A Coverage Guided Mining Approach for Automatic Generation of Succinct Assertions

Abstract—GoldMine is a tool that generates assertions using data mining and static analysis. It uses a decision tree based approach for mining assertions that does not have design coverage related feedback. The assertions are unaware of the design, over-constrained and have low expressiveness. We introduce a coverage guided mining approach for mining assertions in GoldMine. Our approach combines association rule learning, greedy set covering and formal verification.

I. INTRODUCTION

Assertions are summarized statements of design intent that are used in different phases of the design development cycle like pre-Silicon formal verification, dynamic verification, runtime monitoring and post Silicon debug[11]. Among all the solutions for ensuring robustness of hardware systems, assertions have emerged as the most popular candidate solution [12] for Register Transfer Level (RTL) design verification/validation.

Assertions are generated through an iterative, tedious manual process wherever they are used. A recently proposed solution, GoldMine[17] automates assertion generation. GoldMine integrates two solution spaces- statistical, dynamic techniques like data mining and deterministic, static techniques like formal verification. The static methods provide guidance to the data mining part, which is unaware of domain, i.e. hardware design information.

Random or directed simulations of an RTL design constitute the dynamic behavioral data for the data mining phase. The data miner, called A-miner, finds correlations between inputs and outputs to generate candidate assertions. Candidate assertions are passed to a formal verification engine along with the RTL design. The formal verification filters out the true assertions from the false ones. As mentioned in [17], the resulting assertions can be applied in a variety of ways in the verification process. The 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. The data mining algorithms of GoldMine can also be applied at system level or golden RTL, and the candidate assertions can be used for verification against other implementations. The application of GoldMine as a tool to generate differing perspectives from the designers, and as a hint provider for generating manual assertions has also been reported [17].

In GoldMine, the data mining algorithm employed is a decision tree based supervised learning algorithm. This algorithm makes choices of splitting variables while constructing a tree, based purely on statistics, like mean and error values. Although design structure information like the logic cone of an output is provided to streamline the decision tree assertions, there is no feedback to the data miner regarding the applicability of the assertions. Since data mining algorithms perform much better when they learn iteratively, the lack of a feedback mechanism assessing the quality of the generated assertions is a severe underutilization of the potential of these powerful algorithms.

In this work, we present an alternate data mining technique for GoldMine that uses coverage information to guide the assertion generation process. This algorithm uses association rule learning in combination with a greedy set covering technique to generate minimal, high coverage assertions. We define input space coverage (as defined in subsection II-C), which we use as a metric to judge the quality of an assertion. Our algorithm can be applied to temporal assertions as well as combinational assertions.

We demonstrate that our coverage guided mining algorithm addresses the major disadvantages of the decision tree based algorithm. The merit of incorporating coverage information as a guiding tool in a data mining methodology has not been shown before and is presented here. The intention of this work is not to argue for any particular application of GoldMine, but to present the value of an algorithm better than a predecessor, irrespective of the specific use case of GoldMine.

A. Motivation: Why decision tree assertions need improvement

In additional to the lack of assertion quality awareness, the decision tree has more shortcomings. Due to its faithfulness to a (binary) tree structure, it explores every value of each splitting variable. An assertion generated at a leaf node will necessarily have all the splitting variables of the previous levels of the tree. This leads to assertions that are over-constrained, or contain too many propositions (variable, value pairs) in the antecedent. Intermittent poor splitting choices during tree construction can result in irrelevant variables being added in the assertions as well.

For instance, a decision tree would create the assertion \((\text{request} \land \text{we} \land \text{rd} \land \text{branch}) \Rightarrow (\text{gnt})\), where the dependencies on the write enable and read signals are coincidental, but not causal. The desired assertion would be \((\text{request}) \Rightarrow (\text{gnt})\).\(^1\) We made a subjective ranking of generated assertions in [17]. The detailed ranking description will be present in Subsection IV-H. Over-constraining restricts behavior and reduces the input space behavioral coverage of assertions. It also decreases the readability of the assertions. Since individual decision tree assertion has low input space coverage, a large number of assertions are required to cover the design behavior. An increase in the number of assertions is an undesirable side effect, since it implies overhead, whether in pre-Silicon runtimes or in post-Silicon cost.

We present the coverage guided mining algorithm, that is intended to increase the number of succinct assertions and decrease the number of over-constrained assertions produced by GoldMine. This coverage guided association miner replaces the decision tree in the A-miner phase of the GoldMine algorithm. It uses a combination of association rule learning, greedy set covering and formal verification. In each iteration of the coverage guided mining algorithm, the association rule learning finds each assertion that has higher coverage than a specified minimum coverage. In successive iterations, the minimum coverage for each assertion is lowered.

\(^1\)GoldMine produces assertions using Linear Temporal Logic [15]. It can also output SVA as well as PSL assertions[11]. GoldMine will mine over constant size unrolling of the circuit function. We allow “next” operator\((X)\), but not the “eventually” operator\((F)\). The type of assertions will be in the form of \(a \Rightarrow c\), \(a \land b \Rightarrow XXc\) and etc.
This guarantees that the highest coverage assertions are added to the candidate assertion set in a greedy manner every iteration. In addition, a formal verifier[3] is used to verify that candidate assertions added to the solution set are true.

Algorithms based on association rule learning are typically not scalable due to their nature of finding all relations between all variables exhaustively. However, in our algorithm, we constrain the solution space of the association learning by considering only those candidates that fulfill a coverage criterion. We also require that the candidates should be true as attested by formal verification. We also use a heuristic of having minimal propositions in an antecedent for our greedy selection of high coverage candidate assertions. These restrictions sidestep the exhaustive nature of the association learning and result in an efficient, scalable approach.

Our approach produces succinct assertions, with higher expressiveness per assertion. This upgrades the value added by an assertion. Since the value added by an assertion can be quantitatively expressed as input space coverage, this algorithm iteratively refines the set of assertions until it maximizes the coverage achieved by them. The coverage guided mining algorithm, therefore, converges to a set of assertions that are few in number, but high in coverage.

Our contributions in this paper are as follows.

- We present a method to incorporate coverage feedback in the data mining phase of GoldMine assertion generation. The coverage feedback enhances learning and prediction capabilities of the data miner.
- We introduce an algorithm that is a combination of association learning, set covering and formal verification to arrive at the final set of assertions. This algorithm circumvents the exhaustive nature of the association learning by restricting the space of generated rules significantly.
- The assertions generated by the coverage guided association mining algorithm are more succinct, expressive and valuable than the assertions generated by the decision tree.

Our experimental results are shown among other designs, on large modules of the real world processor, OpenSparc T2 [6], demonstrating the scalability of our technique. We show that coverage guided association mining performs competitively against the decision tree method in terms of overall input space coverage and far better than decision trees with respect to input space coverage per assertion, number of propositions per assertion, and subjective designer rankings.

II. BACKGROUND

A. Decision Tree

Since assertions must be true in every possible execution, only the nodes with 100% confidence, i.e. true in every sample in the simulation data, are considered as candidate assertions. An example of this decision tree algorithm is shown in Figure 1 for the given design function \( z = (a \sim c) \& b \). An error(E) function picks the best splitting variable by computing the variance between target output values and the values predicted by decision variables. The predicted value on each node is the mean(M) of output values. A candidate assertion is output when the error value becomes zero. A candidate assertion is a Boolean propositional logic statement computed by following the path from the root to the leaf of the tree. In the example, ASSERT1, ASSERT2 and ASSERT3 are final candidate assertions.

B. Data Mining Concepts

Association rule mining [8] is a data mining method that finds useful correlations between items. If we apply it for signals in simulation traces of circuits, these correlation rules among signals will be natural fits for the assertions format that we are interested in. Though this algorithm has an exponential complexity in the worst case, high efficiency is achieved by applying constraints and using pruning techniques.

The set covering problem refers to a case where you have a set of elements called a universe and many sets that cover several elements, and you wish to find the minimal number of sets that cover all elements in the universe. The complexity of finding the optimally minimal set cover is NP-Complete [14]. However, there are many approximation algorithms which can find a near-optimal solution efficiently. The greedy set covering algorithm works by choosing the set that covers the largest number of uncovered element until all elements have been covered.

Gain is a data mining concept that refers to the value of adding some rule to the solution set of rules. In data mining, we only want to add a rule to our solution set if its gain is higher than any other potential rules. This concept fits well with our concept of input space coverage since we can define a notion of coverage gain. The coverage gain of a rule (assertion) refers to the change in total coverage of a set given that the rule is added to that set. The greedy algorithm of the set covering problem can be also stated with the concept of the coverage gain. The algorithm is to iteratively find the set with the maximum coverage gain.

Generating a set of assertions can be treated as the set covering problem. Let the universe be the rows of truth tables with all inputs, and the sets are assertions whose elements are the covered rows. Then, a minimum set cover becomes a minimum set of assertions. Finding a minimum set of assertions leads interesting properties that will be discussed in the experiment section.

When we want to solve the set covering problem with assertions, we first need to find all possible assertions with association rule mining, and then solve the set covering problem. However, the number of assertions are exponential to the number of input variables. If we put some thresholds in the association rule mining to get small number of assertions, it will affect the result of the set covering in the second step. Similar problems are identified in discriminative pattern mining [9]. Thus, instead of having two-step approach in which we find assertions and do set covering, we propose an iterative association rule mining integrated with the greedy set covering algorithm. Also, to restrict the number of rules, we apply several constraints. Our first constraint, as in [17], is that only rules with 100% confidence can be considered as candidate assertions for association rule mining. We now include coverage feedback as a constraint. We impose a minimum coverage gain to drastically limit the number of candidate assertions. We then gradually relax this constraint until we have reached a desired coverage value. The greedy set covering algorithm will always choose the highest coverage assertions in each iteration.
C. Input Space Coverage

Input space (or truth table) coverage is a metric which has been adopted for the purpose of evaluating a set of assertions in relation to some output. It should be noted that this metric has no relation to standard coverage metrics such as code, branch, or path coverage. The reason for this is that those metrics are used for judging the quality of a directed test suite, which means that they cannot be applied to an individual assertion. It should be noted that we are only targeting at improving the input space coverage of each individual true assertion instead of whole output assertion set. Decision tree, as a classifier, should cover the entire input space.

Assuming a truth table with all inputs, an assertion must cover a percentage of these entries for a specified output. This percentage is referred to as the input space coverage. Consider the assertion \( a_1 : (a \Rightarrow f) \). We know that this assertion must cover 50% of the truth table entries since \( a \) is equivalent to 1 in 50% of the truth table.

III. THE COVERAGE GUIDED MINING ALGORITHM

We run this algorithm to generate assertions for a specified output in a design, \( z \). The assertions will be in the format where a set of propositions describing input variables and their respectively assigned values imply that the output, \( z \) will be a certain value.

\( A_s \) is defined as the solution set of assertions. The total input space coverage of \( A_s \) is defined as \( c(A_s) \). We define \( g(A_s, A_s') \) as the input space coverage gain between two sets of assertions where \( A_s' = A_s + a \) and \( a \) is an assertion. We also define \( g_{\text{min}} \) as the minimum coverage gain. The minimum coverage gain ensures that any assertion that is mined must raise the total coverage of \( A_s \) by \( g_{\text{min}} \). We set a minimum coverage gain threshold \( g_{\text{threshold}} \) and a maximum total coverage threshold \( c_{\text{threshold}} \) which result in algorithm termination when reached. Our goal is to maximize the total input space coverage \( c(A_s) \) by maximizing the \( g(A_s, A_s') \) in each iteration while minimizing the total number of assertions and propositions in the antecedent of each assertion.

A. Algorithm Explanation

The basic flow of the algorithm is shown in Figure 2. We will apply the algorithm as it is explained to the simulation trace in Figure 3, which is the same example in the decision tree explanation in subsection II-A. We set the maximum total coverage threshold to 99% and the minimum coverage gain threshold to 1%.

The algorithm starts by initializing the \( g_{\text{min}} = 50\% \), \( A_s = \{\} \), and \( c(A_s) = 0\% \). We know that at least one proposition must be in the antecedent of the assertion which means that the maximum coverage gain must be 50%. We do not consider assertions without any propositions in the antecedent since those assertions are trivial.

In the next step, \( g_{\text{candidates}} \) is generated for \( F, P, E \) by Algorithm 1, and \( A_c \) is generated at line 4 of Algorithm 2.

```
Algorithm 1 Association Miner

\( g_{\text{candidates}}(F, P, E) \)
1: for each \( \{\text{input variable, value}\} \) pair in \( F \), \( f_i \) do
2: \( \text{if } g(A_c, A_c \cup \text{assertion}) \geq g_{\text{min}} \text{ then} \)
3: \( \text{if } \forall e_c \in E, P \cup f_i \Rightarrow \{z \} \text{ then} \)
4: \( A_c = A_c \cup \text{assertion} \)
5: \text{else if } \forall e_c \in E, P \cup f_i \Rightarrow \{z \} \text{ then} \)
6: \( A_c = A_c \cup \text{assertion} \)
7: \( \text{end else} \)
8: \( \text{gen_candidates}(F - f_i, P \cup f_i, E) \)
9: end if
10: end if
11: end for
```

Fig. 3: Dataset and function for example and candidate assertions

The algorithm is described in Algorithm 1, which is invoked in Algorithm 2. It is based on the correlation and the algorithm returns. The algorithm also returns when the coverage gain falls below the minimum coverage gain because adding more propositions to the antecedent can only decrease the coverage gain.

In line 1, \( f_i = \{a, 0\} \). The coverage gain of the assertion \( (a = 0) \Rightarrow (z = X) \) is calculated to 50% in line 2, which is equal to \( g_{\text{min}} \). At line 3, we can see that for the data in every cycle, \( e_j \), \((a = 0) \Rightarrow (z = 0)\), which means that there is a correlation between \( a = 0 \) and \( z = 0 \) which indicates a candidate assertion. The candidate assertion \( a_1 \) is added to \( A_c \), the set of candidate assertions, in line 4.

Now, back at line 1, \( f_i = \{a, 1\} \). Even though the coverage gain of assertion \( (a = 1) \Rightarrow (z = X) \) is also 50%, neither the rule \((a = 1) \Rightarrow (z = 0)\) nor \((a = 1) \Rightarrow (z = 1)\) is true for each cycle of data, \( e_j \). This means that the conditions in lines 3 and 5 are not satisfied. The algorithm recurses at line 8 with \( P = \{\{a, 1\}\} \) and \( F = \{\{b, 0\}, \{b, 1\}, \{c, 0\}, \{c, 1\}\} \).

Now the coverage gains of assertions \((a = 1 \land b = 0) \Rightarrow (z = X)\), \((a = 1 \land b = 1) \Rightarrow (z = X)\), \((a = 1 \land c = 0) \Rightarrow (z = X)\), and \((a = 1 \land c = 1) \Rightarrow (z = X)\) are each 25% since each has two propositions in the antecedent. The minimum coverage gain is never satisfied in lines 2, and the algorithm returns.

The algorithm is continued from line 1 for the remaining \{input variable, value\} pairs resulting in the candidates \( a_2 \) and \( a_3 \) being added to \( A_c \). The assertions in \( A_c \) are sorted by the number of propositions to keep the number of propositions per assertion to a minimum. In the example, the list remains unchanged since each candidate has the same number of propositions.
Algorithm 2 recalibrate_add

```
recalibrate_add(As, Ac) 
1: for all a ∈ As do 
2:   if g(As ∪ a) ≥ gmin then 
3:     Ac = Ac ∪ a 
4: end if 
5: end for
```

In the next step, recalibrate_add adds candidate assertions with coverage gain greater than or equal to \( g_{\text{min}} \) to the solution set as shown in Algorithm 2. Because coverage gain, \( g(\cdot) \) is relative to the solution set \( A_s \), as soon as the solution set changes, the coverage gain of all assertions must be recalculated based on the new solution set. For this reason, even though all assertions in \( A_c \) must have coverage gain greater than or equal to \( g_{\text{min}} \) with respect to the \( A_s \) before this function is called, the coverage gain of any assertion may decrease below \( g_{\text{min}} \) as other assertions are added to \( A_s \). Because of this, \( A_c \) must be recalibrated with regards to coverage gain of each assertion before an assertion may be added to \( A_s \).

In our example \( a_1 \) is added to the solution set, \( A_s \), since \( A_s \) remains the same as before the function was called. After adding that candidate to the solution set, the coverage gain of next candidate, \( a_2 \), is recalculated based on the new \( A_s \). Since \( A_s \) contains assertion \( a_1 \) with the antecedent \( (a = 0) \), it should be noted that the truth table entries where \( a = 0 \) and \( b = 0 \) are already covered. Therefore, the assertion \( a_2 \) with antecedent \( (b = 0) \) can only cover the truth table entries where \( a = 1 \) and \( b = 0 \), resulting in decreased coverage gain of only 25%. By the same logic, the coverage gain of assertion \( a_3 \) with antecedent \( (c = 1) \) is also reduced to 25%. Since both candidates have coverage gain less than \( g_{\text{min}} \), they are both discarded.

In the final step of the first iteration, \( A_s \) is cleared and the minimum coverage gain, \( g_{\text{min}} \), is reduced by half. In the example, \( g_{\text{min}} \) is reduced from 50% to 25%, which is still greater than the minimum gain threshold. The total coverage of \( A_s \) is 50%, which is less than the maximum total coverage threshold, \( c_{\text{threshold}} \). Since neither threshold is passed, the algorithm continues to the second iteration.

In the second iteration, gen_candidates is performed again with the reduced \( g_{\text{min}} \). This generates the following candidate assertions which are added to \( A_s \): \( a_4, a_5, a_6, a_7, a_2, \) and \( a_3 \). These candidate assertions are added to \( A_s \) and then sorted by number of propositions per assertion with resulting order is \( a_2, a_3, a_4, a_5, a_6, a_7 \).

Assertion \( a_2 \) is added to \( A_s \). The coverage gain of the remaining candidate assertions is recalculated, causing \( a_3, a_6 \) and \( a_7 \) each drop to 12.5% and \( a_4 \) drop to 0. This leaves only the assertion \( a_5 \) that remains at 25%, which is also added to \( A_s \). It should now be noted that the total input space coverage of \( A_s \) has reached 100%, which is above the total coverage gain threshold. The means that the algorithm can exit, producing the following assertions: \( a_1, a_2, \) and \( a_5 \).

It should be noted that this algorithm can be applied to temporal assertions much like in the decision tree algorithm [17]. For temporal assertions, the circuit is unrolled a user specified number of times. The number of times the circuit is unrolled is known as the lookback amount. A separate set of inputs is created for each clock cycle that the circuit is unrolled where each new set of inputs represents the value of that signal relative to the current time. For example \( a[t] \) represents signal \( a \) in the current cycle and \( a[t-1] \) represents the value of \( a \) in the previous cycle. With this data transformation, the data mining algorithm can treat the newly added signals as separate from the signals in the current time and use the same algorithm as is used on combinational signals.

B. Integration of Formal Verification

In our greedy set covering approach, we only choose candidate assertions based on coverage. Because these candidate assertions are only necessarily true with respect to a simulation trace, it is possible that a spurious assertion may be added to the solution set. Additionally, adding this spurious assertion to the solution set will prevent true assertions that cover the same input space from being added to the solution set, which adversely affect overall coverage.

Consider the example presented in subsection III-A. While \( a_5 \) and \( a_2 \) are true, \( a_1 \) is not. Even though the input space coverage of the solution set is 100%, the actual coverage is reduced to 75% since the \( a_1 \) is untrue. We want to be able to check whether any assertions are true before ever adding them to the solution set.

The solution to this problem is to integrate the formal verifiers into the algorithm to validate candidate assertion choice. We modify the recalibrate_add function to include a formal verification check in Algorithm 2. After the association rule miner produces the set of candidate assertions, the formal verifier is used to prune the false candidates while retaining the true assertions. This guarantees that any assertion that is added to the solution set is going to be true. If we use this modified algorithm on our example presented in the previous subsection, we produce the assertions \( a_2, a_5, (b = 1 \land c = 1) \Rightarrow (z = 1), \) and \( (a = 0 \land c = 1) \Rightarrow (z = 0) \) which results in 100% input space coverage.

C. Scalability

For \( N \) input variables in a given simulation trace, searching through the space of all antecedents \((3^N)\) is not scalable. In our algorithm, however, the minimum coverage gain helps guide and focus our antecedent search on important assertions. By definition of coverage gain, an assertion with \( k \) propositions in its antecedent covers at most \( \frac{N}{2^k} \) of the whole input space. In general, the number of antecedents with \( k \) propositions is \( O(\frac{N}{k}) \) and their coverage gains are at most \( \frac{1}{2^k} \). Thus, if the minimum coverage gain is \( \frac{1}{8} \), the maximum number of possible antecedents in the search space is \( O(\frac{2^k(\frac{N}{k})}{k}) = O((2N)^k) \), which is polynomial for a fixed \( g_{\text{threshold}} \). Moreover, because the consistency check between candidate assertions and simulation traces in line 3 and line 5 of Algorithm 1 can help prune search space, the actual number of antecedents searched in practice is much smaller than this theoretical bound.

Our algorithm’s scalability is only restricted by formal verification. Although formal verification technology is sensitive to state space, we find that in practice, we are able to effectively verify many modules of large designs, like the OpenSparc MMU. So far, the only module that was too large to verify is the OpenSparc L2 cache. The reason for this is that the L2 cache contains many RAM elements. In these infrequent cases, there are several options. One option is to individually verify the submodules of the limiting module. Another option is to disable formal verification of candidate assertions. The candidate assertions can then be simulated to determine if they are valid.

IV. EXPERIMENTAL RESULTS

We compare the decision tree and coverage guided methods for multiple designs. The designs used for testing include fetch_stage(fetch) and wb_stage(wb) from a many-core processor design \(^3\), b10, b13, and b15 from the ITC99 [4], icache controller(b100), dcache controller(b101), wishbone interface(b102), and exception control(b103) from the OpenRisc1200 CPU [5],

\(^3\)Actual name undisclosed for blind review
TABLE II: Sample assertions generated by using decision tree and coverage guided association mining. [dt] means the assertion is generated using decision tree based method. [arm] means the assertion is generated using coverage guided association rule mining.

<table>
<thead>
<tr>
<th>Module</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR1200 - b100</td>
<td>122</td>
<td>7</td>
<td>88</td>
</tr>
<tr>
<td>OR1200 - b101</td>
<td>163</td>
<td>11</td>
<td>1178</td>
</tr>
<tr>
<td>OR1200 - b102</td>
<td>234</td>
<td>9</td>
<td>1223</td>
</tr>
<tr>
<td>OR1200 - b103</td>
<td>596</td>
<td>9</td>
<td>3324</td>
</tr>
<tr>
<td>many-core CPU - fetch stage</td>
<td>458</td>
<td>6</td>
<td>4165</td>
</tr>
<tr>
<td>many-core CPU - wb stage</td>
<td>963</td>
<td>3</td>
<td>269</td>
</tr>
<tr>
<td>SpaceWire - SPW_FSM</td>
<td>46</td>
<td>7</td>
<td>342</td>
</tr>
<tr>
<td>SpaceWire - Receiver</td>
<td>75</td>
<td>15</td>
<td>979</td>
</tr>
<tr>
<td>SpaceWire - Transmitter</td>
<td>96</td>
<td>5</td>
<td>896</td>
</tr>
<tr>
<td>ITC - b10</td>
<td>27</td>
<td>2</td>
<td>282</td>
</tr>
<tr>
<td>ITC - b13</td>
<td>55</td>
<td>6</td>
<td>720</td>
</tr>
<tr>
<td>ITC - b15</td>
<td>534</td>
<td>994</td>
<td>57</td>
</tr>
<tr>
<td>OpenSparc - MMU</td>
<td>3393</td>
<td>16</td>
<td>66395</td>
</tr>
</tbody>
</table>

TABLE I: Characteristics of each module used for experiments and Transmitter(tran), Receiver(rcv), and SPW_FSM(spw) from SpaceWire codec[2]. We have included results for the OpenSparc T2, which is an open source industrial design. The module tested is the memory management unit (MMU). The number of inputs bits, outputs, and Gate number can be viewed in table I.

All tests were run on an Intel Core 2 Q6600 with 4GB of RAM. Each simulation trace contains 10,000 cycles of data. The parameters are configured such that the minimum support is set to 0.1%, the minimum coverage gain threshold is 0.2%, and the coverage threshold is set to 99%. Although we generate assertions for circuits’ outputs in our experiments, any intermediate signal can also be set as target for assertion generation.

A. Input Space Coverage as a Function of Iterations

In the first experiment, we show the number of iterations the algorithm takes to converge. The results for this experiment are taken from the OR1200 data cache controller module. The results are shown in Figure 4. It is clear that there is a logarithmic increase in input space coverage each iteration since the minimum gain is decreased in each cycle.

B. Runtime and Memory Requirements of our Algorithm

For runtime, we recorded the time when the algorithm starts to the time that the algorithm exits as defined in Figure 2. Formal verification is enabled in this test. To record the maximum memory usage, we used the Massif tool in the Valgrind [7]. The runtime and the maximum memory usage are shown in table III. It should be noted that the maximum memory usage is the peak value instead of the sum value. It always happens when the simulation traces are read into memory for mining in our experiments even if the formal verification is integrated into the algorithm.

Though the runtime of the coverage guided mining algorithm is not as fast as the decision tree [17], the tool is still very scalable, even with formal verification enabled. If runtime is a concern, the formal verification can be disabled. This produces assertions much more quickly although there will be no feedback on the validity of the candidate assertions. Maximum memory usage is also very low. This is due to memory usage scaling with the size of the simulation trace (inputs × number of cycles). If a bigger simulation trace is used, the maximum memory usage will increase linearly with the number of cycles.

C. Sample assertions

In this experiment, we show some sample assertions for several target outputs. In table II, it can be observed that coverage guided association mining tends to use fewer number of variables to imply the target output comparing with decision tree method. The variables in bold in table are unnecessary and also lower the input space coverage of the assertion since other variables have already been able to decide the target output. Decision tree always adds such unnecessary variables in assertions due to its tree structure. From assertion (3) and (4), the variables eoc, S1[2] and S1[1] have been in decision tree and then variable send_data is also added to the tree as new decision variable. As a result, the output assertion always has to include the variables eoc, S1[2] and S1[1]. Although it is a true assertion, single variable send_data is already able to decide the output. Association rule mining, however, uses coverage to guide the variable selection and thus is able to generate more succinct assertion.

D. Comparison of Succinctness of Assertions

Since a primary intent of the coverage guided mining algorithm is to improve assertion quality, we compare the average number of
propositions in the antecedent between the two algorithms. A low number of propositions in the antecedent indicates a high input space coverage and also means that the assertion is more concise and thus easier to read by a human. The results of the test are shown in figure 5. These results show that the coverage guided mining algorithm produces a lower average number of propositions in every module tested.

E. Comparison of Information per Unit: Average Input Space Coverage per Assertion

It is interesting to see what the average input space coverage per assertion is. This metric is based on the total input space coverage divided by the number of assertions in the set. The results in figure 6 show that the coverage guided algorithm produces higher coverage assertions than the decision tree method.

![Fig. 6: Comparison of the average input space coverage per assertion using each algorithm. High input space coverage shows more information per assertion. In some cases, the coverage guided mining algorithm assertions have average coverage per assertion up to 20-30% more than the decision tree algorithm.](image)

**F. Comparison of Number of Assertions Triggered in Directed Tests**

In this experiment, we produce a set of assertions for the fetch_stage and wb_stage of the many-core processor design. We then run the directed test suite created by the designers to determine how many assertions are triggered. If an assertion is triggered, it indicates that the assertion is checking behavior that would be likely to occur in a realistic environment. The results of this test are shown in figure 7.

![Fig. 7: This chart compares both algorithms in terms of the percentage of assertions triggered in the many-core CPU directed test suite. Assertions generated by coverage guided mining are triggered at least once, meaning that they are more likely to be triggered in a realistic environment than those generated by the decision tree algorithm.](image)

G. The Final Test: Subjective Designer Rankings

For this experiment, we generated assertions for the fetch_stage and wb_stage of the many-core processor design and then asked a designer to rank a set of assertions generated by the decision tree method and a set generated by the coverage guided mining method. The designer was not informed of the difference between the two sets. The rankings were assigned from 1-3 as described below.

1) Uninteresting assertion, would not use in the design
2) True design constraint, might use to test the design
3) Captures subtle design intent, would definitely use to test the design

The results in figure 8 show that the coverage guided algorithm produces a much higher percentage of rank 3 assertions than the decision tree algorithm. Any assertions that were good, but included more propositions in the antecedent than necessary were reduced from a rank 3 to a rank 2, which was the case for many decision tree assertions. Overall, the designer commented that he would use the set of assertions generated by the coverage guided method over the assertions generated by the decision tree method.

![Fig. 8: This chart shows the subjective ranking by a designer of the set of assertions generated by each algorithm. All datapath assertions were considered a rank 1 by this designer because he did not consider them valuable. The coverage guided mining algorithm produces a significantly higher percentage of assertions which are at rank 3, which was the original motivation of the technique.](image)

V. RELATED WORK AND CONCLUSIONS

Assertion generation has been studied in the context of static analysis of RTL source code as in [18]. Dynamic analysis to generate assertions has been studied in [13], [16], [10], [1]. However, these techniques do not use data mining and the generated assertions are typically low level invariants. While we would like to compare GoldMine against these other methods, it is not possible since there are no publicly available hardware assertion generation tools for us to compare against. Regardless, the assertions generated with these methods are so low level that they can not be compared to GoldMine assertions.

The analysis and experiments show that the coverage guided mining algorithm is superior in quality to the decision tree algorithm in GoldMine. The results show that our method will produce a high coverage, near minimal set of high quality assertions. The coverage guided mining algorithm is the next step towards high quality automatic assertion generation.

REFERENCES