Mining High Quality Assertions Using Best-Gain Decision Forests

***Abstract*—We introduce the Best-Gain Decision Forest algorithm, an assertion mining methodology that generates high quality assertions. Our methodology uses static analysis and a novel machine learning technique to mine assertions from register-transfer level (RTL) simulation traces. Our machine learning technique is inspired by decision tree algorithms and generates concise, high coverage RTL assertions. The Best-Gain Decision Forest algorithm induces assertions from all decision trees optimized for maximum gain and uses a set containment algorithm to minimize redundancy. We show that our methodology generates assertions with up to 2 fewer propositions and 10% greater functional coverage than those generated by existing methodologies.**

I. Introduction

Verification accounts for the majority of hardware design effort. Among many verification methodologies, assertion-based verification is becoming increasingly popular [1]. *Assertions* express intended design behavior and are used in both pre and post-silicon validation [2].

Although assertion-based verification is an effective verification methodology, generating high-quality assertions is non-trivial. Verification engineers spend a considerable amount of time manually generating concise, high-coverage assertions. Recently, methodologies have been proposed to automatically generate assertions at the register transfer level (RTL) [3]–[12]. The generated assertions can be used to expose subtle RTL design bugs or guide the design's evolution.

A number of assertion generation methodologies use machine learning algorithms to extract assertions from RTL simulation traces [**?**], [5], [7], [9]–[12]. Machine learning algorithms statistically analyze observational data to learn a mathematical model that uses a set of feature variables to predict the value of a target variable. Assertion generation methodologies use static analysis techniques to select a set of feature variables from a design for a given target variable. Afterward, these methodologies will attempt to infer causal relationships between the feature and target variables from the RTL simulation traces.

Decision tree algorithms [13], [14] are ideal for assertion generation because they are simple, scalable, and represent the data in a compact and intuitive way [15]. Decision tree learning algorithms perform a greedy search to quickly identify local regions in the data space. These algorithms recursively partition the data space by assigning values to feature variables until the value of the target variable is consistent within a data subspace. Assertion generation methodologies that use decision tree algorithms generate both propositional and temporal assertions [10], [11].

To illustrate, consider the example in figure 1. Initially, f is not consistent within the data space. Therefore, we cannot generate an assertion that predicts the value of f. However, whenever $a = 0$, $f = 0$. To infer this relationship, a decision tree algorithm partitions the data space into two data subspaces. Since the value of f is consistent in the first data subspace, the decision tree algorithm generates the assertion `~a |-> ~f`[1]. In the second data subspace a casual relationship between variables a and f does not exist. In such cases, the decision tree learning algorithm will continue to partition the data space.

Since decision tree algorithms partition the data space in a hierarchical manner, they can introduce irrelevant propositions into a set of assertions. Consider the assertion `a && ~b |-> ~f` generated in the previous example. Regardless of the value of a, whenever $b = 0$, $f = 0$. Therefore, the variable a is irrelevant. Decision tree learning algorithms often generate such verbose and overconstrained assertions. These assertions are often difficult to comprehend and have low functional coverage. Consequently, they may be discarded by verification engineers.

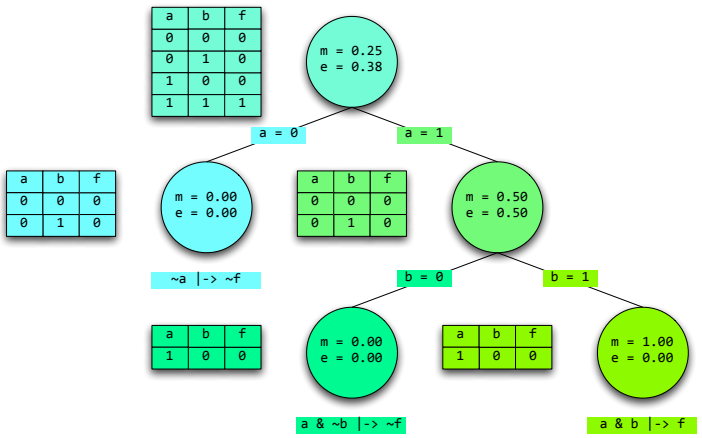

Fig. 1: A decision tree. Each node is pictured with the data subspace it represents, and is labeled with its mean and error. Each branch is labeled with the variable and value used to partition the data subspace represented by its parent.

The random forest algorithm in [17], [18] constructs multiple decision trees in randomly selected data subspaces. In [19], the authors propose a methodology to efficiently construct a *decision forest* by adding nodes shared by multiple trees only once. These methodologies construct decision tree forests to avoid overfitting the data space. However, neither addresses how to induce rules from such a decision forest. In addition, these methodologies rely on randomization or parameters to construct a decision forest.

In this work, we introduce the *Best-Gain Decision Forest algorithm* to generate concise, high-coverage RTL assertions. The Best-Gain Decision Forest algorithm is inspired by the decision tree algorithm. Decision tree algorithms compute the gain of each feature variable to determine which will best partition the data space. When multiple feature variables partition the data space equally well, decision tree algorithms select one such variable to partition the data space. For example, in figure 1 the decision tree algorithm initially selects a to partition the data space despite the fact that b partitions the data space equally well. The Best-Gain Decision Forest algorithm partitions the data space using all such variables. Consequently, our algorithm builds all decision trees optimized for maximum gain. To maintain efficiency, our algorithm adds nodes shared between multiple decision trees only once. Our algorithm extracts a complete set of assertions from the decision forest and uses a set containment algorithm to generate a minimal set of concise, high coverage assertions.

Previous methodologies have used static analysis techniques such as the cone of influence [20] to reduce the search space of decision tree algorithms. In this work, we apply the bounded cone of influence [21], [22] to this problem. The bounded cone of influence includes only variables on which the consequent of an assertion depends within a finite amount of time. Since our methodology generates assertions within a fixed temporal window [10], [11], this method is very effective in focusing the Best-Gain Decision Forest algorithm further.

We present experimental results that demonstrate the effectiveness of our methodology. Our experimental results show that the Best-Gain Decision Forest algorithm is able to generate a minimal set of concise, high coverage assertions.

[1] In this work, we present assertions using the SystemVerilog Assertion language [16].

We show that our methodology generates assertions with up to 2 fewer propositions and 10% greater functional coverage than those generated by the methodology in [10] on average. In addition, our methodology generates sets of assertions with up to 45% greater functional coverage than equivalent sets generated by the methodology in [10].

Our contributions in this work are as follows.

- We introduce the Best-Gain Decision Forest algorithm, which builds a decision forest without relying on randomization or parameters.
- We introduce an algorithm to infer a minimal set of concise rules from a decision forest.
- Our methodology automatically generates concise, high coverage RTL assertions with high readability.
- Our methodology uses the bounded cone of influence to improve the quality of automatically generated assertions.

II. Background

In this section we give a brief overview of the work necessary to understand our methodology

A. Machine Learning

Machine learning algorithms statisically analyze observational data to build mathematical models [23]. Rule induction machine learning algorithms induce a set of rules which predict the value of a *target variable*. Variables used to predict the value of the target variable are *feature variables*. Typically, machine learning algorithms use heuristics to select feature variables from the data space.

B. Decision Tree

Decision tree algorithms are rule induction machine learning algorithms. These algorithms recursively partition the data space by assigning values to feature variables until the value of the target variable is consistent within a data subspace. Decision tree algorithms use decision trees to represent the dataset. A decision tree is a multary tree that consists of branch and leaf nodes. Each node represents a data subspace while each edge represents a partition in the data subspace of its parent.

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

C. Assertions

We define assertions as follows. Let $\mathbf{B} = \{0, 1\}$ denote the set of Boolean values and let $b \in \mathbf{B}$ denote an arbitrary Boolean value. Let $\mathbf{V}$ denote the set of feature variables and let $v_t \in \mathbf{V}$ denote the target variable. Finally, let *proposition* $p_i = (v_i \in \mathbf{V}, b)$ denote a variable-value pair that assigns a Boolean value to variable v_i, and let $\mathbf{A}$ denote a set of propositions.

We let the formula $\mathbf{A} \implies (v_t, b)$ define an assertion. The *antecedent* of the assertion is defined by conjoining the propositions in the set $\mathbf{A}$, while its *consequent* is defined by the proposition (v_t, b). For example, consider the assertion $a_0 = \{(v_0, 1), (v_1, 1)\} \implies (v_t, 1)$. The assertion a_0 states "*if* v_0 *is equal to* 1 *and* v_1 *is equal to* 1*, then* v_t *is equal to* 1."

D. Input Space Coverage

We define *input space coverage* to measure the functional coverage of an assertion as follows. Consider a truth table that computes the value of the target variable as a function of the feature variables. We say that an assertion covers an entry in such a truth table if the values of the feature variables in that entry satisfy the antecent of the assertion. The input space coverage of an assertion refers to the percentage of truth table entries covered by that assertion.

III. Methodology

Our methodology has three phases. The first phase uses a static analysis technique to select a subset of feature variables for a user-selected target variable. In the second phase, the Best-Gain Decision Forest algorithm uses the selected feature variables to construct all decision trees optimized for maximum gain and generates all assertions from them. The third phase uses a set containment algorithm to minimize redundancy in the generated assertions.

A. Bounded Cone of Influence

Our methodology uses the bounded cone of influence to statically analyze a design and select a subset of feature variables for a user-selected target variable. The bounded cone of influence is an extension of the classic cone of influence reduction technique employed by most unbounded model checking algorithms. The classic cone of influence is a special case of the localization reduction in [20]. The classic cone of influence uses an RTL dependency graph to transitively compute which variables can affect the consequent of an assertion. Variables that do not affect the consequent of the assertion are used to prune the state space. The bounded cone of influence is similar, but transitively computes which variables can affect the consequent of an assertion for a fixed number of time steps. Since our methodology unrolls the RTL design for a fixed number of time steps, we use the bounded cone of influence to prune the set of feature variables.

B. Best-Gain Decision Forest Algorithm

Algorithm 1 Best-Gain Decision Forest Algorithm

```
1:  procedure decision_forest(V, E, P)
2:      m ← mean(v_t, E)
3:      e ← error(v_t, E)
4:      if e = 0 then
5:          A ← A ∪ (P ⟹ (v_t, m))
6:          return
7:      end if
8:      S ← {∅}
9:      g_best = −∞
10:     for all v ∈ V do
11:         g ← e − error(v_t, E_{v=0}) − error(v_t, E_{v=1})
12:         S ← S ∪ (v, g)
13:         if g > g_best then
14:             g_best ← g
15:         end if
16:     end for
17:     for all (v, g) ∈ S do
18:         if g = g_best then
19:             decision_forest(V \ v, E_{v=0}, P ∪ (v, 0))
20:             decision_forest(V \ v, E_{v=1}, P ∪ (v, 1))
21:         end if
22:     end for
23: end procedure
```

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

The Best-Gain Decision Forest algorithm requires the functions *mean* and *error*, which are defined as follows. The $mean(v_i, \mathbf{E})$ function computes the mean value of v_i using the vectors in $\mathbf{E}$. The $error(v_i, \mathbf{E})$ function computes the absolute deviation of v_i from its mean value. We let $\mathbf{S}$ denote a set of variable value pairs such that $(v, g) \in \mathbf{S}$ assigns variable v a gain value g.

a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 2: A set of Boolean vectors

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

If the error of $v_t \neq 0$, then the algorithm determines the best variables to partition the data space. For each variable $v_i \in \mathbf{V}$, the algorithm computes its gain g_i and adds the pair (v_i, g_i) to the set $\mathbf{S}$. While computing the gain of each variable, the algorithm records g_{best}, the highest gain.

The algorithm recurses after computing $\mathbf{S}$ and g_{best}. For each $(v_i, g_i) \in \mathbf{S}$ with $g_i = g_{best}$, the algorithm makes two recursive calls. Both the first and second call remove v_i from $\mathbf{V}$. The first call removes vectors from $\mathbf{E}$ where $v_i = 1$, and adds the proposition $(v_i, 0)$ to $\mathbf{P}$, while the second call removes vectors from $\mathbf{E}$ where $v_i = 0$, and adds the proposition $(v_i, 1)$ to $\mathbf{P}$. In other words, these calls partition the data space so that one partition includes data where $v_i = 0$, and the other partition includes data where $v_i = 1$.

C. Set Containment

Our methodology minimizes redundancy in the generated assertions using a set containment algorithm. We say that assertion a_i contains assertion a_j if the antecedent of a_i is a subset of the antecedent of a_j and their consequents are equal. That is to say, if a_i contains a_j, then a_i captures the same behavior as a_j in a more concise manner. Consequently, each assertion need only be compared with those that have a greater number of propositions in their antecedents. In other words, assertion a_i should remove assertion a_j only if the size of a_i is less than or equal to the size of a_j.

D. Example

In this section we present an example of the Best-Gain Decision Forest algorithm. Let $\mathbf{V} = \{a, b, f\}$ and let $v_t = f$. Let $\mathbf{E}$ be the set of Boolean vectors in figure 2. Based on the figure, we can see that $f = a \wedge b$.

The Best-Gain Decision Forest algorithm begins by computing the mean and error of f in $\mathbf{E}$. Upon inspection, we can see that the mean of $f = 0.25$. The error of f is computed as follows. First, for each vector in $\mathbf{E}$, the value of f is subtracted from the mean value of f. Next, the absolute value of these differences is summed. Finally, the sum is divided by $|\mathbf{E}|$. In this example, the error of $f = 0.375$. Since the error of $f \neq 0$, no assertions are added to $\mathbf{A}$ and the algorithm continues.

Next, the algorithm computes the gain of each variable $v_i \in \mathbf{V}$. We begin with variable a. When $a = 0$, the mean and error of f are both 0. When $a = 1$, the mean and error of f are both 0.5. Therefore, the gain of $a = -0.125$. However, since using a to partition the data space will add an assertion to $\mathbf{A}$, we explicitly maximize its gain by setting it equal to ∞.

Next, the algorithm computes the gain of the remaining variables in $\mathbf{E}$. Upon inspection, we see that the gain of b is equivalent to the gain of a. The algorithm does not compute the gain of f since it is the target variable. After exiting the loop on line 16, $\mathbf{S} = \{(a, \infty), (b, \infty)\}$ and $g_{best} = \infty$.

Next, The algorithm recurses. Since the gain of both a and b is equal to g_{best}, the algorithm will use both of them to partition the data space. Note that the decision tree learning algorithm would only select one of these variables to partition the data space. In the recursive call where $a = 0$, the error of f is also equal to 0. Therefore, the assertion $\{(a, 0)\} \implies (f, 0)$ is added to $\mathbf{A}$ and the call terminates.

In the recursive call where a equals 1, the error of $f \neq 0$. Therefore, the algorithm recurses on b since it is the only remaining variable in $\mathbf{V}$. In both recursive calls on b, the error of $f = 0$. Therefore, the algorithm adds two assertions to $\mathbf{A}$. The recursive call where $b = 0$ adds the assertion $\{(a, 1), (b, 0)\} \implies (f, 0)$ and the recursive call where $b = 1$ adds the assertion $\{(a, 1), (b, 1)\} \implies (f, 1)$.

Figure 3 depicts the decision forest constructed by the algorithm. The set $\mathbf{A}$ contains the following assertions:

$$
\begin{aligned}
a_0 &= \{(a, 0)\} \implies (f, 0) \\
a_1 &= \{(b, 0)\} \implies (f, 0) \\
a_2 &= \{(a, 1), (b, 0)\} \implies (f, 0) \\
a_3 &= \{(a, 1), (b, 1)\} \implies (f, 1) \\
a_4 &= \{(b, 1), (a, 0)\} \implies (f, 0) \\
a_5 &= \{(b, 1), (a, 1)\} \implies (f, 1)
\end{aligned}
$$

Note that the set $\mathbf{A}$ contains redundant assertions. For example, consider the assertions a_0 and a_4. Both assertions express the same implication, but a_0 is more concise. In other words, the proposition $(b, 1)$ in the antecedent of a_4 does not affect the consequent of a_4. The Best-Gain Decision Forest removes such redundant assertions using a set containment algorithm. After set containment, the set $\mathbf{A}$ contains the following assertions:

$$
\begin{aligned}
a_0 &= \{(a, 0)\} \implies (f, 0) \\
a_1 &= \{(b, 0)\} \implies (f, 0) \\
a_3 &= \{(a, 1), (b, 1)\} \implies (f, 1)
\end{aligned}
$$

E. Analysis

In this section, we analyze the complexity and properties of the Best-Gain Decision Forest algorithm. Let n denote the number of variables in the data space. In the worst case, each branch node in a decision tree at will produce two child nodes. Since the worst case depth of a decision tree is equal to n, the decision tree can have at most 2^n leaf nodes. Therefore, the worst case size of a decision tree will be on the order of $O(2^n)$.

The Best-Gain Decision Forest algorithm constructs all optimal decision trees simultaneously. In the worst case, the algorithm will partition the data space using every variable at every depth. At depth k, there are $n - k$ variables remaining to partition the data space. Hence, in the worst case, the number of unique decision trees is equal to $n!$. Therefore, the size of a decision forest will be on the order of $O(n! \cdot 2^n)$.

The Best-Gain Decision Forest algorithm will make $n! \cdot 2^n$ recursive calls at most. Each recursive call must compute the gain of each variable. Therefore, the worst case run time complexity of the Best-Gain Decision Forest algorithm is $O(n!^2 \cdot 2^n)$. We find in our experimental results that the Best-Gain Decision Forest algorithm is practically efficient despite high theoretical worst case complexity.

The Best-Gain Deecision Forest algorithm has two interesting properties. We first show that the Best-Gain Decision Forest algorithm discards only functionally redundant assertions. Suppose the algorithm generates the assertions $\mathbf{A} \implies (v_t, b)$ and $\mathbf{A}' \implies (v_t, b)$. In addition, suppose that $\mathbf{A} \supset \mathbf{A}'$. Let f be a function that evaluates to true when either $\mathbf{A}$ or $\mathbf{A}'$ evaluate to true. Since $\mathbf{A} \supset \mathbf{A}'$, and both $\mathbf{A}$ and $\mathbf{A}'$ define a conjunction, it follows from Boolean algrebra that $\mathbf{A}'$ is sufficient to satisfy f. Therefore, the assertion $\mathbf{A} \implies (v_t, b)$ is functionally redundant.

We next show that the Best-Gain Decision Forest algorithm generates an assertion that is either equivalent to or more concise than each assertion generated by the decision tree algorithm. Since the Best-Gain Decision Forest algorithm builds all optimal decision trees, it will generate every assertion generated by the decision tree algorithm. Suppose the decision tree algorithm generates the assertion $\mathbf{A} \implies (v_t, b)$ and the Best-Gain Decision Forest algorithm generates the assertion $\mathbf{A}' \implies (v_t, b)$ such that $\mathbf{A} \supset \mathbf{A}'$. We can conclude from the previous property that the Best-Gain Decision Forest algorithm generates an assertion that is either equivalent to or more concise than each assertion generated by the decision tree algorithm.

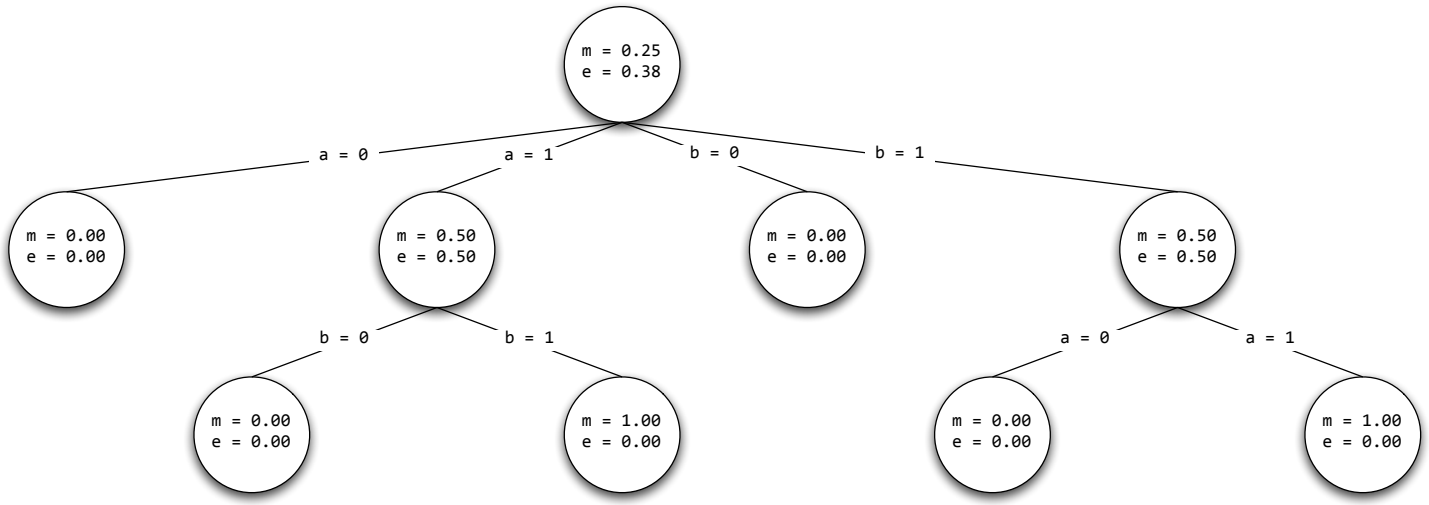

Fig. 3: A decision forest. Each node is labeled with its mean and error. Each branch is labeled with the variable and value used to partition the data subspace represented by its parent. Variables a and b are both used to partition the data space represented by the root node since they do so equally well.

IV. Experimental Results

In this section, we present experimental results. Henceforth, we will refer to the number of variables used to partition the data space in each iteration of the Best-Gain Decision Forest algorithm as the number of **partitions**. The first set of experiments examine how explicitly limiting the number of partitions affects various statistics. Note that if the number of partitions is limited to 1, then the Best-Gain Decision Forest algorithm emulates the decision tree learning algorithm. The second set of experiments examine how the bounded cone of influence and set containment algorithms affect the performance of the Best-Gain Decision Forest algorithm.

We present results for three modules selected from various open-source hardware designs. The first module is the master state machine from the Peripheral Component Interconnect (PCI) protocol. The second module is the state machine from the SpaceWire (SWR) protocol. The final module is the protocol engine from the Universal Serial Bus (USB) protocol. Table 4 presents some basic information for each design. For each experiment, we used a random test bench to generate 10000 cycles of data. In addition, we explicitly limit the depth of the decision forest to 5. We conducted all experiments using a 2.67 gigahertz quad core Intel Core i5 with 16 gigabytes of memory.

A. Average Number of Propositions Removed from an Assertion as a Function of the Maximum Number of Partitions

This experiment examines the average number of propositions removed per assertion by the Best-Gain Decision Forest algorithm as a function of the maximum number of partitions. Henceforth, we will refer to the average number of propositions removed per assertion as the average reduction per assertion. In general, we expect the average reduction per assertion to increase with the maximum number of partitions. As the maximum number of partitions increases, the number of concise assertions will increase as well. For example, if the Best-Gain Decision Forest algorithm would select n variables to partition the data at depth d, but the maximum number of partitions is explicitly limited to k, then the algorithm could potentially discard $n - k$ assertions of length $d + 1$. The discarded assertions may cover other assertions with a greater number of propositions. Therefore, we expect that the average reduction per assertion to increase with the maximum number of partitions.

Figure 5 shows how the average reduction per assertion varies as a function of the maximum number of partitions. We can see that in general, the average reuduction per assertion increases with the number of partitions. Interestingly, the average reduction per assertion reaches a maximum before the maximum number of partitions reaches an optimal value. We can explain this as follows. In general, we expect the total number of generated assertions to increase with the maximum number of partitions. Consequently, the number of assertions without reduction will increase as well. Therefore, the average reduction per assertion will decrease.

B. Average Input Space Coverage Gain of an Assertion as a Function of the Maximum Number of Partitions

This experiment examines the average input space coverage gain per assertion as a function of the maximum number of partitions. The average input space coverage gain per assertion has a strong correlation with the average reduction per assertion. In other words, as propositions are removed from an assertion, we expect its input space coverage to increase proportionally. Therefore, we expect the average input space coverage gain per assertion and the average reduction per assertion to trend similarly against the maximum number of partitions.

Figure 6 shows how the average coverage gain per assertion varies as a function of the maximum number of partitions. We can see that in general, the average coverage gain per assertion increases with the number of partitions. As expected, the average coverage gain per assertion and average reduction per assertion trend similarly against maximum number of partitions.

C. Total Input Space Coverage of a Set of Assertions as a Function of the Maximum Number of Partitions

This experiment examines the total input space coverage of a set of assertions as a function of the maximum number of partitions. We compute the total input space coverage for a set of assertions by enumerating each unique Boolean vector over the set of variables supplied to the Best-Gain Decision Forest algorithm and compute the portion covered by the assertions. Consequently, computing input space coverage for some sets of assertions is intractable because they may depend on many variables.

In general, we expect the total input space coverage of a set of assertions to increase with the maximum number of partitions. If an assertion is reduced, then we expect its input space coverage to increase. Since the average reduction per assertion increases with the maximum number of partitions, we expect the total input space coverage of the set of assertions to increase as well.

Figure 7 shows how the total coverage of a set of assertions varies as a function of the maximum number of partitions. We computed the total input space coverage of sets of assertions generated for several outputs in the pci_master32_sm module in the PCI design. We excluded outputs for which either total input space coverage was 100% regardless of the maximum number of partitions, or for which input space coverage could not be computed.

In general, we can see that the total input space coverage increases with the maximum number of partitions. Interestingly, most outputs are maximumally covered when the maximum number of partitions is equal to 5. In addition,

Design Name	Number of Lines	Number of Modules	Module Name	Number of Outputs	Number of Submodules
PCI	29443	53	pci_master32_sm	17	8
SWR	243	1	SPW_FSM	13	0
USB	7657	15	usbf_pe	19	0

Fig. 4: Statistics for the PCI, SWR, and USB designs

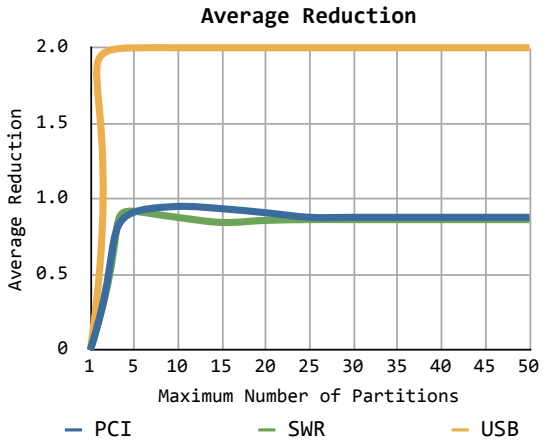

Fig. 5: The average number of propositions removed from an assertion as a function of the maximum number of partitions.

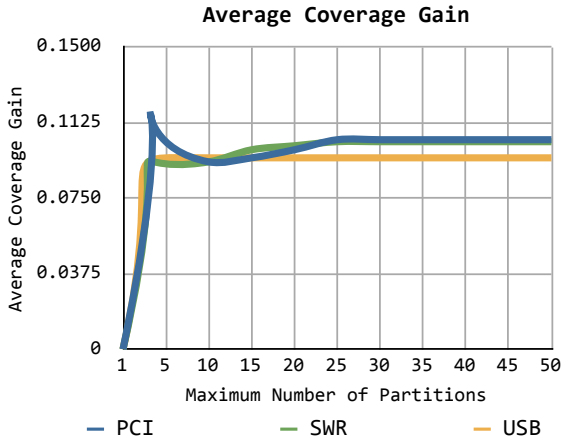

Fig. 6: The average input coverage gain per assertion as a function of the maximum number of partitions.

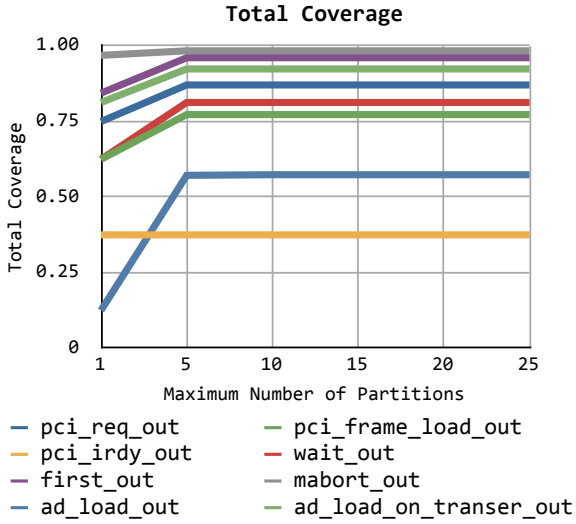

Fig. 7: The average input coverage gain per assertion as a function of the maximum number of partitions.

the total input space coverage does not decrease once it has reached a maximum value. We expect such behavior since the input space coverage of assertions can only increase with the maximum number of partitions.

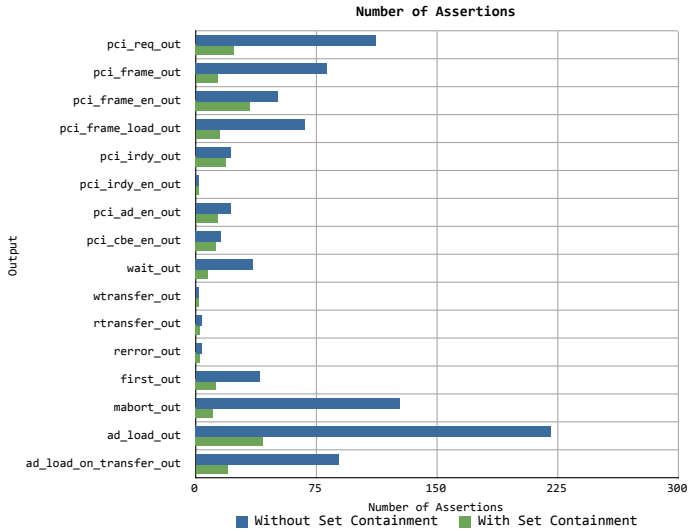

Fig. 8: The total number of assertions generated by the Best-Gain Decision Forest algorithm with and without set containment.

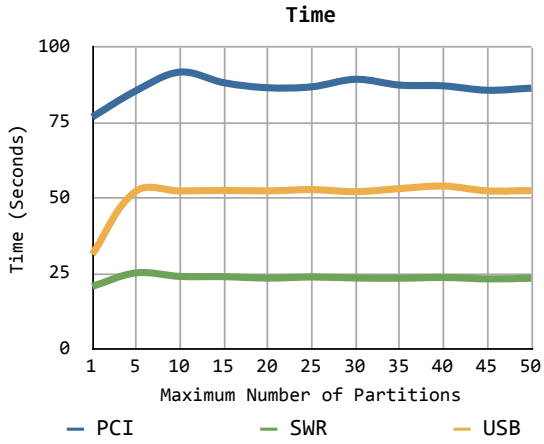

Fig. 9: The run time of the Best-Gain Decision Forest algorithm as a function of the maximum number of partitions

D. Evaluation of Set Containment

This experiment evaluates the set containment algoritm. In general, we expect the set containment algorithm to remove a large number of assertions. Figure 8 shows how the set containment algorithm affects the total number of assertions for each output in the pci_master_32 module in the PCI design. We can see that in general, the set containment algorithm removes a large number of assertions. Hence, the set containment algorithm is a crucial part of the Best-Gain Decision Forest algorithm.

It is worth noting that figure 8 excludes the output variable retry_out. We exclude retry_out because the Best-Gain Decision Forest algorithm generates a large number of assertions for the output, which makes figure 8 unreadable. The Best-Gain Decision Forest algorithm generates 2522 assertions for retry_out without set containment, while with set containment the algorithm generates 52 assertions. Such a result further justifies the inclusion of the set containment algorithm.

E. Runtime and Memory Requirements

This experiment evaluates the runtime and memory requirements of the Best-Gain Decision Forest algorithm. In

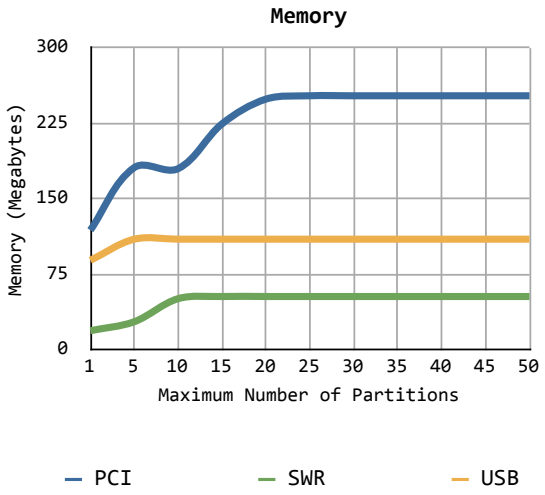

Fig. 10: The memory requirements of the Best-Gain Decision Forest algorithm as a function of the maximum number of partitions

general, we expect the run time and memory requirements of the Best-Gain Decision Forest algorithm to increase with the maximum number of partitions. The number of recursive calls and nodes in the algorithm will increase with the maximum number of partitions. Therefore, we expect the memory and run time requirements to increase as well.

Figures 9 and 10 show how the memory and time vary as a function of the maximum number of partitions. We can see that both run time and memory requirements increase until the maximum number of partitions reaches an optimal value. Though the worst case complexity of the Best-Gain Decision Forest algorithm is high, in practice, the run time and memory requirements are reasonable.

V. Related Work and Conclusions

Recently, academia and industry have proposed a number of methodologies to automatically generate RTL assertions [3]–[12]. In [7], Chang et al use sequential mining to find and infer causal relationships between frequently occurring sequences of input and output events. Since their methodology only seeks relationships between inputs and outputs, it may not generate meaningful assertions for designs with high temporal depth. In addition, since their methodology enumerates each unique input and output event, it might not scale for large designs. Since Chang et al do not present any generated assertions or quantitative measures of functional coverage, we could not properly compare our methodologies.

In [10], Vasudevan et al use static analysis, a decision tree algorithm, and formal verification to automatically generate RTL assertions. Their methodology randomly simulates an RTL design for a fixed number of cycles and uses static analysis to select a set of feature variables for a user-selected target variable. Their methodology uses a decision tree algorithm to extract assertions from the simulation traces and eliminates false assertions using formal verification. Since their methodology uses a decision tree algorithm, it might generate verbose, low-coverage assertions.

Although commercial methodologies to automatically generate RTL assertions including NextOp's BugScope [9] and Jasper's ActiveProp [12] exist, they were not available for comparison.

In this work, we presented the Best-Gain Decision Forest algorithm. We showed that the Best-Gain Decision Forest algorithm generates a concise set of high coverage RTL assertions.

References

[1] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-Based Design*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[2] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proceedings of the 8th International Symposium on Quality Electronic Design*, ser. ISQED '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 613–620. [Online]. Available: http://dx.doi.org/10.1109/ISQED.2007.38

[3] 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 *Proceedings of the 35th annual Design Automation Conference*, ser. DAC '98. New York, NY, USA: ACM, 1998, pp. 534–537. [Online]. Available: http://doi.acm.org/10.1145/277044.277188

[4] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "Iodine: a tool to automatically infer dynamic invariants for hardware designs," in *Proceedings of the 42nd annual Design Automation Conference*, ser. DAC '05. New York, NY, USA: ACM, 2005, pp. 775–778. [Online]. Available: http://doi.acm.org/10.1145/1065579.1065786

[5] X. Cheng and M. S. Hsiao, "Simulation-directed invariant mining for software verification," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 682–687. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403541

[6] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke, "Automatic generation of complex properties for hardware designs," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 545–548. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403506

[7] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ser. ASPDAC '10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 607–612. [Online]. Available: http://dl.acm.org/citation.cfm?id=1899721.1899864

[8] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 755–760. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837466

[9] (2010) Assertion synthesis. [Online]. Available: http://www.nextopsoftware.com/BugScope-assertion-synthesis.html

[10] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: automatic assertion generation using data mining and static analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010, pp. 626–629. [Online]. Available: http://dl.acm.org/citation.cfm?id=1870926.1871074

[11] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan, "Towards coverage closure: Using goldmine assertions for generating design validation stimulus," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1 –6.

[12] (2012) Activeprop assertion-based verification system. [Online]. Available: http://www.jasper-da.com/products/activeprop-assertion-based-verification-system

[13] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986. [Online]. Available: http://dx.doi.org/10.1023/A:1022643204877

[14] ——, *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[15] J. Han and M. Kamber, *Data mining: concepts and techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[16] (2012) Systemverilog. [Online]. Available: http://www.systemverilog.org

[17] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 8, pp. 832–844, Aug. 1998. [Online]. Available: http://dx.doi.org/10.1109/34.709601

[18] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1010933404324

[19] H. Zhao and A. Sinha, "An efficient algorithm for generating generalized decision forests," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 35, no. 5, pp. 754 – 762, sept. 2005.

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

[21] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu, "Verifiying safety properties of a power pc microprocessor using symbolic model checking without bdds," in *Proceedings of the 11th International Conference on Computer Aided Verification*, ser. CAV '99. London, UK, UK: Springer-Verlag, 1999, pp. 60–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=647768.733940

[22] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Form. Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, Jul. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011276507260

[23] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.