMine aid in design debugging and bug localization. A verification engineer can use valuable data structures from the static analyzer such as CDFG, VDG, and COI to understand complex and subtle design bugs that require temporal reasoning. In addition to that, the assertion miner can be used to summarize failing traces in terms of temporal assertions that are representative of bug symptoms. The SRank analyzer component of assertion analyzer can be used to identify the design statements that are in the scope of those bug symptoms. Such automation-assisted localization can facilitate much of the manual debugging procedure.

In Figure 2a, when the `_BUG_` pragma is enabled, the CDFG remains unchanged as that of Figure 2b. However the inspection of the VDG (c.f. Figure 2d) and the COI of `gnt1` (c.f. Figure 2f) show that the intended dependence of `gnt1` on `req2` is not present, thereby the design is failing to capture the "arbiting" design intent of the arbiter. On simulating the buggy design, the monitors for the primary output `gnt1` failed. When we summarize the failing traces of `gnt1`, we find that the failure symptoms in Figure 4b shows a trivial relationship between `req1` and `gnt1` that refutes design intent, thus localizing the root cause of the failure.

III. Conclusion

GoldMine generates multiple design perspectives as a part of the assertion generation task. It does so, by combining two diverse technologies- static analysis and data-driven statistical reasoning in complementary, synergistic ways. These perspectives are of high value to a verification engineer. While we have outlined some example verification use cases that can use the different components of GoldMine, the algorithms and data structures in GoldMine can be creatively employed to a wide range of verification and design analysis applications.

References

[1] V. Athavale, S. Ma, S. Hertz, and S. Vasudevan. Code coverage of assertions using RTL source code analysis. *Design Automation Conf. (DAC)*, 2014.

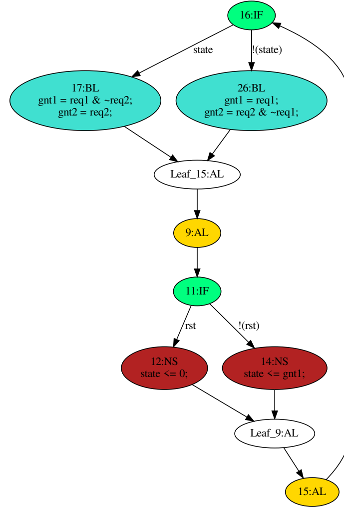
[2] W. Chen, S. Ray, J. Bhadra, M. S. Abadir, and L. Wang. Challenges and trends in modern soc design verification. *IEEE Des. Test*, 2017.

[3] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. *Design, Automation, and Test in Europe (DATE)*, 2015.

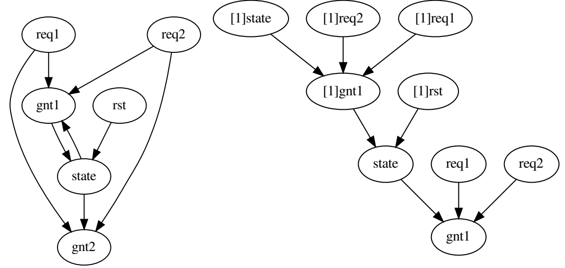
[4] S. Hertz. Enhancing quality of assertion generation: Methods for automatic assertion generation and evaluation. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL USA, 2013.

```verilog
1   `define _BUG_
2   module arb2(clk,rst,req1,
3   req2,gnt1, gnt2);
4   input clk, rst;
5   input req1, req2;
6   output gnt1, gnt2;
7   reg state;
8   reg gnt1, gnt2;
9   always@(posedge clk or
10    posedge rst)
11    if(rst)
12      state <= 0;
13    else
14      state <= gnt1;
15  always@(*)
16    if (state)
17    begin
18      `ifdef _BUG_
19      gnt1 = req1;
20      `else
21      gnt1 = req1 & ~req2;
22      `endif
23      gnt2 = req2;
24    end
25    else
26    begin
27      gnt1 = req1;
28      gnt2 = req2 & ~req1;
29    end
30  endmodule
```

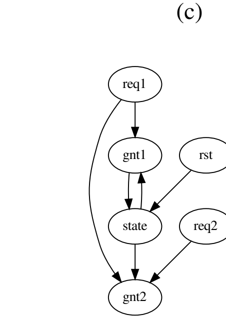
(a)



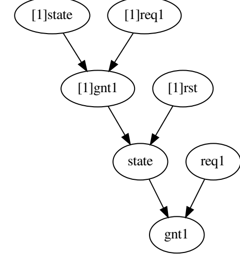
(b)

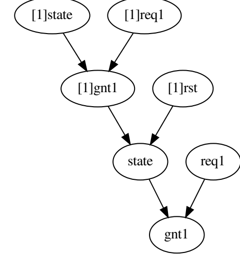


(c) (e)



(d)





(f)

Fig. 2: **Various outputs of GoldMine on a arbiter design** – (a) Verilog code for 2-port arbiter with a _BUG_ pragma. When _BUG_ pragma is defined, line 19 will be included during design compilation whereas when _BUG_ pragma is undefined, line 21 will be included during design compilation. (b) CDFG of 2-port arbiter with _BUG_ undefined. (c) VDG with _BUG_ undefined. (d): VDG with _BUG_ defined. (e) COI of `gnt1` with _BUG_ undefined. (f) COI of `gnt1` with _BUG_ defined.

Var	Var_def_chain
'state':	'CDeps': [[[], ['rst']], [[], ['rst']]], 'CLines': [[None, '11:C'], [None, '11:C']], 'Clocked': True, 'DDeps': [[], ['gnt1']], 'DLines': ['12:D', '14:D']

(a)

Var	Var_use_chain
'state':	'DefVars': [None, None] 'Lines': ['17:C', '17:C'] 'Sensitivities': [['state'], ['state']]

(b)

Fig. 3: **Static analysis data structures** – (a): variable definition chain for variable `state`, **CDeps**: control-dependencies of the variable, **CLines**: Line number of control-dependencies, **Clocked**: Indicates if procedural block is clock sensitive, **DDeps**: Data-dependencies of the definition, **DLines**: Line numbers of data-dependencies. (b): variable use chain for variable `state`, **DefVars**: Usages in variable definitions, **Lines**: Line numbers of usages, **Sensitivities**: All usages of a variable.

ID	Assertion	I	C	RS	R
1.	$(req2 == 1 \wedge state == 1)$ ##1 $(req1 == 1) \rightarrow (gnt1 == 1)$	3.30	13	0.254	1
2.	$(req1 == 1 \wedge req2 == 0) \rightarrow (gnt1 == 1)$	0.72	5	0.144	4

(a)

ID	Assertion	I	C	RS	R
1.	$(req1 == 1) \rightarrow (gnt1 == 1)$	0.505	4	0.126	1
2.	$(req1 == 0) \rightarrow (gnt1 == 0)$	0.505	4	0.126	2

(b)

Fig. 4: **Comparing assertions for the target variable** `gnt1` – (a) Assertions for non-buggy design. (b) Symptoms for buggy design. **I** Assertion importance score. **C**: Assertion complexity score. **RS**: Assertion rank score. **R** Assertion rank.

[5] S. Hertz, D. Pal, S. Offenberger, and S. Vasudevan. A figure of merit for assertions in verification. In *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2019.

[6] S. Hertz, D. Sheridan, and S. Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2013.

[7] L. Liu and S. Vasudevan. Automatic generation of system level assertions from transaction level models. *J. Electronic Testing*, 2013.

[8] J. Malburg, T. Flenker, and G. Fey. Property mining using dynamic dependency graphs. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2017.

[9] E. E. Mandouh and A. G. Wassal. Automatic generation of hardware design properties from simulation traces. *ISCAS*, 2012.

[10] NextOp. www.nextopsoftware.com/BugScope-assertion-synthesis.html.

[11] D. Pal, S. Offenberger, and S. Vasudevan. Assertion ranking using RTL source code analysis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2020.

[12] D. Pal and S. Vasudevan. Symptomatic bug localization for functional debug of hardware designs. *Int'l Conference on VLSI Design*, 2016.

[13] S. Vasudevan, D. Sheridan, S. J. Patel, D. Tcheng, and D. R. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. *Design, Automation, and Test in Europe (DATE)*, 2010.