ENHANCING QUALITY OF ASSERTION GENERATION: METHODS FOR AUTOMATIC ASSERTION GENERATION AND EVALUATION

BY

SAMUEL L. HERTZ

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Assistant Professor Shobha Vasudevan
ABSTRACT

We present methods for automatically generating and evaluating register transfer level (RTL) assertions. We detail the GoldMine methodology and each of its data mining algorithms. We introduce the Best-Gain Decision Forest algorithm to mine concise RTL assertions. We develop an assertion ranking methodology. We define assertion importance, complexity, rank and ideality and we detail methods to compute each of them. We present a case study and experimental results to demonstrate the effectiveness of assertion rank. We develop an assertion rank aggregation methodology. We define assertion coverage and expectedness. We aggregate rankings for assertion importance, complexity, coverage and expectedness. We present experimental results to demonstrate the value of these metrics and the rank aggregation methodology. We rigorously analyze the performance of each data mining algorithm in GoldMine. We present experimental results that demonstrate each algorithm’s performance with respect to various metrics.
For my family and friends.
ACKNOWLEDGMENTS

I would like to thank my advisor Shobha Vasudevan. Without her encouragement, I would not have attended graduate school. Without her vision and support, GoldMine would not be the amazing tool that it is today. She always motivated me to elevate my ideas and research.

I would like to thank Dave Sheridan. Dave was very supportive and patient while I was translating GoldMine from Java to C++. Without Dave, my inheritance of GoldMine would have been far more difficult.

I would like to thank Viraj Athavale. Viraj’s assertion coverage implementation greatly strengthened GoldMine and helped inspire assertion ranking. His insight was invaluable while I developed GoldMine’s Verilog parser and static analyzer.

Finally, I would like to thank Lingyi Liu. Our countless conversations regarding research and graduate school proved invaluable. Many of his ideas have significantly improved GoldMine.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Assertion-Based Verification</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Automatic Assertion Generation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Assertion Evaluation</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>3</td>
</tr>
<tr>
<td>CHAPTER 2 ASSERTION GENERATION</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Related Work and Background</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Methodology</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Decision Tree</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Best-Gain Decision Forest</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Coverage Mining</td>
<td>18</td>
</tr>
<tr>
<td>2.6 Prism</td>
<td>18</td>
</tr>
<tr>
<td>2.7 Experimental Results</td>
<td>20</td>
</tr>
<tr>
<td>CHAPTER 3 ASSERTION RANKING</td>
<td>28</td>
</tr>
<tr>
<td>3.1 Motivation</td>
<td>30</td>
</tr>
<tr>
<td>3.2 Related Work and Background</td>
<td>31</td>
</tr>
<tr>
<td>3.3 Methodology</td>
<td>33</td>
</tr>
<tr>
<td>3.4 Case Study</td>
<td>46</td>
</tr>
<tr>
<td>3.5 Experimental Results</td>
<td>49</td>
</tr>
<tr>
<td>CHAPTER 4 ASSERTION RANK AGGREGATION</td>
<td>56</td>
</tr>
<tr>
<td>4.1 Related Work and Background</td>
<td>56</td>
</tr>
<tr>
<td>4.2 Methodology</td>
<td>60</td>
</tr>
<tr>
<td>4.3 Experimental Results</td>
<td>64</td>
</tr>
<tr>
<td>CHAPTER 5 EFFECTIVENESS OF DATA MINING ALGORITHMS</td>
<td>73</td>
</tr>
<tr>
<td>CHAPTER 6 CONCLUSION</td>
<td>79</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

2.1 The GoldMine methodology ........................................... 9
2.2 An and gate ............................................................... 12
2.3 A decision tree for the Verilog in figure 2.2. Each node is pictured with the data it represents and is labeled with its mean and entropy. Each branch is labeled with the feature variable and value used to partition the data represented by its parent ................................................... 13
2.4 A decision forest for the Verilog in figure 2.2. Each node is labeled with its mean and entropy. Each branch is labeled with the feature variable and value used to partition the data represented by its parent. The BGDF algorithm selects both a and b to partition the root node’s data since they do so equally well ................................................... 16
2.5 An example of the prism algorithm for the Verilog in figure 2.2. Each node is labeled with its mean and entropy. Each branch is labeled with the feature variable and value used to partition the data represented by its parent ................................................... 19
2.6 The size of modules usbf_pe, pci_master32_sm and or1200_ctrl 20
2.7 The number of assertions generated by each data mining algorithm ................................................... 21
2.8 The average antecedent size of generated assertions for each data mining algorithm ................................................... 22
2.9 The average temporal length of generated assertions for each data mining algorithm ................................................... 23
2.10 The average input space coverage of generated assertions for each data mining algorithm ................................................... 24
2.11 The hit rate of generated assertions for each data mining algorithm ................................................... 25
2.12 The run time of each data mining algorithm ................................................... 26
2.13 The memory use of data mining algorithm ................................................... 27
3.1 A 2-port arbiter ............................................................... 30
3.2 An example of PageRank ................................................... 31
3.3 The global dependency graph for the 2-port arbiter. Each node’s $I_g$ value denotes its global importance score.  
3.4 The relative dependency graph for gnt1. Each node’s $I_r$ and $C_r$ values denote its relative importance and complexity scores.  
3.5 An implementation of the PCI bridge master state machine.  
3.6 The positional rank and positional rank percentile of each output based upon the output’s global importance score.  
3.7 Assertion complexity as a function of importance.  
3.8 The average antecedent size of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity and rank percentiles.  
3.9 The average temporal length of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity and rank percentiles.  
3.10 The average importance score of assertions in the 99th, 95th, 90th, 75th and 50th rank percentile.  
3.11 The average complexity score of assertions in the 99th, 95th, 90th, 75th and 50th rank percentile.  
3.12 The ideality score of the lowest, middle, and highest ranked true assertions for each target variable.  

4.1 An example of the Kuhn-Munkres algorithm. The left bipartite graph uses each $W(p, r)$ value to define all rank aggregations. The right bipartite graph uses an optimal set of $W(p, r)$ values to define $\tau_a = [p_1 \geq p_0 \geq p_2]$.  
4.2 Assertion complexity as a function of importance.  
4.3 Assertion coverage as a function of importance.  
4.4 Assertion coverage as a function of complexity.  
4.5 Assertion expectedness as a function of importance.  
4.6 Assertion expectedness as a function of complexity.  
4.7 Assertion expectedness as a function of coverage.  
4.8 Kendall tau distance between a ranking for assertion rank and a rank aggregation for assertion importance and complexity.  
4.9 The average importance of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles.  
4.10 The average complexity of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles.  
4.11 The average coverage of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles.
4.12 The average expectedness of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles. .......................... 71

5.1 The average normalized importance of 25% highest ranked assertions that passed formal verification for each data mining algorithm. ......................................................... 74

5.2 The average normalized complexity of 25% highest ranked assertions that passed formal verification for each data mining algorithm. ......................................................... 75

5.3 The average normalized ideality of 25% highest ranked assertions that passed formal verification for each data mining algorithm. ......................................................... 76

5.4 The average normalized coverage of 25% highest ranked assertions that passed formal verification for each data mining algorithm. ......................................................... 77

5.5 The average normalized expectedness of 25% highest ranked assertions that passed formal verification for each data mining algorithm. ......................................................... 78
LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABV</td>
<td>Assertion-Based Verification</td>
</tr>
<tr>
<td>BGDF</td>
<td>Best-Gain Decision Forest</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IFV</td>
<td>Incisive Formal Verifier</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>VCD</td>
<td>Value Change Dump</td>
</tr>
<tr>
<td>VCS</td>
<td>Verilog Compiler Simulator</td>
</tr>
<tr>
<td>VHDL</td>
<td>Verilog Hardware Description Language</td>
</tr>
</tbody>
</table>
1.1 Assertion-Based Verification

Digital systems continue to revolutionize the world. However, most take for
granted the amount of effort required to verify these systems. Pre-silicon
verification constitutes well over 50% of the effort required to design modern
chips [1]. Therefore, verification is the primary bottleneck in digital system
design. Since the complexity of digital systems is perpetually increasing,
verification will continue to pose new challenges.

Pre-silicon verification validates the correctness of a design’s implementa-
tion with respect to its specification before the design is fabricated. Typically,
the design is specified at the register-transfer level (RTL) using a hardware
description language (HDL). Hardware description languages are a class of
programming language that enable automated simulation and analysis of dig-
ital systems. Popular HDLs include VHDL [2], Verilog [3] and SystemVerilog
[4].

Pre-silicon verification uses many methods to verify a design’s correctness.
Popular functional validation methods include simulation and formal veri-
ification. Simulation uses a test bench to apply input vectors to an RTL
design and verifies the correctness of output values. Directed test benches
apply a predetermined set of input vectors to target specific design behavior.
In contrast, random test benches apply random input vectors to maximize
behavioral coverage. Simulation is computationally efficient, but inherently
incomplete since it cannot exercise the design’s entire behavior. Typically,
verification engineers use coverage metrics to estimate the completeness of a
simulation.

Formal verification uses mathematical methods to verify a design’s cor-
rectness. Equivalence checking formally verifies whether or not two imple-
mentations of a design are functionally equivalent. Model checking formally verifies whether or not a design satisfies a specification. Consequently, formal verification is a complete, but computationally inefficient solution.

Assertions or invariants [5] specify the desirable or required properties the design should satisfy. Assertion-based verification (ABV) [6, 7] uses assertions in both simulation and formal verification. Simulation reports violations of an assertion during the execution of a test bench. Formal verification statically analyzes the design’s state space and reports conditions that can violate an assertion. Assertion-based verification is among the most popular verification methodologies [8].

1.2 Automatic Assertion Generation

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, various works have proposed methodologies to automatically generate RTL assertions [9, 10, 11, 12, 13, 14, 15, 16, 17, 18]. The generated assertions can expose subtle bugs in the design or guide the design’s evolution.

Many of these methodologies use data mining algorithms to mine assertions from RTL simulation traces. Data mining is the process of discovering interesting patterns and knowledge from large amounts of data [19]. Assertion generation methods use data mining algorithms to induce a set of predictive rules from a dataset. These rules use a set of feature variables to predict the value of a target variable. Assertion generation methods form assertions from the induced rules.

GoldMine [20] is a unique assertion generation method since it uses static and formal analysis in addition to data mining. GoldMine uses static analysis to guide the data mining engine. Since it generates assertions based on a subset of the design’s behavior, GoldMine uses formal analysis to verify the generated assertions. In this thesis, we provide an overview of the GoldMine methodology and detail each of its data mining algorithms. We also analyze each algorithm’s performance.
1.3 Assertion Evaluation

Verification is only as good as the properties we wish to verify. As such, writing quality assertions is tantamount to writing clear specifications, which is a problem as old as the verification problem itself. Classic verification literature has long discussed and debated questions regarding the quality of a specification, the information contained in a property, or the behavior covered by a property [21].

In this thesis, we present methods to evaluate the quality of RTL assertions. These methods are primarily motivated by the emergence of automatic assertion generation. Automatic assertion generation methods frequently produce a large number of assertions with non-uniform quality. These assertions might cover unimportant behavior or be incomprehensible. Also, a large number of assertions can discourage designers from parsing them. An assertion ranking method based on qualitative parameters that human beings value will greatly ease the use of automatic assertion generation technology.

In chapter 3, we present an assertion ranking methodology that considers two parameters — assertion *importance* and *complexity*. In short, assertion importance and complexity estimate the importance and understandability of the behavior conveyed by an assertion respectively. We demonstrate the effectiveness of the assertion rank methodology using a detailed case study and experimental results.

In chapter 4 we present an assertion rank aggregation methodology that considers an arbitrary number of parameters. Rank aggregation algorithms combine ranking results from various sources to generate an optimal ranking [22, 23]. We use the rank aggregation methodology to aggregate rankings for assertion importance, complexity, coverage and expectedness. We demonstrate the effectiveness of the assertion rank aggregation methodology using experimental results.

1.4 Contributions

We contribute the following in this thesis.

- We detail the GoldMine methodology and each of its data mining algorithms. We introduce the Best-Gain Decision Forest algorithm to mine
concise RTL assertions.

- We develop an assertion ranking methodology. We define assertion importance, complexity, rank and ideality and we detail methods to compute each of them. We present a case study and experimental results to demonstrate the effectiveness of assertion rank.

- We develop an assertion rank aggregation methodology. We define assertion coverage and expectedness. We aggregate rankings for assertion importance, complexity, coverage and expectedness. We present experimental results to demonstrate the value of these metrics and the rank aggregation methodology.

- We rigorously analyze the performance of each data mining algorithm in GoldMine. We present experimental results that demonstrate each algorithm’s performance with respect to various metrics.
CHAPTER 2

ASSERTION GENERATION

In this chapter, we detail the GoldMine methodology and its data mining algorithms.

2.1 Related Work and Background

An assertion $p$ is a linear temporal logic (LTL) [24] formula of the form $G(A \implies X^kC)$, where the antecedent $A$ can be propositional or temporal and the consequent $C$ is a single proposition. A proposition in this formula is a $(variable, value)$ pair. A propositional logic formula can have a conjunction, disjunction or negation.

Let $\Pi$ denote a Verilog design with variables $V$. We refer to $\Pi$ as the target design. Let $v_t \in V$ denote the target variable. The target variable is the variable for which we would like to generate assertions. A feature variable is a variable in $V$ used to predict the value of $v_t$. Let $t$ denote the temporal window length. The temporal window length denotes the maximum number of cycles for which generated assertions can capture temporal behavior.

We generate temporal assertions by unrolling $\Pi$ for $t$ temporal frames. Unrolling the design duplicates $V$ in each temporal frame. Let temporal variable $v^k \in V$ with temporal index $k$ denote $v^k$ temporal frames relative to the current frame.

We define a simulation $S$ of $\Pi$ as follows. Let $\mathbb{B} = \{0, 1\}$ denote the Boolean number set. Let $S$ be a set of Boolean vectors and let $s \in S$ be a Boolean vector in $\mathbb{B}^{t|V|}$ that assigns a Boolean value to each $v \in V$. Let $s(v)$ denote the value of variable $v$ in $s$. 
2.1.1 Automatic Assertion Generation

Work in deductive program verification [25, 26, 27] has studied assertion generation through static analysis of source code since the seventies. Work in software uses static analysis techniques to generate assertions for assisting program verification [28, 29]. Additional work in software uses dynamic analysis [30, 31, 11] to generate assertions.

Previous work [9, 32, 33] has used static analysis to generate hardware assertions. IODINE [10] generates low-level dynamic assertions for hardware designs, but it does not use data mining. Instead, IODINE analyzes dynamic program behavior with respect to standard property templates such as one-hot encoding or mutex. The methodology in [12] uses dynamic simulation trace data to generate assertions, but it does not use data mining. Instead, the methodology tries to generalize design behavior based on simulation data.

Inferno [34] uses simulation data to extract the semantic protocol from a communication interface RTL design. Inferno infers a set of transaction diagrams which it uses to generate a set of assertions. Since Inferno only generates transactional assertions, it might not generate meaningful assertions for designs that have high temporal depth. Consequently, Inferno assertions might complicate the search for bugs in such designs. In addition, Inferno is limited to designs that implement a communication protocol, which constitute a small subset of hardware.

In [13], the authors use sequential pattern mining to infer causal relationships between frequently occurring sequences of input and output events. Since their methodology only seeks relationships between inputs and outputs, it might 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.

In [35], the authors use symbolic simulation to extract assertions from test bench constraints. The authors use the generated assertions to optimize the design under test. Since their methodology extracts assertions from a constrained test bench, the assertions might reflect a bug in the test bench or overlook untested behavior. In addition, the generated assertions might have limited value since their methodology extracts them using simple templates and verifies them using bounded formal verification.

Several commercial methodologies including NextOp’s BugScope [15] and
Jasper’s ActiveProp [18] generate hardware assertions.

2.1.2 Data Mining

Overview

Data mining is the process of discovering interesting patterns and knowledge from large amounts of data [19]. Data mining encompasses a large set of techniques and algorithms. In this chapter, we are most interested in rule induction algorithms. Rule induction algorithms induce a set of predictive rules from a dataset. Typically, these rules are of the form $A \implies C$. Here, $A$ is a conjunction of propositions that assign concrete values to feature variables and $C$ is a proposition that assigns a value to the target variable. Consequently, rule induction algorithms are ideal for assertion generation.

Data mining usually attempts to functionalize data that is difficult to describe using a mathematical model. For example, we might use data mining to derive a weather prediction model. In such cases, we would expect and tolerate error in the model. However, we can describe Verilog using well-defined mathematical models. As a result, we cannot tolerate error in generated assertions. Therefore, GoldMine will generate an assertion only if it has 100% confidence. The confidence of assertion $p = A \implies C$ is the fraction of $S$ that satisfies $p$ when $A$ is satisfied.

Entropy and Information Gain

Entropy measures the uncertainty in a random variable and was first proposed by Shannon in [36]. Entropy originated in information theory, but rule induction algorithms use it to measure the uncertainty of the value of $v_t$ in a dataset. Formally, let $v \in V$ denote a variable, let $S_{v=b} = \{s \in S \mid s(v) = b\}$ denote the subset of $S$ where $v = b$ and let $p_b(S, v) = \frac{|S_{v=b}|}{|S|}$ denote the fraction of $S$ in which $v = b$. We compute the entropy $H(S)$ of $v_t$ in $S$ as follows:

$$H(S) = -p_0(S, v_t) \log_2(p_0(S, v_t)) - p_1(S, v_t) \log_2(p_1(S, v_t))$$

(2.1)

If $H(S) = 0$, then the value of $v_t$ does not change in $S$. In such cases, rule
induction algorithms will induce a rule. Otherwise, they will use information gain as an optimization criteria to reduce $H(S)$. Information gain measures the entropy reduction between $S$ and partitions of $S$ in which a feature variables are assigned concrete values. We compute the information gain $G(S, v)$ between $S$ and partitions $S_{v=0}$ and $S_{v=1}$ of $S$ as follows:

$$G(S, v) = H(S) - p_0(S, v)H(S_{v=0}) - p_1(S, v)H(S_{v=1})$$ (2.2)

We compute the information gain $G_\beta(S, v, b)$ between $S$ and partition $S_{v=b}$ of $S$ with respect to assignment $v_t = \beta$ as follows:

$$G_\beta(S, v, b) = \frac{p_\beta(S_{v=b}, v_t)}{p_\beta(S, v_t)}$$ (2.3)

Rule Induction Methods

Decision tree algorithms [37, 38] are ideal for assertion generation because they are simple, scalable, and represent the data in a compact and intuitive way [19]. Decision tree algorithms perform a greedy search to quickly identify local regions in the data space. These algorithms recursively partition the dataset by assigning values to feature variables until the value of the target variable is consistent. 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 subset of the data and each edge partitions the data represented by its incident parent node.

Branch nodes represent a subset of the data in which the target variable is assigned multiple values. Consequently, decision tree algorithms cannot induce a rule from a branch node. Therefore, they partition the node’s dataset by assigning concrete values to the feature variable with the greatest information gain. They assign each partition to a child node. Leaf nodes represent a subset of the data in which the target variable is assigned only one value. Consequently, decision tree algorithms induce a rule by walking from the root of the tree to a leaf. Each edge in the walked path defines a proposition in the rule’s antecedent and the leaf node defines its consequent.

Decision tree algorithms can introduce irrelevant propositions into a set of rules since they partition the dataset in a hierarchically. Previous work has proposed methods to overcome this limitation. In [39, 40], the authors pro-
pose an algorithm that constructs multiple decision trees, or a decision forest, using random data subspaces. In [41], the authors propose a methodology to efficiently construct a decision forest by sharing nodes between decision trees. These methodologies construct decision forests to avoid overfitting the dataset. However, neither of these methodologies addresses how to induce rules from a decision forest.

In [42], the author presents the prism algorithm to mine concise rules. The prism algorithm is similar to the tree based algorithms, but induces one rule at a time for a (target variable, value) pair. Unlike the tree based algorithms, the prism algorithm does not select feature variables based on their ability to partition the dataset. Instead, the algorithm selects the (feature variable, value) pair that reduces the dataset’s entropy most. Therefore, the prism algorithm recognizes that not all values of a feature variable are relevant to the target variable. Consequently, the algorithm does not introduce irrelevant propositions into a set of assertions.

### 2.2 Methodology

In this section, we detail the GoldMine methodology. Figure 2.1 depicts each component of the methodology.

![Figure 2.1: The GoldMine methodology.](image)

In this section, we detail the GoldMine methodology. Figure 2.1 depicts each component of the methodology.

#### 2.2.1 Static Analyzer

The static analyzer parses and extracts various static information from a Verilog [3] design. First, the static analyzer determines basic design information to make GoldMine more usable. This includes discerning the top module in
the design hierarchy, speculating which input variables are clocks and resets and selecting a set of target variables.

Next, the static analyzer selects a set of feature variables for the target variable. Originally, the static analyzer used the classic cone of influence for this purpose. The classic cone of influence [43] uses the design’s dependency graph to transitively compute which variables can affect another variable’s value. Model checking algorithms use this information to prune the design’s state space. However, we used the classic cone of influence to avoid selecting feature variables that cannot affect the target variable’s value.

Currently, the static analyzer uses the bounded cone of influence to select feature variables. The bounded cone of influence extends the classic cone of influence. Instead of computing all of a variable’s dependencies, it computes only those within a bounded number of temporal frames. Because we generate assertions with a bounded temporal length, the bounded cone of influence selects a set of feature variables superior to that of the classic cone of influence.

Other components of GoldMine rely on the static analyzer to accomplish their tasks. The data generator uses information from the static analyzer to generate a test bench for the design and parse simulation data. The data miner uses the static analyzer to select a minimal set of feature variables for the target variable. Finally, the assertion evaluator uses the static analyzer to compute an assertion’s importance, complexity and coverage (see chapter 3).

2.2.2 Data Generator

The data generator generates and parses simulation data. The data generator creates an unconstrained random test bench for the design if a directed or constrained random test bench is unavailable. The unconstrained random test bench assigns random values to each input variable for 10000 cycles of simulation. The data generator uses the created (supplied) test bench to simulate the design.

The data generator also parses value change dump (VCD) simulation data. First, the data generator parses the entire simulation. Next, it summarizes the data by retaining only that which coincides with the clock edge. Finally,
the data generator unrolls the data into temporal frames of length \( t \) and compresses it by discarding duplicate frames. The data miner relies on this process to simplify its task.

2.2.3 Data Miner

The data miner searches for causal relationships between feature variables and the target variable in the simulation data. If the data miner finds a relationship with 100% confidence, then it will generate an assertion. Any suitable rule induction algorithm can mine a set of assertions from the simulation data. In sections 2.3, 2.4, 2.5 and 2.6, we discuss the algorithms GoldMine uses to generate assertions.

2.2.4 Formal Verifier

The data miner generates assertions based on a subset of the design’s functionality. Consequently, we cannot guarantee that generated assertions are true system invariants. Therefore, the formal verifier uses Cadence Incisive Formal Verifier (IFV) [44] to verify the generated assertions. The formal verifier generates code that checks the generated assertions at each clock edge. The generated code also constrains the design’s reset signal to prevent IFV from generating trivial counterexamples. The formal verifier reports assertions that pass formal verification as true system invariants. If an assertion fails formal verification, then the data miner can use the assertion’s counterexample to refine it. This technique is detailed in [45].

2.2.5 Assertion Evaluator

GoldMine attempts to automate a traditionally manual process. Consequently, the assertion evaluator evaluates GoldMine’s performance and the quality of the generated assertions. Originally, the assertion evaluator used a superficial set of metrics to evaluate an assertion’s quality. These metrics include antecedent size, temporal length, input space coverage and hit rate. Antecedent size refers to the number of propositions in an assertion’s antecedent. Temporal length refers to the number of temporal frames spanned
by an assertion’s antecedents. The assertion evaluator reports the average antecedent size and temporal length for a set of generated assertions to estimate their readability.

Input space coverage estimates the functional coverage of an assertion. Suppose we create a truth table that computes the target variable’s value as a function of the feature variables. The input space coverage of assertion \( p = A \implies C \) is equal to the fraction of entries in the table that satisfy \( A \). We can use the formula \( \frac{1}{2^m} \) to compute the input space coverage \( p \). Intuitively, an assertion with fewer propositions in its antecedent will cover a greater fraction of the input space. Though computing an assertion’s input space coverage is trivial, doing so for an entire set of assertions is not since the coverage of two assertions might not be mutually exclusive.

Hit Rate refers to the fraction of generated assertions that passed formal verification. Hit rate evaluates the data miner’s performance and completeness of the simulation data. In other words, we prefer a data mining algorithm that can find true assertions over one that can find high quality false assertions. Regardless, a data mining algorithm’s performance will be limited by the completeness of the simulation data.

Currently, the assertion evaluator uses importance, complexity, coverage and expectedness to evaluate a set of assertions. We detail these metrics in chapters 3 and 4.

2.3 Decision Tree

```plaintext
module and_(input a, b, output f);
assign f = a & b;
endmodule
```

Figure 2.2: An and gate.

In [20], we detail GoldMine’s decision tree algorithm. Here, we present an example of the algorithm. Consider the Verilog and its decision tree in figures 2.2 and 2.3. The root node assigns multiple values to \( f \). Therefore, the algorithm cannot generate an assertion and partitions the simulation data. The
Figure 2.3: A decision tree for the Verilog in figure 2.2. Each node is pictured with the data it represents and is labeled with its mean and entropy. Each branch is labeled with the feature variable and value used to partition the data represented by its parent.

root’s left child node always assigns \( f = 0 \). Therefore, the algorithm generates the assertion \( \neg a \implies \neg f \). Again, the root’s right child node assigns multiple values to \( f \). Therefore the algorithm will partition the simulation data again. Each new child node assigns a single value to \( f \). Therefore, the algorithm generates assertions \( a \land \neg b \implies \neg f \) and \( a \land b \implies f \).

The decision tree algorithm can introduce irrelevant propositions into a set of assertions. For example, consider the assertion \( a \land \neg b \implies \neg f \) in figure 2.3. Regardless of the value of \( a \), \( b = 0 \implies f = 0 \). Therefore, the inclusion of \( a \) in this assertion is unnecessary. As a result, the assertion is verbose and constrained — limiting its use and value.

2.4 Best-Gain Decision Forest

We introduce the Best-Gain Decision Forest algorithm (BGDF) to generate concise assertions. Decision tree algorithms select one feature variable with maximum information gain to partition the dataset. In contrast, the BGDF algorithm partitions the dataset using all such variables. Consequently, the
BGDF algorithm builds all decision trees optimized for maximum information gain. To maintain efficiency, the algorithm shares nodes between decision trees. The BGDF algorithm extracts all assertions from a decision forest and uses a set containment algorithm to discard those that are redundant.

Algorithm

Algorithm 1 Best-Gain Decision Forest Algorithm

1: `procedure decision_forest(V, S, A)`
2:     if `entropy(S) = 0` then
3:         `P ← P ∪ (A ⊃ (v, mean(S)))`
4:     return
5: end if
6: `G ← {∅}`
7: `g_{best} ← -∞`
8: for all `v ∈ V` do
9:     `g ← gain(v, S_v=0, S_v=1)`
10:    `G ← G ∪ (v, g)`
11:    if `g_{best} < g` then
12:        `g_{best} ← g`
13: end if
14: end for
15: for all `(v, g) ∈ G` do
16:     if `g = g_{best}` then
17:         `V ← V \ v`
18:         `decision_forest(V, S_v=0, A ∪ (v, 0))`
19:         `decision_forest(V, S_v=1, A ∪ (v, 1))`
20: end if
21: end for
22: `end procedure`

Algorithm 1 shows the Best-Gain Decision forest algorithm. The algorithm requires inputs `V`, `S`, and `A`. Let `V` and `S` denote the set of variables and simulation defined by section 2.1 and let `v_t ∈ V` denote the target variable. Let `A` denote a set of propositions and let `(v ∈ V, b ∈ B) ∈ A` denote a proposition that assigns `v = b`. Finally, let `P` denote the set of assertions generated by the Best-Gain Decision Forest algorithm.

The Best-Gain Decision Forest algorithm requires the following additional definitions. Let the function `entropy(S)` compute the entropy of `v_t` in `S` and
let the function $\text{mean}(S)$ compute the mean value of $v_t$ in $S$. Let the function $\text{gain}(S, v)$ compute information gain $G(S, v)$. Finally, let $G$ denote a set of (variable, information gain) pairs.

The Best-Gain Decision Forest algorithm begins by computing the entropy of $v_t$ in $S$. If $\text{entropy}(S) = 0$, then the algorithm adds assertion $A \implies (v_t, \text{mean}(S))$ to $P$ and terminates. If $\text{entropy}(S) \neq 0$, then the algorithm selects feature variables to partition $S$. The algorithm computes the information gain $g$ of each variable $v \in V$ and adds the pair $(v, g)$ to $G$. While computing each variable’s gain, the algorithm records $g_{\text{best}}$ — the maximum gain.

Next, the algorithm recurses. For each $(v, g) \in G$ for which $g = g_{\text{best}}$, the algorithm removes $v$ from $V$ and makes two recursive calls. The first call removes vectors from $S$ where $v = 1$ and adds the proposition $(v, 0)$ to $P$. The second call removes vectors from $S$ where $v = 0$ and adds the proposition $(v, 1)$ to $P$.

The BGDF algorithm minimizes redundancy in $P$ using set containment. Assertion $p_i = A_i \implies C_i$ contains assertion $p_j = A_j \implies C_j$ if $A_i \subset A_j$ and $C_i \equiv C_j$. That is, if $p_i$ contains $p_j$, then $p_i$ conveys the same behavior as $p_j$ more concisely. Consequently, it is impossible for $p_i$ to contain $p_j$ if $|A_i| > |A_j|$. The BGDF algorithm uses this fact to reduce the number of set containment checks between assertions.

Example

We revisit the Verilog in figure 2.2 to illustrate the BGDF algorithm. Let $V = \{a, b, f\}$ and let $v_t = f$. Let $S$ be the set of Boolean vectors to the left of the root node in figure 2.3.

The Best-Gain Decision Forest algorithm begins by computing $\text{entropy}(S) = -\frac{3}{4}\log_2\left(\frac{3}{4}\right) - \frac{1}{4}\log_2\left(\frac{1}{4}\right) = 0.81$. Since the $\text{entropy}(S) \neq 0$, the algorithm partitions $S$. To do so, the algorithm computes each variable’s information gain and records $g_{\text{best}}$.

\[
\text{gain}(a, S) = 0.81 - 0.50 \cdot 0.00 - 0.50 \cdot 1.00 = 0.31
\]

\[
\text{gain}(b, S) = 0.81 - 0.50 \cdot 0.00 - 0.50 \cdot 1.00 = 0.31
\]

Now, $G = \{(a, 0.31), (b, 0.31)\}$ and $g_{\text{best}} = 0.31$. 

15
Next, the algorithm recurses. Since the gains of both $a$ and $b$ are equal to $g_{\text{best}}$, the algorithm uses both of them to partition $S$. In the recursive call where $a = 0$, the error of $f$ is also equal to 0. Therefore, the algorithm adds assertion $p_0 = \neg a \implies \neg f$ to $P$ and terminates.

In the recursive call where $a = 1$, the $H(S) \neq 0$. Therefore, the algorithm selects $b$ to partition $S$ since it is the only remaining variable in $V$. In both subsequent recursive calls, $H(S) = 0$. Therefore, the algorithm adds the assertions $p_1 = a \wedge \neg b \implies \neg f$ and $p_2 = a \wedge b \implies f$ to $P$.

Figure 2.4: A decision forest for the Verilog in figure 2.2. Each node is labeled with its mean and entropy. Each branch is labeled with the feature variable and value used to partition the data represented by its parent. The BGDF algorithm selects both $a$ and $b$ to partition the root node’s data since they do so equally well.

Figure 2.4 depicts the decision forest constructed by the algorithm. Set $P$
contains the following assertions.

\begin{equation}
\begin{align*}
p_0 &= \neg a \implies \neg f \\
p_1 &= a \land \neg b \implies \neg f \\
p_2 &= a \land b \implies f \\
p_3 &= \neg b \implies \neg f
\end{align*}
\end{equation}

Consider the \( p_1 \) and \( p_3 \). Both assertions have the same consequent, but \( p_3 \) is more concise. Therefore, the proposition \((b, 1)\) in the antecedent of \( p_1 \) is unnecessary. After the algorithm uses set containment to remove such assertions, \( P \) contains \( p_0, p_2 \) and \( p_3 \).

Analysis

We analyze the Best-Gain Decision Forest algorithm. First, we analyze the algorithm's complexity. Consider the construction of a worst case decision forest with variables \( V = \{v_0, v_1, \cdots, v_n\} \). Since the BGDF will select all variables to partition \( S \), it cannot select \( v_0 \) to partition \( S \) in the subforest of any other \( v_i \in V \). Consequently, if we disregard \( v_0 \), then the algorithm will construct a worst case decision forest with \( n - 1 \) feature variables. Similarly, each child of \( v_0 \) will be a worst case decision forest with \( n - 1 \) feature variables. Therefore, the worst case size of a decision forest is \( O(3^{|V|}) \).

Next, we show that the BGDF algorithm discards only functionally redundant assertions. Let \( p = A \implies (v_t, b) \) and \( p' = A' \implies (v_t, b) \) be assertions generated by the BGDF algorithm and let \( A \supset A' \). Let \( f = A \lor A' \). Since \( A \supset A' \) and both \( A \) and \( A' \) are conjunctive, it follows from Boolean algebra that \( A' \) is sufficient to satisfy \( f \). Therefore, \( p \) is functionally redundant.

Finally, we show that the BGDF algorithm generates assertions that are either equivalent to or more concise than those generated by the decision tree algorithm. Let \( P \) and \( P' \) denote the sets of assertions generated by the BGDF and decision tree algorithms respectively. Since the BGDF algorithm builds all optimal decision trees, \( P \supseteq P' \). Now, the BGDF algorithm discards only functionally redundant assertions and \( P \supseteq P' \). Therefore, the assertions in \( P \) are either equivalent to or more concise than those in \( P' \).
2.5 Coverage Mining

In [46], the author details GoldMine’s coverage mining algorithm. The coverage mining algorithm searches the assertion space breadth-first to greedily generate assertions with high input space coverage. In each iteration, the algorithm generates an assertion whose input space coverage gain with respect to $S$ is highest. The algorithm terminates after satisfying a coverage criteria for $S$.

The tree based algorithms are greedy in the data space since they generate the minimum number of assertions required to cover $S$. In contrast, the coverage mining algorithm is greedy in the assertion space since it generates the minimum number of concise assertions to cover $S$. Consequently, the tree based algorithms are significantly more efficient than the coverage mining algorithm. However, the coverage mining algorithm might find a more intuitive set of assertions than the tree based algorithms.

2.6 Prism

We adapt the prism algorithm in [42] to generate assertions. The prism algorithm is similar to the tree based algorithms, but induces one rule at a time for assignment $v_t = \beta$. In each iteration, the algorithm adds a proposition $(v \in V, b)$ for which $G_\beta(S, v, b)$ is maximum to the candidate rule until $H(S) = 0$. After the algorithm generates a rule, it discards vectors in $S$ that satisfy the rule. The algorithm repeats this process until $S_{v_t=\beta}$ is empty.
Figure 2.5 shows an example of the prism algorithm for the Verilog in figure 2.2. First, the algorithm generates an assertion for \( f = 0 \). The algorithm computes the information gain of propositions \((a, 0), (a, 1), (b, 0)\) and \((b, 1)\) as follows:

\[
G_0(S, a, 0) = \frac{p_0(S_{a=0}, v_1)}{p_0(S, v_1)} = \frac{1}{3/4} = \frac{4}{3} \\
G_0(S, a, 1) = \frac{2}{3} \\
G_0(S, b, 0) = \frac{4}{3} \\
G_0(S, b, 1) = \frac{2}{3}
\]

The algorithm adds \((a, 0)\) to the assertion’s antecedent and generates the assertion \( p_0 = \neg a \implies \neg f \) since \( H(S) = 0 \).

Next, the algorithm removes the first and second rows from \( S \) since they satisfy \( p_0 \). Because \( S \) contains another row where \( f = 0 \), the algorithm generates another assertion for \( f = 0 \). Again, the algorithm computes the information gain of propositions \((a, 0), (a, 1), (b, 0)\) and \((b, 1)\) and adds \((b, 0)\) to the assertion’s antecedent. The algorithm generates the assertion \( p_1 = \neg b \implies \neg f \) and removes the third row from \( S \). Now, since \( S \) does not contain any rows where \( f = 0 \), the algorithm generates the assertion \( p_2 = a \land b \implies f \) for \( f = 1 \) and terminates.
2.7 Experimental Results

<table>
<thead>
<tr>
<th>Design</th>
<th>Module</th>
<th>Number of Variables</th>
<th>Number of Bits</th>
<th>Number of Output Variables</th>
<th>Number of Output Bits</th>
<th>Number of Module Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB</td>
<td>usbf_pe</td>
<td>152</td>
<td>785</td>
<td>25</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>PCI</td>
<td>pci_master32_sm</td>
<td>133</td>
<td>370</td>
<td>20</td>
<td>85</td>
<td>8</td>
</tr>
<tr>
<td>OR1200</td>
<td>or1200_ctrl</td>
<td>68</td>
<td>435</td>
<td>45</td>
<td>315</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.6: The size of modules *usbf_pe*, *pci_master32_sm* and *or1200_ctrl*

We present experimental results to compare the assertion mining algorithms. We used each algorithm to generate assertions for three Verilog modules. The first module is the protocol engine from the Universal Serial Bus (USB) protocol, the second module is the master state machine from the Peripheral Component Interconnect (PCI) protocol and the final module is the decode pipeline stage from the OpenRisc 1200 (OR1200) CPU. Figure 2.6 shows the size of each of these modules. For all experiments, the data generator simulated each module for 10000 cycles using an unconstrained random test bench. We explicitly limited the antecedent size and temporal length of all assertions to 5 and 2 respectively. We conducted all experiments using a 2.67 gigahertz quad core Intel Core i5 with 16 gigabytes of memory.
Figure 2.7: The number of assertions generated by each data mining algorithm. The decision tree algorithm consistently generates a small number of assertions. We expect this since the decision tree algorithm generates the minimum number of assertions required to describe the dataset. The decision forest and coverage mining algorithms generate varying number of assertions. We expect this since the decision forest and coverage mining algorithms generate assertions that have low antecedent sizes instead of generating those that minimally describe the dataset. For all designs, the prism algorithm generates more assertions than any other algorithm. We expect this since the prism algorithm generates the minimum number of assertions that have low antecedent sizes to describe the dataset.
Figure 2.8 shows the average antecedent size of generated assertions for each data mining algorithm. For all designs, the coverage mining algorithm generates assertions that have lower antecedent sizes than those generated by any other algorithm. We expect this since the coverage miner optimizes for input space coverage. For USB, the prism algorithm generates assertions that have larger antecedent sizes than those generated by any other algorithm. This is reasonable since the prism algorithm typically generates a large number of assertions. Similarly, the decision forest algorithm generates assertions that have larger antecedent sizes than those generated by the decision tree algorithm.
Figure 2.9 shows the average temporal length of generated assertions for each data mining algorithm. For all designs, the prism algorithm generates assertions that have higher temporal length than those generated by any other algorithm. Again, this is reasonable since the prism algorithm generates a large number of assertions. Similarly, for PCI and OR1200, the decision forest algorithm generates assertions that have higher temporal length than those generated by the decision tree algorithm. For USB and OR1200, the coverage mining algorithm generates assertions that have the lower temporal length than those generated by any other algorithm. This is reasonable since we expect temporal assertions to have large antecedent sizes.
Figure 2.10 shows the average input space coverage of generated assertions for each data mining algorithm. Since input space coverage is inversely proportional to antecedent size, these results provide an alternate interpretation of those in figure 2.8. They also validate that the coverage mining algorithm indeed optimizes for input space coverage.
Figure 2.11 shows the hit rate of generated assertions for each data mining algorithm. The decision tree algorithm consistently generates a high percentage of true assertions. This suggests that data mining algorithms that optimize for antecedent size can generate a high percentage of false assertions. For PCI and OR1200, the coverage mining algorithm generates the highest percentage of true assertions. This suggests that assertions that cover a large fraction of the dataset are more likely to pass formal verification.
Figure 2.12: The run time of each data mining algorithm.
Figures 2.12 and 2.13 show the run time and memory use of each data mining algorithm. For most designs, every algorithm executes in less than 3 minutes. However, for PCI, the coverage miner executes in more than 10 minutes. We expect this since the coverage miner mines assertions breadth-first. Similarly, most algorithms require less than 100 Mb of memory to execute. However, for USB and PCI, the coverage miner requires nearly 500 and 2000 Mb respectively.
In this chapter, we present an assertion ranking methodology that considers two parameters — assertion *importance* and *complexity*. In short, assertion importance and complexity estimate the importance and understandability of the behavior conveyed by an assertion respectively.

To compute assertion importance, we first compute a *global importance score* for each variable in an RTL design using an algorithm inspired by Google’s PageRank. PageRank \([47, 48]\) is a seminal graph ranking algorithm used by Google to rank web search results. PageRank represents the world wide web using a graph, where each node denotes a web page and each directed edge denotes a hyperlink from one page to another. PageRank analyzes the hyperlink structure of the world wide web graph to compute an importance score for each web page. Intuitively, PageRank will rank graph nodes with many incoming and outgoing edges higher than those with fewer such edges. We adapt the PageRank algorithm to work for an RTL variable dependency graph instead of the web graph.

In the context of assertion ranking, each assertion constitutes a search query. In other words, the relationship between the variables in an assertion’s antecedent and those in its consequent is relevant. Since PageRank is a query-independent ranking algorithm, it does not consider such relationships. Therefore, we compute a *relative importance score* that captures how important a variable is with respect to a target variable. Relative importance scores consider the spatial distance, temporal distance and importance of execution paths between references to the given variable and assignments to the target variable. We use relative importance scores to compute an assertion’s importance score.

An assertion with a high importance score will likely have low understandability. Consequently, we balance assertion importance with assertion complexity. To compute assertion complexity, we first compute a relative
complexity score that captures the understandability of the dependencies between a variable and a target variable. The relative complexity score captures how understandable a variable is with respect to a target variable. Relative complexity scores consider the spatial distance, temporal distance and understandability of execution paths between references to the given variable and assignments to the target variable. We use relative complexity scores to compute an assertion’s complexity score.

We use assertion importance and complexity scores to compute assertion rank. An assertion’s rank score estimates its value with respect to the design for which it was written. As the name implies, assertion rank scores also provide a means to rank a set of assertions. We normalize an assertion’s rank score with respect to a maximum rank score. We refer to an assertion’s normalized rank score as its ideality score. An assertion’s ideality score estimates its “completeness.”

We show a detailed case study using an open source RTL implementation of the PCI protocol. The case study demonstrates that the proposed assertion quality metrics agree with human judgement. We present an analysis of the choices made by the ranking methodology for assertions generated using GoldMine. In addition to the case study, we present experimental results that further verify the effectiveness of the methodology.
3.1 Motivation

We motivate assertion ranking with an example. Figure 3.1 depicts the Verilog source code for a 2-port arbiter. The arbiter uses the temporary variable `gnt_` to guarantee that neither request line is starved. Consider

```verilog
module arb2(clk, rst, req1, req2, gnt1, gnt2);

input clk, rst;
input req1, req2;
output gnt1, gnt2;

reg gnt_;
reg gnt1, gnt2;

always @(posedge clk,posedge rst)
    if (rst)
        gnt_ <= 0;
    else
        gnt_ <= gnt1;

always @(*)
    if (gnt_)
        begin
            gnt1 = req1 & ~req2;
            gnt2 = req2;
        end
    else
        begin
            gnt1 = req1;
            gnt2 = req2 & ~req1;
        end

endmodule
```

Figure 3.1: A 2-port arbiter.
properties \( p_0 \) and \( p_1 \), defined as follows. Property \( p_0 \) expresses \textit{if the first request line is low, then the first grant line is low}. Property \( p_1 \) expresses \textit{if the second request line is high and the first grant line was high previously and one cycle later the first request line is high, then the first grant line is high}.

Both properties are true and express important design intent. However, many designers would agree that \( p_0 \) is inferior to \( p_1 \). Property \( p_0 \) is a trivial recitation of the source code expressing combinational behavior, while \( p_1 \) expresses subtle temporal behavior. Also, \( p_0 \) conveys nothing about the second request line while \( p_1 \) conveys the main functionality of an arbiter —– a contention scenario. Finally, \( p_0 \) does not utilize \( gnt_- \), the most important design variable, while \( p_1 \) does.

### 3.2 Related Work and Background

#### 3.2.1 PageRank

PageRank [47, 48] is a seminal graph ranking algorithm used by Google to rank web search results. PageRank proposes that a web page is important if it has incoming hyperlinks from many important pages. This definition recognizes that not all hyperlinks should have equal weight. For example, if a page has a single incoming hyperlink from the Apple home page, it should have a higher importance score than those with many incoming hyperlinks from obscure pages.

![Figure 3.2: An example of PageRank.](image)
PageRank computes an importance score for each web page based on its incoming hyperlinks. Intuitively, each web page distributes its importance score equally amongst its outgoing hyperlinks. Consequently, the importance score for a web page is equal to the sum of the importance scores distributed to its incoming hyperlinks. The simplified PageRank formula formalizes this relationship.

Let $p$ denote a web page. Let $B(p)$ denote the set of pages that have an outgoing hyperlink to $p$ and let $F(p)$ denote the set of pages that $p$ has outgoing hyperlinks to. We define the simplified PageRank $PR(p)$ of $p$ as follows:

$$PR(p) = \sum_{p_i \in B(p)} \frac{PR(p_i)}{|F(p_i)|}$$  \hspace{1cm} (3.1)

Figure 3.2 illustrates equation 3.1. For example, $PR(c) = PR(a) + \frac{PR(b)}{2} = 128$ since $a$ has one outgoing hyperlink and $b$ has two outgoing hyperlinks.

We explain the intuition behind PageRank using a random surfer model. Suppose a web surfer browses the web by clicking hyperlinks at random. The simplified PageRank of a given web page represents the probability that the random surfer will navigate to that page. If the random surfer is caught in a cycle of web pages, then the surfer is unlikely to continue in such a cycle forever. Consequently, the PageRank formula adds an additional term to equation 3.1 to account for such behavior.

Let $\epsilon$ be a constant between 0 and 1 and let $n$ denote the total number of web pages. We define the PageRank $PR(p)$ of $p$ as follows:

$$PR(p) = (1 - \epsilon) \sum_{p_i \in B(p)} \frac{PR(p_i)}{|F(p_i)|} + \frac{\epsilon}{n}$$  \hspace{1cm} (3.2)

The additional term in equation 3.2 represents the probability of the random surfer growing bored and navigating to a random page in the web graph.

### 3.2.2 Dependency Graph

We analyze a design’s dependency graph to rank its variables. We define a dependency graph based on the semantics of the Verilog Hardware Description Language [3]. An expression is a function defined over values, variables and operators. A left reference refers to a variable reference that appears
in an expression on the left side of a Verilog assignment. A right reference
refers to all variable references that are not left references.

Let \( v_i \) and \( v_j \) denote two Verilog variables. We say that \( v_i \) depends on
\( v_j \) if there exists a Verilog assignment to \( v_i \) that will execute only if a right
reference to \( v_j \) is evaluated. Formally, we define a dependency graph as a
directed graph \( G = (V, E) \) with vertices \( V \) and directed edges \( E \). Let each
vertex \( v \in V \) denote a Verilog variable and let each directed edge \( (v_i, v_j) \in E \)
denote a dependence between variables \( v_i \) and \( v_j \). If \( (v_i, v_j) \in E \), then \( v_j \)
depends on \( v_i \).

We construct two types of dependency graphs. A global dependency graph
expresses dependencies between all variables in a Verilog design. The global
dependency graph is analogous to the dependency graph used by the classic
cone of influence computation [43]. A relative dependency graph transitively
expresses dependencies for a target variable within a bounded number of tem-
poral frames. The relative dependency graph is analogous to the dependency
graph used by the bounded cone of influence computation [49, 50].

If target variable \( v_t \) is temporal, then it will have temporal dependencies.
Let temporal variable \( v^k \) with temporal index \( k \) denote \( v^k \) temporal frames
relative to the current frame. We treat each such variable as a unique vari-
able. For example, \( v^{-1} \neq v \). If there is a cycle in the dependency graph of
\( v_t \), then it will depend on an infinite number of temporal variables. Con-
sequently, we construct the relative dependency graph of \( v_t \) for a bounded
number of temporal frames.

### 3.3 Methodology

We compute the rank score of an assertion in three phases. The first phase
computes the assertion’s importance score. The second phase computes the
assertion’s complexity score. The third phase uses an assertion’s importance
and complexity scores to compute its rank score.

#### 3.3.1 Importance

We first compute the importance score of an assertion. This phase includes
three subphases. The first subphase computes a global importance score
for each variable in the design. The second subphase computes a relative importance score for each variable in the assertion’s antecedent. The final subphase uses the relative importance scores of the variables in the assertion’s antecedent to compute its importance score.

Global Variable Importance

We adapt the PageRank algorithm [47, 48] to compute a global importance score for each variable in a Verilog design. The PageRank formula is derived from the intuition that important web pages have incoming hyperlinks from many important pages. Similarly, we propose that important variables depend on many important variables. For example, a designer might intuitively consider a variable representing the next state in a design important. Consequently, such a variable would depend on other important variables, including those that determine the current state of the design.

The global importance computation requires a global dependency graph. We represent a global dependency graph using an adjacency matrix. Let $a_{ij}$ denote the number of right references to variable $i$ in all assignments to variable $j$. Let $a_i$ denote the number of right references to variable $i$ in all assignments. Let $A$ be an $n \times n$ square matrix with rows and columns corresponding to variables. Let $A_{ij} = \frac{a_{ij}}{a_i}$ if $a_i > 0$ and let $A_{ij} = \frac{1}{n}$ otherwise.

Intuitively, $A_{ij}$ is equal to the fraction of right references to variable $i$ that exist in all assignments to variable $j$. If no references to variable $i$ exist in the design, then we assume a right reference to variable $i$ exists in an assignment to each other variable. Hence, $A_{ij} = \frac{1}{n}$ when $a_i = 0$. PageRank refers to such nodes with no outgoing edges as dangling nodes and manages them in a similar manner to ensure the importance score distribution models a probability distribution.

The global importance computation iteratively computes the global importance score of each variable in the design. Let $0 < \epsilon < 1$ be a constant source of global importance. We have found through experimentation that when $\epsilon = 0.5$, the global importance score distribution agrees with designer intuition. Let $r$ denote a global importance score vector over variables and let $r^k$ denote $r$ in the $k$th iteration of the global importance computation. Let $r^0_i = \frac{1}{n}$. We compute the global importance score of each variable as
follows:
\[ r^{k+1} = (1 - \epsilon)Ar^k + \frac{\epsilon}{n} \] (3.3)

Equation 3.3 can iterate to convergence. However, we have found that 100 iterations computes the global importance score of each variable with a precision of up to 5 decimal places.

PageRank is scalable to hundreds of millions of pages [51, 52]. Consequently, an appropriate implementation of the global importance computation would be scalable to hundreds of millions of variables. However, we implement equation 3.3 in a straightforward manner. Such an implementation is sufficient since RTL assertions rarely specify complex intermodular behavior.

The global importance computation does not consider the semantics of variable references. Therefore, a variable’s global importance score estimates its structural importance or connectedness with respect to the design. In other words, removing a variable with a high global importance score from the design would significantly alter the design’s functionality while removing a variable with a low global importance score would not.

Relative Variable Importance

We are usually interested in a variable’s importance within a specific functional context. However, global importance scores cannot convey such information. For example, consider a design with inputs \( a \) and \( b \), and outputs \( f_a \) and \( f_b \). Suppose the design trivially assigns \( f_a = a \) and \( f_b = b \). Intuitively, the global importance scores of \( a \) and \( b \) would be equivalent since they are both inputs. However, if we consider the importance of \( a \) and \( b \) with respect to \( f_a \), then we would expect the importance score of \( a \) to be greater than that of \( b \) since the value of \( f_a \) is not affected by the value of \( b \). We compute relative importance scores to measure the relative importance of one variable with respect to another.

We present additional examples to illustrate why global importance scores are insufficient for estimating an assertion’s importance. Each example presents a set of assertions. For each assertion, we discuss the relative importance score of each variable in the assertion’s antecedent with respect to the variable in its consequent. We assume that each variable in each assertion’s
antecedent is an input variable. Consequently, the global importance score of each such variable will be equivalent.

1. **Spatial distance** — Consider the assertions $a \rightarrow f$ and $b \rightarrow f$. Suppose the spatial distance between $a$ and $f$ is 3 assignments while that between $b$ and $f$ is 1 assignment. Intuitively, $a$ should have a higher importance score than $b$ with respect to $f$ since the spatial distance between $a$ and $f$ is greater than that between $b$ and $f$.

2. **Temporal distance** — Consider the assertions $a \rightarrow Xf$ and $a \rightarrow f$. Both assertions reference $a$ in their antecedents. However, the first assertion references $a$ in a previous temporal frame. Therefore, we can rewrite the first assertion as $a^{-2} \rightarrow f$. Since the spatial distance between $a^{-2}$ and $f$ is greater than that between $a$ and $f$, $a^{-2}$ should have a higher importance score than $a$ with respect to $f$.

3. **Execution path importance** — Consider the assertions $a \rightarrow f$ and $b \rightarrow f$. Suppose the spatial distances between $a$ and $f$ and $b$ and $f$ are both equal to 3 assignments. If the assignments between $a$ and $f$ reference variables with higher global importance scores than those between $b$ and $f$, then $a$ should have a higher importance score than $b$ with respect to $f$.

Global importance scores cannot capture the concepts presented in the previous set of examples. Therefore, we compute relative importance scores.

Let $I_{r}(v, v_t)$ denote the relative importance score of variable $v$ with respect to target variable $v_t$. In any practical design, $v_t$ would depend on many variables. It is unreasonable to consider each of these variables equally important with respect to $v_t$. Consequently, we compute $I_{r}(v, v_t)$ based on the global importance scores of the variables referenced between references to $v$ and assignments to $v_t$.

Algorithm 2 constructs a relative dependency graph for a target variable and computes a relative importance score for each variable in the graph. The algorithm constructs relative dependency graph $G_r = (V_r, E_r)$. The algorithm requires inputs $v, k$ and $v_t$. Input $v$ denotes the current variable in the depth first construction of the relative dependency graph, $k$ denotes the current temporal index, and $v_t$ denotes the target variable.
Algorithm 2 Relative Variable Importance

1: procedure relative_importance(v, k, v_t)
2:     if k < k_{max} then
3:         V ← dependencies(v)
4:         for all v_i ∈ V do
5:             i_r ← I_g(v_i) + I_r(v, v_t)
6:             V_r ← V_r ∪ (v_i^{-k}, i_r)
7:             E_r ← E_r ∪ (v_i^{-k}, v)
8:         end for
9:         for all v_i ∈ V do
10:             if temporal(v_i) then
11:                 relative_importance(v_i^{-k}, k + 1, v_t)
12:             else
13:                 relative_importance(v_i^{-k}, k, v_t)
14:             end if
15:         end for
16:     end if
17: end procedure

Algorithm 2 requires the following additional definitions. Let k_{max} denote the maximum temporal length of the relative dependency graph. Let G_g = (V_g, E_g) denote the global dependency graph and let the function dependencies(v) = \{v_i ∈ V_g \mid (v_i, v) ∈ E_g\}. That is, dependencies(v) returns the set of variables on which v depends within 1 temporal frame. Let the function I_g(v) return the global importance score of v. Finally, let the function temporal(v) be satisfied only if assignments to v span multiple temporal frames.

Algorithm 2 begins by checking if k < k_{max}. If k < k_{max}, then the algorithm terminates. Otherwise, for each variable on which v depends, the algorithm adds a new node and edge to G_r. The algorithm computes the relative importance score i_r of v_i^{-k} by summing the global importance score of v_i and relative importance score of v with respect to v_t. Next, for each variable v_i on which v depends, the algorithm increases k if v_i is temporal and recurses.

Assertion Importance

An important assertion should have high spatial distance, high temporal distance, and/or cover important execution paths between the satisfaction of
its antecedent and consequent. Therefore, we use relative importance scores to compute the importance score of an assertion since they measure these attributes.

Let \( p \) denote an assertion, let \( V_a \) denote the set of variables in the antecedent of \( p \) and let \( v_c \) be the variable in the consequent of \( p \). We compute the importance score \( I(p) \) of \( p \) as follows:

\[
I(p) = \sum_{v_i \in V_a} I_r(v_i, v_c)
\]  

Equation 3.4 considers all variables in the antecedent of an assertion equally. In other words, it makes no difference how variables are related to one another via operators. Each variable contributes its relative importance score to an assertion each time it appears in the assertion’s antecedent.

An assertion’s importance score reflects the relative importance of the variables in its antecedent with respect to the variable in its consequent. A variable’s relative importance score depends transitively on the relative importance scores of the variables it assigns. Therefore, a variable with a high relative importance score constitutes an important execution path. Consequently, an assertion’s importance score estimates its important execution path coverage. As a result, an assertion with a high importance score conveys behavior that is critical with respect to the design.

### 3.3.2 Complexity

Assertions with high importance scores might be difficult to understand. To understand such assertions, a designer might need to parse complex expressions and reason about multiple temporal frames. Consequently, these assertions will have limited use and value. Therefore, we compute the complexity score of an assertion to convey its understandability.

We compute the complexity score of an assertion in two subphases. The first subphase computes a relative complexity score for each variable in the assertion’s antecedent. The second subphase uses the relative complexity scores of the variables in the assertion’s antecedent to compute its complexity score.
Relative Variable Complexity

We devised relative complexity scores based on the process required to understand the meaning of an assertion. Suppose a designer would like to understand the assertion \( a \implies XXf \). First, they would need to search the design for all assignments to \( f \). Next, they would need to transitively parse the set of assignments that assign to \( f \). The designer would continue this process until they encountered an assignment that references \( a \). After finding all such assignments, the designer would reason about them to understand how they satisfy the assertion.

We compute a variable’s relative complexity score based on the following observations. First, understanding the semantics of a variable reference in a large expression will require more effort than doing so for a small expression. Therefore, a variable’s relative complexity score should be proportional to the size of expressions that reference it. Second, understanding the dependence between variables that are spatially or temporally distant requires more effort than doing so for those that are spatially or temporally close. Therefore, a variable’s relative complexity score should depend on variables that it assigns.

Algorithm 3 constructs a relative dependency graph for a target variable and computes a relative complexity score for each variable in the graph. Algorithms 2 and 3 share definitions for \( G_r, v, k, v_t, k_{max}, \text{dependencies}(v) \) and \( \text{temporal}(v) \). Algorithm 3 requires the following additional definitions. Let the function \( \text{expressions}(v) \) return the set of expressions on which \( v \) depends within 1 temporal frame. Let the function \( \text{sensitivities}(v) \) return the set of expressions that reference \( v \).

Algorithm 3 begins by checking if \( k < k_{max} \). If \( k < k_{max} \), then the algorithm terminates. Otherwise, for each variable \( v_i \) on which \( v \) depends, the algorithm adds a new node and edge to \( G_r \). The algorithm computes the relative complexity score \( c_r \) of \( v_i^{-k} \) by summing the sizes of the expressions in \( X \cap S \), which includes expressions that both assign to \( v \) and reference \( v_i \). Next, for each variable \( v_i \) on which \( v \) depends, the algorithm increases \( k \) if \( v_i \) is temporal and recurses.
Algorithm 3 Relative Variable Complexity

1: procedure relative_complexity($v, k, v_t$)
2:     if $k < k_{\max}$ then
3:         $X \leftarrow$ expressions($v$)
4:         $V \leftarrow$ dependencies($v$)
5:         for all $v_i \in V$ do
6:             $S \leftarrow$ sensitivities($v_i$)
7:             $c_r \leftarrow 0$
8:             for all $X_i \in X \cap S$ do
9:                 $c_r \leftarrow c_r + |X_i|$
10:            end for
11:         $V_r \leftarrow V_r \cup (v_i^{k}, c_r)$
12:         $E_r \leftarrow E_r \cup (v_i^{k}, v)$
13:         end for
14:     for all $v_i \in V$ do
15:         if temporal($v_i$) then
16:             relative_complexity($v_i^{k}, k + 1, v_t$)
17:         else
18:             relative_complexity($v_i^{k}, k, v_t$)
19:         end if
20:     end for
21: end if
22: end procedure
Assertion Complexity

A complex assertion should have high spatial distance, high temporal distance, and/or cover complex execution paths between the satisfaction of its antecedent and consequent. Therefore, we use relative complexity scores to compute the complexity score of an assertion since they measure these attributes.

Let $p$ denote an assertion, let $V_a$ be the set of variables in the antecedent of $p$ and let $v_c$ be the variable in the consequent of $p$. Let $C_r(v, v_t)$ denote the relative complexity score of variable $v$ with respect to target variable $v_t$. We compute the complexity score $C(p)$ of $p$ as follows:

$$C(p) = \sum_{v_i \in V_a} C_r(v_i, v_c)$$  \hspace{1cm} (3.5)

Equation 3.5 considers all variables in the antecedent of an assertion equally. In other words, it makes no difference how variables are related to one another via operators. Each variable contributes its relative complexity score to an assertion each time it appears in the assertion’s antecedent.

An assertion’s complexity score reflects the relative complexity of the variables in its antecedent with respect to the variable in its consequent. A variable’s relative complexity score depends transitively on the relative complexity scores of the variables it assigns. Therefore, a variable with a high relative complexity score constitutes a complex execution path. Consequently, an assertion’s complexity score estimates its complex execution path coverage. As a result, an assertion with a high complexity score conveys behavior that is difficult to understand with respect to the design.

3.3.3 Rank

Assertion importance and complexity scores are ideal for estimating the value of an assertion. If a designer evaluated a set of assertions based on their importance and complexity, then they would prefer assertions that maximize importance with respect to complexity. We utilize this idea to compute the rank score of an assertion.
Let $p$ denote an assertion. We compute the rank score $R(p)$ of $p$ as follows:

$$R(p) = \frac{I(p)}{C(p)} \tag{3.6}$$

Equation 3.6 is simple but effective. Intuitively, an assertion that conveys the most important behavior in the least complex way should have greater value than any other assertion.

Assertion rank scores do not evaluate the semantics of an assertion. Such an analysis would be subjective and arbitrary. Instead, the assertion rank computation relies on the structural analysis techniques used by the assertion importance and complexity computations to compute an assertion’s rank score. Consequently, the computation is general, consistent and efficient. In addition, it is not limited by assertions with complicated semantics.

### 3.3.4 Ideality

Assertion rank scores are absolute. This raises a question regarding the existence of an ideal assertion. An *ideal assertion* is one whose rank score is equal to the maximum assertion rank score for a given target variable and temporal bound. We can use the ideal assertion’s rank score to compute another assertion’s *ideality score*.

Let $V$ denote the finite set of variables target variable $v_t$ depends on within temporal bound $k$. Let $\pi$ denote the ideal assertion for $v_t$ and $k$. Based on equation 3.6, the rank score of $\pi$ has the following form.

$$R(\pi) = \frac{\sum_{v_i \in V} I_r(v_i, v_t)}{\sum_{v_i \in V} C_r(v_i, v_t)} \tag{3.7}$$

We can compute $R(\pi)$ by selecting the subset $V_\pi$ of $V$ that maximizes equation 3.7. Next, we prove that $V_\pi$ includes only the variable with the maximum relative importance score to complexity score ratio.

Let $v_\pi \in V$ denote the variable with the maximum relative importance score to complexity score ratio. Let $R_r(v, v_t) = \frac{I_r(v, v_t)}{C_r(v, v_t)}$ denote the relative rank score of $v$ with respect to $v_t$. We would like to prove $V_\pi = \{v_\pi\}$, given $R_r(v_\pi, v_t) \geq R_r(v_i, v_t), \forall v_i \in V$. To do so, we must show that including
another variable in $V_\pi$ can only decrease $R(\pi)$. Therefore, we must prove the following.

$$(R_r(v_\pi, v_t) \geq R_r(v_i, v_t) \implies R_r(v_\pi, v_t) \geq \frac{I_r(v_\pi, v_i) + I_r(v_i, v_t)}{C_r(v_\pi, v_t) + C_r(v_i, v_t)}, \forall v_i \in V)$$

**Proof.** Let $I_\pi, C_\pi$ and $R_\pi$ denote $I_r(v_\pi, v_t), C_r(v_\pi, v_t)$ and $R_r(v_\pi, v_t)$ respectively. Let $I_i, C_i$ and $R_i$ denote $I_r(v_i, v_t), C_r(v_i, v_t)$ and $R_r(v_i, v_t)$ for any $v_i \in V$ respectively. Now,

$$R_\pi \geq R_i$$
$$\frac{I_\pi}{C_\pi} \geq \frac{I_i}{C_i}$$
$$I_\pi \cdot C_i \geq I_i \cdot C_\pi$$
$$I_\pi \cdot C_i + I_\pi \cdot C_\pi \geq I_i \cdot C_\pi + I_\pi \cdot C_\pi$$
$$I_\pi \cdot (C_\pi + C_i) \geq C_\pi \cdot (I_\pi + I_i)$$
$$\frac{I_\pi}{C_\pi} \geq \frac{I_i + I_i}{C_\pi + C_i} \quad \square$$

Let $p$ denote an assertion. We compute the ideality $\nu(p)$ score of $p$ as follows:

$$\nu(p) = \frac{R(p)}{R(\pi)} \quad (3.8)$$

Equation 3.8 normalizes an assertion’s rank score with respect to the ideal assertion’s rank score. Consequently, an assertion’s ideality score conveys its completeness with respect to the design.

### 3.3.5 Example

In this section, we revisit the 2-port arbiter example from section 3.1. We would like to rank the following set of assertions.

\begin{verbatim}
a0: ( req2 == 1 && gnt_ == 1 ) ##1
    ( req1 == 1 ) |->
    ( gnt1 == 1 )
a1: ( req1 == 0 ) ##1 ( req1 == 1 ) |->
    ( gnt1 == 0 )
a2: ( req1 == 0 ) |-> ( gnt1 == 0 )
\end{verbatim}
Figure 3.3: The global dependency graph for the 2-port arbiter. Each node’s $I_g$ value denotes its global importance score.

\[ a3: ( \text{req1} == 1 \&\& \text{req2} == 0 ) \implies ( \text{gnt1} == 1 ) \]

Assertions $a0$ and $a1$ express nontrivial temporal properties while $a2$ and $a3$ express trivial combinational properties. Therefore, we expect both the importance and complexity scores of $a0$ and $a1$ to be higher than those of $a2$ and $a3$.

Figure 3.3 shows the global dependency graph for the arbiter. Each node in the graph is labeled with its respective variable and global importance score. Edge weights denote the number of dependencies between two variables. For example, since $\text{gnt1}$ depends on $\text{req1}$ in both lines 19 and 24 of the Verilog, the weight of edge $(\text{req1}, \text{gnt1})$ is equal to 2. The weights of edges without a specified weight are equal to 1.

We discuss algorithms 2 and 3 simultaneously since they are similarly structured and share definitions. Figure 3.4 depicts the relative dependency graph constructed for $\text{gnt1}$ by both algorithms. Since the maximum temporal length of all assertions is equal to 2, $k_{\text{max}} = 2$. The algorithms begin with $v = \text{gnt1}$, $k = 0$ and $v_t = \text{gnt1}$. Line 2 checks if $k < k_{\text{max}}$. Since $k < k_{\text{max}}$, the algorithms continue.

Line 3 of algorithm 3 initializes $X$ to the set of expressions that assign $\text{gnt1}$ within 1 temporal frame. Therefore, $X = \{ \text{gnt\_}, \text{req1} \& \neg \text{req2}, \text{req1} \}$. Line 3 (4) in algorithm 2 (3) initializes $V$ to the set of variables that assign
Figure 3.4: The relative dependency graph for gnt1. Each node’s $I_r$ and $C_r$ values denote its relative importance and complexity scores.

gnt1 within 1 temporal frame. Therefore, $V = \{\text{req1, req2, gnt}\}$. Lines 4 (5) through 8 (13) add nodes for each of these variables to the relative dependency graph.

Consider req1. In algorithm 2, line 5 computes the relative importance score of req1 as $I_r(\text{req1}, \text{gnt1}) = 2 \times 0.20 + 0.09 = 0.49$. Since gnt1 refers req1 in two assignments, gnt1 contributes its relative importance score to the relative importance score of req1 twice. In algorithm 3, line 6 initializes $S$ to the set of expressions that reference req1. Therefore, $S = \{\text{req1 & ~req2, req1}\}$. Line 8 computes $X \cap S = S$ and lines 8 through 10 compute the relative complexity score of req1 as $C_r(\text{req1, gnt1}) = 2 + 1 = 3$.

Lines 9 (14) through 15 (20) recurse algorithm 2 (3). Since req1 and req2 are inputs, they do not depend on any variables. Therefore, the algorithms will terminate in each of these recursive calls. When the algorithms recurse on gnt_, they will increment $k$ since gnt_ is temporal. Algorithms 2 and 3 terminate when $k \geq k_{\text{max}}$ or when $V = \emptyset$.

Next, we compute the importance score of each assertion as follows:

$$I(\text{a0}) = 0.92 + 1.89 + 0.49 = 3.30$$

$$I(\text{a1}) = 2.24$$

$$I(\text{a2}) = 0.49$$

$$I(\text{a3}) = 0.72$$
Similarly, we compute the complexity score of each assertion as follows:

\[ C(a0) = 5.00 + 5.00 + 3.00 = 13.00 \]
\[ C(a1) = 9.00 \]
\[ C(a2) = 3.00 \]
\[ C(a3) = 5.00 \]

Finally, we compute the rank score of each assertion as follows:

\[ R(a0) = \frac{I(a0)}{C(a0)} = 0.254 \]
\[ R(a1) = \frac{I(a1)}{C(a1)} = 0.249 \]
\[ R(a2) = \frac{I(a2)}{C(a2)} = 0.163 \]
\[ R(a3) = \frac{I(a3)}{C(a3)} = 0.144 \]

3.4 Case Study

In this section we qualitatively analyze a set of ranked assertions. We used GoldMine to generate assertions for the Peripheral Component Interconnect (PCI) bridge master state machine. We used the methodology in section 3.3 to rank the assertions. Figure 3.5 shows the relevant Verilog source code for the PCI bridge master state machine module.

The PCI bridge connects a PCI host device with a PCI bus. The master state machine of the PCI bridge is responsible for executing requests to target devices. To execute a request, the state machine acquires control of the PCI bus. Next, the state machine broadcasts the target address and command, and waits for the target to respond. Finally, the state machine transfers data between the host and target.

We analyze the highest and lowest ranked assertions for output `pci_frame_en_out` from the master state machine. Output `pci_frame_en_out` is the enable signal for `pci_frame_out`. Output `pci_frame_out` signals that the master state machine is transferring data on the bus. During a transfer, `pci_frame_out` should be enabled unless a master abort occurs.
module pci_master32_sm (clk_in, reset_in, pci_gnt_in, pci_frame_in,
    pci_frame_out_in, pci_irdy_in, pci_trdy_in,
    pci_stop_in, req_in, rdy_in,
    output pci_frame_out, pci_frame_en_out);

reg sm_idle;
reg sm_address;
reg sm_data_phases;
reg sm_turn_arround;
wire u_dont_have_pci_bus = pci_gnt_in || ~pci_frame_in || ~pci_irdy_in;
wire u_have_pci_bus = ~pci_gnt_in && pci_frame_in && pci_irdy_in;
wire frame_en_slow = (sm_idle && u_have_pci_bus && req_in && rdy_in) ||
    sm_address || (sm_data_phases && ~pci_frame_out_in);
wire frame_en_keep = sm_data_phases && pci_frame_out_in && ~mabort1 && ~mabort2;
assign pci_frame_en_out = frame_en_slow ||
    frame_en_keep && pci_stop_in && pci_trdy_in;

reg [3:0] cur_state;
reg [3:0] next_state;
parameter S_IDLE = 4'h1 ;
parameter S_ADDRESS = 4'h2 ;
parameter S_TRANSFER = 4'h4 ;
parameter S_TA_END = 4'h8 ;
wire ch_state_slow = sm_address || sm_turn_arround ||
    sm_data_phases || (pci_frame_out_in || ~mabort1 || ~mabort2);
wire ch_state_med = ch_state_slow ||
    sm_idle && u_have_pci_bus && req_in && rdy_in;
wire change_state = ch_state_med ||
    sm_data_phases && (~pci_trdy_in && pci_stop_in);
always @ (posedge reset_in or posedge clk_in)
begin
  if (reset_in)
    cur_state <= S_IDLE;
  else if (change_state)
    cur_state <= next_state;
end
always @ (cur_state or do_write or pci_frame_out_in)
begin
  sm_idle = 1'b0 ;
  sm_address = 1'b0 ;
  sm_data_phases = 1'b0 ;
  sm_turn_arround = 1'b0 ;
case (cur_state)
  S_IDLE:
    begin
      sm_idle = 1'b1 ;
      next_state = S_ADDRESS ;
    end
  S_ADDRESS:
    begin
      sm_address = 1'b1 ;
      next_state = S_TRANSFER ;
    end
  S_TRANSFER:
    begin
      sm_data_phases = 1'b1 ;
      if (pci_frame_out_in)
        next_state = S_TA_END ;
      else
        next_state = S_TRANSFER ;
    end
  S_TA_END:
    begin
      sm_turn_arround = 1'b1 ;
      next_state = S_IDLE ;
    end
  default:
    begin
      next_state = S_IDLE ;
    end
endcase
endmodule

Figure 3.5: An implementation of the PCI bridge master state machine.
GoldMine generated 36 true assertions for \texttt{pci\_frame\_en\_out} — ranked as follows:

\begin{verbatim}
a1: ( cur_state[3] == 1 ) #1
    ( rdy_in == 0 ) |->
    ( pci_frame_en_out == 0 )
a36: ( rdy_in == 0 ) #1
    ( pci_frame_out_in == 1 &&
      pci_trdy_in == 0 &&
      rdy_in == 0 ) |->
    ( pci_frame_en_out == 0 )
\end{verbatim}

Assertions \texttt{a1} and \texttt{a36} were the highest and lowest ranked assertions respectively. We justify this ranking and show the difference in quality between these assertions.

Assertion \texttt{a1} expresses the property \textit{if the state machine’s current state is turn around and one cycle later the host is not ready to send or receive data, then disable \texttt{pci\_frame\_out}}.

Consider the first cycle of \texttt{a1}. The proposition \texttt{cur_state[3] == 1} conveys that the design’s current state is turn around. Careful inspection of the Verilog reveals that the design will change state only if signal \texttt{change\_state} is asserted. Lines 43 through 48 reveal that since the design’s current state is turn around, \texttt{change\_state} is asserted. Consequently, the design’s next state is idle.

Consider the second cycle of \texttt{a1}. Since the design’s current state is idle, the proposition \texttt{rdy_in == 0} ensures that \texttt{frame\_en\_slow} is not asserted on line 29. In addition, \texttt{frame\_en\_keep} is not asserted on line 30. Since both \texttt{frame\_en\_slow} and \texttt{frame\_en\_keep} are not asserted, \texttt{pci\_frame\_en\_out} is not asserted on lines 32 and 33. Therefore, \texttt{pci\_frame\_out} will be disabled.

Assertion \texttt{a36} expresses the property \textit{if the host is not ready to transfer data and one cycle later the host is still not ready to transfer data and the target is not ready to transfer data and the master state machine is not transferring data, then disable \texttt{pci\_frame\_out}}.

Consider the second cycle of \texttt{a36}. The proposition \texttt{pci\_trdy\_in == 0} ensures that the second disjunctive term on line 33 is not asserted. Therefore, we consider the logic for \texttt{frame\_en\_slow} on lines 29 through 30. The propositions \texttt{rdy\_in == 0} and \texttt{pci\_frame\_out\_in == 1} ensure that the first and
third disjunctive terms on lines 29 through 30 are not asserted. Therefore, \textit{pci\_frame\_en\_out} is asserted only if the design’s current state is address. Consequently, \textit{pci\_frame\_en\_out} is asserted only if the design’s previous state was idle.

Consider the first cycle of \textit{a36}. The proposition \textit{rdy\_in == 0} ensures that \textit{change\_state} on lines 47 through 48 is not asserted. Since the design’s current state is idle, the second disjunctive term on line 48 is not asserted. Therefore, we consider the logic for \textit{ch\_state\_med} on lines 45 through 46. The proposition \textit{rdy\_in == 0} ensures that the second disjunctive term on line 46 is not asserted. Therefore, we consider the logic for \textit{ch\_state\_slow} on lines 43 through 44. Since the design’s current state is idle, \textit{ch\_state\_slow} is not asserted. Since the design’s current state is idle and \textit{change\_state} is not asserted, the design’s next state cannot be address. Therefore, \textit{pci\_frame\_en\_out} will not be asserted and \textit{pci\_frame\_out} will be disabled.

Assertion \textit{a1} is consistent with PCI specification. In addition, it is not trivial recitation of the Verilog. In contrast, \textit{a36} is convoluted with respect to the PCI specification. While the assertion is not a trivial recitation of the Verilog, it is difficult to understand. Hence, \textit{a1} is ranked higher than \textit{a36}. Intuitively, a verification engineer would consider \textit{a1} more valuable than \textit{a36}.

### 3.5 Experimental Results

In this section we present experimental results for the assertion ranking methodology. We used GoldMine to generate 260 assertions for the PCI bridge master state machine. We used the assertion ranking methodology to rank the assertions.
3.5.1 Global Variable Importance

<table>
<thead>
<tr>
<th>Output</th>
<th>Positional Rank</th>
<th>Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_ad_out</td>
<td>3</td>
<td>98.1203</td>
</tr>
<tr>
<td>pci_cbe_out</td>
<td>5</td>
<td>96.6165</td>
</tr>
<tr>
<td>pci_frame_en_out</td>
<td>14</td>
<td>89.8496</td>
</tr>
<tr>
<td>pci_cbe_en_out</td>
<td>15</td>
<td>89.0977</td>
</tr>
<tr>
<td>pci_frame_out</td>
<td>19</td>
<td>86.0902</td>
</tr>
<tr>
<td>pci_irdy_out</td>
<td>21</td>
<td>84.5865</td>
</tr>
<tr>
<td>pci_frame_load_out</td>
<td>22</td>
<td>83.8346</td>
</tr>
<tr>
<td>pci_ad_en_out</td>
<td>23</td>
<td>83.0827</td>
</tr>
<tr>
<td>ad_load_out</td>
<td>26</td>
<td>80.8271</td>
</tr>
<tr>
<td>retry_out</td>
<td>27</td>
<td>80.0752</td>
</tr>
<tr>
<td>data_out</td>
<td>35</td>
<td>74.0602</td>
</tr>
<tr>
<td>rtransfer_out</td>
<td>47</td>
<td>65.0376</td>
</tr>
<tr>
<td>first_out</td>
<td>52</td>
<td>60.5263</td>
</tr>
<tr>
<td>wait_out</td>
<td>56</td>
<td>57.5188</td>
</tr>
<tr>
<td>pci_irdy_en_out</td>
<td>58</td>
<td>56.015</td>
</tr>
<tr>
<td>ad_load_on_transfer_out</td>
<td>61</td>
<td>53.0075</td>
</tr>
<tr>
<td>rerror_out</td>
<td>62</td>
<td>52.2556</td>
</tr>
<tr>
<td>pci_req_out</td>
<td>70</td>
<td>44.7368</td>
</tr>
<tr>
<td>mabort_out</td>
<td>71</td>
<td>43.985</td>
</tr>
<tr>
<td>wtransfer_out</td>
<td>81</td>
<td>30.0752</td>
</tr>
</tbody>
</table>

Figure 3.6: The positional rank and positional rank percentile of each output based upon the output’s global importance score.

The first experiment validates the global variable importance computation. Figure 3.6 shows the positional rank and positional rank percentile of each output in the PCI bridge master state machine. A variable’s positional rank is equal to its numerical position in a list of all variables descendingly ordered according to their global importance scores. In general, figure 3.6 shows that output variables are ranked highly.
3.5.2 Ranking

The next set of experiments validate rank score computation. Figure 3.7 correlates assertion importance and complexity. The figure shows that assertion importance weakly correlates with complexity. Therefore, despite the similarity of algorithms 2 and 3, assertion importance and complexity are not dual metrics.
Figure 3.8: The average antecedent size of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity and rank percentiles.
Figures 3.8 and 3.9 show the average antecedent size and temporal length of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity and rank percentiles. We observe the following from these figures. First, assertions that have high importance also have high antecedent sizes and high temporal length. Second, assertions that have low complexity also have low antecedent sizes and low temporal length. Finally, assertions that have high rank also have moderate antecedent sizes and high temporal length.
Figures 3.10 and 3.11 show the average importance and complexity of assertions in the 99th, 95th, 90th, 75th and 50th rank percentile. We observe
the following from these figures. First, assertions that have high rank also have high importance. Second, assertions that have high rank also have low complexity. Finally, assertions that have high rank will not necessarily have the highest importance.

3.5.3 Ideality

<table>
<thead>
<tr>
<th>Target</th>
<th>Minimum Ideality</th>
<th>Median Ideality</th>
<th>Maximum Ideality</th>
</tr>
</thead>
<tbody>
<tr>
<td>pci_irdy_out</td>
<td>0.809296</td>
<td>0.809407</td>
<td>0.809519</td>
</tr>
<tr>
<td>wtransfer_out</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pci_ad_en_out</td>
<td>0.00137253</td>
<td>0.394661</td>
<td>0.792062</td>
</tr>
<tr>
<td>pci_frame_load_out</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pci_cbe_en_out</td>
<td>0.0296962</td>
<td>0.373479</td>
<td>0.67663</td>
</tr>
<tr>
<td>pci_irdy_en_out</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pci_frame_out</td>
<td>0.408212</td>
<td>0.424985</td>
<td>1</td>
</tr>
<tr>
<td>wait_out</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>first_out</td>
<td>0.888156</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rerror_out</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pci_frame_en_out</td>
<td>0.0297634</td>
<td>0.36018</td>
<td>0.618062</td>
</tr>
<tr>
<td>retry_out</td>
<td>0.117951</td>
<td>0.117951</td>
<td>0.117951</td>
</tr>
<tr>
<td>ad_load_on_transfer_out</td>
<td>0.072491</td>
<td>0.503257</td>
<td>1</td>
</tr>
<tr>
<td>ad_load_out</td>
<td>0.0530941</td>
<td>0.0579914</td>
<td>0.7411</td>
</tr>
<tr>
<td>pci_req_out</td>
<td>0.0684741</td>
<td>0.377218</td>
<td>0.844746</td>
</tr>
<tr>
<td>rtransfer_out</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.12: The ideality score of the lowest, middle, and highest ranked true assertions for each target variable.

Figure 3.12 shows the ideality of the lowest, middle and highest ranked true assertions for each target variable. For most target variables, GoldMine was able to generate an ideal or nearly ideal assertion. GoldMine was unable to generate an ideal assertion for retry_out.
In chapter 3, we introduced an assertion ranking methodology. Though effective, the methodology considers only two parameters — importance and complexity. Consequently, the methodology might misjudge an assertion’s value. For example, if a designer cares only about an assertion’s importance and coverage regardless of its complexity, then the assertion ranking methodology will be ineffective.

In this chapter, we introduce an assertion rank aggregation methodology. Rank aggregation algorithms combine ranking results from various sources to generate an optimal ranking [22, 23]. Rank aggregation algorithms are capable of combining an arbitrary number of rankings. The proposed methodology aggregates assertion rankings for importance, complexity, coverage and expectedness.

4.1 Related Work and Background

Let \( \Pi \) denote a Verilog design with variables \( V \). We refer to \( \Pi \) as the target design. We define a simulation \( S \) of \( \Pi \) as follows. Let \( \mathbb{B} = \{0, 1\} \) denote the Boolean number set. Let \( S \) be a set of Boolean vectors and let \( s \in S \) be a Boolean vector in \( \mathbb{B}^{|V|} \) that assigns a Boolean value to each \( v \in V \). Let \( s(v) \) denote the value of variable \( v \) in \( s \).

4.1.1 Assertion Coverage

Assertion coverage estimates the fraction of design functionality covered by an assertion. Previous work [53, 54] has proposed methods to compute the fraction of the state space covered by an assertion. However, these methods are intractable since they must enumerate the entire state space of the de-
sign. In [55], the authors propose the *correctness based coverage algorithm* to compute the fraction of RTL statements covered by an assertion.

The correctness based coverage algorithm computes the set of statements executed after the satisfaction of an assertion’s antecedent. If an executed statement transitively satisfies the assertion’s consequent, then the assertion covers that statement. The algorithm proceeds as follows. Let $p$ denote an assertion with temporal length $k$. First, the algorithm initializes $S$ so that it satisfies the antecedent of $p$. Next, the algorithm performs an event-based simulation of $\Pi$ for $k$ temporal frames. During simulation, the algorithm records the statement that assigns each $v \in V$ in each temporal frame. Finally, the algorithm adds the transitive set of statements that satisfied each proposition in the consequent of $p$ to the set of statements it covers.

4.1.2 Rank Aggregation

Overview

Social choice theory establishes the theoretical basis for rank aggregation. Informally, social choice theory analyzes the problem of using individual preferences to make a collective decision. Social choice theory dates back to the late eighteenth century. Seminal works include those of Borda [56], Condorcet [57], and Arrow [58].

We formalize rank aggregation as follows. Let $X$ denote a finite set of $n$ alternatives, let $\geq$ denote an ordering relation on $X$ and let $\tau = [x_1 \geq x_2 \geq \cdots \geq x_n]$ be an ordering of the alternatives in $X$. We refer to $\tau$ as a *ranking*. Let $\tau(x_i)$ denote the position of $x_i$ in ranking $\tau$. We refer to $\tau(x_i)$ as the *rank* of $x_i$. Let $T$ denote a finite set of rankings of the alternatives in $X$. Rank aggregation uses the rankings in $T$ to find an optimal ranking $\tau_\alpha$.

Properties

There has been considerable research regarding the desirable properties of a rank aggregation. In [57], Condorcet proposed that the winning alternative should be the one preferred by the majority of voters in pairwise contests against all other alternatives. Rank aggregation methods that can find such an alternative satisfy the *Condorcet criterion*.
In [59], Truchon proposed an extension to the Condorcet criterion. Suppose we partitioned the alternatives into subsets \( Y \) and \( Z \). In addition, suppose that for any \( y \in Y \) and \( z \in Z \), the majority of voters prefer \( y \) over \( z \). Rank aggregation methods that can find such a partition satisfy the extended Condorcet criterion. The extended Condorcet criterion is less strict than the Condorcet criterion.

In [58], Arrow proposed a set of desirable properties for rank aggregation methods and proved that no method choosing between more than 2 alternatives can satisfy the properties simultaneously. Social choice theory refers to this result as Arrow’s impossibility theorem. Arrow’s impossibility theorem implies that a “perfect” rank aggregation method cannot exist. In other words, all rank aggregation methods will be unfair in some sense.

Arrow’s criteria included non-dictatorship, unrestricted-domain, independence of irrelevant attributes, and Pareto efficiency. We summarize these criteria as follows:

- **Non-dictatorship** — a rank aggregation method should consider the preferences of each ranking equally.

- **Unrestricted-domain** — a rank aggregation method should consider the preferences of all rankings.

- **Independence of irrelevant attributes** — a rank aggregation method’s preference between alternatives \( x_1 \) and \( x_2 \) should depend only on the rankings’ preferences between \( x_1 \) and \( x_2 \).

- **Pareto efficiency** — a rank aggregation method should prefer \( x_1 \) over \( x_2 \) if all rankings prefer \( x_1 \) over \( x_2 \).

**Distance Measures**

The Spearman footrule distance and Kendall tau distance are the most common measures used to quantify the distance between a pair of rankings. The Spearman footrule distance computes the absolute difference in an alternative’s rank between a pair of rankings and sums this quantity for all alternatives. Formally, we compute the Spearman footrule distance \( S(\tau, v) \)
between rankings \( \tau \) and \( v \) as follows:

\[
S(\tau, v) = \sum_{x_i \in X} | \tau(x_i) - v(x_i) | \quad (4.1)
\]

The Kendall tau distance computes the number of pairwise disagreements between a pair of rankings. The Kendall tau distance is also known as the “bubble sort” distance since it computes the number of swaps required by the bubble sort algorithm to transform one ranking into another. Formally, we compute the Kendall tau distance \( K(\tau, v) \) between rankings \( \tau \) and \( v \) as follows:

\[
K(\tau, v) = \left| \{(x_i, x_j) \mid i < j, \text{sgn}(\tau(x_i) - \tau(x_j)) \neq \text{sgn}(v(x_i) - v(x_j))\} \right| \quad (4.2)
\]

Methods

In [56], Borda proposed a simple rank aggregation method known as the Borda count method. The Borda count method computes a Borda score \( B(x) \) for each alternative \( x_i \in X \). We define \( B(x) \) as follows:

\[
B(x) = \sum_{\tau_i \in T} \tau_i(x) \quad (4.3)
\]

The Borda count method aggregates the rankings in \( T \) by ascendingly ordering the alternatives in \( X \) according to their Borda scores. The Borda count method satisfies neither the Condorcet nor the extended Condorcet criteria. In addition, Previous work has shown that this method is relatively ineffective [22].

In [60], Kemeny proposed a rank aggregation method known as Kemeny-Young method. The Kemeny-Young method computes a Kemeny optimal aggregation by minimizing the sum of the Kendall tau distances between the aggregation and rankings. This method satisfies the Condorcet, extended Condorcet, non-dictatorship, unrestricted-domain and Pareto efficiency criteria. Consequently, the Kemeny-Young method computes a high-quality aggregation. However, it is computationally prohibitive when \( |X| > 3 \) [22].
4.2 Methodology

In this section, we detail the assertion rank aggregation methodology. The methodology consists of three phases. The first phase computes a set of rankings for a set of assertions. The second phase computes a rank aggregation for the assertions. The final phase uses a process called local Kemenization to optimize the aggregation.

4.2.1 Ranking

The first phase of the rank aggregation methodology computes a set of rankings for the assertions. In general, this phase will depend highly on the metrics selected for aggregation. We compute an assertion rank aggregation using four metrics — assertion importance, complexity, coverage and expectedness. We detail assertion importance and complexity in chapter 3. In short, assertion importance and complexity estimate the importance and understandability of the behavior conveyed by an assertion respectively.

Coverage

We modify the correctness-based coverage algorithm to consider only the set of assignments covered by an assertion. Let \( \alpha \) denote the total number of assignments in \( \Pi \) and let \( \alpha_p \) denote the number of assignments covered by \( p \). We compute the coverage \( R(p) \) of \( p \) as follows:

\[
R(p) = \frac{\alpha_p}{\alpha}
\]  

Since each assignment constitutes an execution path in the design, equation 4.4 effectively computes an assertion’s execution path coverage.

Expectedness

Assertion expectedness estimates the probability that an assertion’s antecedent will be satisfied — assuming the design’s inputs vary uniform-randomly. Assertions with low expectedness convey design behavior that occurs rarely. Consequently, we rank assertions with low expectedness higher than those
with high expectedness. We use simulation techniques to estimate an assertion’s expectedness.

Let \( p \) denote an assertion and let \( A \) denote the set of propositions in the antecedent of \( p \). We compute the expectedness \( E(p) \) of \( p \) as follows:

\[
E(p) = \frac{\left| \{ s \in S \mid \forall (v, b) \in A, s(v) = b \} \right|}{|S|} \tag{4.5}
\]

Equation 4.5 computes an assertion’s simulation trace coverage.

### 4.2.2 Aggregation

We use the footrule aggregation algorithm to aggregate a set of assertion rankings since it computes an aggregation that approximates that of the Kemeny-Young method [22]. Let \( \tau_a \) denote a rank aggregation of \( n \) assertions. Let \( G = (P, R, E) \) denote a bipartite graph with nodes \( P \) and \( R \) and weighted edges \( E \). Each node \( p \in P \) denotes an assertion and each node \( r \in R \) denotes one of \( n \) available ranks in \( \tau_a \). Note that a perfect matching for \( G \) implicitly defines an assertion ranking.

Let \( T \) denote a set of assertion rankings. We compute the weight \( W(p, r) \) of edge \( (p, r) \in E \) as follows:

\[
W(p, r) = \sum_{\tau_i \in T} |\tau_i(p) - r| \tag{4.6}
\]

We explain the intuition behind equation 4.6 as follows. Suppose we assigned assertion \( p \) to rank \( r \) in a hypothetical \( \tau_a \). For each \( \tau_i \in T \), equation 4.6 computes the Spearman footrule distance between \( r \) and \( \tau_i(p) \). Because the equation’s final result is equal to the sum of these distances, it effectively computes the cost of assigning assertion \( p \) to rank \( r \) in a hypothetical \( \tau_a \).

An optimal assertion rank aggregation will minimize \( W(p, r) \) for each \( p \in P \). Therefore, we would like to compute a minimum cost perfect matching for \( G \). Such a matching will implicitly define an optimal assertion rank aggregation. We use the Kuhn-Munkres algorithm [61] to compute a minimum cost perfect matching for \( G \). Since this algorithm is well-documented, we do not detail it here.
4.2.3 Local Kemenization

We use a process called local Kemenization [22, 23] to optimize the assertion rank aggregation. Local Kemenization transforms an aggregation into a locally Kemeny optimal aggregation. Such an aggregation is a relaxation of a Kemeny optimal aggregation and is guaranteed to satisfy the extended Condorcet criteria. Unlike the Kemeny-Young aggregation method, local Kemenization is computationally tractable.

Let $K(a; T)$ denote the sum of the Kendall tau distances between $a$ and the rankings in $T$. Aggregation $a$ is locally Kemeny optimal if swapping an adjacent pair of assertions in $a$ cannot reduce $K(a; T)$. Consequently, we can locally Kemenize $a$ by swapping adjacent pairs of assertions in the aggregation until $K(a; T)$ cannot be reduced further.

4.2.4 Example

We present an example of the assertion rank aggregation methodology. Let $p_0$, $p_1$ and $p_2$ denote assertions and let $\tau_0 = [p_0 \geq p_1 \geq p_2]$, $\tau_1 = [p_1 \geq p_0 \geq p_2]$ and $\tau_2 = [p_2 \geq p_1 \geq p_0]$ denote rankings for $p_0$, $p_1$ and $p_2$. The rank aggregation methodology computes $W(p, r)$ for each assertion and rank as follows:

$$W(p_0, 0) = |\tau_0(p_0) - 0| + |\tau_1(p_0) - 0| + |\tau_2(p_0) - 0| = 0 + 1 + 2 = 3$$
$$W(p_0, 1) = 2$$
$$W(p_0, 2) = 3$$
$$W(p_1, 0) = 2$$
$$W(p_1, 1) = 1$$
$$W(p_1, 2) = 4$$
$$W(p_2, 0) = 4$$
$$W(p_2, 1) = 3$$
$$W(p_2, 2) = 2$$

Next, the rank aggregation methodology uses the Kuhn-Munkres algorithm to compute $\tau_a$. Figure 4.1 shows an example of the algorithm. The left bipartite graph in the figure uses each $W(p, r)$ value to define all rank
aggregations. The algorithm optimizes the left graph until the right bipartite graph in figure 4.1 remains. The right graph uses an optimal set of $W(p, r)$ values to define $\tau_a = [p_1 \geq p_0 \geq p_2]$.

Next, the rank aggregation methodology locally Kemenizes $\tau_a$. The methodology computes $K(\tau_a, T)$ as follows:

\[
\begin{align*}
K(\tau_a, \tau_0) &= | \{(p_0, p_1)\}| = 1 \\
K(\tau_a, \tau_1) &= 0 \\
K(\tau_a, \tau_2) &= 2 \\
K(\tau_a, T) &= K(\tau_a, \tau_0) + K(\tau_a, \tau_1) + K(\tau_a, \tau_2) = 3
\end{align*}
\]

The rank aggregation methodology checks whether or not swapping any adjacent pair of assertions in $\tau_a$ will reduce $K(\tau_a, T)$. The methodology computes $K(\tau_a' = [p_0 \geq p_1 \geq p_2], T)$ as follows:

\[
\begin{align*}
K(\tau_a', \tau_0) &= 0 \\
K(\tau_a', \tau_1) &= 1 \\
K(\tau_a', \tau_2) &= 3 \\
K(\tau_a', T) &= 4
\end{align*}
\]
Since $K(\tau'_a, T) \not< K(\tau_a, T)$, the rank aggregation methodology tries swapping another adjacent pair of assertions in $\tau_a$. The methodology computes $K(\tau'_a = [p_1 \geq p_2 \geq p_0], T)$ as follows:

$$K(\tau'_a, \tau_0) = 1$$
$$K(\tau'_a, \tau_1) = 1$$
$$K(\tau'_a, \tau_2) = 1$$
$$K(\tau'_a, T) = 3$$

Again, $K(\tau'_a, T) \not< K(\tau_a, T)$. Therefore, $\tau_a$ is locally Kemeny optimal since it does not contain another adjacent pair of assertions.

### 4.3 Experimental Results

In this section we present experimental results for the rank aggregation methodology. We used GoldMine to generate 260 assertions for the PCI bridge master state machine. We used the rank aggregation methodology to aggregate rankings for assertion importance, complexity, coverage and expectedness.

![Complexity versus Importance](image)

Figure 4.2: Assertion complexity as a function of importance.
Figure 4.3: Assertion coverage as a function of importance.

Figure 4.4: Assertion coverage as a function of complexity.
Figure 4.5: Assertion expectedness as a function of importance.

Figure 4.6: Assertion expectedness as a function of complexity.
Figures 4.2 through 4.7 correlate each pair of assertion metrics. These figures show that each assertion metric weakly correlates with each other metric. Therefore, these metrics’ rankings are ideal for aggregation since no ranking will credit or discredit any other.

![Expectedness versus Coverage](image)

Figure 4.7: Assertion expectedness as a function of coverage.

![Kendall Tau Distance between Rankings](image)

Figure 4.8: Kendall tau distance between a ranking for assertion rank and a rank aggregation for assertion importance and complexity.
Figure 4.8 shows the Kendall tau distance between a ranking for assertion rank and a rank aggregation for assertion importance and complexity. The figure shows that the ranking methodology presented in chapter 3 and the rank aggregation methodology are not equivalent. Therefore, the rank aggregation methodology interprets assertion importance and complexity differently than the assertion ranking methodology.

Figure 4.9: The average importance of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles.
Comparison of Average Complexity for all Rankings

Figure 4.10: The average complexity of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles.
Figure 4.11: The average coverage of assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles.
Figures 4.9, 4.10, 4.11 and 4.12 show the average importance, complexity, coverage and expectedness for assertions in the for assertions in the 99th, 95th, 90th, 75th and 50th importance, complexity, coverage, expectedness and rank aggregation percentiles. We observe the following from these figures. First, assertions that have high importance also have high complexity, moderate coverage and moderate expectedness. Second, assertions that have low complexity also have low importance, low coverage and high expectedness. Third, assertions that have high coverage also have moderate importance, high complexity and low expectedness. Fourth, assertions that have low expectedness also have moderate importance, high complexity and moderate coverage. Fifth, highly ranked assertions have high importance, low complexity, high coverage and low expectedness. Second, lowly ranked assertions never outperform highly ranked assertions with respect to any met-
ric. Finally, the rank aggregation methodology fairly compromises between the metrics.
In this chapter, we present experimental results that compare GoldMine’s data mining algorithms with respect to importance, complexity, ideality, coverage and expectedness. We used each algorithm to generate assertions for three Verilog modules. We used the methodology from chapter 4 to aggregate rankings for assertion importance, complexity, coverage and expectedness. The first module is the protocol engine from the Universal Serial Bus (USB) protocol, the second module is the master state machine from the Peripheral Component Interconnect (PCI) protocol and the final module is the decode pipeline stage from the OpenRisc 1200 (OR1200) CPU. For all experiments, the data generator simulated each module for 10000 cycles using an unconstrained random test bench. We explicitly limited the antecedent size and temporal length of all assertions to 5 and 2 respectively. We conducted all experiments using a 2.67 gigahertz quad core Intel Core i5 with 16 gigabytes of memory.
Figure 5.1: The average normalized importance of 25% highest ranked assertions that passed formal verification for each data mining algorithm.

Figure 5.1 shows the average normalized importance of the 25% highest ranked assertions that passed formal verification for each data mining algorithm. We observe the following from these results. First, there is no algorithm that always generates assertions that have high importance. Therefore, we should select a data mining algorithm based on the target design’s behavior. Second, the coverage mining algorithm does not generate assertions that have high importance. We expect this since concise assertions are more likely to have lower importance than verbose assertions.
Figure 5.2 shows the average normalized complexity of the 25% highest ranked assertions that passed formal verification for each data mining algorithm. We observe the following from these results. First, we should select the coverage mining algorithm if we wish to generate assertions that have low complexity. Second, the decision forest algorithm usually generates assertions that have lower complexity than those generated by the decision tree algorithm. Finally, the prism algorithm usually generates assertions that have low complexity.
Figure 5.3: The average normalized ideality of 25% highest ranked assertions that passed formal verification for each data mining algorithm.

Figure 5.3 shows the average normalized ideality of the 25% highest ranked assertions that passed formal verification for each data mining algorithm. We observe the following from these results. First, we should select the coverage mining or prism algorithms if we wish to generate assertions that consistently have high ideality. Finally, the decision tree based algorithms perform similarly with respect to ideality.
Figure 5.4 shows the average normalized coverage of the 25% highest ranked assertions that passed formal verification for each data mining algorithm. We observe the following from these results. First, there is no algorithm that always generates assertions that have high coverage. Therefore, we should select a data mining algorithm based on the target design’s behavior. Second, the decision tree based algorithms perform similarly with respect to coverage. Finally, the coverage mining and prism algorithms also perform similarly with respect to coverage.
Figure 5.5: The average normalized expectedness of 25% highest ranked assertions that passed formal verification for each data mining algorithm.

Figure 5.5 shows the average normalized expectedness of the 25% highest ranked assertions that passed formal verification for each data mining algorithm. We observe the following from these results. First, we should select the decision tree based algorithms if we wish to generate assertions that consistently have low expectedness. Second, the coverage mining algorithm always generates assertions that have high expectedness.
In this work, we detailed the GoldMine methodology and its algorithms. We introduced various metrics and two methodologies for evaluating a set of assertions. Together, GoldMine and the assertion ranking methodologies can change Verification forever. Instead of spending effort refining manually generated assertions, we can use these methods to generate assertions that surpass those crafted by human beings. Eventually, GoldMine and the assertion ranking methodologies will bring the day when we no longer require human effort to generate assertions.
In this chapter, we discuss how to obtain and use GoldMine to generate and rank a set of assertions.

7.1 Obtaining GoldMine

To obtain GoldMine, navigate to http://goldmine.csl.illinois.edu, register for an account and sign in.

7.2 Executing GoldMine

7.2.1 Requirements

- **Synopsys Verilog Compiler Simulator (VCS)** – GoldMine uses VCS to generate random simulation data. GoldMine can accept VCD files on the command line if VCS is not available.

- **Cadence Incisive Formal Verifier (IFV)** – GoldMine uses IFV to formally verify the assertions it generates. GoldMine will label the assertions it generates as “unverified” if IFV is not available.

7.2.2 Quick Start

GoldMine can be executed using the following command:

```
goldmine verilog/arb2.v
```

GoldMine does not require VCS to generate assertions for the Verilog file `arb2.v` because we have provided the VCD file `goldmine.out/arb2/arb2.vcd`. 
7.2.3 Usage

GoldMine can be executed using the following command:

```bash
goldmine [options] <input_files>
```

GoldMine will parse the input files, simulate a random testbench and generate a set of assertions for the top module in the design. GoldMine will attempt to find the top module, clock signal and reset signal automatically. By default, GoldMine will select each scalar output from the top module as a target for assertion generation. Each of these execution parameters can be specified using command line options.

GoldMine will generate the following files in the directory goldmine.out:

- `<top_module>/benchmark.v` – The testbench used to simulate `<top_module>`.
- `<top_module>/gold` – A detailed analysis of each assertion generated for `<top_module>`. The assertions are listed descendingly according to rank.
- `<top_module>/log` – The command line output generated by GoldMine.
- `<top_module>/vcd` – The simulation data used to generate the assertions.
- `<top_module>/rank` – The global importance score for each variable in `<top_module>`.
- `<top_module>/cone` – A detailed analysis of each variable in the bounded cone of influence for `<target>`.
- `<top_module>/gold` – A detailed analysis of each assertion generated for `<target>`. The assertions are listed descendingly according to rank.

7.2.4 Options

- `-h [ --help ]` – GoldMine will print the list of command line options and exit.
-p [–parse ] – GoldMine will parse the input files and exit.

-m [–module ] arg – GoldMine will generate assertions using <arg> as the top module.

-c [–clock ] arg – GoldMine will generate assertions using <arg> as the clock signal.

-r [–reset ] arg – GoldMine will generate assertions using <arg> as the reset signal.

-target-vectors – GoldMine will generate assertions for both output scalars and vectors.

-t [–targets ] arg – GoldMine will generate assertions using <arg> as the target signals.

-v [–vcd ] arg – GoldMine will generate assertions using <arg> as the simulation data.

-e [–engine ] arg – GoldMine will generate assertions using <arg> as the assertion mining engine. You can select either the decision tree <arg = tree>, best-gain decision forest <arg = forest>, coverage miner <arg = coverage> or prism <arg = prism> engine.

-rank – GoldMine will rank assertions using the assertion ranking methodology instead of the rank aggregation methodology.

### 7.2.5 Configuration

GoldMine will parse any additional options in the configuration file goldmine.cfg. You can maximize a numerical parameter by assigning it a value of +. A description of the options in the configuration file follows.

- vcs_home:arg – VCS_HOME environment variable

- synopsys_license:arg – LM_LICENSE_FILE environment variable

- ifv_home:arg – IFV_ROOT environment variable

- cadence_license:arg – CDS_LICENSE_FILE environment variable
- **engine:**\texttt{arg} – Assertion mining engine
- **num\_examples:**\texttt{arg} – Maximum number of simulation cycles
- **num\_cycles:**\texttt{arg} – Maximum temporal length of an assertion
- **num\_propositions:**\texttt{arg} – Maximum antecedent size of an assertion
- **num\_partitions:**\texttt{arg} – Maximum number of partitions per iteration of the Decision Forest algorithm
- **num\_counterexamples:**\texttt{arg} – Maximum number of counterexamples used to refine assertions
REFERENCES


