Behavioral Signature-based Framework for Identifying Unsatisfiable Variable Mappings between Digital Designs

Vaibhav Uday Tendulkar
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/552

This Dissertation is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.
Behavioral Signature-based Framework for Identifying Unsatisfiable Variable Mappings between Digital Designs

A dissertation proposal submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

By

VAIBHAV UDAY TENDULKAR
M.S., Electrical Engineering, Wright State University, 2006

2012
Wright State University

__________________________
Travis Doom, Ph.D.
Dissertation Director

__________________________
Arthur A. Goshtasby, Ph.D.
Director, Ph.D. Program,
Department of Computer Science and Engineering

Committee:

__________________________
Dr. Andrew Hsu, Ph.D.
Dean, Graduate School

__________________________
Travis Doom, Ph.D.
Department of Computer Science and Engineering

__________________________
John Gallagher, Ph.D.
Department of Computer Science and Engineering

__________________________
Meilin Liu, Ph.D.
Department of Computer Science and Engineering

__________________________
Henry Chen, Ph.D.
Department of Electrical Engineering

__________________________
Sridhar Ramachandran, Ph.D.
Coordinator of Department of Informatics, Indiana University Southeast
ABSTRACT


Throughout its design process (from specification to implementation) a digital circuit goes through a variety of structural changes. These changes are introduced primarily due to the use of automated tools in the design process. Checking whether the Boolean functions representing the two designs are equivalent is thus necessary to verify if a design implementation adheres to its specification. Combinational Equivalence Checking (CEC) – a process of determining whether two combinational design functions are equivalent, has been one of the most researched Boolean matching problems. The well-known CEC techniques that have been proposed adopt some kind of a formal approach such as Canonical Form (ROBDD, BMDs, etc.) isomorphism or Boolean Satisfiability in order to prove or disprove equivalence. Hybrid techniques that adopt a combination of the above mentioned two techniques have also been proposed.

Knowing the exact nature of variable mappings / correspondences between the two designs a priori is advantageous during the CEC process. However, situations may arise wherein the knowledge of these mappings is unknown or lost. Not knowing the variable mappings between the two designs a priori increases the computational complexity of CEC techniques. The CEC problem, under unknown variable mappings, is a more complex Boolean matching problem – the problem of determining if an input and an output variable mapping/permutation exists under which the two designs are functionally
equivalent. The use of signatures/filters has proven to be a well-known approach taken by the design verification community quickly detect and prune those variable mappings that do not make the two designs equivalent.

In our work we propose and implement three novel output behavior based signatures known as Behavioral signatures. Behavioral signatures are computed solely based on the binary output behavior exhibited by the designs and thus distance themselves from relying on Boolean equations, canonical forms or any other functional representations for their computation. This property makes the Behavioral signatures useful to all digital design domains including those that might not possess any knowledge of the Boolean functions representing the designs that face the problem of determining design equivalence under unknown variable correspondences. The Grouped Row Sum and the Row Difference signatures that we propose are used to identify unsatisfiable input variable mappings between the two designs. Our Grouped Column Sum signature on the other hand is used to identify the unsatisfiable output variable mappings.

The success of a Behavioral signature lies in the fineness with which it summarizes the design behavior. However there exists a tradeoff between the degree of fineness achieved and the execution time. For instance, amongst the input variable signatures that we propose, the Row Difference signature is more adept in summarizing the design behavior than its counterpart the Grouped Row Sum signature. However the computation cost associated with the Row Difference signature is higher as compared to the Grouped
Row Sum signature. We thus define a framework for using these signatures based on the nature of the designs under consideration.
# TABLE OF CONTENTS

**Chapter 1. Introduction** ................................................................. 1

1.1. *P*-Equivalence Checking ......................................................... 3

1.2. Signatures ................................................................................. 5

1.3. Performance Measures .............................................................. 6

1.4. Applications .............................................................................. 8

1.5. Problem Structure ................................................................. 10

1.6. Outline ...................................................................................... 11

**Chapter 2. Background** ............................................................... 12

2.1. CEC Techniques ................................................................. 12

2.1.1. Canonical form-based equivalence check ............................... 13

2.1.2. SAT-based equivalence checking ........................................... 15

2.2. Boolean Signatures ................................................................. 18

2.2.1. Boolean signature and SAT-based approach of Katebi and Markov (2010) ......................................................... 18

2.2.2. Boolean signatures reviewed by Mohnke et al. (2001) .......... 21

2.3. Behavioral Signature based approach of Doom and Leighber (2001) … 22

2.3.1. Terminologies ........................................................................ 23

2.3.2. Algorithm Structure .............................................................. 27

2.3.3. Behavioral Signatures of Doom and Leighber (2001) ............ 27
Chapter 6. Conclusion and Future Work ............................................ 67

6.1. Framework ............................................................................. 67

6.1.1. Framework Proposal for determining Input Variable Mappings ...... 67

6.1.2. Framework Proposal for determining Output Variable Mappings ..... 70

6.2. Future Work ......................................................................... 73

6.2.1. Design Structure Knowledge ................................................. 73

6.2.2. Improved Input Vectors ......................................................... 74

6.2.3. Decoder-like output behavior ................................................. 76

6.2.4. Symmetric variables ........................................................... 77

6.2.5. Parallel Implementation ....................................................... 78

6.3. Conclusion ......................................................................... 78

APPENDIX A: Source Code .......................................................... 80

BIBLIOGRAPHY .......................................................................... 113
<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Structure of Behavioral signature-based approach for identifying unsatisfiable I/O mappings.</td>
<td>10</td>
</tr>
<tr>
<td>2.1.a, 2.1.b</td>
<td>ROBDD representations of function $f = a \cdot \overline{b}$ under input orderings $(a,b)$ and $(b,a)$ respectively.</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>ROBDD representations of the specification function (left) $Z_s = (\overline{r} \cdot \overline{q}) + (\overline{r} \cdot p)$ and its optimized implementation function $Z_I = ((x_1 \cdot x_2) + x_0)$ (right).</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Transforming the combinational equivalence checking problem to a Boolean SAT problem.</td>
<td>17</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison of computational speedups achieved by Grouped Row Sum and Row Difference Signatures over the Weighted Row Sum Signature</td>
<td>69</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparison of percentage coverage of 1-1 mapped inputs achieved by Weighted Row Sum, Grouped Row Sum and Row Difference Signatures</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>Comparison of computational speedups achieved by the Grouped Column Sum signature when coupled with Weighted Row Sum (dotted line) and Grouped Row Sum signature (solid line) over the Column Sum Signature</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparison of percentage coverage of 1-1 mapped outputs achieved by the Grouped Column Sum signature when coupled with Weighted Row Sum and Grouped Row Sum signature over the Column Sum Signature</td>
<td>72</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table no.</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1, 2.2</td>
<td>Behavior of 9-input 6-output Designs $\vec{F}$ and $\vec{G}$ respectively, when input vectors to test each of their inputs are applied.</td>
<td>29</td>
</tr>
<tr>
<td>2.3, 2.4</td>
<td>Behavior of 9-input 6-output Designs $\vec{F}$ and $\vec{G}$ respectively, when input vectors to test each of their inputs are applied wherein the design inputs have been ordered on the basis of their decreasing $RSum$ values.</td>
<td>29</td>
</tr>
<tr>
<td>2.5, 2.6</td>
<td>Behavior of 9-input 6-output Designs $\vec{F}$ and $\vec{G}$ respectively, when input vectors to test each of their inputs are applied. The outputs have been ordered based on decreasing values of their $CSum$ signature.</td>
<td>31</td>
</tr>
<tr>
<td>3.1, 3.2</td>
<td>Behavior of 4-input 6-output Designs $\vec{F}$ and $\vec{G}$ respectively when input vectors used to test each of their inputs are applied. The inputs have been ordered based on decreasing values of their $RSum$ and $WRSum$ values whereas the outputs have been ordered based on decreasing values of their $CSum$ signature.</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Grouped Row Sum signatures of $\vec{F}$ and $\vec{G}$’s input variables. The signatures are computed based on the behavior of $\vec{F}$ and $\vec{G}$ illustrated in Tables 3.1 and 3.2 respectively.</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Improvement in number of input variables mapped when Group Row Sum signatures are used for the mentioned LGSynth’93 and ITC’99 circuits.</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Computation time required by Weighted Row Sum and Grouped Row Sum Signatures for circuits mentioned in Table 3.4.</td>
<td>42</td>
</tr>
<tr>
<td>3.6</td>
<td>Speedup achieved by the use of Grouped Row Sum signature in comparison to the Weighted Row Sum signature.</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>Performance comparison of Grouped Row Sum signatures to the signature based approach proposed by Katebi and Markov (2010) for ITC’99 TORINO benchmark circuits.</td>
<td>43</td>
</tr>
<tr>
<td>4.1, 4.2</td>
<td>Behavior of 6-input 6-output Designs $\vec{F}$ and $\vec{G}$ respectively when input vectors used to test their inputs are applied. The inputs have been ordered based on decreasing values of their $RSum$, $WRSum$ and $GRSum$ signature values whereas the outputs have been ordered based on decreasing values of their $CSum$ signature</td>
<td>46</td>
</tr>
<tr>
<td>Table no.</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.3, 4.4</td>
<td>Computed Row Difference signature arrays (RDiff₀ and RDiff₁) of Design $\bar{F}$ and $\bar{G}$’s input variables. Behavior of $\bar{F}$ and $\bar{G}$ illustrated in Tables 4.1 and 4.2 is considered.</td>
<td>49</td>
</tr>
<tr>
<td>4.5</td>
<td>Improvement of Row Difference signature over Weighted Row Sum and Grouped Column Sum signature in mapping input variables.</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>Computation time required by Weighted Row Sum, Grouped Row Sum Signatures and Row Difference signatures to map inputs of the circuits mentioned in Table 4.5.</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Performance comparison of Row Difference signatures to the signature based approach proposed by Katebi and Markov (2010) for the ITC’99 TORINO benchmark circuits.</td>
<td>54</td>
</tr>
<tr>
<td>5.1, 5.2</td>
<td>Behavior of 6-input 6-output Designs $\bar{F}$ and $\bar{G}$ respectively, when input vectors used to test each of their inputs are applied.</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Grouped Column Sum ($GCSum$) signatures of $\bar{F}$ and $\bar{G}$’s output variables. These signatures are computed based on the behavior of $\bar{F}$ and $\bar{G}$ illustrated in Tables 5.1 and 5.2 respectively.</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance improvement achieved by $GCSum$ signature in detecting unsatisfiable output mappings over the $CSum$ signature when they are coupled alongside the $RSum$ and $WRSum$ input variable signatures for the shown LGSynth93 and ITC’99 Benchmarks.</td>
<td>61</td>
</tr>
<tr>
<td>5.5</td>
<td>The computational speedup achieved by $GCSum$ signature in detecting unsatisfiable output mappings over the $CSum$ signature when they are coupled alongside the $RSum$ and $WRSum$ input variable signatures for the shown LGSynth93 and ITC’99 Benchmarks.</td>
<td>62</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance improvement achieved by $GCSum$ signature in detecting unsatisfiable output mappings over the $CSum$ signature when they are coupled alongside the $RSum$ and $GRSum$ input variable signatures for the shown LGSynth93 and ITC’99 Benchmarks.</td>
<td>63</td>
</tr>
<tr>
<td>Table no.</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.7</td>
<td>The computational speedup achieved by $GCSum$ signature in detecting unsatisfiable output mappings over the $CSum$ signature when they are coupled alongside the $RSum$ and $WRSum$ input variable signatures for the shown LGSynth93 and ITC’99 Benchmarks.</td>
<td>64</td>
</tr>
<tr>
<td>5.8</td>
<td>Performance comparison of $GCSum$ signature to the signatures proposed by Katebi and Markov (2010) in detecting unsatisfiable output mappings.</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>Circuits exhibiting equal-sized input and output variable groups.</td>
<td>77</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my dissertation director and advisor Dr. Travis Doom for his unconditional support, motivation and guidance. I am also thankful to all the committee members (Dr. Sridhar Ramachandran, Dr. John Gallagher, Dr. Henry Chen and Dr. Meilin Liu) for their valuable suggestions and constant encouragement. Last but not the least I would like to sincerely thank the chair (Dr. Mateen Rizki) and the administration (special mentions: Val Smith, Paula Price, Jennifer Limoli, Matt Kijowski) of the Department of Computer Science and Engineering at Wright State University for their firm support.
DEDICATED TO
GOD, MY PARENTS, FAMILY and FRIENDS
Chapter 1

Introduction

Throughout its design process from specification to tape-out, a digital circuit may undergo a series of structural changes at various levels (Eijk, 1997). These structural changes can be introduced into the circuit because of reasons like inclusion of additional testing circuitry, changes realized during synthesis and optimization, etc. (Hulgaard et al., 1999; Eijk, 1997). It thus becomes mandatory to check if these changes introduce an erroneous change in the behavior of the circuit. In other words, it becomes mandatory to verify whether the architectural implementation (gate-level netlist) obtained at the end of a particular stage (logic synthesis) in the design process is equivalent to its behavioral specification (RTL design). A simple way to perform such a check is to observe whether the binary output behavior exhibited by the two designs when a complete set of input vectors are applied is the same, i.e. to check if they are behaviorally equivalent. However, for designs with large number of inputs, checking their behavior over a complete set of input vectors becomes infeasible.
Conversely, checking whether the Boolean functions representing the two designs are equivalent is proven to be a more efficient alternative in verifying whether an implementation of a design adheres to its specification. Over the past decade, Combinational Equivalence Checking (CEC) – a process of determining whether two combinational designs are functionally equivalent, has been one of the most researched Boolean matching problems. The well-known functional CEC techniques that have been proposed adopt some kind of a formal approach such as Canonical Form (ROBDD, BMDs, etc.) isomorphism (Abdollahi and Pedram, 2005; Debnath and Sasao, 2004; Ciric and Sechen, 2003; Debnath and Sasao, 1999; Hinsberger and Kolla, 1998; Malik et al., 1988) or Boolean Satisfiability (Goldberg et al., 2001) in order to prove or disprove equivalence. These techniques will be discussed in more detail in Chapter 2. Hybrid techniques that adopt a combination of the above mentioned two techniques have also been proposed (Reda and Salem, 2001; Paruthi and Kuehlmann, 2000; Burch and Long, 1998).

Knowing the exact nature of variable mappings / correspondences between the two designs \textit{a priori} is advantageous during the CEC process. In fact, knowing the register/latch (or register variable) mappings between two sequential designs \textit{a priori}, facilitates the use of CEC techniques in determining their functional equivalence as the state transition functions of these sequential components are combinational in nature (Molitor and Mohnke, 2004, Page 5). However, many-a-times situations arise wherein the knowledge of these mappings is unknown or lost. Molitor and Mohnke (2004, Page 5) noted that often the tools used to synthesize and optimize the designs have restricted naming conventions and thus tend to shorten or change the original variable and wire
names. These naming conversions, if not recorded appropriately, may result in the loss of variable mappings between a specification and its implementation. Most of the modern day synthesis and optimization tools avoid such shortcomings by recording the naming conversions. However, there still exist many different applications for which determination of functional equivalence between two designs is essential but the variable mappings between them are unknown. Such applications have been discussed briefly in Section 1.4.

Not knowing the variable mappings between the two designs a priori increases the computational complexity of CEC techniques. For instance, if the variable mappings are unknown, a ROBDD-based CEC technique must take into consideration each of those \( n! \) ROBDDs of an \( n \) –input design which have been constructed over every possible input variable ordering. This makes the CEC process computationally expensive for large-sized circuits. The CEC problem, under unknown variable mappings is a more complex Boolean matching problem – the PP-equivalence checking problem (Benini and De Micheli, 1997). PP-equivalence checking requires determining if there exists an input and an output variable mapping/permutation under which the two designs are functionally equivalent.

1.1 PP-Equivalence Checking

Definition 1.1: Two multi-output design functions are input-permutation output-permutation (PP) equivalent if they are functionally equivalent under a certain mapping / permutation of their respective input and output variables.
Consider a \( n \)-input \( m \)-output specification function \( \tilde{F}(\tilde{i}) = \tilde{o} \) with input, output and function vectors of \( \tilde{i} = \{i_0, i_1, ..., i_{n-1}\} \), \( \tilde{o} = \{o_0, o_1, ..., o_{m-1}\} \), \( \tilde{F} = \{f_0, f_1, ..., f_{m-1}\} \) respectively and its equal-sized implementation function \( \tilde{G}(\tilde{x}) = \tilde{y} \) with input, output and function vectors of \( \tilde{x} = \{x_0, x_1, ..., x_{n-1}\} \), \( \tilde{y} = \{y_0, y_1, ..., y_{m-1}\} \) and \( \tilde{G} = \{g_0, g_1, ..., g_{m-1}\} \) respectively. The specification and the implementation are \( PP \)-equivalent iff an input mapping \( \Delta_i: \{i_0, i_1, ..., i_{n-1}\} \rightarrow \{x_0, x_1, ..., x_{n-1}\} \) and an output mapping \( \Delta_o: \{o_0, o_1, ..., o_{m-1}\} \rightarrow \{y_0, y_1, ..., y_{m-1}\} \) exist such that \( \forall_k, 0 \leq k < m, f_k(\tilde{i}) = \Delta_o(g_k)(\Delta_i(\tilde{i})) \).

**Example 1.1:** Consider a 3-input 2-output specification function \( \tilde{F}(\tilde{i}) = \tilde{o} \) whose input, output and function vectors are denoted by \( \tilde{i} = \{i_0, i_1, i_2\} \), \( \tilde{o} = \{o_0, o_1\} \) and \( \tilde{F} = \{f_0, f_1\} \) respectively and an equal-sized implementation function \( \tilde{G}(\tilde{x}) = \tilde{y} \) whose input, output, and function vectors are represented by \( \tilde{x} = \{x_0, x_1, x_2\} \), \( \tilde{y} = \{y_0, y_1\} \) and \( \tilde{G} = \{g_0, g_1\} \) respectively. The two functions \( \tilde{F} \) and \( \tilde{G} \) are represented by the following single-output functions:

\[
\begin{align*}
    f_0 &= f(o_0) = i_0 + i_1 \\
    f_1 &= f(o_1) = i_2 \\
    g_0 &= g(y_0) = x_1 \\
    g_1 &= g(y_1) = x_2 + x_0
\end{align*}
\]

There exists an input permutation function \( \Delta_i: \{i_0, i_1, i_2\} \rightarrow \{x_0, x_2, x_1\} \) and an output permutation function \( \Delta_o: \{o_0, o_1\} \rightarrow \{y_1, y_0\} \) that makes \( \tilde{F} \) and \( \tilde{G} \) \( PP \)-equivalent.
1.2 Signatures

If the variable mappings between two \( n \)-input \( m \)-output designs are not known \( a \) priori a total of \( n! \) input and \( m! \) output variable mapping combinations must be considered during the CEC process thus making it computationally more complex. The use of signatures has proven to be well-known approach taken by the design verification community to reduce this complexity by quickly detecting and pruning those variable mappings that do not make the two designs equivalent.

Signatures are permutation-independent metrics associated to a variable which are computed based on its role in the design’s behavior. A variable mapping in which a variable from one design does not possess the same signature as the corresponding variable of the other design featured in that mapping does not satisfy the equivalence criteria and can be eliminated from consideration during the CEC process. For instance, an input variable mapping between two \( n \)-input designs \( \Delta_{i}:(i_{0}, i_{1}, \ldots, i_{n-1}) \leftrightarrow (x_{0}, x_{1}, \ldots, x_{n-1}) \) is unsatisfiable and can hence be eliminated from consideration if for any \( k \) (\( 0 \leq k \leq n - 1 \)), \( \text{Sig}(i_{k}) \neq \text{Sig}(x_{k}) \). Fast and effective signature computation methodologies always aim towards proposing signatures that take minimum computation time and eliminate most of the unsatisfiable mappings. Two designs are functionally not equivalent if all of the possible input (or output) variable mappings are identified as unsatisfiable.

Signatures can be broadly classified into two categories – Boolean signatures and Behavioral signatures. The more popular Boolean signatures (Katebi and Markov, 2010;
Abdollahi and Pedram 2008; Abdollahi 2008; Chai and Kuehlmann, 2006; Abdollahi and Pedram, 2005; Mohnke et al., 2001) are computed based on the characteristics exhibited by the design functions or their canonical forms or any other function representations (like CNF). On the other hand, the Behavioral signatures (Doom and Leighber, 2001) are computed based only on the binary output behavior exhibited by the designs when a specific set of binary input vectors are applied to them. This computational approach makes the Behavioral signatures useful to domains that face the problem of determining an input and an output variable mapping between two designs that satisfy the equivalence criteria but do not possess any knowledge of functions representing the two designs. Our research goal is to propose and implement a set of novel and effective Behavioral Signatures.

1.3 Performance Measures

A one-to-one / bijection input (output) variable mapping is said to exist between two designs if it is the only mapping that makes the two designs equivalent. In a bijection variable mapping every input (output) of one design possesses a signature that is only equal to the signature of the corresponding input of the other design that features in that mapping. More formally, an input variable mapping say $\Delta_i : (i_0, i_1, \ldots, i_{n-1}) \leftrightarrow (x_0, x_1, \ldots, x_{n-1})$ between two $n$-input $m$-output design functions $\tilde{F}(\bar{I}) = \bar{o}$ and $\tilde{G}(\bar{x}) = \bar{y}$ is a bijection input variable mapping iff it satisfies the equivalence criteria and if for every $k, (0 \leq k \leq n - 1)$, $\text{Sig}(i_k) = \text{Sig}(x_k)$ only. In the absence of symmetric variables, an effective signature should be competent to detect all the unsatisfiable variable mappings except the bijection mapping. In the presence of symmetric variables there
is more than one variable mapping which will make the two designs equivalent. In such case the signature must eliminate all unsatisfiable mappings except the ones that satisfy the equivalence criteria. The performance of a signature is measured as a percentage of the number of unsatisfiable mappings identified by it out of the actual number of unsatisfiable mappings present. A signature should always strive to achieve a percentage coverage of 100%.

Percentage unsatisfiable input mapping coverage =

\[
\frac{\text{Total no. of unsatisfiable input mappings detected by the signature}}{(\text{Total no. of possible input mappings}) - (\text{Actual no. of satisfiable input mappings present})} \times 100
\]

Percentage unsatisfiable output mapping coverage =

\[
\frac{\text{Total no. of unsatisfiable output mappings detected by the signature}}{(\text{Total no. of possible output mappings}) - (\text{Actual no. of satisfiable output mappings present})} \times 100
\]

If a bijection mapping exists between the two designs, actual number of satisfiable mappings equals 1. For instance, \(\Delta_I : (i_0, i_1) \leftrightarrow (x_1, x_0)\) is the only satisfiable bijection input variable mapping which will make the two single-output design functions \(f = i_0 + i_1\) and \(g = \overline{x_1} + x_0\) equivalent.

If symmetric variables are present in the designs, actual number of satisfiable mappings is greater than 1. For instance, \(\Delta_I : (i_0, i_1, i_2) \leftrightarrow (x_0, x_1, x_2)\) and \(\Delta_I : (i_0, i_1, i_2) \leftrightarrow (x_0, x_2, x_1)\) are both satisfiable input mappings which makes the two 3-input single-output design functions \(f = i_0 + (i_1 \cdot i_2)\) and \(g = x_0 + (x_1 \cdot x_2)\) equivalent. Thus, of the
total 3! possible input mapping possibilities between the aforementioned two designs, only two mappings are satisfiable. In the above case an effective signature should be able to identify all of the remaining four unsatisfiable input mappings.

If actual number of satisfiable mappings is equal to 0, there are no variable mappings present which will make the two designs equivalent (i.e. the designs are not equivalent). In such case the signature should be able to identify all of the possible mappings as unsatisfiable and thus provide a coverage of 100%.

If a mapping overage of less than 100% is achieved it implies that not all of the unsatisfiable variable mappings were identified by the signature that was used. The unsatisfiable variable mappings that could not be identified by a signature will be henceforth referred to as residual mappings.

1.4 Applications

Behavioral signatures are computationally simple, time-efficient, and, more importantly, effective in eliminating unsatisfiable variable mappings between two mid- or large-sized designs. The use of Behavioral signature can thus prove to be an effective pre-processing step for many of the existing well-known but computationally expensive Boolean signatures. The efficiency of these Boolean signatures will increase since now they only need to focus on determining unsatisfiable variable mappings from a set of residual variable mappings that were obtained after the Behavioral signatures were applied.
For many designs, the Behavioral signatures themselves exhibit a mapping coverage of 100% thus eliminating the need for the use of any Boolean signatures.

Library binding or technology mapping (Benini and De Micheli, 1997) is an important logic synthesis process that expresses a given behavioral (RTL netlist) specification in terms of standard library cells. The Behavioral signatures that we propose can facilitate quick determination of variable mappings between a module or a sub-module and a standard library cell consequently facilitating an increase in the size and variety of standard cells that can be included in the library for reuse.

Another process that finds the use of Behavioral signatures advantageous is the Design Recovery process (Doom et al., 1999). Design recovery uses a variant of the library binding technique wherein a given circuit or a sub-circuit with a ‘black box’ functionality is checked for functional equivalence with a known library cell functionality. In such a case determining that there does not exist any variable mapping between the two that will make them equivalent significantly reduces the complexity of the matching process. Behavioral signatures are versatile and find their use in eliminating the unsatisfiable variable mappings even if the underlying functionality of the designs is unknown as encountered in design recovery.
1.5 Problem Structure

In Figure 1.1, $F(i) = \vec{o}$ and $G(x) = \vec{y}$ are two $n$-input $m$-output design functions with unknown variable mappings. The Behavioral signature based approach we propose applies a set of specific binary input vectors to both these designs. Behavioral signatures are computed based on the output behavior exhibited by these designs following which the unsatisfiable variable mappings between them are identified.

Figure 1.1.: Structure of Behavioral signature-based approach for identifying unsatisfiable I/O mappings.
Problem Statement

Given two \(n\)-input \(m\)-output combinational designs \(\tilde{F}(\vec{i}) = \tilde{\vec{o}}\) and \(\tilde{G}(\vec{x}) = \tilde{\vec{y}}\), find every possible input \((\Delta_i : \vec{i} \rightarrow \vec{x})\) and output variable mapping \((\Delta_o : \tilde{\vec{o}} \rightarrow \tilde{\vec{y}})\) between them that does not satisfy the equation \(\forall_k, 0 \leq k < m, f_k(\vec{i}) = \Delta_o(g_k)(\Delta_i(\vec{i}))\). The unsatisfiable variable mappings must be identified based on the Behavioral signatures of the variables.

Constraints

a. Functionality or any functional representation of the two designs is unknown.

b. Prior knowledge of the binary output behavior of both the designs is unknown.

Goal

a. Identify maximum number of unsatisfiable variable mappings between the designs.

b. Minimize the computation time.

1.6 Outline

Chapter 2 acquaints the reader with the required background associated to our work. In Chapters 3 and 4 we propose two novel Behavioral signatures to identify unsatisfiable input variable mappings between two designs. Chapter 5 proposes a novel Behavioral signature for identifying unsatisfiable output mappings between two designs. In Chapter 6 we propose a Behavioral signature-based framework for identifying unsatisfiable I/O mappings between two designs and discuss the relevant future work.
Chapter 2

Background

This chapter briefly describes a couple of popular modern day combinational equivalence checking (CEC) techniques, discusses a few of the well-known Boolean signatures used in design verification and acquaints the reader with background research associated to our work.

2.1 CEC Techniques

The techniques described in the following sub-sections are the most basic modern-day approaches that are used to verify if a design implementation adheres to its specification. For each of these techniques there are many modifications that have been proposed over the years by the design verification community in order to make them fast and effective. However these modifications have not been discussed in the following sections since a basic knowledge of these CEC techniques is sufficient to understand our work. It
is worth noting that for both these techniques not knowing the variable mappings between the two designs \textit{a priori} increases their complexity.

2.1.1 Canonical form-based equivalence check

A canonical form is a unique representation of a design function. There can be only one canonical form for a function for a given input variable ordering. The most widely-used canonical form representation is the Reduced Ordered Binary Decision Diagrams (ROBDDs) (Bryant, 1986). The structure of a function’s ROBDD is static for a given input variable ordering but changes if the ordering is changed because the input variable ordering that is chosen influences the placement of input nodes in the ROBDD.

\textit{Example 2.1.}: Consider a function $f = a \cdot \overline{b}$. The ROBDD representation of this function is illustrated in Figure 2.1.a. This ROBDD is constructed based on an input variable ordering of $(a,b)$. Likewise, the ROBDD illustrated in Figure 2.1.b has been constructed based on an input ordering of $(b,a)$. The two structures are clearly different (not isomorphic (Bryant, 1986)).
Two designs are considered functionally equivalent if the ROBDD representations of their functions are isomorphic. Knowing the input mapping between the designs a priori provides a set of input variable orderings (one for each design) based on which their respective ROBDDs can be constructed and subsequently compared for isomorphism.

Example 2.2.: Consider a pair of 3-input 1-output specification \( Z_s \) and its optimized implementation \( Z_l \) represented by Boolean equations \( Z_s = (\overline{r} \cdot q) + (\overline{r} \cdot p) \) and \( Z_l = ((x_1 \cdot x_2) + x_0) \) respectively. Let us assume that we are aware of the fact that the input variables \( r, q \) and \( p \) of function \( Z_s \) map onto inputs \( x_0, x_1 \) and \( x_2 \) respectively of function \( Z_l, a priori \). To confirm if these two functions are equivalent it is sufficient to construct and compare their respective ROBDDs based on input orderings \( r, q, p \) and \( x_0, x_1, x_2 \) as shown in Figure 2.2. Since the two ROBDDs are isomorphic the two functions are equivalent.
However, in the absence of prior knowledge regarding the input variable mappings between the two functions, it becomes mandatory to construct the ROBDDs for one of the functions over every possible input variable and then compare each of them to the ROBDD of the other function which has been constructed over any one arbitrarily-chosen ordering of its input variables. Thus for two n-input designs, $n!$ ROBDDs must be constructed for one of the functions involved.

2.1.2 SAT based equivalence checking

Boolean Satisfiability (SAT) refers to a problem of determining whether there exists an input variable assignment which makes the Boolean function evaluate to a true value or, proving that no such assignment exists. Boolean SAT is an NP complete problem.
However, many modern day algorithms efficiently cover large search spaces by exploiting the function structures\(^1\). The combinational equivalence checking problem can be transformed into a Boolean SAT problem by adding the *miter* circuitry as shown in Figure 2.3.

In Figure 2.3, Designs A and B are two \(n\)-input \(m\)-output combinational designs which must be checked for equivalence. The problem of checking for their equivalence can now be transformed into a Boolean SAT problem of determining if an input variable assignment \((a_0, a_1, \ldots, a_{n-1})\) exists for which at least one of the miter circuit outputs \((H_0, H_1, \ldots, H_{m-1})\) will become true. If such an assignment exists the two designs are not equivalent since it signifies that for a given input assignment the two designs have exhibited varied output behavior. As an e.g. if miter output \(H_1\) becomes true it implies that outputs \(Y_1\) and \(Z_1\) have exhibited varied behavior for a given input assignment. If the SAT solver proves that there does not exist any input assignment for which at least one of the miter outputs become true the two designs are deemed equivalent.

---

\(^1\) The modern day SAT solvers convert the Boolean functions to their Conjunctive Normal Forms (CNF)
Figure 2.3: Transforming the combinational equivalence checking problem to a Boolean SAT problem.

In order for the aforementioned Boolean SAT based CEC approach to work efficiently the input and output variable mappings between the two designs must be known \textit{a priori}. In Figure 2.3 the input variable mapping \((X_0, X_1, \ldots, X_{n-1}) \leftrightarrow (l_0, l_1, \ldots, l_{n-1})\) and the output variable mapping \((Y_0, Y_1, \ldots, Y_{m-1}) \leftrightarrow (Z_0, Z_1, \ldots, Z_{m-1})\) between the two designs is assumed to be known \textit{a priori}. However, the complexity of the above setup increases manifold if these variable mappings are unknown. In Section 2.2, we discuss a
few popular Boolean signatures which have been proposed in the past to effectively eliminate unsatisfiable variable mappings. These signatures primarily rely on Boolean resources for computation.

2.2 Boolean Signatures

These Boolean signatures have been categorized based on the authors that proposed them.

2.2.1 Boolean signature and SAT-based approach of Katebi and Markov (2010)

Katebi and Markov (2010) proposed a two-step approach for identifying variable mappings viz. signature-based and SAT-based. The signature-based approach considers the support and degree of a variable for computation of signatures and then eliminates the unsatisfiable mappings based on their comparison.

Definition 2.1.: Input $x$ is a support variable of output $z$ and output $z$ is a support variable of input $x$ if there exists an input Vector $V$ such that flipping the value of $x$ in $V$ flips the value of $z$.

Definition 2.2.: The support of an input (or output) $x$, denoted by $Supp(x)$, is the set of all the support variables of $x$. The cardinality of the support of $x$ denoted by $|Supp(x)|$, is the
number of I/Os in $Supp(x)$. The degree of $x$ denoted by $D(x)$, is defined as the cardinality of its support.

Based on the aforementioned two definitions the signature of a variable $x$ whose support variables are $(z_0, z_1, ..., z_{m-1})$ is computed as:

$$\text{Sig}(x) = (|Supp(z_0)|, |Supp(z_1)|, ..., |Supp(z_{m-1})|) = (D(z_0), D(z_1), ..., D(z_{m-1}))$$

**Example 2.3.** Consider a circuit with input set $X = \{x_0, x_1, x_2\}$ and output set $Z = \{z_0, z_1, z_2\}$ where, $z_0 = \overline{x_0}$, $z_1 = x_0 \cdot x_1$ and $z_2 = \overline{x_1} \cdot \overline{x_2}$. The supports and degree of input variables are as follows. $Supp(x_0) = \{z_0, z_1\} \Rightarrow D(x_0) = 2$, $Supp(x_1) = \{z_1, z_2\} \Rightarrow D(x_1) = 2$, $Supp(x_2) = \{z_2\} \Rightarrow D(x_2) = 1$. Likewise the supports and degree of the output variables are as follows. $Supp(z_0) = \{x_0\} \Rightarrow D(z_0) = 1$, $Supp(z_1) = \{x_0, x_1\} \Rightarrow D(z_1) = 2$, $Supp(z_2) = \{x_1, x_2\} \Rightarrow D(z_2) = 2$. The signatures of the input variables are thus computed as $\text{Sig}(x_0) = (D(z_0), D(z_1)) = (1, 2)$, $\text{Sig}(x_1) = (D(z_1), D(z_2)) = (2, 2)$ and $\text{Sig}(x_2) = (D(z_2)) = (2)$. The output signatures are computed in a similar manner.

The SAT-based approach reduces the design functions to cofactor sub-functions represented only in terms of input variables that have been uniquely (one-to-one) mapped. This is achieved by allocating all input variables that haven’t been uniquely mapped to either a constant 0 or a constant 1. In other words the input set of the original functions is now reduced to include only those inputs that have been uniquely mapped based on their signatures. These cofactor sub-functions are known as Smallest Matching
Sub-circuits (SMS). Once the SMS are derived, their input set is extended to include the non-uniquely mapped input variables one at a time.

Example 2.4.: Consider two 4-input 1-output circuits \((N_1 \text{ and } N_2)\) with input sets \(X = \{x_0, x_1, x_2, x_3\}\) and \(Y = \{y_0, y_1, y_2, y_3\}\) respectively and outputs \(z_0\) and \(m_0\) respectively. Based on their support and degree signatures let us assume that the input variables have been grouped into Group 0: \((x_0,y_1)\), Group 1: \((x_1,y_0)\), Group 2: \((x_2,x_3,y_2,y_3)\) respectively. Inputs \(x_0\) and \(x_1\) form a one-to-one mapping with inputs \(y_1\) and \(y_0\) respectively. Let \(N_1'\) and \(N_2'\) represent the SMS of \(N_1\) and \(N_2\). \(N_1'\) and \(N_2'\) are derived from \(N_1\) and \(N_2\) by considering only the one-to-one mapped input variables and assigning the non-one-to-one mapped inputs to a constant 0 or 1. Thus \(N_1'\) is a sub-function of \(N_1\) with an input set of \(X' = \{x_0, x_1\}\) and \(N_2'\) is a sub-function of \(N_2\) with an input set of \(Y' = \{y_1, y_0\}\). Please note that input sets \(X'\) and \(Y'\) have been ordered based on the one-to-one mapping between the input variables of the two designs. Once the SMS are obtained they are extended to include a pair (one each from both the designs) of non-one-to-one mapped inputs that belong to same input variable group at a time. For instance, let us consider an arbitrarily-picked pair (one from each circuit) of input variables which belong to the same input variable group of non-one-to-one mapped inputs, say \(x_3\) and \(y_2\). Consequently let \(N_1''\) and \(N_2''\) represent the cofactor sub-functions of \(N_1\) and \(N_2\) that have been derived based on the inclusion of \(x_3\) and \(y_2\) to the SMS input sets \(X'\) and \(Y'\). Thus \(N_1''\) is a sub-function of \(N_1\) with an input set of \(X'' = X' \cup \{x_3\} = \{x_0, x_1, x_3\}\) and \(N_2''\) is a sub-function of \(N_2\) with an input set of \(Y'' = Y' \cup \{y_2\} = \{y_1, y_0, y_2\}\). \(N_1''\) and \(N_2''\) are then passed off on to a miter and presented to the SAT solver for equivalence checking. If the \(N_1''\) and
$N_2''$ are proved to be functionally equivalent it implies that $x_3$ and $y_2$ map onto each other.

2.2.2 Boolean signatures reviewed by Mohnke et al. (2001)

In this section we briefly review some of the popular Boolean signatures that have been proposed. A detailed review of these signatures was carried out by Mohnke et al. (2001).

Cofactor Satisfy Count Signatures (Cheng and Sadowska, 1993; Mohnke and Malik, 1993; Lai et al., 1992)

Consider a $n$-input $m$-output design function $\tilde{F} = \{f_0, f_1, ..., f_{m-1}\}$ with an input set of $I = \{i_0, i_1, ..., i_{n-1}\}$. The positive (negative) cofactor functions of $\tilde{F}$ with respect to $i_0$ are obtained by assigning a value of 1 (0) to $i_0$ in each of its single-output functions and is denoted by:

$$\tilde{F}|_{i_0=1} = \{f_0|_{i_0=1}, f_1|_{i_0=1}, ..., f_{m-1}|_{i_0=1}\} (\tilde{F}|_{i_0=0} = \{f_0|_{i_0=0}, f_1|_{i_0=0}, ..., f_{m-1}|_{i_0=0}\})$$

The satisfy count signature uses the number of input assignments that satisfy these cofactor functions for signature computation.
Breakup signatures (Mohnke, 1999; Mohnke and Malik, 1993):

The Breakup signatures are an extension of the aforementioned cofactor satisfy count signatures. Apart from considering the number of input assignments that satisfy the co-factor functions, Breakup signatures consider the hamming distance of these satisfiable input assignments from the origin of Boolean space.

Cross Signatures (Wang et al., 1996; Schlichtmann et al., 1993):

Cross signature of an input variable $i_0$ considers the satisfy counts of the following functions.

$$\tilde{F}|_{i_0=1,i_k=0} = \{ f_0|_{i_0=1,i_1=0}, f_0|_{i_0=1,i_2=0}, \ldots, f_0|_{i_0=1,i_{n-1}=0} \} \ldots$$

$$\{ f_m-1|_{i_0=1,i_1=0}, f_m-1|_{i_0=1,i_2=0}, \ldots, f_m-1|_{i_0=1,i_{n-1}=0} \}$$

2.3.: Behavioral Signature based approach of Doom and Leighber (2001)

Behavioral signatures are used to establish variable mappings between two combinational designs if no prior knowledge of these mappings is available. Once the mappings are identified the two combinational designs can be checked for equivalence by any of the modern state-of-art CEC techniques. Behavioral signatures are computed based only on the output behavior exhibited by the two designs when a set of specifically formulated input vectors are applied to them. This property makes the Behavioral signatures applica-
able to domains that may not possess the knowledge of a design’s functions or its canonical forms or any other Boolean resources. Behavioral signatures do not even assume that a complete knowledge of the design’s binary output behavior is known *a priori.*

Doom and Leighber (2001) proposed an algorithm to eliminate unsatisfiable variable mapping possibilities based on a three-step approach: formulate a set of specific input vectors that must be applied to the designs, compute the Behavioral signatures based on the output exhibited by the designs in response to these vectors and finally, eliminate the unsatisfiable variable mapping possibilities based on the Behavioral signatures of the variables. The approach we use is similar to the one proposed by Doom and Leighber (2001) with respect to input vector formation and elimination of unsatisfiable variable mappings. Our approach however differs in the techniques used to compute the Behavioral signatures. For a better understanding of our novel Behavioral signatures it is essential to understand some of the terminologies, the algorithm structure and the behavioral signatures that were defined by Doom and Leighber (2001).

### 2.3.1 Terminologies

The names and/or the description of some of these terminologies have been slightly modified with their author’s permission for ease-of-understanding. The original names are mentioned in parenthesis.
Identity Input Vectors (unit input vectors)

Initially when no unsatisfiable variable mappings are known, the designs are subjected to a set of identity input vectors. The identity input vector set for an $n$-input design comprises of $n$ binary input vectors. Every identity input vector is formulated to test a particular input and is formed by initializing that input to active high while assigning all the other inputs to a binary ‘0’. The output vector generated by application of an identity input vector that is formulated to test a particular input is used to compute the behavioral signatures for that input.

Example 2.5.: For a 3-input design $(i_0, i_1, i_2) = \{(1,0,0), (0,1,0), (0,0,1)\}$ represents a complete set of identity input vectors. In this set, vector $(i_0, i_1, i_2) = (1,0,0)$ has been specifically formulated to test input $i_0$ and so on. The output behavior exhibited by the design when the identity input vector formulated to test $i_0$ ($i_0, i_1, i_2 = (1,0,0)$) is applied to it is used to compute the behavioral signatures of $i_0$.

Variable Groups (equivalence/correspondence groups)

Input variables that possess equal signatures are always grouped together. Consequently, inputs that belong to different groups cannot be mapped to each other since their signatures are not equal. Thus an input variable mapping between the two designs that comprises of inputs from different groups being placed at corresponding locations in the mapping is unsatisfiable and can be eliminated from consideration. Conversely, all those
input variable mappings that feature inputs belonging to the same group being placed at corresponding locations must be considered.

*Example 2.6.* Consider two 4-input designs with \( \vec{i} = [i_0, i_1, i_2, i_3] \) and \( \vec{x} = [x_0, x_1, x_2, x_3] \) as their input vectors. Of these inputs let us assume that \( i_1, i_2, i_3, x_0, x_2 \) and \( x_3 \) possess equal signatures. Likewise, let us assume that inputs \( i_0 \) and \( x_1 \) have equal signatures too. Based on this information the inputs can be grouped into Group 0: \((i_1, i_2, i_3, x_0, x_2, x_3)\) and Group 1: \((i_0, x_1)\) respectively. Input variables that belong to the same group have equal signatures. Similarly inputs that belong to different groups possess unequal signatures. An input variable mapping \( \Delta_I: (i_0, i_1, i_2, i_3) \leftrightarrow (x_0, x_2, x_3, x_1) \) is unsatisfiable since inputs \( i_0 \) and \( x_0 \) that have been placed at corresponding (first) locations in the mapping belong to different groups. Conversely, input variable mappings \( \Delta_I: (i_0, i_1, i_2, i_3) \leftrightarrow (x_1, x_0, x_2, x_3), \Delta_I: (i_0, i_1, i_2, i_3) \leftrightarrow (x_1, x_0, x_3, x_2), \Delta_I: (i_0, i_1, i_2, i_3) \leftrightarrow (x_1, x_2, x_0, x_3), \Delta_I: (i_0, i_1, i_2, i_3) \leftrightarrow (x_1, x_2, x_3, x_0) \), \( \Delta_I: (i_0, i_1, i_2, i_3) \leftrightarrow (x_1, x_3, x_0, x_2) \), \( \Delta_I: (i_0, i_1, i_2, i_3) \leftrightarrow (x_1, x_3, x_2, x_0) \) cannot be pruned out and must be considered since all these mappings have inputs with equal signatures being placed at corresponding positions.

The number of input variable mappings that must be considered during equivalence check is thus directly dependent upon the number of input variables that are grouped together. The concept of variable groups can also be similarly extended to output variables.
Iterative Input Vectors

If a bijection input variable mapping is achieved every input variable group will contain exactly one distinct input variable from one design and its one-to-one mapped input variable from the other design. In such an ideal scenario, no other input variable mapping needs to be considered by the CEC technique. The above observations can be extended towards output variables too.

The behavioral signatures computed based on the output behavior obtained when identity input vectors are applied is often not enough to identify all of the unsatisfiable variable mappings. Doom and Leighber (2001) proposed an iterative application of additional input vectors to avoid this shortcoming. The Behavioral signatures are recomputed based on the behavior of the designs in response to these additional vectors. Any variables that previously possessed equal signatures but now possess different recomputed signatures are regrouped into smaller size groups consequently reducing the number of mappings that must be considered for equivalence check.

These additional vectors are formulated based on the input variable groups that were derived in previous iteration. The input vectors are formulated in a manner such that inputs from both the designs which belong to the same input variable group are always assigned the same randomly chosen binary value. The reader is encouraged to read through the manuscript of Doom and Leighber (2001) for further details regarding the formation of additional input vectors.
2.3.2 Algorithm Structure

Our work adopts the same algorithm structure proposed by Doom and Leighber (2001) but differs in the Behavioral signatures used to achieve the goal.

<table>
<thead>
<tr>
<th>Step 1: Formulate and apply the identity input vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2: Compute Behavioral Signatures of the I/Os</td>
</tr>
<tr>
<td>Step 3: Form Input and Output Variable Groups based on the Behavioral signatures possessed by the I/Os</td>
</tr>
<tr>
<td>Step 4: Formulate and apply the iterative input vectors</td>
</tr>
<tr>
<td>Step 5: Iterate through Steps 2 to 5 until either a bijection input variable mapping between the two designs is achieved or if a predetermined loop count is hit.</td>
</tr>
<tr>
<td>Step 6: Display the final variable groups.</td>
</tr>
</tbody>
</table>

*Algorithm 2.1*: Behavioral signature-based approach proposed by Doom and Leighber (2001) for identification of unsatisfiable variable mappings.

2.3.3 Behavioral Signatures of Doom and Leighber (2001)

The names and/or the description of these Behavioral Signatures have been slightly modified with their author’s permission for ease-of-understanding. The original names have been mentioned in parenthesis.
**Row Sum Signatures (Horizontal Signatures)**

**Definition 2.3.:** The *Row Sum Signature* of an input variable is the sum of function outputs when an input vector (e.g., identity input vector) formulated to test that input is applied to the function. For ease of understanding we will hereby define the computation of these signatures based on the application of identity input vectors. However all these definitions also hold true when additional iterative input vectors are applied.

Consider an $n$-input $m$-output design represented by function $\vec{F}(\vec{i}) = \vec{\sigma}$ such that, $\vec{i} = \{i_0, i_1, ..., i_{n-1}\}$, $\vec{\sigma} = \{o_0, o_1, ..., o_{m-1}\}$ and $\vec{F} = \{f_0, f_1, ..., f_{|\vec{\sigma}|}\}$ represent its input, output and function vectors respectively. The row sum signature of an input variable $i_j$ ($0 \leq j \leq n - 1$) is computed as follows. In this computation $\vec{u}_{i_j}$ represents the identity input vector that has been formulated to test input $i_j$.

$$RSum(i_j) = \sum_{k=0}^{m-1} f_k(\vec{u}_{i_j})$$

**Example 2.7.:** Tables 2.1 and 2.2 illustrate the output behavior of a 9-input 6-output design $\vec{F}(\vec{i}) = \vec{\sigma}$ with input, output and function vectors $\vec{i} = \{i_0, i_1, ..., i_8\}$, $\vec{\sigma} = \{o_0, o_1, ..., o_5\}$ and $\vec{F} = \{f_0, f_1, ..., f_5\}$ and an equal-sized design $\vec{G}(\vec{x}) = \vec{y}$ with input, output and function vectors $\vec{x} = \{x_0, x_1, ..., x_8\}$, $\vec{y} = \{y_0, y_1, ..., y_5\}$ and $\vec{G} = \{g_0, g_1, ..., g_5\}$ respectively when identity input vectors that test each of their inputs are applied. For example, row 2 of Table 2.1 illustrates the output behavior exhibited by $\vec{F}$.
when an identity input vector to test input $i_0$ ($\overrightarrow{u_i}$) is applied to it. The $RSum$ signature of $i_0$ is thus computed as follows. The inputs are then ordered based on the decreasing value of $RSum$ signature as shown in Tables 2.3 and 2.4.

$$RSum(i_0) = \sum_{k=0}^{5} f_k(\overrightarrow{u_i}) = f_0(\overrightarrow{u_i}) + f_1(\overrightarrow{u_i}) + \cdots + f_5(\overrightarrow{u_i})$$

$$= 1 + 1 + 0 + 0 + 1 + 0 = 3$$

<table>
<thead>
<tr>
<th>$\vec{F}$</th>
<th>$RSum$, WRSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$</td>
<td>1 1 0 0 1 0</td>
</tr>
<tr>
<td>$i_1$</td>
<td>1 0 1 1 0 0</td>
</tr>
<tr>
<td>$i_2$</td>
<td>1 0 0 0 1 0</td>
</tr>
<tr>
<td>$i_3$</td>
<td>1 0 0 1 0 0</td>
</tr>
<tr>
<td>$i_4$</td>
<td>0 0 1 0 1 0</td>
</tr>
<tr>
<td>$i_5$</td>
<td>0 1 0 0 1 0</td>
</tr>
<tr>
<td>$i_6$</td>
<td>1 1 1 1 0 0</td>
</tr>
<tr>
<td>$i_7$</td>
<td>1 1 1 1 0 0</td>
</tr>
<tr>
<td>$i_8$</td>
<td>0 1 1 0 0 1</td>
</tr>
<tr>
<td>$CSum$</td>
<td>6 5 5 4 4 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\vec{G}$</th>
<th>$RSum$, WRSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_0$</td>
<td>1 0 0 1 0 0</td>
</tr>
<tr>
<td>$y_1$</td>
<td>1 0 1 0 0 1</td>
</tr>
<tr>
<td>$y_2$</td>
<td>0 1 0 0 1 1</td>
</tr>
<tr>
<td>$y_3$</td>
<td>0 1 0 0 1 0</td>
</tr>
<tr>
<td>$y_4$</td>
<td>0 0 0 1 1 0</td>
</tr>
<tr>
<td>$y_5$</td>
<td>1 0 0 1 1 0</td>
</tr>
<tr>
<td>$x_0$</td>
<td>0 0 0 1 0 1</td>
</tr>
<tr>
<td>$x_1$</td>
<td>1 1 0 0 1 1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>1 1 0 0 1 1</td>
</tr>
<tr>
<td>$x_3$</td>
<td>0 1 0 0 1 0</td>
</tr>
<tr>
<td>$x_4$</td>
<td>0 0 0 1 1 0</td>
</tr>
<tr>
<td>$x_5$</td>
<td>1 0 0 1 0 0</td>
</tr>
<tr>
<td>$x_6$</td>
<td>0 0 0 1 0 1</td>
</tr>
<tr>
<td>$CSum$</td>
<td>5 4 1 4 6 5</td>
</tr>
</tbody>
</table>

Table 2.1 (left) and 2.2 (right): Behavior of 9-input 6-output Designs $\vec{F}$ and $\vec{G}$ respectively when input vectors to test each of their inputs are applied.

<table>
<thead>
<tr>
<th>$\vec{F}$</th>
<th>$RSum$, WRSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_6$</td>
<td>1 1 1 1 0 0</td>
</tr>
<tr>
<td>$i_7$</td>
<td>1 1 1 1 0 0</td>
</tr>
<tr>
<td>$i_0$</td>
<td>1 1 0 0 1 0</td>
</tr>
<tr>
<td>$i_1$</td>
<td>1 0 1 1 0 0</td>
</tr>
<tr>
<td>$i_8$</td>
<td>0 1 1 0 0 1</td>
</tr>
<tr>
<td>$i_2$</td>
<td>1 0 0 0 1 0</td>
</tr>
<tr>
<td>$i_3$</td>
<td>1 0 0 1 0 0</td>
</tr>
<tr>
<td>$i_4$</td>
<td>0 0 1 0 1 0</td>
</tr>
<tr>
<td>$i_5$</td>
<td>0 1 0 0 1 0</td>
</tr>
<tr>
<td>$CSum$</td>
<td>6 5 5 4 4 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\vec{G}$</th>
<th>$RSum$, WRSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_0$</td>
<td>1 1 0 0 1 1</td>
</tr>
<tr>
<td>$y_1$</td>
<td>1 1 0 0 1 1</td>
</tr>
<tr>
<td>$y_2$</td>
<td>0 1 0 0 1 1</td>
</tr>
<tr>
<td>$y_3$</td>
<td>0 1 0 0 1 0</td>
</tr>
<tr>
<td>$y_4$</td>
<td>0 0 0 1 1 0</td>
</tr>
<tr>
<td>$y_5$</td>
<td>1 0 0 1 0 0</td>
</tr>
<tr>
<td>$x_7$</td>
<td>1 1 0 0 1 1</td>
</tr>
<tr>
<td>$x_8$</td>
<td>1 1 0 0 1 1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>0 1 0 0 1 1</td>
</tr>
<tr>
<td>$x_5$</td>
<td>1 0 0 1 1 0</td>
</tr>
<tr>
<td>$x_1$</td>
<td>1 0 1 0 0 1</td>
</tr>
<tr>
<td>$x_3$</td>
<td>0 1 0 0 1 1</td>
</tr>
<tr>
<td>$x_4$</td>
<td>0 0 0 1 1 0</td>
</tr>
<tr>
<td>$x_0$</td>
<td>1 0 0 1 0 0</td>
</tr>
<tr>
<td>$x_6$</td>
<td>0 0 0 1 0 1</td>
</tr>
<tr>
<td>$CSum$</td>
<td>5 4 1 4 6 5</td>
</tr>
</tbody>
</table>

Table 2.3 (on left) and 2.4 (right): Behavior of 9-input 6-output Designs $\vec{F}$ and $\vec{G}$ respectively when input vectors to test each of their inputs are applied wherein the design inputs have been ordered on the basis of their decreasing $RSum$ values.
Based on their \( RSum \) signatures the inputs can be grouped into Group 0: \((i_6, i_7, x_7, x_8)\); Group 1: \((i_0, i_1, i_8, x_2, x_5, x_1)\) and Group 2: \((i_2, i_3, i_4, i_5, x_3, x_4, x_0, x_6)\). All input mappings which contain inputs from the same group being present at corresponding positions must be considered.

**Column Sum Signature (Vertical Signature)**

**Definition 2.4.** A Column Sum Signature records the number of times a particular output variable outputs a binary ‘1’, when input vectors formulated to test every input \( i_k \) \((0 \leq k \leq n-1)\) of an \( n \)-input design are applied to the design. The column sum signature of an \( n \)-input \( m \)-output design’s \( \widehat{F}(\vec{i}) = \vec{o} \) output variable \( o_j \) is computed as follows.

\[
CSum(o_j) = \sum_{k=0}^{n-1} f_j(\vec{u}_{ik})
\]

**Example 2.7. (contd...)** The last rows of Tables 2.3 and 2.4 illustrate the column sum signature of each of the Design \( \widehat{F} \) and Design \( \widehat{G} \)’s outputs. The computation of column sum signature for Design \( \widehat{F} \)’s output variable \( o_0 \) is shown below. In this calculation \( \vec{u}_{i_0}, \vec{u}_{i_1}, \ldots, \vec{u}_{i_8} \) represent the identity input vectors formulated to test input variables \( i_0, i_1, \ldots, i_8 \) respectively.
\[ CSum(o_0) = \sum_{k=0}^{8} f_0(u_{i_k}) = f_0(u_{i_0}) + f_0(u_{i_1}) + \cdots + f_0(u_{i_8}) \]

\[ = 1 + 1 + 1 + 1 + 0 + 0 + 1 + 1 + 0 + 0 = 6 \]

The outputs of both the designs are then ordered according to decreasing values of their CSum signature. Tables 2.5 and 2.6 illustrate the ordered outputs.

Based on their CSum signatures the outputs can be grouped into Group 0: \((o_0, y_4)\); Group 1: \((o_1, o_2, y_0, y_5)\); Group 2: \((o_3, o_4, y_3, y_1)\) and Group 3: \((o_5, y_2)\). It is worth noting that based on their CSum signature outputs \(o_0\) and \(y_4\) can now be one-to-one mapped. Same can be observed for \(o_5\) and \(y_2\).
Weighted Row Sum Signature (Anti-Aliasing Signatures)

The computation of Weighted Row Sum signature takes into account only those output variables from both the designs that share the same signature with only one other output from the other design, i.e. it considers only those output variables that have been so far uniquely mapped. The uniquely-mapped output variables from both designs that possess same signature are then assigned same integer weights.

Example 2.7. (contd…): Consider the behavior of Design \( \vec{F} \) and Design \( \vec{G} \) (shown in Tables 2.5 and 2.6 respectively) when identity input vectors formulated to test each of their input variables is applied. Based on the comparison of their column sum signatures Design \( \vec{F} \)'s output variables \( o_0 \) and \( o_5 \) have been uniquely mapped to Design \( \vec{G} \)'s output variables \( y_4 \) and \( y_2 \) respectively. The uniquely-mapped outputs are assigned unique but equal integer weights and ordered as per their decreasing weights during computation. Thus, \( \text{Weight}(o_5) = \text{Weight}(y_2) = 0 \) and \( \text{Weight}(o_0) = \text{Weight}(y_4) = 1 \). The weighted row sum signature of (say) Design \( \vec{F} \)'s input \( i_3 \) is computed as follows. \( f_0(\vec{u}_{i_3}) \) and \( f_5(\vec{u}_{i_3}) \) represent the behavior of outputs \( o_0 \) and \( o_5 \) when identity input vector formulated to test input \( i_3 \) is applied to Design \( \vec{F} \) (refer row 8 of Table 2.4).

\[
WRSum \ (i_3) = (2^{\text{weight}(o_0)} \cdot f_0(\vec{u}_{i_3})) + (2^{\text{weight}(o_5)} \cdot f_5(\vec{u}_{i_3}))
\]

\[
= (2^1 \cdot 1) + (2^0 \cdot 0) = 2
\]
More formally, the Weighted Row Sum Signature of an $m$-output design’s input variable $i_j$ is computed as follows:

\begin{verbatim}
for (0 ≤ k < m)
    if $o_k$ has the same column sum signature as only one output variable of the other design:
        $WRSum (i_j) = \left( WRSum (i_j) \right) + (2^{weight(o_k)} \cdot f_k(\vec{u}_{ij}))$
    else $WRSum (i_j) = \left( WRSum (i_j) \right) + 0$
end for loop
\end{verbatim}

**Algorithm 2.2:** Computation of Weighted Row Sum signature proposed by Doom and Leighber (2001).

**Example 2.7. (contd...):** The $WRSum$ signature of the design inputs have been illustrated in the last column of Tables 2.5 and 2.6. Once the $WRSum$ signature of the inputs is computed, the input variables are ordered based on the decreasing values of their $RSum$ and $WRSum$ signatures. Based on the comparison of their $RSum$ and $WRSum$ signatures Design $\tilde{F}$’s input variable $i_8$ and Design $\tilde{G}$’s input variable $x_1$ now share a unique mapping between them. Inputs $i_2, i_3, i_4, i_5, x_3, x_4, x_0, x_6$ that initially had to be grouped together because of their equal $RSum$ signatures can now be split up into two different groups viz. $(i_2, i_3, x_3, x_4)$ and $(i_4, i_5, x_0, x_6)$ because of their different $WRSum$ signature. It is now safe to state that a Design $\tilde{F}$’s input from one of these two groups can only be mapped to a Design $\tilde{G}$ input which belongs to the same group.
Chapter 3

Grouped Row Sum Signature

In this chapter, we introduce the first of our novel Behavioral signatures. The Grouped Row Sum Signature (GRSum) is used to identify unsatisfiable input variable mappings that exist between two designs.

3.1 Introduction

Behavioral Signatures essentially summarize the output behavior exhibited by the designs when input vectors formulated to test the inputs are applied. The effectiveness of a Behavioral signature depends on the degree of fineness with which the signature can summarize the design behavior. The Row (RSum) and Column Sum (CSum) signatures proposed by Doom and Leighber (2001) summarize the output behavior exhibited by the two designs in a coarse manner by summing up the output behavior exhibited. The Weighted Row Sum (WRSum) signature summarizes the output behavior at a finer level than the RSum signature by considering behavior at every individual output which has
been one-to-one mapped with the other design. However, it ignores the behavior exhibited at outputs which have not been one-to-one mapped.

Example 3.1: Consider two 4-input 6-output designs \( \tilde{F}(\mathbf{i}) = \bar{\mathbf{o}} \) and \( \tilde{G}(\mathbf{x}) = \bar{\mathbf{y}} \). The output behavior of these designs when input vectors formulated to test each of their inputs are applied is illustrated in Tables 3.1 and 3.2. The \( RSum \) and \( WRSum \) signature of every input has been illustrated in the last column of the two tables. Likewise, the \( CSum \) signature of every output has been illustrated in the last row of the two tables. Based on their \( RSum \) and \( WRSum \) signatures inputs \( i_0 \) and \( i_3 \) can now be one-to-one mapped on to \( x_0 \) and \( x_3 \) respectively. Similarly, based on their \( CSum \) signatures outputs \( o_0 \) and \( o_1 \) are now one-to-one mapped onto \( y_0 \) and \( y_1 \) respectively.

<table>
<thead>
<tr>
<th>( \tilde{F} )</th>
<th>( RSum, WRSum )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_0 )</td>
<td>1 1 1 0 0 1</td>
</tr>
<tr>
<td>( i_1 )</td>
<td>1 1 0 0 1 0</td>
</tr>
<tr>
<td>( i_2 )</td>
<td>1 1 0 1 0 0</td>
</tr>
<tr>
<td>( i_3 )</td>
<td>1 0 1 1 0 0</td>
</tr>
<tr>
<td>( CSum )</td>
<td>4 3 2 2 1 1</td>
</tr>
<tr>
<td>( \tilde{G} )</td>
<td>( RSum, WRSum )</td>
</tr>
<tr>
<td>( y_0 )</td>
<td>1 1 0 1 0 1</td>
</tr>
<tr>
<td>( y_1 )</td>
<td>1 1 1 0 0 0</td>
</tr>
<tr>
<td>( y_2 )</td>
<td>1 1 0 1 0 0</td>
</tr>
<tr>
<td>( y_3 )</td>
<td>1 0 1 1 0 0</td>
</tr>
<tr>
<td>( y_4 )</td>
<td>( CSum )</td>
</tr>
</tbody>
</table>

Tables 3.1.(left) and 3.2.(right): Behavior of 4-input 6-output Designs \( \tilde{F} \) and \( \tilde{G} \) respectively when input vectors used to test each of their inputs are applied. The inputs have been ordered based on decreasing values of their \( RSum \) and \( WRSum \) values whereas the outputs have been ordered based on decreasing values of their \( CSum \) signature.

Inputs \( i_1, i_2, x_1, x_2 \) possess equal \( RSum \) and \( WRSum \) signatures and hence must be grouped together. Even though it is evident that the behavior exhibited by Design \( \tilde{F} \) when input vectors corresponding to \( i_1 \) and \( i_2 \) are applied is different (see rows 3 and 4 of Table 3.1), their \( RSum \) and \( WRSum \) signatures are unable to identify this difference. The \( RSum \) signature of \( i_1 \) and \( i_2 \) does not recognize this difference because of its coarse ap-
proach of summarizing the design behavior. On the other hand, their \textit{WRSum} signature fails to identify this difference because of its incompetency in considering the behavior exhibited at non one-to-one mapped outputs. Since the behavior exhibited when input vectors corresponding to \(i_1\) and \(i_2\) are applied differs only at outputs that have not been one-to-one mapped \((o_2, o_3, o_4, o_5)\) there is a need for a signature that will summarize the behavior exhibited by the non-one-to-one mapped outputs. The Grouped Row Sum signature that we propose in the following section fulfills this requirement by considering the behavior of designs exhibited at both one-to-one mapped and non-one-to-one mapped outputs.

### 3.2 Grouped Row Sum Signatures

The \textit{Grouped Row Sum} signature (\textit{GRSum}) is an extension of the Row Sum signature which takes into account the behavior at both one-to-one and non-one-to-one mapped outputs. The behavior exhibited at outputs that have not been uniquely mapped and which belong to the same group must however be collectively considered. This is achieved by considering the sum of binary behavior exhibited at outputs that belong to the same output variable group. The procedure is repeated for every output variable group.

The \textit{GRSum} signature of an input variable \(i_j\) is an integer array \((\alpha_0, \alpha_1, \ldots, \alpha_{\text{Number of Output Variable Groups} - 1})\) of size equal to the number of output variable groups. Output variable groups that were formed based on the Column Sum signature of
the outputs are considered. The first element of the array \((\alpha_0)\) is computed by summing up the behavior exhibited only at outputs which belong to the first output variable group. More formally, the \(GRSum\) signature array elements of an \(n\)-input \(m\)-output design \(\bar{F}(\overline{I}) = \overline{\delta}\)'s input variable \(i_j\) are computed as follows. \(f_x(\overline{u_{ij}})\) represents the output behavior at \(o_x\) when input vector formulated to test input \(i_j\) is applied.

1. \textbf{for} \(\exists\) Output Variable Group \(k\) (\(0 \leq k < \text{Number of Output Variable Groups}\))
2. \textbf{for} \(\exists\) Output \(o_x \in \bar{F}(\overline{I}) = \bar{\delta}\) (\(0 \leq x < \text{Number of Outputs}\))
3. \textbf{if} \((o_x \in \text{Output Variable Group k})\)
4. \(\alpha_k = \alpha_k + f_x(\overline{u_{ij}})\)
5. \textbf{else} \(\alpha_k = \alpha_k + 0\)
6. \textbf{end for}
7. \textbf{end for}

\textit{Algorithm 3.1: Computation of Grouped Row Sum signature}

\textit{Example 3.1.} (continued): Based on their \(CSum\) signatures illustrated in Tables 3.1 and 3.2 the output variables of Designs \(\bar{F}\) and \(\bar{G}\) can be grouped into four groups viz. Group 0: \((o_0, y_0)\), Group 1: \((o_1, y_1)\), Group 2: \((o_2, o_3, y_2, y_3)\) and Group 3: \((o_4, o_5, y_4, y_5)\). The Grouped Row Sum signature of every input variable will thus comprise of four elements \((\alpha_0, \alpha_1, \alpha_2, \alpha_3)\). The first element \((\alpha_0)\) of every input variable’s Grouped Row Sum signature is computed by summing up the behavior exhibited by outputs which belong to output variable group 0 and so on. For instance, the third element \((\alpha_2)\) of input variable
$i_0$’s Grouped Row Sum signature is computed as follows. For computing $\alpha_2$ the behavior of $\hat{F}$ at outputs ($o_2, o_3$) which belong to output variable group 2 must be considered. The behavior of $\hat{F}$ when input vector formulated to test $i_0$ is applied (see row 2 of Table 3.1) is considered. Table 3.3 illustrates the computed Grouped Row Sum signatures of the two designs.

$$\alpha_0(i_0) = 0 + 0 + f_2(\overline{u_{i_0}}) + f_3(\overline{u_{i_0}}) + 0 + 0 = 0 + 0 + 1 + 0 + 0 + 0 = 1$$

<table>
<thead>
<tr>
<th>Inputs ($\tilde{F}$)</th>
<th>Grouped Row Sum Sig. ($\alpha_0, \alpha_1, \alpha_2, \alpha_3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$</td>
<td>(1,1,1,1)</td>
</tr>
<tr>
<td>$i_1$</td>
<td>(1,1,0,1)</td>
</tr>
<tr>
<td>$i_2$</td>
<td>(1,1,1,0)</td>
</tr>
<tr>
<td>$i_3$</td>
<td>(1,0,2,0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs ($\tilde{G}$)</th>
<th>Grouped Row Sum Sig. ($\alpha_0, \alpha_1, \alpha_2, \alpha_3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>(1,1,1,1)</td>
</tr>
<tr>
<td>$x_1$</td>
<td>(1,1,1,0)</td>
</tr>
<tr>
<td>$x_2$</td>
<td>(1,1,0,1)</td>
</tr>
<tr>
<td>$x_3$</td>
<td>(1,0,2,0)</td>
</tr>
</tbody>
</table>

Table 3.3.: Grouped Row Sum signatures of $\tilde{F}$ and $\tilde{G}$’s input variables. The signatures are computed based on the behavior of $\tilde{F}$ and $\tilde{G}$ illustrated in Tables 3.1 and 3.2 respectively.

Based on the comparison of their Grouped Row Sum (Table 3.3), Row Sum (Table 3.1 and 3.2) and Weighted Row Sum (Table 3.1 and 3.2) signatures the input variables of Designs $\tilde{F}$ and $\tilde{G}$ can now be grouped into Group 0: $(i_0, x_0)$, Group 1: $(i_1, x_2)$, Group 2: $(i_2, x_1)$ and Group 3: $(i_3, x_3)$ giving us a bijection mapping of $(i_0, i_1, i_2, i_3) \leftrightarrow (x_0, x_2, x_1, x_3)$.
3.3 Experimental Setup

Our work adopts the algorithm structure discussed in Section 2.3.2. The algorithm was implemented in C and executed on a Linux system with an Intel® Xeon® 2.93 GHz processor. The output behavior of the two designs in response to the application of identity and iterative input vectors was obtained by simulating the And-Inverter Graph (AIG) representations of the designs. The AIGs were constructed and simulated using a set of utility applications provided in the open-source AIGER package (Biere et al., 2007). The role of AIGs in our work is strictly limited to simulation only. Any other design representations like ROBDDs, BMDs, CNFs or even look-up tables can easily substitute for the use of AIGs in our algorithm. In fact, during the early stages of our work we relied on the use of ROBDDs (constructed and simulated using CUDD package (Somenzi, 1997)) for simulation but later changed to AIGs because of their compact nature. The choice of design representation does not affect the computation of Behavioral signatures and the final mappings obtained. The input to our algorithm is the AIG representations of the two designs between whom the variable mappings must be identified. For experimentation purposes the same design is used as a specification and implementation. Our algorithm terminates if a pre-determined loop count of 500 is hit or whenever a bijection mapping is identified. As described in Section 2.3.2, during each iteration, the input vectors are applied, the behavioral signatures are recomputed and the variable groups are restructured. This experimental setup is common to all the experiments that will be carried out throughout this dissertation.
3.4 Results

Table 3.4 illustrates the LGSynth’93 and the ITC’99 TORINO benchmark circuits for which the use of Grouped Row Sum signature (GRSum) along with Row Sum (RSum) and Column Sum (CSum) signatures was more effective in eliminating unsatisfiable input variable mappings over the use of Row Sum, Column Sum, and Weighted Row Sum (WRSum) signatures. The performance is measured based on the number of input variables that have been one-to-one mapped (see columns 3 and 6 of Table 3.4) and their percentage (see columns 4 and 7 of Table 3.4) and the group sizes of inputs that could not be one-to-one mapped (see columns 5 and 8 of Table 3.4). The comparison of time taken by the GRSum and the WRSum signature to identify the variable mappings of these circuits has been illustrated in Table 3.5.

The ITC’99 TORINO benchmark circuits (circuits b14 and b15 in Tables 3.4 and 3.5 and circuits b01 through b15 in Table 3.7) contain latches which are synthesized as combinational logic. For such circuits, in addition to the primary I/Os it is also essential to identify the latch I/O variable mappings between the two designs as it aids the underlying CEC mechanism to effectively and efficiently determine design equivalence. Thus, for a circuit with latches the number of inputs and outputs that must be mapped increases by a factor of the number of latches present. As an example, for the b14 circuit illustrated in Table 3.4 which comprises of 32 inputs, 54 outputs and 245 latches, a total of 277 (32+245) input variables and 299 (54+245) output variables must be considered for map-
ping. The percentage of one-to-one mapped inputs for these circuits is determined based on the revised number of input variables.

For the des, b14 and b15 circuits an improvement in the number of one-to-one mapped inputs is observed. Even though there is no increase in the number of one-to-one mapped inputs in the apex7 and example2 circuits, a reduction in the size of non-one-to-one mapped input variable groups is observed. For instance, in circuit example2, a group of 15 inputs that have the same RSum and WRSum signatures is disintegrated into two smaller groups of size 8 and 7 respectively when RSum and GRSum signatures are used.

The use of Grouped Row Sum signature yields a bijection input variable mapping for the 256-input 245-output des circuit at an impressive speed up of 50 (see Table 3.5). The improvement in mapping coverage for the b14 circuit also comes with a speedup of 2 (see Table 3.5).

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Weighted Row Sum Sig.</th>
<th>Grouped Row Sum Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># of 1-to-1 mapped inputs</td>
<td>% of 1-to-1 mapped inputs</td>
</tr>
<tr>
<td>apex7</td>
<td>49,37,0</td>
<td>39</td>
<td>79.6%</td>
</tr>
<tr>
<td>des</td>
<td>256,245,0</td>
<td>248</td>
<td>96.9%</td>
</tr>
<tr>
<td>example2</td>
<td>85,66,0</td>
<td>33</td>
<td>38.8%</td>
</tr>
<tr>
<td>b14</td>
<td>32,54,245</td>
<td>164</td>
<td>59.2%</td>
</tr>
<tr>
<td>b15</td>
<td>36,70,449</td>
<td>201</td>
<td>41.44%</td>
</tr>
</tbody>
</table>

Table 3.4.: Improvement in number of input variables mapped when Group Row Sum signatures are used for the mentioned LGSynth’93 and ITC’99 circuits.
Table 3.5.: Computation time required by Weighted Row Sum and Grouped Row Sum Signatures for circuits mentioned in Table 3.4.

For circuits shown in Table 3.6 the use of $R_{Sum}$, $W_{RSum}$ and $G_{RSum}$ signatures yields a computational speedup over the use of $R_{Sum}$, $C_{Sum}$ and $W_{RSum}$ signatures. However, no change in the final input variable mappings obtained and mapping coverage is observed.

Table 3.6.: Speedup achieved by the use of Grouped Row Sum signature in comparison to the Weighted Row Sum signature

Table 3.7 compares the performance of Grouped Row Sum signatures to the Boolean signatures proposed by Katebi and Markov (2010) for the shown ITC’99 benchmark circuits. Except for $b12$ and $b15$ circuits a computational speed up is observed for all the
other circuits when Grouped Row Sum signatures along with the pre-existing Row and Column Sum signatures are used to identify the input variable mappings. More importantly, the use of $RSum$, $CSum$ and $GRSum$ signatures provide equal or better one-to-one mapping coverage for the $b02$, $b04$, $b05$, $b08$, $b09$, $b11$ and $b12$ circuits. For the $b06$, $b07$, $b13$ and $b14$ circuits, the one-to-one mapping coverage achieved by the use of $RSum$, $CSum$ and $GRSum$ signatures is marginally less than the coverage of the signature-based approach proposed by Katebi and Markov (2010). The approach proposed by Katebi and Markov (2010) proves to be more computational efficient as well as more effective in identifying input variable mappings for circuits with more than 300 inputs as observed for the $b15$ circuit.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Grouped Row Sum Sig.</th>
<th>Signature based approach of Katebi &amp; Markov (2010)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>% of 1-to-1 mapped inputs</td>
<td>Time in Sec.</td>
<td>% of 1-to-1 mapped inputs</td>
</tr>
<tr>
<td>$b01$</td>
<td>2,2,5</td>
<td>42.9%</td>
<td>0.00004</td>
<td>83%</td>
</tr>
<tr>
<td>$b02$</td>
<td>1,1,4</td>
<td>100%</td>
<td>0.00002</td>
<td>100%</td>
</tr>
<tr>
<td>$b03$</td>
<td>4,4,30</td>
<td>100%</td>
<td>0.0097</td>
<td>54%</td>
</tr>
<tr>
<td>$b04$</td>
<td>11,8,66</td>
<td>100%</td>
<td>0.082</td>
<td>100%</td>
</tr>
<tr>
<td>$b05$</td>
<td>1,36,34</td>
<td>100%</td>
<td>0.004</td>
<td>54%</td>
</tr>
<tr>
<td>$b06$</td>
<td>2,6,9</td>
<td>45.5%</td>
<td>0.03721</td>
<td>50%</td>
</tr>
<tr>
<td>$b07$</td>
<td>1,8,49</td>
<td>92%</td>
<td>0.061</td>
<td>94%</td>
</tr>
<tr>
<td>$b08$</td>
<td>9,4,21</td>
<td>100%</td>
<td>0.04615</td>
<td>100%</td>
</tr>
<tr>
<td>$b09$</td>
<td>1,1,28</td>
<td>100%</td>
<td>0.00618</td>
<td>46%</td>
</tr>
<tr>
<td>$b10$</td>
<td>11,6,17</td>
<td>64.3%</td>
<td>0.00288</td>
<td>100%</td>
</tr>
<tr>
<td>$b11$</td>
<td>7,6,31</td>
<td>100%</td>
<td>0.01661</td>
<td>100%</td>
</tr>
<tr>
<td>$b12$</td>
<td>5,6,121</td>
<td>46%</td>
<td>4.35222</td>
<td>41%</td>
</tr>
<tr>
<td>$b13$</td>
<td>10,10,53</td>
<td>95.2%</td>
<td>0.0822</td>
<td>97%</td>
</tr>
<tr>
<td>$b14$</td>
<td>32,54,245</td>
<td>82.7%</td>
<td>0.982</td>
<td>89%</td>
</tr>
<tr>
<td>$b15$</td>
<td>36,70,449</td>
<td>65.97%</td>
<td>178.5</td>
<td>94%</td>
</tr>
</tbody>
</table>

*Table 3.7.*: Performance comparison of Grouped Row Sum signatures to the signature based approach proposed by Katebi and Markov (2010) for ITC’99 TORINO benchmark circuits.
3.5 Discussion

For the ITC’99 TORINO benchmark circuits with less than 300 inputs the Grouped Row Sum signature we propose performs at par with the Boolean signature based approach proposed by Katebi and Markov (2010). The SAT-based approach proposed by them (see Section 2.2.1) is effective in identifying one-to-one mappings between variables which could not identified by their signature based approach. However, the complexity of this SAT approach directly depends on the number of input variables that have been one-to-one mapped based on the signature based approach. A signature-based approach that can quickly provide an effective coverage of one-to-one mapped inputs to the SAT based approach can considerably bring down the overall execution time of the mapping process. For circuits with less than 300 inputs the use of Grouped Row Sum Behavioral signature can possibly better compliment the SAT based approach proposed by Katebi and Markov (2010) than their Boolean signature based approach. However, for circuits with more than 300 inputs our approach takes more than 500 iterations which increases the runtime of our approach far beyond the runtime reported by Katebi and Markov (2010). For these circuits the mapping coverage achieved at the end of the 500th iteration is notably less than their Boolean signature-based approach.
Chapter 4

Row Difference Signature

In this chapter we propose an effective input variable signature - the Row Difference signature that helps identify unsatisfiable input variable mappings between two digital designs.

4.1 Introduction

The Grouped Row Sum (GRSum) signature proposed in Chapter 3 sums up the behavior of the outputs which belong to the same output variable group. However in this process it loses out on critical precision needed to summarize the behavior of the outputs which belong to the same output variable group.

Example 4.1.: The behavior of two 6-input 6-output designs $\tilde{F}(\tilde{y}) = \tilde{\alpha}$ and $\tilde{G}(\tilde{x}) = \tilde{\beta}$ when input vectors formulated to test each of their inputs are applied is shown in Tables 4.1 and 4.2 respectively.
Based on their RSum, WRSum and GRSum signatures the input variables can now be grouped into three groups, viz. Group 0: \((i_0, i_1, x_0, x_1)\), Group 1: \((i_2, i_3, x_2, x_3)\) and Group 2: \((i_4, i_5, x_4, x_5)\). Likewise, based on their CSum signature the output variables can be grouped into Group 0: \((o_0, o_1, y_0, y_1)\), Group 1: \((o_2, o_3, y_2, y_3)\), Group 2: \((o_4, y_4)\) and Group 3: \((o_5, y_5)\). It can be observed that the behavior exhibited by outputs \(o_0\) and \(o_1\) when input vectors used to test inputs \(i_4\) and \(i_5\) are applied is different. However, since outputs \(o_0\) and \(o_1\) belong to the same output variable group the GRSum signature sums up the behavior exhibited at these two outputs. Unfortunately, the sum of behavior exhibited by \(o_0\) and \(o_1\) when input vectors used to test \(i_0\) and \(i_1\) are applied is same making it difficult for the GRSum to distinguish \(i_0\) and \(i_1\) apart.

The Row Difference signature that we propose in Section 4.2 eliminates this short coming by considering the behavior of every individual output rather than considering the sum of behavior exhibited by outputs which belong to the same group. The Row Difference signature primarily captures the difference in output behavior exhibited by the de-
sign at every individual output when input vectors formulated to test each of its inputs are applied.

4.2 Row Difference Signature (RDiff)

An input variable’s Row Difference Signature comprises of two numeric arrays viz. the zero-equality (RDiff\textsubscript{0}) and the one-equality (RDiff\textsubscript{1}) signature array respectively. The RDiff\textsubscript{0} signature array of an n-input m-output design’s (\overrightarrow{\mathcal{F}}(\overrightarrow{i}) = \overrightarrow{\delta}) input variable \(i_j\) is a n-sized array. The value of \(i_j\)’s \(k\)\textsuperscript{th} RDiff\textsubscript{0} signature array element RDiff\textsubscript{0}[k] is obtained by comparing the output behavior exhibited by the design when input test vectors formulated to test input variables \(i_j\) and \(i_k\) (denoted by \(\overrightarrow{u}_{i_j}\) and \(\overrightarrow{u}_{i_k}\) respectively) are applied to the design. This \(k\)\textsuperscript{th} value of input \(i_j\)’s RDiff\textsubscript{0} array represents the number of output variables that consistently output a “0” value when input test vectors corresponding to \(i_j\) and \(i_k\) are applied to the design. Likewise, the RDiff\textsubscript{1} signature array of an n-input m-output design’s (\overrightarrow{\mathcal{F}}(\overrightarrow{i}) = \overrightarrow{\delta}) input variable \(i_j\) is also a n-sized array. The value of \(i_j\)’s \(k\)\textsuperscript{th} RDiff\textsubscript{1} signature array element is denoted by RDiff\textsubscript{1}[k] and represents the number of output variables that consistently output a “1” value when input test vectors corresponding to \(i_j\) and \(i_k\) (denoted by \(\overrightarrow{u}_{i_j}\) and \(\overrightarrow{u}_{i_k}\) respectively) are applied to the design.

The computation of RDiff\textsubscript{0} and RDiff\textsubscript{1} signature arrays of an n-input m-output design’s (\overrightarrow{\mathcal{F}}(\overrightarrow{i}) = \overrightarrow{\delta}) input variable \(i_j\) can be formalized as shown below. The computed
Row Difference signature arrays are then sorted to eliminate the influence of the order in which the behaviors are compared.

```
1.  \( k \leftarrow 0 \)
2.  \textbf{for } \exists i_k \ (0 \leq k < n) \in \bar{F}(\bar{t}) = \bar{\sigma} \textbf{ do } \\
3.  \quad RDif f_0[k] = RDif f_1[k] = 0 \quad // \text{ Initialization } \\
4.  \textbf{for } \exists o_p \ (0 \leq p < m) \in \bar{F}(\bar{t}) = \bar{\sigma} \textbf{ do } \\
5.  \quad \textbf{if } f_{op}(\bar{u}_{t_j}) = f_{op}(\bar{u}_{i_k}) = 0 \textbf{ then } \\
6.  \quad \quad RDif f_0[k] \leftarrow RDif f_0[k] + 1 \\
7.  \quad \textbf{else if } f_{op}(\bar{u}_{t_j}) = f_{op}(\bar{u}_{i_k}) = 1 \textbf{ then } \\
8.  \quad \quad RDif f_1[k] \leftarrow RDif f_1[k] + 1 \\
9.  \textbf{else continue } \\
10. \textbf{end for } \\
11.  \quad k \leftarrow k + 1 \\
12. \textbf{end for }
```

**Algorithm 4.1:** Computation of Row Difference signature

*Example 4.1 (continued):* Consider the behavior of the 6-input 6-output design \( \bar{F}(\bar{t}) = \bar{\sigma} \) illustrated in Table 4.1. The \( RDif f_0 \) (\( RDif f_i \)) signature array of its input variable say \( i_2 \) is a numeric array of size 6 (same as the number of design inputs). The first element of \( i_2 \)'s \( RDif f_0 \) (\( RDif f_i \)) signature array is denoted by \( RDif f_0[0] \) (\( RDif f_i[0] \)) and is obtained by comparing the design behavior when input test vectors corresponding to \( i_2 \) and \( i_0 \) are applied to the design (see rows 4 and 2 of Table 4.1). The value of \( RDif f_0[0] \) (\( RDif f_i[0] \)) rep-
resents the number of output variables that consistently output a “0” (“1”) value when input vectors corresponding to \(i_2\) and \(i_0\) are applied to the design. On comparing the output behavior illustrated in rows 4 and 2 of Table 4.1 it is observed that there are two (two) such output variables that consistently output a “0” (“1”) value i.e. \(RDiff_0[0] = 2\) \((RDiff_1[0] = 2)\). Likewise, the second element of \(i_2\)’s signature array is denoted by \(RDiff_0[1]\) \((RDiff_1[1])\) and is obtained by comparing the design behavior when input test vectors corresponding to \(i_2\) and \(i_1\) are applied to the design (see rows 4 and 3 of Table 4.1) and so on. Thus the computed values of \(i_2\)’s \(RDiff_0\) and \(RDiff_1\) signature array elements are \(RDiff_0(0,1,2,3,4,5) = (2,1,3,3,2,2)\) and \(RDiff_1(0,1,2,3,4,5) = (2,1,3,3,1,1)\) respectively. Tables 4.3 and 4.4 illustrate the sorted Row Difference signature arrays of Designs \(\vec{F}\) and \(\vec{G}\)’s input variables, respectively. These arrays have been computed based on the behavior illustrated in Tables 4.1 and 4.2 respectively. Based on their Row Difference signatures the input variables can now be grouped as Group 0: \((i_0, x_0)\), Group 1: \((i_1, x_0)\), Group 2: \((i_2, i_3, x_2, x_3)\), Group 3: \((i_4, x_4)\) and Group 4: \((i_5, x_5)\). This classification could not have been achieved based on the previously proposed signatures.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>((RDiff_0), (RDiff_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_0)</td>
<td>(3,2,2,2,1,1), (3,2,2,1,1,0))</td>
</tr>
<tr>
<td>(i_1)</td>
<td>(3,3,2,1,1,1), (3,2,1,1,1,1))</td>
</tr>
<tr>
<td>(i_2)</td>
<td>(3,3,2,2,2,1), (3,3,2,1,1,1))</td>
</tr>
<tr>
<td>(i_3)</td>
<td>(3,3,2,2,2,1), (3,3,2,1,1,1))</td>
</tr>
<tr>
<td>(i_4)</td>
<td>(4,3,2,2,2,1), (2,1,1,1,1,1))</td>
</tr>
<tr>
<td>(i_5)</td>
<td>(4,3,3,2,2,1), (2,2,1,1,1,0))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs</th>
<th>((RDiff_0), (RDiff_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)</td>
<td>(3,3,2,1,1,1), (3,2,1,1,1,1))</td>
</tr>
<tr>
<td>(x_1)</td>
<td>(3,2,2,2,1,1), (3,2,2,1,1,0))</td>
</tr>
<tr>
<td>(x_2)</td>
<td>(3,3,2,2,2,1), (3,3,2,1,1,1))</td>
</tr>
<tr>
<td>(x_3)</td>
<td>(3,3,2,2,2,1), (3,3,2,1,1,1))</td>
</tr>
<tr>
<td>(x_4)</td>
<td>(4,3,2,2,2,2), (2,1,1,1,1,1))</td>
</tr>
<tr>
<td>(x_5)</td>
<td>(4,3,3,2,2,1), (2,2,1,1,1,0))</td>
</tr>
</tbody>
</table>

*Table 4.3* (left) and *4.4* (right): Row Difference signature arrays \((RDiff_0\) and \(RDiff_1)\) of Design \(\vec{F}\) and \(\vec{G}\)’s input variables computed based on the behavior of \(\vec{F}\) and \(\vec{G}\) shown in Tables 4.1 and 4.2.
The approach of comparing the output behavior exhibited by the design when input vector used to test a particular input is applied to the output vectors exhibited by the design when input vectors used to test all other inputs are applied is proven to give us many varied comparison possibilities rather than comparing against a constant output vector of all 0’s or 1’s.

### 4.3 Results

Table 4.5 illustrates the improvement exhibited by the Row Difference signature over the Weighted Row Sum and Grouped Row Sum signatures for the shown LGSynth’93 and the ITC’99 TORINO benchmark circuits. The performance is measured based on the number of input variables that have been one-to-one mapped (see columns 3, 6 and 9 of Table 4.5) and their percentage (see columns 4, 7 and 10 of Table 4.5), the group sizes of inputs that could not be one-to-one mapped (see columns 5, 8 and 11 of Table 4.5) and the time taken (see Table 4.6).

For all the circuits shown in Table 4.5 (except b15) an improvement in the number of one-to-one mapped inputs is observed when \( RDiff \) signatures are used to eliminate unsatisfiable mappings. Even though the number of one-to-one mapped inputs identified by the Row Difference signature is slightly less than the number of one-to-one mapped inputs identified by the Grouped Row Sum signature in the case of b15 circuit, a considerable reduction in the size of non-one-to-one mapped input variable groups is observed. As observed from Table 4.6 and based on observations made for over 50 circuits the Row
Difference signature is observed to be the most computationally expensive signature of the three. This is because in order to compute the Row Difference signature of one input of an $n$-input $m$-output design, $n$ output vector comparisons must be performed. During each of those comparisons $m$ binary output values must be compared. In other words, in order to compute the Row Difference signature of an input, $n \cdot m$ comparisons must be performed. Furthermore, in order to compute the Row Difference signature of $n$-inputs a total of $n \cdot (n \cdot m)$ comparisons must be made.
<table>
<thead>
<tr>
<th>Circuit</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Weighted Row Sum Signature</th>
<th>Grouped Row Sum Signature</th>
<th>Row Difference Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># of 1-to-1 mapped inputs</td>
<td>% of 1-to-1 mapped inputs</td>
<td>Group sizes of non-1-to-1 mapped inputs</td>
</tr>
<tr>
<td>apex2</td>
<td>39,3,0</td>
<td>20</td>
<td>51.28%</td>
<td>8,5,2³</td>
</tr>
<tr>
<td>too_large</td>
<td>38,3,0</td>
<td>20</td>
<td>52.63%</td>
<td>7,5,2³</td>
</tr>
<tr>
<td>b14</td>
<td>32,54,245</td>
<td>164</td>
<td>59.2%</td>
<td>30,28,21,18,14,2</td>
</tr>
<tr>
<td>b15</td>
<td>36,70,449</td>
<td>201</td>
<td>41.44%</td>
<td>32,31²,30,28,26,25,14,8,4,3³,2²</td>
</tr>
</tbody>
</table>

Table 4.5 illustrates the improvement of Row Difference signature over Weighted Row Sum and Grouped Column Sum signature in mapping input variables.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Weighted Row Sum Signature</th>
<th>Grouped Row Sum Signature</th>
<th>Row Difference Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time in seconds</td>
<td>Time in seconds</td>
<td>Time in seconds</td>
</tr>
<tr>
<td>apex2</td>
<td>39,3,0</td>
<td>2.685</td>
<td>4.39</td>
<td>4.987</td>
</tr>
<tr>
<td>too_large</td>
<td>38,3,0</td>
<td>10.22</td>
<td>13.96</td>
<td>14.20</td>
</tr>
<tr>
<td>b14</td>
<td>32,54,245</td>
<td>2.64</td>
<td>0.982</td>
<td>47.24</td>
</tr>
<tr>
<td>b15</td>
<td>36,70,449</td>
<td>175.4</td>
<td>178.5</td>
<td>2628</td>
</tr>
</tbody>
</table>

Table 4.6 illustrates the computation time required by Weighted Row Sum, Grouped Row Sum Signatures and Row Difference signatures to map inputs of the circuits mentioned in Table 4.5.
Table 4.7 (see next page) compares the performance of Row Difference signatures to the Boolean signatures proposed by Katebi and Markov (2010) for the given ITC’99 TORINO benchmark circuits. Except for $b12$ and $b14$ circuits a computational speed up is recorded for all the other circuits when the Row Difference signature along with the pre-existing Row and Column Sum signatures is used to identify the unsatisfiable input variable mappings. More importantly, the use of $RSum$, $CSum$ and $RDiff$ signatures provide equal or better one-to-one mapping coverage for the $b02$, $b03$, $b04$, $b05$, $b08$, $b09$, $b11$ and $b12$ circuits. For the $b06$, $b07$, $b13$ and $b14$ circuits, the one-to-one mapping coverage achieved by the use of $RSum$, $CSum$ and $RDiff$ signatures is marginally less than the coverage of the signature-based approach proposed by Katebi and Markov (2010). The speed up for $b01$, $b10$ and $b15$ is not recorded since there is a big difference in the percentage coverage achieved by signatures of Katebi and Markov (2010) over the Row Difference signature.
| Circ -uit | #Inputs, #Outputs, #Latches | Row Difference Signature | Katebi & Markov’s (2010) Signature based approach | Spee -dup |
|-----------|-----------------------------|---------------------------|-----------------------------------------------|
|           |                             | % of 1-to-1 mapped inputs | Time in seconds | % of 1-to-1 mapped inputs | Time in seconds | |
| b01       | 2,2,5                       | 42.9%                     | 0.0000075      | 83%                        | 0.37            | -         |
| b02       | 1,1,4                       | 100%                      | 0.000038       | 100%                       | 0.28            | >20k      |
| b03       | 4,4,30                      | 100%                      | 0.025          | 54%                        | 0.39            | 15        |
| b04       | 11,8,66                     | 100%                      | 0.32           | 100%                       | 0.47            | 1.47      |
| b05       | 1,36,34                     | 100%                      | 0.025          | 54%                        | 0.54            | 21        |
| b06       | 2,6,9                       | 45.5%                     | 0.0004         | 50%                        | 0.39            | 975       |
| b07       | 1,8,49                      | 92%                       | 0.275          | 94%                        | 0.46            | 1.67      |
| b08       | 9,4,21                      | 100%                      | 0.141          | 100%                       | 0.53            | 3.75      |
| b09       | 1,1,28                      | 100%                      | 0.0186         | 46%                        | 0.42            | 22        |
| b10       | 11,6,17                     | 64.3%                     | 0.011          | 100%                       | 0.37            | -         |
| b11       | 7,6,31                      | 100%                      | 0.0269         | 100%                       | 0.41            | 15        |
| b12       | 5,6,121                     | 46%                       | 10.93          | 41%                        | 1.43            | 0.13      |
| b13       | 10,10,53                    | 95.2%                     | 0.435          | 97%                        | 0.4             | 1         |
| b14       | 32,54,245                   | 82.7%                     | 47.24          | 89%                        | 11.88           | 0.25      |
| b15       | 36,70,449                   | 63.92%                    | 2628           | 94%                        | 62              | -         |

Table 4.7: Performance comparison of Row Difference signatures to the signature based approach proposed by Katebi and Markov (2010) for the ITC’99 TORINO benchmark circuits.

### 4.4 Discussion

For over 50 circuits, the performance of **RDiff** signature is observed to either parallel or better (for circuits shown in Table 4.5) the performance of **WRSum** signature proposed by Doom and Leighber (2001) and the **GRSum** signature proposed in Chapter 3 in terms of input mapping coverage. However, the **RDiff** signature does not measure up to the performance of either the **WRSum** or the **GRSum** signature in terms of execution time for over 50 circuits tested. This drawback is observed because in order to compute the **RDiff** signature of an input variable, the behavior obtained when input vector used to test that input variable is applied is compared to the behavior obtained when input vectors used to test all the other inputs are applied. In other words, for each input of an \( n \)-input
design $n$ comparisons must be carried out. This drawback limits the applicability of $RDiff$ signature to designs with lesser than 300 inputs. There is a significant execution time overhead associated with the $RDiff$ signature for circuits with more than 300 inputs.
Chapter 5

Grouped Column Sum Signature

In this chapter, we introduce our first Behavioral output variable signature - the Grouped Column Sum Signature (GCSum) - to identify unsatisfiable output variable mappings that exist between two designs.

5.1 Introduction

The Column Sum (CSum) signature proposed by Doom and Leighber (2001) sums up the behavior exhibited by an output when input vectors used to test all of the design inputs are applied to the design. CSum signatures are sometimes unable to identify the differences in behavior exhibited by two or more outputs. Consequently, it may be necessary to have more finely grained Behavioral signatures that will more precisely summarize the behavior exhibited by outputs. The GCSum signature that we propose in this chapter attempts to more precisely capture the behavior of an output.
Example 5.1.: Consider two 6-input 6-output designs $\mathbf{F}(\mathbf{I}) = \mathbf{\hat{o}}$ and $\mathbf{G}(\mathbf{x}) = \mathbf{\hat{y}}$. The output behavior of these designs when input vectors formulated to test each of their inputs are applied is illustrated in Tables 5.1. and 5.2. The $CSum$ signature of every design output has been illustrated in the last row of the two tables. Based on their $CSum$ signature the design outputs can be grouped into Group 0: $(o_0, o_1, y_0, y_1)$, Group 1: $(o_2, o_3, y_2, y_3)$, Group 2: $(o_4, y_4)$ and Group 3: $(o_5, y_5)$. It is worth noting that even though the behavior exhibited by Design $\mathbf{F}$ at outputs $o_2$ and $o_3$ is different (see columns 4 and 5 of Table 5.1) their $CSum$ signature is unable to recognize this difference because of its coarseness in summarizing the behavior of these outputs. However, it can be observed that the output behavior exhibited by $o_2$ and $o_3$ (see columns 4 and 5 of Table 5.1) when input vectors used to test inputs $i_2$ and $i_3$ (which belong to the same input variable group because of their equal $RSum$, $WRSum$ and $GRSum$ signatures) are applied is different. The Row Sum ($RSum$), Weighted Row Sum ($WRSum$) and Grouped Row Sum ($GRSum$) signature of every input has been illustrated in the last column of the two tables.

<table>
<thead>
<tr>
<th>$i_0$</th>
<th>$o_0$</th>
<th>$o_1$</th>
<th>$o_2$</th>
<th>$o_3$</th>
<th>$o_4$</th>
<th>$o_5$</th>
<th>$\mathbf{\hat{o}}$</th>
<th>$RSum$, $WRSum$, $GRSum$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3,2, (1,1,1,0)</td>
<td>$3,2, (1,1,1,0)$</td>
</tr>
<tr>
<td>$i_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3,0, (2,1,0,0)</td>
<td>$3,0, (2,1,0,0)$</td>
</tr>
<tr>
<td>$i_3$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3,0, (2,1,0,0)</td>
<td>$3,0, (2,1,0,0)$</td>
</tr>
<tr>
<td>$i_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2,0, (1,1,0,0)</td>
<td>$2,0, (1,1,0,0)$</td>
</tr>
<tr>
<td>$i_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2,0, (1,1,0,0)</td>
<td>$2,0, (1,1,0,0)$</td>
</tr>
<tr>
<td>$\mathbf{\hat{o}}$</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>$\leftarrow CSum$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$y_0$</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$y_4$</th>
<th>$y_5$</th>
<th>$\mathbf{\hat{y}}$</th>
<th>$RSum$, $WRSum$, $GRSum$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3,2, (1,1,1,0)</td>
</tr>
<tr>
<td>$x_1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3,2, (1,1,1,0)</td>
</tr>
<tr>
<td>$x_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3,0, (2,1,0,0)</td>
</tr>
<tr>
<td>$x_3$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3,0, (2,1,0,0)</td>
</tr>
<tr>
<td>$x_4$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2,0, (1,1,0,0)</td>
</tr>
<tr>
<td>$x_5$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2,0, (1,1,0,0)</td>
</tr>
<tr>
<td>$\mathbf{\hat{y}}$</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>$\leftarrow CSum$</td>
</tr>
</tbody>
</table>

*Tables 5.1.(left) and 5.2.(right): Behavior of 6-input 6-output Designs $\mathbf{F}$ and $\mathbf{G}$ respectively when input vectors used to test each of their inputs are applied.*
5.2 Grouped Column Sum Signatures

The *Grouped Column Sum* signature of an output (*GCSum*) is an extension of the Column Sum (*CSum*) signature that takes into account the behavior exhibited by the output when a set of binary input vectors formulated to test each of the design inputs are applied. The behavior exhibited by an output when input vectors formulated to test inputs which belong to the same input variable group are applied is collectively considered. This is achieved by considering the sum of binary behavior exhibited by the output when input vectors formulated to test inputs that belong to the same input variable group are applied. The procedure is repeated for every input variable group.

The *GCSum* signature of an output variable *o_j* is an integer array \((\alpha_0, \alpha_1, ..., \alpha_{\text{Number of Input Variable Groups}-1})\) of size equal to the number of input variable groups. Input variable groups that were formed based on the *RSum*, *WRSum* and *GRSum* signatures of the inputs are considered. The first element \(\alpha_0\) of the array is computed by summing up the behavior exhibited by output *o_j* when input vectors formulated to test all those input variables that belong to the first input variable group are applied to the design. More formally, the *GCSum* signature elements of an \(n\)-input \(m\)-output design \(\tilde{F}(\tilde{I}) = \tilde{\sigma}\)'s output variable \(o_j\) is computed as follows. \(f_j(\textbf{u}_{i_x})\) denotes the output behavior exhibited by output \(o_j\) when input vector formulated to test design input \(i_x\) is applied.
Algorithm 5.1: Algorithm for computing Grouped Column Sum Signatures

Example 5.1.(continued): Based on their RSum, WRSum and GRSum signatures shown in Tables 5.1 and 5.2, the design inputs can be grouped into three groups viz. Group 0: \((i_0, i_1, x_0, x_1)\), Group 1: \((i_2, i_3, x_2, x_3)\) and Group 2: \((i_4, i_5, x_4, x_5)\). The GCSum signature of every input variable will thus comprise of three elements \((\alpha_0, \alpha_1, \alpha_2)\). The first element \((\alpha_0)\) of every output variable’s GCSum signature is computed by summing up the behavior exhibited by that output when input vectors formulated to test those inputs which belong to input variable group 0 are applied. For instance, the third element \((\alpha_2)\) of output variable \(o_0\)’s GCSum signature is computed as follows. For computing \(\alpha_2\) the behavior exhibited by \(o_0\) when input vectors corresponding to those inputs \((i_4\) and \(i_5)\) which belong to input variable group 2 are applied, is considered. The behavior of \(o_0\) when input vectors formulated to test inputs \(i_4\) and \(i_5\) are applied is illustrated in rows 6 and 7 of column 2 in Table 5.1. The GCSum signatures of all the design outputs have been illustrated in Table 4.3.
\[ \alpha_0(o_0) = f_0(u_{i4}) + f_0(u_{i5}) = 1 + 0 = 1 \]

<table>
<thead>
<tr>
<th>Outputs of $\vec{F}$</th>
<th>GCSum $(\alpha_0, \alpha_1, \alpha_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_0$</td>
<td>(1,2,1)</td>
</tr>
<tr>
<td>$o_1$</td>
<td>(1,2,1)</td>
</tr>
<tr>
<td>$o_2$</td>
<td>(1,2,0)</td>
</tr>
<tr>
<td>$o_3$</td>
<td>(1,0,2)</td>
</tr>
<tr>
<td>$o_4$</td>
<td>(2,0,0)</td>
</tr>
<tr>
<td>$o_5$</td>
<td>(0,0,0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs of $\vec{G}$</th>
<th>GCSum $(\alpha_0, \alpha_1, \alpha_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_0$</td>
<td>(1,2,1)</td>
</tr>
<tr>
<td>$y_1$</td>
<td>(1,2,1)</td>
</tr>
<tr>
<td>$y_2$</td>
<td>(1,2,0)</td>
</tr>
<tr>
<td>$y_3$</td>
<td>(1,0,2)</td>
</tr>
<tr>
<td>$y_4$</td>
<td>(2,0,0)</td>
</tr>
<tr>
<td>$y_5$</td>
<td>(0,0,0)</td>
</tr>
</tbody>
</table>

*Table 5.3*: Grouped Column Sum (GCSum) signatures of $\vec{F}$ and $\vec{G}$'s output variables. These signatures are computed based on the behavior of $\vec{F}$ and $\vec{G}$ illustrated in Tables 5.1 and 5.2 respectively.

Based on the comparison of their GCSum (see Table 5.3) and CSum (see Tables 5.1 and 5.2) signatures, the output variables of Designs $\vec{F}$ and $\vec{G}$ can now be grouped into Group 0: $(o_0, o_1, y_0, y_1)$, Group 1: $(o_2, y_2)$, Group 2: $(o_3, y_3)$, Group 3: $(o_4, y_4)$ and Group 4: $(o_5, y_5)$.

### 5.3 Results

The results indicating the effectiveness of the GCSum signature in identifying unsatisfiable output variable mappings can be classified into three sections. In the first section, we compare the performance of the CSum and GCSum signatures in identifying unsatisfiable output variable mappings when they are coupled with the RSum and WRSum input variable signatures. In the second section we compare the performance of the CSum and GCSum signatures in identifying unsatisfiable output variable mappings when they are coupled with the RSum and GRSum input variable signatures. It is essential to consider the use of output variable signatures alongside the use of input variable signatures be-
cause during each iteration, the input variable groups that are obtained influence the manner in which binary input vectors that must be applied in the next iteration are formed. The input variable groups obtained during each iteration when $RSum$ and $WRSum$ signatures are used can be different than the input variable groups obtained based on $RSum$ and $GRSum$ signatures and may hence cause different input vectors to be explored in the Boolean space. The third section compares the performance of $GCSum$ with the signatures proposed by Katebi and Markov (2010).

5.3.1 Comparison of Column Sum and Grouped Column Sum signatures (when coupled with Row Sum and Weighted Row Sum signatures)

<table>
<thead>
<tr>
<th>Name</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Column Sum Sig.</th>
<th>Grouped Column Sum Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% of 1-to-1 mapped outputs</td>
<td>% of 1-to-1 mapped outputs</td>
<td>Group sizes of non-1-to-1 mapped outputs</td>
</tr>
<tr>
<td>$c1908$</td>
<td>33,25,0</td>
<td>25</td>
<td>100%</td>
</tr>
<tr>
<td>$des$</td>
<td>256,245,0</td>
<td>211</td>
<td>86.12%</td>
</tr>
<tr>
<td>$apex7$</td>
<td>49,37,0</td>
<td>33</td>
<td>89.19%</td>
</tr>
<tr>
<td>$b05$</td>
<td>1,36,34</td>
<td>47</td>
<td>67.14%</td>
</tr>
<tr>
<td>$b08$</td>
<td>9,4,21</td>
<td>25</td>
<td>100%</td>
</tr>
<tr>
<td>$b11$</td>
<td>7,6,31</td>
<td>37</td>
<td>100%</td>
</tr>
<tr>
<td>$b14$</td>
<td>32,54,245</td>
<td>239</td>
<td>79.93%</td>
</tr>
</tbody>
</table>

Table 5.4: Performance improvement achieved by $GCSum$ signature in detecting unsatisfiable output mappings over the $CSum$ signature when they are coupled alongside the $RSum$ and $WRSum$ input variable signatures for the shown LGSynth93 and ITC’99 Benchmarks.
Table 5.5: The computational speedup achieved by \textit{GCSum} signature in detecting unsatisfiable output mappings over the \textit{CSum} signature when they are coupled alongside the \textit{RSum} and \textit{WRSum} input variable signatures for the shown LGSynth93 and ITC’99 Benchmarks.

For the 256-input 245-output \textit{des} benchmark circuit the use of \textit{GCSum} signature yields a 100% output mapping coverage as compared to the 86% mapping coverage yielded by the use of \textit{CSum} signature alone (see Table 5.4) at an impressive speedup of 33x (see Table 5.5). More importantly, a group of 22 outputs that could not be uniquely mapped when \textit{CSum} signatures were used can now be mapped with the aid of \textit{GCSum} signature. A 7% improvement in mapping coverage is also reported for the \textit{b05} benchmark circuit at the expense of negligible increase (0.00002 sec) in computation time. For all the other circuits the use of \textit{GCSum} yields the same mapping coverage as the \textit{CSum} signature but at a speedup gain ranging from 1.5 to 3.

5.3.2 Comparison of Column Sum and Grouped Column Sum signatures (when coupled with Row Sum and Grouped Row Sum signatures)

Here too, for the 256-input 245-output \textit{des} benchmark circuit the use of \textit{GCSum} signature yields a 100% output mapping coverage as compared to the 64.5% mapping coverage achieved by the use of \textit{CSum} signature alone (see Table 5.6). For the \textit{des} circuit, the \textit{GCSum} signature when coupled alongside the \textit{RSum} and \textit{GRSum} input signatures
yields a 100% mapping coverage (see Table 5.6) in 0.23 seconds (see Table 5.7) which is faster compared to the 0.6 seconds (see Table 5.5 of Section 5.3.1) taken by the \textit{GCSum} signature when it is coupled alongside the \textit{RSum} and \textit{WRSum} input signatures. An improvement in mapping coverage is also observed for the \textit{b14} (75\% to 83\%) and \textit{b15} (51.6\% to 64.7\%) circuits when \textit{GCSum} signature is used. Additionally, these improvements are achieved in nearly the same amount of computation time as the \textit{CSum} signature (see Table 5.7). For all other circuits the use of \textit{GCSum} signature yields a speedup gain ranging from 1 to 6 over the use of \textit{CSum} signature.

<table>
<thead>
<tr>
<th>Name</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Column Sum Sig.</th>
<th>Grouped Column Sum Sig.</th>
<th>Grouped Column Sum Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># of 1-to-1 mapped outputs</td>
<td>% of 1-to-1 mapped outputs</td>
<td>Group sizes of non-1-to-1 mapped outputs</td>
</tr>
<tr>
<td>\textit{myadder}</td>
<td>33,17,0</td>
<td>17</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>\textit{x1}</td>
<td>51,35,0</td>
<td>31</td>
<td>88.5%</td>
<td>2$^2$</td>
</tr>
<tr>
<td>\textit{x3}</td>
<td>135,99,0</td>
<td>99</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>\textit{apex6}</td>
<td>135,99,0</td>
<td>99</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>\textit{c1908}</td>
<td>33,25,0</td>
<td>25</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>\textit{des}</td>
<td>256,245,0</td>
<td>158</td>
<td>64.49%</td>
<td>16,10,5$^2$, 4$^3$, 3$^3$, 2$^{15}$</td>
</tr>
<tr>
<td>\textit{i9}</td>
<td>88,63,0</td>
<td>63</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>\textit{b04}</td>
<td>11,8,66</td>
<td>70</td>
<td>94.59%</td>
<td>2$^2$</td>
</tr>
<tr>
<td>\textit{b07}</td>
<td>1,8,49</td>
<td>53</td>
<td>92.98%</td>
<td>4</td>
</tr>
<tr>
<td>\textit{b09}</td>
<td>1,1,28</td>
<td>29</td>
<td>100%</td>
<td>1</td>
</tr>
<tr>
<td>\textit{b12}</td>
<td>5,6,121</td>
<td>55</td>
<td>43.3%</td>
<td>2$^{36}$</td>
</tr>
<tr>
<td>\textit{b14}</td>
<td>32,54,245</td>
<td>225</td>
<td>75.25%</td>
<td>30,14,13, 8,5, 2$^2$</td>
</tr>
<tr>
<td>\textit{b15}</td>
<td>36,70,449</td>
<td>268</td>
<td>51.6%</td>
<td>31$^3$, 30,28, 24$^2$, 12,8, 5, 3, 2$^{12}$</td>
</tr>
</tbody>
</table>

\textit{Table 5.6.}: Performance improvement achieved by \textit{GCSum} signature in detecting unsatisfiable output mappings over the \textit{CSum} signature when they are coupled alongside the \textit{RSum} and \textit{WRSum} input variable signatures for the shown LGSynth93 and ITC’99 Benchmarks.
<table>
<thead>
<tr>
<th>Name</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Column Sum Sig.</th>
<th>Grouped Column Sum Sig.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time in seconds</td>
<td>Time in seconds</td>
<td></td>
</tr>
<tr>
<td>myadder</td>
<td>33,17,0</td>
<td>0.0075</td>
<td>0.0065</td>
<td>1.15</td>
</tr>
<tr>
<td>x1</td>
<td>51,35,0</td>
<td>0.0321</td>
<td>0.0243</td>
<td>1.32</td>
</tr>
<tr>
<td>x3</td>
<td>135,99,0</td>
<td>0.318</td>
<td>0.2556</td>
<td>1.24</td>
</tr>
<tr>
<td>apex6</td>
<td>135,99,0</td>
<td>0.179</td>
<td>0.146</td>
<td>1.22</td>
</tr>
<tr>
<td>c1908</td>
<td>33,25,0</td>
<td>0.00418</td>
<td>0.0016</td>
<td>2.6</td>
</tr>
<tr>
<td>des</td>
<td>256,245,0</td>
<td>0.673</td>
<td>0.23</td>
<td>2.92</td>
</tr>
<tr>
<td>i9</td>
<td>88,63,0</td>
<td>0.0433</td>
<td>0.038</td>
<td>1.14</td>
</tr>
<tr>
<td>b04</td>
<td>11,8,66</td>
<td>0.088</td>
<td>0.043</td>
<td>2.05</td>
</tr>
<tr>
<td>b07</td>
<td>1,8,49</td>
<td>0.0656</td>
<td>0.0478</td>
<td>1.37</td>
</tr>
<tr>
<td>b09</td>
<td>1,1,28</td>
<td>0.0067</td>
<td>0.0044</td>
<td>1.5</td>
</tr>
<tr>
<td>b12</td>
<td>5,6,121</td>
<td>4.306</td>
<td>0.709</td>
<td>6.1</td>
</tr>
<tr>
<td>b14</td>
<td>32,54,245</td>
<td>20.45</td>
<td>18.83</td>
<td>1.1</td>
</tr>
<tr>
<td>b15</td>
<td>36,70,449</td>
<td>175.02</td>
<td>178.37</td>
<td>0.981</td>
</tr>
</tbody>
</table>

Table 5.7.: The computational speedup achieved by GCSum signature in detecting unsatisfiable output mappings over the CSum signature when they are coupled alongside the RSum and WRSum input signatures for the shown LGSynth93 and ITC’99 circuits.

5.3.3 Comparison of Grouped Column Sum signatures and the signatures proposed by Katebi and Markov (2010)

<table>
<thead>
<tr>
<th>Circ - uit</th>
<th>#Inputs, #Outputs, #Latches</th>
<th>Grouped Column Sum Signature</th>
<th>Katebi &amp; Markov’s (2010) Signature based approach</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>% of 1-to-1 mapped outputs</td>
<td>Time in secs.</td>
<td>% of 1-to-1 mapped outputs</td>
</tr>
<tr>
<td>b01</td>
<td>2,2,5</td>
<td>71.4%</td>
<td>0.0193</td>
<td>85%</td>
</tr>
<tr>
<td>b02</td>
<td>1,1,4</td>
<td>100%</td>
<td>0.000032</td>
<td>100%</td>
</tr>
<tr>
<td>b03</td>
<td>4,4,30</td>
<td>100%</td>
<td>0.0052</td>
<td>47%</td>
</tr>
<tr>
<td>b04</td>
<td>11,8,66</td>
<td>100%</td>
<td>0.043</td>
<td>100%</td>
</tr>
<tr>
<td>b05</td>
<td>1,36,34</td>
<td>48.7%</td>
<td>0.004</td>
<td>47%</td>
</tr>
<tr>
<td>b06</td>
<td>2,6,9</td>
<td>46.7%</td>
<td>0.00018</td>
<td>53%</td>
</tr>
<tr>
<td>b07</td>
<td>1,8,49</td>
<td>93%</td>
<td>0.046</td>
<td>95%</td>
</tr>
<tr>
<td>b08</td>
<td>9,4,21</td>
<td>100%</td>
<td>0.0318</td>
<td>100%</td>
</tr>
<tr>
<td>b09</td>
<td>1,1,28</td>
<td>100%</td>
<td>0.0044</td>
<td>48%</td>
</tr>
<tr>
<td>b10</td>
<td>11,6,17</td>
<td>65.2%</td>
<td>0.0027</td>
<td>100%</td>
</tr>
<tr>
<td>b11</td>
<td>7,6,31</td>
<td>100%</td>
<td>0.0084</td>
<td>100%</td>
</tr>
<tr>
<td>b12</td>
<td>5,6,121</td>
<td>43.3%</td>
<td>0.709</td>
<td>41%</td>
</tr>
<tr>
<td>b13</td>
<td>10,10,53</td>
<td>95.2%</td>
<td>0.072</td>
<td>97%</td>
</tr>
<tr>
<td>b14</td>
<td>32,54,245</td>
<td>83.6%</td>
<td>18.83</td>
<td>90%</td>
</tr>
<tr>
<td>b15</td>
<td>36,70,449</td>
<td>64.7%</td>
<td>208.37</td>
<td>94%</td>
</tr>
</tbody>
</table>

Table 5.8.: Performance comparison of GCSum signature to the signatures proposed by Katebi and Markov (2010) in detecting unsatisfiable output mappings

64
The GCSum signature outperforms the signatures proposed by Katebi and Markov (2010) in terms of both mapping coverage and execution time for the b03, b05, b09, b12 circuits. The GCSum signature is superior to the signatures proposed by Katebi and Markov (2010) in terms of execution time for the b02, b04, b08 and b11 circuits wherein the mapping coverage achieved by both approaches is 100%. In all other cases the signatures proposed by Katebi and Markov (2010) outperform the GCSum signatures either in terms of execution time or mapping coverage or both. The difference is significant for the large-sized b15 circuit.

5.4 Discussion

Theoretically, the computation of GCSum signature is as efficient as the computation of the CSum signature as both these processes primarily compute the sum of behavior exhibited by an output variable when input vectors used to test all of the design inputs are applied. This fact is reaffirmed by the fact that the speedup achieved by the GCSum signature over the CSum signature is greater than or equal to one for the circuits tested when WRSum signature is used to map input variables. For circuits in which the use of GCSum signature yields a speedup of more than one over the CSum signature, the GCSum signature is successful in achieving either equal better or improved mapping coverage in lesser number of iterations than its counterpart. For instance, in the case of 245-output des circuit the use of GCSum signature coupled with the WRSum signature yields a 100% mapping coverage in just 10 iterations as compared to the CSum signature that yields a 1-1 mapping coverage of only 86% at the end of the allocated 500 iterations.
When the $GCSum$ signature is used in conjunction with the $GRSum$ signature it produces a better mapping coverage for circuits with larger number of outputs like the $b14$ and $b15$ circuits as compared to the $GCSum$ signature when coupled alongside the $WRSum$ signature. This improvement is observed because for these circuits, the use of $GRSum$ signature records a notable improvement in input variable mapping coverage over the $WRSum$ signature. An improved input variable mapping coverage facilitates our algorithm to apply more varied input vectors to the designs and achieve more varied output behavior which in turn facilitates better output variable mapping coverage. Consequently, for circuits in which the use of $GRSum$ signature is observed to be more computationally expensive than the use of $WRSum$ signature, the use of $GCSum$ signature when coupled with the $GRSum$ signature has also proven to be more computationally expensive as compared to when it is coupled with the $WRSum$ signature.
Chapter 6

Conclusion and Future Work

6.1 Framework

6.1.1 Framework Proposal for determining Input Variable Mappings

Figure 6.1 (see next page) illustrates the computational speed up achieved by the Grouped Row Sum ($GRSum$) and the Row Difference ($RDiff$) signatures over the Weighted Row Sum signature in identifying unsatisfiable input variable mappings between two designs. The use of $RDiff$ has a considerable computational overhead associated to it. As seen in Figure 6.1 the computational speed up associated with the $RDiff$ (see dotted line graph in Figure 6.1) signature is much less than the computational speedup achieved by the $GRSum$ signature. As a matter of fact the speedup achieved by the $RDiff$ signature over the $WRSum$ signature is less than 1 for most circuits indicating that the $RDiff$ signature is the most computationally expensive signature amongst the three. However, the $RDiff$ signature is more effective in identifying input variable mappings between
circuits with large (greater than 200) number of inputs (as indicated by results for the \textit{b14} and \textit{b15} circuits in Chapter 4).

On the other hand, the \textit{GRSum} signature shows a higher speed up (see solid line graph in Figure 6.1) as compared to the \textit{RDiff} signatures for all of the tested circuits. Circuits for which the use of \textit{GRSum} signature yields a speed up of less than one are circuits in which the use of \textit{GRSum} signature is more computationally expensive than the \textit{WRSum}. There are 10 such circuits that report this anomaly out of which for two circuits (including the 277 input \textit{b14} circuit) an improvement in input mapping coverage is reported. Of the remaining eight, four are circuits with more than 100 outputs. Since the \textit{GRSum} signature takes into consideration the behavior exhibited by every output as against the \textit{WRSum} signature which considers the behavior exhibited by only one-to-one mapped outputs, such slowdown is expected. Lastly, for all the tested circuits the use of \textit{GRSum} either yields the same or better input mapping coverage than the \textit{WRSum} signature. This observation affirms the fact that the \textit{GRSum} signature summarizes the behavior exhibited by the non-one-to-one mapped outputs as well the behavior exhibited by the one-to-one mapped outputs. In other words, the functionality of \textit{GRSum} signature encompasses the functionality of the \textit{WRSum} signature. We thus propose to replace the existing Weighted Row signatures with the Grouped Row Sum signature.
Figure 6.1: Comparison of computational speedups achieved by Grouped Row Sum and Row Difference Signatures over the Weighted Row Sum Signature

Figure 6.2: Comparison of percentage coverage of 1-1 mapped inputs achieved by Weighted Row Sum, Grouped Row Sum and Row Difference Signatures
Additionally, as illustrated in Figure 6.2, the percentage coverage of 1-1 mapped inputs achieved by either the GRSum or the RDiff signature matches the percentage coverage achieved by WRSum signature. For the circuits with more than 200 inputs the percentage coverage yielded by the GRSum and the RDiff signatures is higher than the coverage achieved by WRSum signature. Based on these observations we define the following framework for mapping input variables.

*Framework proposal for mapping input variables*

- Use of Row Sum and Grouped Row Sum signatures to determine unsatisfiable input variable mappings for circuits with less than 200 inputs.

- Use of Row Sum, Grouped Row Sum and Row Difference signatures to determine unsatisfiable input variable mappings for circuits with less than 200 inputs if the aforementioned combination does not yield a 100% mapping.

- Use of Row Sum, Grouped Row Sum and Row Difference signatures to determine unsatisfiable input variable mappings for circuits with more than 200 inputs.

**6.1.2 Framework Proposal for determining Output Variable Mappings**

Figure 6.3 (see next page) illustrates the computational speed up achieved by the Grouped Column Sum (GCSum) signature when it is coupled with Grouped Row Sum
(GRSum) and Weighted Row Sum (WRSum) signature over the use of Column Sum (CSum) signature in identifying unsatisfiable output variable mappings between two designs. The speedup achieved by coupling the GCSum signature with the WRSum signature over the use of CSum signature is illustrated by a dotted line in Fig. 6.3. Except for two circuits the speedup achieved by coupling GCSum and WRSum signatures together is either equal to or greater than one. In other words, for majority of circuits the GCSum signature when coupled with the WRSum signature performs as efficiently as the CSum signature.

On the other hand, the speedup achieved by coupling the GCSum and GRSum signatures over the use of CSum signature (see solid line in Figure 6.3) falls below 1 for a considerable number of circuits. This is primarily due to the computational overhead associated to GRSum signatures in mapping input variables; it is based on these mapped inputs that the future vectors are formed and applied. Even though the 299-output b14 circuit exhibits the lowest speedup, an improvement in mapping coverage is observed (see Figure 6.4).

As illustrated in Figure 6.4, the percentage coverage of 1-1 mapped outputs achieved by the GCSum signature when coupled with either WRSum or GRSum signature is always either same or better than the coverage achieved by the CSum signature. Based on the aforementioned, observations we propose the replacement of the existing CSum signature with our novel GCSum signature.
Figure 6.3: Comparison of computational speedups achieved by the Grouped Column Sum signature when coupled with Weighted Row Sum (dotted line) and Grouped Row Sum signature (solid line) over the Column Sum Signature

Figure 6.4: Comparison of percentage coverage of 1-1 mapped outputs achieved by the Grouped Column Sum signature when coupled with Weighted Row Sum and Grouped Row Sum signature over the Column Sum Signature
Framework proposal for mapping output variables

- Use of Grouped Column Sum signature coupled with the Weighted Row Sum signature in order to determine unsatisfiable output variable mappings for circuits with less than 200 outputs.

- Use of Grouped Column Sum signature coupled with the Grouped Row Sum signature to determine unsatisfiable output variable mappings for circuits with less than 200 outputs if the aforementioned combination does not yield a 100% mapping.

- Use of Grouped Column Sum signature coupled with the Grouped Row Sum signature to determine unsatisfiable output variable mappings for circuits with more than 200 outputs.

6.2 Future Work

6.2.1. Design Structure Knowledge

The graphs shown in Figures 6.1, 6.2, 6.3 and 6.4 show the percentage of one-to-one mapped variables and the speedup achieved by our proposed signatures for circuits with number of inputs ranging between 5 and 485 and number of outputs ranging between 3 and 519. Since no information regarding Boolean functions or the structure of
the designs is assumed to be known the number of variables is the only factor based on which our frameworks have been established.

Design’s exhibiting varied behavior in response to the binary input vectors that are applied is a crucial factor for the success of our approach. However, the output behavior exhibited by certain basic digital components like multiplexer does not help in effectively mapping its I/O variables. For instance, in the case of two simple 1-bit \([n:1]\) stand-alone multiplexers only their 0\(^{th}\) and \(n^{th}\) input can be one-to-one mapped onto each other based on the output behavior exhibited. Their select lines however cannot be one-to-one mapped and are always grouped to each other. The rest of the data inputs (1 to \(n-1\)) form another group. In such case the remaining data inputs and the select lines can be one-to-one mapped if all possible brute-force combinations of input vectors are applied. In other words the presence of certain structures can hamper the variedness of the output behavior exhibited which may in turn hamper the ability of Behavioral signatures to better map the design variables. Knowledge of the structure of the design can thus prove to be a valuable asset to our approach. Investigating how knowing the structure of a design would help improve the effectiveness of our approach will be an interesting future direction of our research.

6.2.2. Improved Input Vectors

During each loop of our algorithm exactly \(n\) input vectors are applied to test the \(n\)-input design. Since our predetermined loop count is 500, a maximum of \(500n\) input vec-
tors are applied. For circuit with as less as say 20 inputs a total of \(2^{20} = 1048576\) input vectors are possible of which our algorithm explores at most of \(500 \times 20 = 10000\) vectors. Since our algorithm explores a small fraction of Boolean space it is highly dependent on the behavior exhibited by the design in response to these small set of input vectors.

Furthermore, because the exact nature of mappings between the inputs of two designs which belong to the same input variable group is unknown they cannot be assigned varied binary values during the input vector formation process. In other words, inputs which belong to the same input variable groups are always assigned the exact same binary value. This drawback many-a-times restricts us from observing a potential crucial change in output behavior necessary to precisely map the variables. Eliminating these aforementioned drawbacks is an important part of our future work.

The use of randomly-generated input vectors may play a crucial role in exploring design behavior which could not be explored because of our constrained way of generating input vectors thus consequently helping us in more precisely mapping the I/O variables. For instance, a mechanism which can assign varied binary values to inputs that belong to the same input variable groups can potentially extract design behavior which will help map those inputs to the inputs of the other design. The computation of Boolean signature based on random simulation proposed Katebi and Markov (2010) can provide a valuable roadmap towards such research.
6.2.3. Decoder-like output behavior

Many-a-times decoder-like output behavior is observed at outputs which belong to the same output variable group. This directly hinders the algorithm’s ability to better map both the input and output behavior. This can be well-explained using the following small example. The output behavior of 3-input 3-output design when input vectors used to test each of the inputs are applied is shown below.

<table>
<thead>
<tr>
<th></th>
<th>$o_0$</th>
<th>$o_1$</th>
<th>$o_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$i_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Let us assume that all three inputs belong to the same input variable group and so do the outputs. This kind of a behavior hinders the ability of all the signatures that we have proposed to precisely map the I/O variables. Such cases yield equal-sized groups of non-one-to-one mapped inputs and outputs. In such cases it would be advantageous to provide the user with a mapping between the inputs and the outputs itself. For instance, in this example based on simple observation of the behavior we can derive the following mappings between the design’s inputs and its own outputs: $i_0 \rightarrow o_0$, $i_1 \rightarrow o_1$ and $i_2 \rightarrow o_2$. Thus later on if any of these inputs are 1-1 mapped the corresponding outputs can be 1-1 mapped or vice versa. Table 6.1 illustrates some of the LGSynth’93 circuits for which such decoder-like output behavior is reported.
### Table 6.1: Circuits exhibiting equal-sized input and output variable groups

<table>
<thead>
<tr>
<th>Circuits</th>
<th>Group sizes of non-one-to-one mapped inputs</th>
<th>Group sizes of non-one-to-one mapped outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count</code></td>
<td>17, 16, 2</td>
<td>16</td>
</tr>
<tr>
<td><code>c432</code></td>
<td>3(^3)</td>
<td>3</td>
</tr>
<tr>
<td><code>c499</code></td>
<td>32, 9</td>
<td>32</td>
</tr>
<tr>
<td><code>example2</code></td>
<td>8(^2), 7(^2), 6(^4), 4</td>
<td>8(^2), 7(^2), 6, 4</td>
</tr>
<tr>
<td><code>i6</code></td>
<td>32(^2), 28(^2), 6(^2)</td>
<td>32, 28, 6</td>
</tr>
<tr>
<td><code>i7</code></td>
<td>32(^2), 28(^2), 4(^3), 2</td>
<td>32, 28, 4, 2</td>
</tr>
</tbody>
</table>

6.2.4. Symmetric variables

Our approach relies on the output behavior exhibited by the designs. It is thus almost impossible to apply all the possible input vectors in order to determine the symmetric input and output variables present. However, an estimate regarding whether a given set of inputs (or outputs) are symmetric can be made by observing the output behavior exhibited by the design in response to our limited set of input vectors. If the Boolean networks/representations of the designs are available the symmetry-detection algorithm proposed by Abdollahi and Pedram (2008) can be used to pinpoint the symmetries in the circuit. If a group of non-one-to-one mapped variables from both the designs are identified as symmetric variables every mapping possibility between them is considered satisfiable. Thus if a group of input variables is identified as symmetric our algorithm can arbitrarily choose any one of the satisfiable mapping possibilities as a basis for 1-1 mapping those inputs, consequently empowering it to explore varied input vectors.
6.2.5. Parallel Implementation

Because of their fundamental difference in summarizing the output behavior, the Grouped Row Sum and Row Difference signatures are known to produce different input variable groups during the same iteration for a given design. Consequently, the input vectors which are formed and applied in the next iteration are different. The two approaches thus tend to explore different points in Boolean space. Given that the two computation methodologies are independent of each other it is apt to execute them in parallel and take an intersection of the input groups yielded by both the approaches. As an example, say the use of Grouped Row Sum Signature yields us the following two input variable groups for a given 3-input design: \((i_0, i_1)\) and \((i_2)\). On the other hand, say the Row Difference signature yields the following input variable groups \((i_0)\) and \((i_1, i_2)\). The final input variable groups obtained in this case are \((i_0), (i_1)\) and \((i_2)\) and the total computation time is the time taken by the slowest approach.

6.3 Conclusion

Our signatures can provide the existing effective but more-expensive Boolean signatures with an initial time-efficient and effective pruning of unsatisfiable variable mappings thus aiding them in cutting down their associated time complexity. More importantly, our work provides a focused and effective approach for determining unsatisfiable variable mappings between two digital designs to applications that do not possess any prior
knowledge regarding the structure, Boolean functions or even the complete output behavior of the designs
Appendix A: Source Code

```c
#include "aiger.h"    // AIGER package by Biere et al. 2007
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <sys/resource.h>
#include <fcntl.h>
#include <errno.h>

#define POS 0
#define NEG 1
#define NEWGRP 1
#define NEWMEM 0
#define RANDOMGEN 1
#define BINARY 0

typedef unsigned long int ULongInt;

//Seed for random binary value generation function
//gen_vector()
int LFSRTAPS[][7] = {
{0,0,0,0,0}, {0,0,0,0,0}, {0,0,0,0,0}, {0,0,0,0,0}, {2,3,2}, {2,4,3}, {2,5,3},
{2,6,5}, {2,7,6}, {4,8,6,5,4}, {2,9,5}, {2,10,7}, {2,11,9}, {4,12,6,4,1},
{4,13,4,3,1}, {4,14,5,3,1}, {2,15,14}, {4,16,15,13,4}, {2,17,14}, {2,18,11}, {4,19,6,2,1}, {2,20,17}, {2,21,19}, {2,22,21},
{2,23,18}, {4,24,23,22,17}, {2,25,22}, {4,26,6,2,1}, {4,27,5,2,1}, {2,28,25}, {2,29,27}, {4,30,6,4,1}, {2,31,28}, {4,32,22,2,1},
{2,33,20}, {4,34,27,2,1}, {2,35,33}, {2,36,25}, {6,37,5,4,3,2,1},
{4,38,6,5,1}, {2,39,35}, {4,40,38,21,19}, {2,41,38}, {4,42,41,2,19},
{4,51,50,36,35}, {2,52,49},
};
```
GLOBAL VARIABLES

static FILE *file; //File pointer for recording

//Binary input vectors
static unsigned char *pcutcurrent;
static unsigned char *ncutcurrent;
static unsigned char *plibcurrent;
static unsigned char *nlibcurrent;

//AIG graphs of two designs
static aiger *model;
static aiger *model2;

static int close_file;

int x, m, res, check, found, print;

//Output Vectors
int **CutOutVector[2];
int **LibOutVector[2];

/* FUNCTIONS PROVIDED BY AIGER PACKAGE FOR SIMULATING AN AIG GRAPH */

static void die(const char *fmt, ...)
{
va_list ap;
fputs("*** [aigsim] ", stderr);
va_start (ap, fmt);
vfprintf (stderr, fmt, ap);
va_end (ap);
putc ('\n', stderr);
exit (1);

} // die() ends

static unsigned char deref(unsigned lit, unsigned char *c)
{
    unsigned res = c[aiger_lit2var (lit)];
    res ^= aiger_sign (lit);
    #ifndef NDEBUG
    if (lit == 0)
        assert (res == 0);
    if (lit == 1)
        assert (res == 1);
    #endif
    return res;
} // deref() ends

static void put(unsigned lit, unsigned char *cu)
{
    unsigned v = deref (lit,cu);
    if (v & 2)
        fputc ('x', stdout);
    else
        fputc ('0' + (v & 1), stdout);
} // put() ends

static const char * aiger_symbol_as_string(aiger_symbol * s)
{
    static char buffer[20];
    if (s->name)
        return s->name;
    sprintf (buffer, "%u", s->lit / 2);
    return buffer;
} // aiger_symbol_as_string() ends

/*Function provided by AIGER to simulate an input vector
   (svector) on an AIG model (smodel)*/

static void Sim(aiger *smodel, int *svector,
                 unsigned char *curr)
{
    int sj,si;
    unsigned sl,sr,tmp;
int *nxt;
nxt=(int *)malloc(smodel->num_latches*sizeof(int));
for (sj = 0; sj < smodel->num_and; sj++)
{
    aiger_and *and = smodel->ands + sj;
    sl = deref (and->rhs0,curr);
    sr = deref (and->rhs1,curr);
    stmp = sl & sr;
    stmp |= sl & (sr << 1);
    stmp |= sr & (sl << 1);
    curr[and->lhs / 2] = stmp;
}
found = 0;
for (sj = 0; !found && sj < smodel->num_outputs; sj++)
    found = (deref (smodel->outputs[sj].lit,curr) == 1);
if (check && found)
    res = 0;
print = 1 && (!check || found);
/* Then first calculate next state values of latches in parallel.*/
for (sj = 0; sj < smodel->num_latches; sj++)
{
    aiger_symbol *symbol = smodel->latches + sj;
    nxt[sj] = deref (symbol->next,curr);
}
si=0;
for(sj=smodel->num_outputs;sj<(smodel->num_outputs+
    smodel->num_latches);sj++)
{
    svector[sj]=nxt[si];
    si++;
}
if(si!=smodel->num_latches)
{
    printf("ERROR\n");
    getchar();
}
/* Then update new values of latches.*/
for (sj = 0; sj < smodel->num_latches; sj++)
{
    aiger_symbol *symbol = smodel->latches + sj;
    //curr[symbol->lit / 2] = nxt[sj];
}
/* Print outputs.*/
for (sj = 0; sj < smodel->num_outputs; sj++)
{
84

```c
svector[sj]=deref(smodel->outputs[sj].lit,curr);
}
free(nxt);
} // Sim() ends

// Structure of an Input variable
typedef struct input {
  // Row Sum signature
  int PRSig;
  int NRSig;
  // Weighted Row Sum signature
  ULongInt PWRSig;
  ULongInt NWRSig;
  // Grouped Row Sum signature
  int *PGRSig;
  int *NGRSig;
  // Row Difference signature
  int *PRDSig[2];
  int *NRDSig[2];
  int myipgrp; // Input variable group number to which the
               // input belongs
} input;

// Structure of an Output variable
typedef struct output {
  // Column Sum signature
  int PCSig;
  int NCSig;
  // Grouped Column Sum signature
  int *PGRSig;
  int *NGRSig;
  int myopgrp; // Output var. group number to which the output
               // belongs
  /* If 1, mask the behavior of the output during the compu-
     tation of weighted row sum signature since the output has
     not been 1-1 mapped */
  int mask;
} output;

// Input/Output Variable Group structure
/* Inputs (outputs) that possess equal signatures are placed
in the same group */
typedef struct eq_group {
  int *cut_mem; // variables from the first design
  int *lib_mem; // variables from the second design
  int size;    // Size of group
} eq_group;
```

84
int new;  // is the group being created in the current
        // iteration
eq_group;

// Create a new variable group or add members to an existing
// variable group
void BuildGroup(eq_group **group, int member, int no,
        int create_new)
{
    eq_group *ptr;
    int *temp_ptr;

    ptr = *group;
    if(create_new==0)
        {
            ptr[no].size++;
            temp_ptr = (int *) malloc((ptr[no].size+1) *
                sizeof(int));
            memcpy(temp_ptr, ptr[no].cut_mem, (ptr[no].size) *
                sizeof(int));
            free(ptr[no].cut_mem);
            ptr[no].cut_mem = temp_ptr;
            ptr[no].cut_mem[ptr[no].size] = member;
        }
    else
    {
        ptr = (eq_group*)realloc(ptr, (no+1) *
            sizeof(eq_group));
        ptr[no].cut_mem = (int *)malloc(sizeof(int));
        ptr[no].cut_mem[0] = member;
        ptr[no].size = 0;
    }
    *group = ptr;
    return;
// BuildGroup() ends

// Print the variable groups
void PrintGroups(eq_group *ptr, eq_group *optr, int ipgpnum,
        int opgpnum, int ***vector,
        int ***lvector, FILE *fp)
{
    int i,j,x,k;

    for(i=0; i<(ipgpnum+1); i++)
        {
            fprintf(fp,"-----------------------------------\n");
            fprintf(fp,"Group %d: [Size = %d]\n", i, ptr[i].size+1);
            for(j=0; j<(ptr[i].size + 1); j++)
                {
                    fprintf(fp,"%d \t", ptr[i].cut_mem[j]);
                    fprintf(fp,"%d \t", ptr[i].lib_mem[j]);
                    fprintf(fp,"\n");
fprintf(fp,"*****************************
");
return;
}// PrintGroups() ends

// Used for sorting an array of integer
// Required for sorting the Row Difference signature arrays
void mysort(int* data, int N)
{
    int i, j;
    int v, t;
    if(N<=1) return;
    // Partition elements
    t=0;
    v = data[0];
    i = 0;
    j = N;
    for(;;)
    {
        while(data[++i] < v && i < N) { }
        while(data[--j] > v) { }
        if(i >= j) break;
        t = data[i]; data[i] = data[j]; data[j] = t;
    }
    t = data[i-1]; data[i-1] = data[0]; data[0] = t;
    mysort(data, i-1);
    mysort(data+i, N-i);
} // mysort() ends

/* Function used to generate an array of random binary input values. Courtesy: Doom and Leighb (2001) */
void gen_vector(int *vector_ptr, int size, int mode)
{
    int i,j, *temp_vector;
    int feedback=1;               //Must initialize to '1'
    temp_vector = (int *) malloc((size+1)*sizeof(int));
    temp_vector[0] = 0;
    if(mode == RANDOMGEN)
    {
        for(j=1; j<LFSRTAPS[size][0]+1; j++)
            feedback = !(feedback^vector_ptr[(LFSRTAPS[size][j]-1)]);
        for(i=0; i<size; i++)
            temp_vector[i+1] = vector_ptr[i];
        for(i=1; i<size; i++) //size
            vector_ptr[i] = temp_vector[i];
        vector_ptr[0] = feedback;
    }
else                          // else, do binary count
    { // Left in program only for evaluation.
        for(i=0; i<size;i++)
            {
                if(vector_ptr[i]==0)
                    {
                        vector_ptr[i] = 1;
                        break;
                    }
                else
                    {
                        vector_ptr[i] = 0;
                    }
            } //end for
    } //end else
free(temp_vector);

/* Row Sum signature computation of design inputs based on 
the output behavior (vector) exhibited by the design */
// Courtesy: Doom and Leighber (2001)
static void RowSum(int ***vector, input *inp, int nIn,
                    int nOut)
    {
        int i,j;
        for(i=0;i<nIn;i++)
            {
                inp[i].PRSig=0;
                inp[i].NRSig=0;
                for(j=0;j<nOut;j++)
                    {
                        inp[i].PRSig+=vector[0][i][j];
                        inp[i].NRSig+=vector[1][i][j];
                    }
        } // RowSum() ends
/* Column Sum signature computation of design outputs based 
on the output behavior (vector) exhibited by the design */
// Courtesy: Doom and Leighber (2001)
static void ColSum(int ***vector, output *op, int nIn,
                    int nOut)
    {
        int i,j;
        for(i=0;i<nOut;i++)
            {
                op[i].PCSig=0;
                op[i].NCSig=0;
                for(j=0;j<nIn;j++)
                    {
                        op[i].PCSig+=vector[0][j][i];
                        op[i].NCSig+=vector[1][j][i];
                    }
    }
/* Weighted Row Sum signature computation of design inputs based on the output behavior (vector) exhibited by the design */
// Courtesy: Doom and Leighber (2001)

ULongInt WtdRow(int *order, output *ptr, int *vector, int size)
{
    int i;
    ULongInt sig=0, sig2=0, sig3=0, sig4=0, sig5=0;
    for(i=0;i<size;i++)
    {
        if(i<33)
        {
            if((vector[order[i]]*ptr[order[i]].mask)==1)
                sig = sig + (2^i);
        }
        else if(i<66)
        {
            if((vector[order[i]]*ptr[order[i]].mask)==1)
                sig2 = sig2 + (2^(i-33));
        }
        else if(i<99)
        {
            if((vector[order[i]]*ptr[order[i]].mask)==1)
                sig3 = sig3 + (2^(i-66));
        }
        else if(i<132)
        {
            if((vector[order[i]]*ptr[order[i]].mask)==1)
                sig4 = sig4 + (2^(i-99));
        }
        else if(i<165)
        {
            if((vector[order[i]]*ptr[order[i]].mask)==1)
                sig5 = sig5 + (2^(i-132));
        }
        else
        {
            sig=0;
        }
        sig = sig + sig2 + sig3 + sig4 + sig5;
    } //end for-loop
    return sig;
} // WtdRow() ends
// Row Difference signature array computation of input 'i'
void ComputeRowDiffSig(input **igroup, int ***Vector,
                        int nIn, int nOut, int i)
{
    int cnt=0;
    int j, k;
    input *ippmem;
    ippmem = *igroup;

    for(j=0; j<nIn; j++)
    {
        for(k=0; k<nOut; k++)
        {
            if(((Vector[POS][i][k])+(Vector[POS][j][k]))==0)
                ippmem[i].PRDSig[0][cnt]++;
            else if(((Vector[POS][i][k])+(Vector[POS][j][k]))==2)
                ippmem[i].PRDSig[1][cnt]++;
        }
        for(k=0; k<nOut; k++)
        {
            if(((Vector[NEG][i][k])+(Vector[NEG][j][k]))==0)
                ippmem[i].NRDSig[0][cnt]++;
            else if(((Vector[NEG][i][k])+(Vector[NEG][j][k]))==2)
                ippmem[i].NRDSig[1][cnt]++;
        }
        cnt++;
    }
    *igroup = ippmem;
    return;
} // ComputeRowDiffSig() ends

// Are the Grouped Row Sum signatures of given two inputs equal?
int isGRSumSigEq(input *mem_one, input *mem_two, int one,
                 int two, int size, int isCOMPUTE)
{
    int yes=1;
    int x;
    if(isCOMPUTE==0)
        yes=1;
    else{
        for(x=0;x<size;x++)
        {
            if((mem_one[one].PGRSig[x]!=mem_two[two].PGRSig[x])|
               (mem_one[one].NGRSig[x]!=mem_two[two].NGRSig[x]))
            {
                yes=0;
                break;
            }
        }
    }
}
return yes;
} // isGRSumSigEq() ends

/* Are the Grouped Column Sum signatures of given two outputs equal? */
int isOpGRSumSigEq(output *mem_one, output *mem_two,
                   int one, int two, int size, int isCOMPUTE)
{
    int yes=1;
    int x;
    if(isCOMPUTE==0)
        yes=1;
    else{
        for(x=0;x<size;x++)
        {
            if((mem_one[one].PGRSig[x]!=mem_two[two].PGRSig[x])||
                (mem_one[one].NGRSig[x]!=mem_two[two].NGRSig[x]))
            {
                yes=0;
                break;
            }
        }
    }
    return yes;
} // isOpGRSumSigEq() ends

/* Are the Row Difference signatures of given two inputs equal? */
int isIpRDSigEq(input *Mem, input *MemI, int one, int two,
                int size, int isCOMPUTE)
{
    int k,x;
    int assig=1;
    if(isCOMPUTE==0)
        return assig;
    for(k=0; k<size;k++)
    {
        if((Mem[one].PRDSig[0][k]!=MemI[two].PRDSig[0][k])||
            (Mem[one].NRDSig[0][k]!=MemI[two].NRDSig[0][k])||
            (Mem[one].PRDSig[1][k]!=MemI[two].PRDSig[1][k])||
            (Mem[one].NRDSig[1][k]!=MemI[two].NRDSig[1][k]))
        {
            assig=0;
            break;
        }
    }
    return(assig);
} // isIpRDSigEq() ends
int main (int argc, char **argv)
{
    const char *model_file_name, *model2_file_name, *error;
    int i, j, k, l;
    input *CutInMem=NULL, *LibInMem=NULL; // Array of design inputs
    output *CutOutMem=NULL, *LibOutMem=NULL; // Array of design outputs
    int nInGps=0, nOutGps=0; // Number of I/O variable groups
    int assigned=0;
    eq_group *InGp=NULL, *OutGp=NULL; // Array of I/O var. groups
    int LibMemberPosition=0;
    int *opgrporder=NULL, *oporder=NULL, *liboporder=NULL;
    int tcnt=0, done=0, cnt=0;
    int *BinVec=NULL;  // Randomly-generated binary input vec.
    int TestCycle=0;   // Iteration Number
    int NumOfGroupsTemp=0, NumOfOPGroupsTemp=0;
    eq_group GpTemp;
    int *tmpcutmem=NULL, *tmplibmem=NULL;
    int tmpsize, tmplibsize;
    struct timeval tot1, tot2, tot4, tot5;
    int rc;
    int CYCLELIMIT=500;
    char *FNAME; // Filename of a file used to record the variable groups and elapsed time
    FILE *testfile; // File pointer to aforementioned file
    int ROW_DIFF=0, GR_SUM=0, COL_DIFF=0, GC_SUM=0, WR_SUM=0;
    stimulus_file_name = model_file_name = model2_file_name=0;
    // read the names of AIG (.aig) files of the two designs
    model_file_name=argv[1];
    model2_file_name = argv[2];

    // read the user preference with regards to which signatures should be used
    ROW_DIFF=atoi(argv[3]);
    GR_SUM=atoi(argv[4]);
    GC_SUM= atoi(argv[5]);
    WR_SUM= atoi(argv[6]);
    FNAME=argv[7];
    testfile = fopen(FNAME, "a+");
// Read the first AIG model from the .aig file
model = aiger_init();
error = aiger_open_and_read_from_file(model,
        model_file_name);
aiger_reencode (model);

// Read the second AIG model from the .aig file
model2 = aiger_init();
error = aiger_open_and_read_from_file (model2,
        model2_file_name);
aiger_reencode (model2);

CYCLELIMIT=500; //Set Loop Count to 500

fprintf(testfile,"No. of CUT Inputs = %d
",
        model->num_inputs);
fprintf(testfile,"No. of CUT Outputs = %d
",
        model->num_outputs);
fprintf(testfile,"No. of CUT Latches = %d
",
        model->num_latches);
printf("No. of LIB Inputs = %d
",
        model2->num_inputs);
printf("No. of LIB Outputs = %d
",
        model2->num_outputs);
printf("No. of LIB Latches = %d
",
        model2->num_latches);

rc=gettimeofday(&tot1, NULL); // Snapshot of current CPU
        // time
if((model->num_inputs)!=(model2->num_inputs))
{
    printf("CUT and LIB designs not of the same
        size....ABORTING\n");
    exit(-1);
}

CutOutVector[POS] = (int **)malloc((model->num_inputs+
        model->num_latches) * sizeof(int *));
CutOutVector[NEG] = (int **)malloc((model->num_inputs+
        model->num_latches) * sizeof(int *));
LibOutVector[POS] = (int **)malloc((model2->num_inputs+
        model2->num_latches) * sizeof(int *));
LibOutVector[NEG] = (int **)malloc((model2->num_inputs+
        model2->num_latches) * sizeof(int *));

for(x=0; x<(model->num_inputs+model->num_latches); x++)
{
    CutOutVector[POS][x] = (int *)calloc((model->num_outputs+
        model->num_latches), sizeof(int));
    CutOutVector[NEG][x] = (int *)calloc((model->num_outputs+
        model->num_latches), sizeof(int));
    LibOutVector[POS][x] = (int *)calloc((model2->num_outputs+
        model2->num_latches), sizeof(int));
    LibOutVector[NEG][x] = (int *)calloc((model2->num_outputs+
        model2->num_latches), sizeof(int));
}
LibOutVector[NEG][x] = (int *)calloc((model2->num_outputs+
model2->num_latches), sizeof(int));

CutInMem = (input *)malloc((model->num_inputs+
model->num_latches)*sizeof(input));
LibInMem = (input *)malloc((model2->num_inputs+
model2->num_latches)*sizeof(input));
CutOutMem = (output *)malloc((model->num_outputs+
model->num_latches)*sizeof(output));
LibOutMem = (output *)malloc((model2->num_outputs+
model2->num_latches)*sizeof(output));

for(i=0; i<(model->num_inputs+model->num_latches); i++)
{
    CutInMem[i].PRSig=0;
    CutInMem[i].NRSig=0;
    CutInMem[i].PWRSig=0;
    CutInMem[i].NWRSig=0;

    LibInMem[i].PRSig=0;
    LibInMem[i].NRSig=0;
    LibInMem[i].PWRSig=0;
    LibInMem[i].NWRSig=0;
}

pcutcurrent = calloc (model->maxvar + 1,
    sizeof (pcutcurrent[0]));
ncutcurrent = calloc (model->maxvar + 1,
    sizeof (ncutcurrent[0]));
plibcurrent = calloc (model2->maxvar + 1,
    sizeof (plibcurrent[0]));
nlibcurrent = calloc (model2->maxvar + 1,
    sizeof (nlibcurrent[0]));

/* Formulate and apply identity input vectors to the two Designs */
for(x=1;x<=(model->num_inputs+model->num_latches);x++)
{
    for(m=1;m<=(model->num_inputs+model->num_latches);m++)
    {
        if(m==x)
        {
            pcutcurrent[m]=1;
            ncutcurrent[m]=0;
        }
        else
        {
            pcutcurrent[m]=0;
            ncutcurrent[m]=1;
        }
    }
}
//Simulate the first AIG module with identity input
//vectors
Sim(model,CutOutVector[POS][x-1],pcutcurrent);
Sim(model,CutOutVector[NEG][x-1],ncutcurrent);
}
for(x=1;x<=(model2->num_inputs+model2->num_latches);x++)
{
    for(m=1;m<=(model2->num_inputs+model2->num_latches);m++)
    {
        if(m==x)
        {
            plibcurrent[m]=1;
            nlibcurrent[m]=0;
        }
        else
        {
            plibcurrent[m]=0;
            nlibcurrent[m]=1;
        }
    }
    //Simulate the first AIG module with identity input
    //vectors
    Sim(model2,LibOutVector[POS][x-1],plibcurrent);
    Sim(model2,LibOutVector[NEG][x-1],nlibcurrent);
}

//Compute Row Sum Signature
RowSum(CutOutVector, CutInMem,(model->num_inputs+
    model->num_latches),(model->num_outputs+
    model->num_latches));
RowSum(LibOutVector, LibInMem,(model2->num_inputs+
    model2->num_latches),(model2->num_outputs+
    model2->num_latches));

//Compute Column Sum signature
ColSum(CutOutVector,CutOutMem, (model->num_inputs+
    model->num_latches),(model->num_outputs+
    model->num_latches));
ColSum(LibOutVector,LibOutMem, (model2->num_inputs+
    model2->num_latches),(model2->num_outputs+
    model2->num_latches));

nOutGps=0;
OutGp = (eq_group*)malloc(sizeof(eq_group));
BuildGroup(&OutGp, 0, nOutGps, NEWGRP);

//Form output variable groups
for(i=1;i<=(model->num_outputs+model->num_latches);i++)
{
    assigned=0;
for(j=0; j<nOutGps+1; j++)
{
    if((CutOutMem[i].PCSig ==
        CutOutMem[OutGp[j].cut_mem[0]].PCSig) &&
        (CutOutMem[i].NCSig ==
        CutOutMem[OutGp[j].cut_mem[0]].NCSig))
    {
        BuildGroup(&OutGp, i, j, NEWMEM);
        assigned=1;
        break;
    }
}
if(assigned==0)
{
    nOutGps++;  
    BuildGroup(&OutGp, i, nOutGps, NEWGRP);
}
}

for(i=0; i<nOutGps+1; i++)
{
    OutGp[i].lib_mem=(int*)malloc((OutGp[i].size+1)*
        sizeof(int));
    LibMemberPosition=0;
    for(j=0; j<(model2->num_outputs+model2->num_latches);
        j++)
    {
        if((LibOutMem[j].PCSig ==
            CutOutMem[OutGp[i].cut_mem[0]].PCSig) &&
            (LibOutMem[j].NCSig ==
            CutOutMem[OutGp[i].cut_mem[0]].NCSig))
        {
            OutGp[i].lib_mem[LibMemberPosition] = j;
            LibMemberPosition++;
        }
    }
    LibMemberPosition=0;
}

for(i=0; i<nOutGps+1; i++)
{
    for(j=0; j<(OutGp[i].size+1); j++)
    {
        CutOutMem[OutGp[i].cut_mem[j]].myopgrp=i;
        LibOutMem[OutGp[i].lib_mem[j]].myopgrp=i;
    }
}
// End Output Variable Group formation

// Computation of Weighted Row Sum signature
opgrporder = (int*)malloc((nOutGps+1)*sizeof(int));
for(i=0; i<(nOutGps+1); i++)
    opgrporder[i]=i;

oporder = (int *)malloc((model->num_outputs+model->num_latches)*sizeof(int));
liboporder = (int *)malloc((model2->num_outputs+model2->num_latches)*sizeof(int));
tcnt=0;
for(i=0;i<nOutGps+1;i++)
    {
        for(j=0; j<OutGp[opgrporder[i]].size+1; j++)
        {
            oporder[tcnt]=OutGp[opgrporder[i]].cut_mem[j];
            tcnt++;
        }
    }
tcnt=0;
for(i=0;i<(model->num_outputs+model->num_latches);i++)
    {
        if(((OutGp[CutOutMem[i].myopgrp].size)+1)==1)
            CutOutMem[i].mask=1;
        else
            CutOutMem[i].mask=0;
    }
for(i=0;i<(model->num_inputs+model->num_latches);i++)
    {
        CutInMem[i].PWRSig = 0;
        CutInMem[i].NWRSig = 0;
        if(WR_SUM==1)
            {
                CutInMem[i].PWRSig = WtdRow(oporder, CutOutMem,
                CutOutVector[POS][i],
                (model->num_outputs+model->num_latches));
                CutInMem[i].NWRSig = WtdRow(oporder, CutOutMem,
                CutOutVector[NEG][i],
                (model->num_outputs+model->num_latches));
            }
    }
for(i=0;i<nOutGps+1;i++)
    {
        for(j=0; j<OutGp[opgrporder[i]].size+1; j++)
        {
            liboporder[tcnt]=OutGp[opgrporder[i]].lib_mem[j];
            tcnt++;
        }
    }
tcnt=0;
for(i=0;i<(model2->num_outputs+model2->num_latches);i++)
{
    if(((OutG[LibOutMem[i].myopgrp].size)+1)==1)
        LibOutMem[i].mask=1;
    else
        LibOutMem[i].mask=0;
}

for(i=0;i<(model2->num_inputs+model2->num_latches);i++)
{
    LibInMem[i].PWRSig = 0;
    LibInMem[i].NWRSig = 0;
    if(WR_SUM==1)
    {
        LibInMem[i].PWRSig = WtdRow(liboporder, LibOutMem,
            LibOutVector[POS][i],
            (model2->num_outputs+model2->num_latches));
        LibInMem[i].NWRSig = WtdRow(liboporder, LibOutMem,
            LibOutVector[NEG][i],
            (model2->num_outputs+model2->num_latches));
    }
}
free(opgrporder);
free(oporder);
free(liboporder);

//Formation of Input variable groups
nInGps=0;
InGp = (eq_group*)malloc(sizeof(eq_group));
BuildGroup(&InGp, 0, nInGps, NEWGRP);
for(i=1;i<(model->num_inputs+model->num_latches);i++)
{
    assigned=0;
    for(j=0;j<nInGps+1; j++)
    {
        if((CutInMem[i].PRSig==
            CutInMem[InGp[j].cut_mem[0]].PRSig) &
            (CutInMem[i].PWRSig==
            CutInMem[InGp[j].cut_mem[0]].PWRSig) &
            (CutInMem[i].NWRSig==
            CutInMem[InGp[j].cut_mem[0]].NWRSig) &
            (CutInMem[i].NRSig==
            CutInMem[InGp[j].cut_mem[0]].NRSig))
        {
            BuildGroup(&InGp, i, j, NEWMEM);
            assigned=1;
            break;
        }
    }
if(assigned==0)
{
    nInGps++;
    BuildGroup(&InGp, i, nInGps, NEWGRP);
}
}
for(i=0; i<nInGps+1; i++)
{
    InGp[i].lib_mem = (int*)malloc((InGp[i].size+1)*sizeof(int));
    LibMemberPosition=0;
    for(j=0; j<(model2->num_inputs+model->num_latches); j++)
    {
        if((LibInMem[j].PRSig==CutInMem[InGp[i].cut_mem[0]].PRSig) &&
           (LibInMem[j].PWRSig==CutInMem[InGp[i].cut_mem[0]].PWRSig) &&
           (LibInMem[j].NWRSig==CutInMem[InGp[i].cut_mem[0]].NWRSig) &&
           (LibInMem[j].NRSig==CutInMem[InGp[i].cut_mem[0]].NRSig))
        {
            InGp[i].lib_mem[LibMemberPosition] = j;
            LibMemberPosition++;
        }
    }
    LibMemberPosition=0;
    // End Input Variable Group Formation

    //Initializing the binary input vectors
    BinVec = (int*)malloc((nInGps+1)*sizeof(int));
    for(i=0; i<nInGps+1; i++)
    {
        BinVec[i]=0;
    }
    BinVec[0]=1;

    /* Snapshot of current CPU time in order to compute the cumulative time taken by the iterations */
    rc=gettimeofday(&tot4, NULL);

    //LOOP BEGINS → ITERATION COUNT = 500
    for(TestCycle=0; TestCycle<CYCLELIMIT; TestCycle++)
    {
        NumOfGroupsTemp=nInGps;
        NumOfOPGroupsTemp=nOutGps;
        for(i=0; i<nInGps; i++)
        {
            for(j=i+1; j<nInGps+1; j++)
            {

        }
if(InGp[i].size > InGp[j].size)
{
    GpTemp = InGp[j];
    InGp[j] = InGp[i];
    InGp[i] = GpTemp;
}
for(i=0; i<nInGps+1; i++)
{
    for(j=0; j<(InGp[i].size+1); j++)
    {
        CutInMem[InGp[i].cut_mem[j]].myipgrp=i;
        LibInMem[InGp[i].lib_mem[j]].myipgrp=i;
    }
}
//Forming binary input vectors to be applied in the
//current iteration
//######################################################################## m starts ################################################################
for(m=0; m<nInGps+1; m++)
{
    if((nInGps<3)||(nInGps>169))
        gen_vector(BinVec,nInGps, BINARY);
    else
        gen_vector(BinVec, nInGps, RANDOMGEN);
    cnt=0;
    //%%%%%%%%%%%%%%%%%%% i-loop starts %%%%%%%%%%%%%%%%%
    for(i=0;i<nInGps+1; i++)
    {
        if(m!=i)
        {
            for(j=0;j<InGp[i].size+1;j++)
            {
                pcutcurrent[InGp[i].cut_mem[j]+1]=BinVec[cnt];
                plibcurrent[InGp[i].lib_mem[j]+1]=BinVec[cnt];
                if(BinVec[cnt]==1)
                {
                    ncutcurrent[InGp[i].cut_mem[j]+1]=0;
                    nlibcurrent[InGp[i].lib_mem[j]+1]=0;
                }
                else
                {
                    ncutcurrent[InGp[i].cut_mem[j]+1]=1;
                    nlibcurrent[InGp[i].lib_mem[j]+1]=1;
                }
            }
            cnt++;
        }
for(k=0; k<InGp[m].size+1; k++)
{
    for(j=0; j<InGp[m].size+1; j++)
    {
        pcutcurrent[InGp[m].cut_mem[j]+1]=0;
        plibcurrent[InGp[m].lib_mem[j]+1]=0;
        ncutcurrent[InGp[m].cut_mem[j]+1]=1;
        nlibcurrent[InGp[m].lib_mem[j]+1]=1;
    }
    pcutcurrent[InGp[m].cut_mem[k]+1]=1;
    plibcurrent[InGp[m].lib_mem[k]+1]=1;
    ncutcurrent[InGp[m].cut_mem[k]+1]=0;
    nlibcurrent[InGp[m].lib_mem[k]+1]=0;
}

//Applying the vectors to the models and recording
//the output
Sim(model, CutOutVector[POS][InGp[m].cut_mem[k]],
    pcutcurrent);
Sim(model, CutOutVector[NEG][InGp[m].cut_mem[k]],
    ncutcurrent);
Sim(model2, LibOutVector[POS][InGp[m].lib_mem[k]],
    plibcurrent);
Sim(model2, LibOutVector[NEG][InGp[m].lib_mem[k]],
    nlibcurrent);

CutInMem[InGp[m].cut_mem[k]].PRSig =0;
CutInMem[InGp[m].cut_mem[k]].NRSig =0;
LibInMem[InGp[m].lib_mem[k]].PRSig =0;
LibInMem[InGp[m].lib_mem[k]].NRSig =0;
}

// Compute the Row Sum signature
RowSum(CutOutVector, CutInMem,
    (model->num_inputs+model->num_latches),
    (model->num_outputs+model->num_latches));
RowSum(LibOutVector, LibInMem,
    (model2->num_inputs+model2->num_latches),
    (model2->num_outputs+model2->num_latches));

// Compute the Column Sum signature
ColSum(CutOutVector, CutOutMem,
    (model->num_inputs+model->num_latches),
    (model->num_outputs+model->num_latches));
ColSum(LibOutVector,LibOutMem,  
(model2->num_inputs+model2->num_latches),  
(model2->num_outputs+model2->num_latches));

// Compute the Grouped Column Sum signature
if(GC_SUM==1){
    for(i=0; i<(model2->num_outputs+  
        model2->num_latches);i++)
        {
            CutOutMem[i].PGRSig=(int *)malloc  
                ((nInGps+1)*sizeof(int));
            CutOutMem[i].NGRSig=(int *)malloc  
                ((nInGps+1)*sizeof(int));
            LibOutMem[i].PGRSig=(int *)malloc  
                ((nInGps+1)*sizeof(int));
            LibOutMem[i].NGRSig=(int *)malloc  
                ((nInGps+1)*sizeof(int));

            for(j=0; j<(nInGps+1);j++)
                {
                    CutOutMem[i].PGRSig[j]=0;
                    CutOutMem[i].NGRSig[j]=0;
                    LibOutMem[i].PGRSig[j]=0;
                    LibOutMem[i].NGRSig[j]=0;

                    for(x=0;x<(InGp[j].size+1);x++)
                        {
                            CutOutMem[i].PGRSig[j] = CutOutMem[i].PGRSig[j]+  
                                CutOutVector[POS][InGp[j].cut_mem[x]][i];
                            CutOutMem[i].NGRSig[j]=CutOutMem[i].NGRSig[j]+  
                                CutOutVector[NEG][InGp[j].cut_mem[x]][i];
                            LibOutMem[i].PGRSig[j]=LibOutMem[i].PGRSig[j]+  
                                LibOutVector[POS][InGp[j].lib_mem[x]][i];
                            LibOutMem[i].NGRSig[j]=LibOutMem[i].NGRSig[j]+  
                                LibOutVector[NEG][InGp[j].lib_mem[x]][i];
                        }
                }
}

} // End Computation of Grouped Column Sum signature

// FORMATION OF OUTPUT VARIABLE GROUPS
for(x=0;x<nOutGps+1;x++)
    OutGp[x].new=0;
for(j=0;j<nOutGps+1;j++)
    {
        if(((OutGp[j].size+1)>1)&&(OutGp[j].new==0))
            {
                for(x=0;x<nOutGps+1;x++)
                    OutGp[x].new=0;
                tmpcutmem= (int *)malloc((OutGp[j].size+1)*sizeof(int));
                tmplibmem= (int *)malloc((OutGp[j].size+1)*sizeof(int));
            }
tmpsize=OutGp[j].size;
for(k=0;k<tmpsize+1;k++)
{
tmpcutmem[k]=OutGp[j].cut_mem[k];
tmplibmem[k]=OutGp[j].lib_mem[k];
}
free(OutGp[j].cut_mem);
free(OutGp[j].lib_mem);
OutGp[j].cut_mem=(int *)malloc(1*sizeof(int));
OutGp[j].size=0;
OutGp[j].cut_mem[0]=tmpcutmem[0];
for(k=1;k<tmpsize+1;k++)
{
  assigned=0;
  if((CutOutMem[tmpcutmem[k]].PCSig==
      CutOutMem[OutGp[j].cut_mem[0]].PCSig) &
      (isOpGRSUmSigEq(CutOutMem,CutOutMem,
                      tmpcutmem[k],OutGp[j].cut_mem[0],nInGps+1,
                      GC_SUM)==1) &
      (CutOutMem[tmpcutmem[k]].NCSig==
       CutOutMem[OutGp[j].cut_mem[0]].NCSig))
    BuildGroup(&OutGp, tmpcutmem[k], j, NEWMEM);
  else
  {
    for(x=0;x<nOutGps+1;x++)
    {
      if((OutGp[x].new==1) &
      (CutOutMem[tmpcutmem[k]].PCSig==
       CutOutMem[OutGp[x].cut_mem[0]].PCSig) &
      (isOpGRSUmSigEq(CutOutMem,CutOutMem,
                      tmpcutmem[k],OutGp[x].cut_mem[0],nInGps+1,
                      GC_SUM)==1) &
      (CutOutMem[tmpcutmem[k]].NCSig==
       CutOutMem[OutGp[x].cut_mem[0]].NCSig))
      {BuildGroup(&OutGp, tmpcutmem[k], x, NEWMEM);
      assigned=1;
      break;
      }
    }
  }
  if(assigned==0)
  {
    nOutGps++;
    BuildGroup(&OutGp, tmpcutmem[k], nOutGps,
                NEWGRP);
    OutGp[nOutGps].new=1;
  }
}
assigned=0;
}
// else ends
free(tmpcutmem);

for(x=0;x<nOutGps+1;x++)
{
    if((x==j)||(OutGp[x].new==1))
    {
        OutGp[x].lib_mem=(int*)malloc
((OutGp[x].size+1)*sizeof(int));
        tmplibsize=0;
        for(k=0;k<tmpsize+1;k++)
        {
            if((CutOutMem[OutGp[x].cut_mem[0]].PCSig==
LibOutMem[tmplibmem[k]].PCSig) &
(isOpGRSumSigEq(CutOutMem,LibOutMem,
OutGp[x].cut_mem[0],tmplibmem[k],
nInGps+1,GC_SUM)==1) &
(CutOutMem[OutGp[x].cut_mem[0]].NCSig==
LibOutMem[tmplibmem[k]].NCSig))
        {
            OutGp[x].lib_mem[tmplibsize]=tmplibmem[k];
            tmplibsize++;
        }
    }
}
free(tmplibmem);
}
//End Output variable group formation

for(i=0; i<nOutGps+1; i++)
{
    for(j=0; j<(OutGp[i].size+1); j++)
    {
        CutOutMem[OutGp[i].cut_mem[j]].myopgrp=i;
        LibOutMem[OutGp[i].lib_mem[j]].myopgrp=i;
    }
}
if(GC_SUM==1){
    for(i=0; i<(model2->num_outputs+
        model2->num_latches); i++)
    {
        free(CutOutMem[i].PGRSig);
        free(CutOutMem[i].NGRSig);
        free(LibOutMem[i].PGRSig);
        free(LibOutMem[i].NGRSig);
    }
// Compute Weighted Row Sum signature
opgrporder = (int*)malloc((nOutGps+1)*sizeof(int));
for(i=0; i<(nOutGps+1); i++)
    opgrporder[i]=i;

oporder = (int *)malloc((model->num_outputs+
    model->num_latches)*sizeof(int));
liboporder = (int *)malloc((model2->num_outputs+
    model2->num_latches)*sizeof(int));
tcnt=0;
for(i=0;i<nOutGps+1;i++)
    for(j=0; j<OutGp[opgrporder[i]].size+1; j++)
        oporder[tcnt]=OutGp[opgrporder[i]].cut_mem[j];
    tcnt++;
for(i=0;i<nOutGps+1;i++)
    for(j=0; j<OutGp[opgrporder[i]].size+1; j++)
        liboporder[tcnt]=OutGp[opgrporder[i]].lib_mem[j];
    tcnt++;
for(i=0;i<nOutGps+1;i++)
    for(j=0; j<OutGp[opgrporder[i]].size+1; j++)
        if(((OutGp[CutOutMem[i].myopgrp].size)+1)==1)
            CutOutMem[i].mask=1;
        else
            CutOutMem[i].mask=0;
for(i=0;i<(model->num_outputs+model->num_latches);i++)
    if(((OutGp[CutOutMem[i].myopgrp].size)+1)==1)
        CutOutMem[i].mask=1;
    else
        CutOutMem[i].mask=0;
for(i=0;i<(model2->num_outputs+model2->num_latches);i++)
    if(((OutGp[LibOutMem[i].myopgrp].size)+1)==1)
        LibOutMem[i].mask=1;
    else
        LibOutMem[i].mask=0;
for(i=0;i<(model->num_inputs+model->num_latches);i++)
    if(WR_SUM==1)
CutInMem[i].PWRSig = WtdRow(oporder, CutOutMem,
CutOutVector[POS][i],
(model->num_outputs+model->num_latches));
CutInMem[i].NWRSig = WtdRow(oporder, CutOutMem,
CutOutVector[NEG][i],
(model->num_outputs+model->num_latches));

for(i=0;i<(model2->num_inputs+model2->num_latches);i++)
{
    LibInMem[i].PWRSig = 0;
    LibInMem[i].NWRSig = 0;
    if(WR_SUM==1)
    {
        LibInMem[i].PWRSig = WtdRow(liboporder, LibOutMem,
        LibOutVector[POS][i],
        (model2->num_outputs+model2->num_latches));
        LibInMem[i].NWRSig = WtdRow(liboporder, LibOutMem,
        LibOutVector[NEG][i],
        (model2->num_outputs+model2->num_latches));
    }
}
free(liboporder);
free(oporder);
free(opgrporder);

if(GR_SUM==1){
    for(i=0; i<(model2->num_inputs+
        model2->num_latches);i++)
    {
        CutInMem[i].PGRSig=(int *)
        malloc((nOutGps+1)*sizeof(int));
        CutInMem[i].NGRSig=(int *)
        malloc((nOutGps+1)*sizeof(int));
        LibInMem[i].PGRSig=(int *)
        malloc((nOutGps+1)*sizeof(int));
        LibInMem[i].NGRSig=(int *)
        malloc((nOutGps+1)*sizeof(int));
        for(j=0; j<(nOutGps+1);j++)
        {
            CutInMem[i].PGRSig[j]=0;
            CutInMem[i].NGRSig[j]=0;
            LibInMem[i].PGRSig[j]=0;
            LibInMem[i].NGRSig[j]=0;
        }
    }
}
for(x=0;x<(OutGp[j].size+1);x++)
{
    CutInMem[i].PGRSig[j]=CutInMem[i].PGRSig[j]+
    CutOutVector[POS][i][OutGp[j].cut_mem[x]];
    CutInMem[i].NGRSig[j]=CutInMem[i].NGRSig[j]+
    CutOutVector[NEG][i][OutGp[j].cut_mem[x]];
    LibInMem[i].PGRSig[j]=LibInMem[i].PGRSig[j]+
    LibOutVector[POS][i][OutGp[j].lib_mem[x]];
    LibInMem[i].NGRSig[j]=LibInMem[i].NGRSig[j]+
    LibOutVector[NEG][i][OutGp[j].lib_mem[x]];
}

//Compute ROW DIFFERENCE SIGNATURE
if(ROW_DIFF==1){
    for(i=0;i<(model2->num_inputs+
              model2->num_latches);i++)
    {
        CutInMem[i].PRDSig[0]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));
        CutInMem[i].PRDSig[1]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));
        CutInMem[i].NRDSig[0]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));
        CutInMem[i].NRDSig[1]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));

        ComputeRowDiffSig(&CutInMem,CutOutVector,
            (model2->num_inputs+model2->num_latches),
            (model2->num_outputs+model2->num_latches),i);

        mysort(CutInMem[i].PRDSig[0],(model2->num_inputs+
              model2->num_latches));
        mysort(CutInMem[i].PRDSig[1],(model2->num_inputs+
              model2->num_latches));
        mysort(CutInMem[i].NRDSig[0],(model2->num_inputs+
              model2->num_latches));
        mysort(CutInMem[i].NRDSig[1],(model2->num_inputs+
              model2->num_latches));

        LibInMem[i].PRDSig[0]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));
        LibInMem[i].PRDSig[1]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));
        LibInMem[i].NRDSig[0]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));
        LibInMem[i].NRDSig[1]=(int *)calloc(
            (model2->num_inputs+model2->num_latches),
            sizeof(int));
    }
}
LibInMem[i].PRDSig[1] = (int *)calloc((model2->num_inputs+model2->num_latches), sizeof(int));
LibInMem[i].NRDSig[0] = (int *)calloc((model2->num_inputs+model2->num_latches), sizeof(int));
LibInMem[i].NRDSig[1] = (int *)calloc((model2->num_inputs+model2->num_latches), sizeof(int));

ComputeRowDiffSig(&LibInMem, LibOutVector,
(model2->num_inputs+model2->num_latches),
(model2->num_outputs+model2->num_latches), i);

mysort(LibInMem[i].PRDSig[0], (model2->num_inputs+
model2->num_latches));
mysort(LibInMem[i].PRDSig[1], (model2->num_inputs+
model2->num_latches));
mysort(LibInMem[i].NRDSig[0], (model2->num_inputs+
model2->num_latches));
mysort(LibInMem[i].NRDSig[1], (model2->num_inputs+
model2->num_latches));

for (x=0; x<nInGps+1; x++)
InGp[x].new = 0;

// INPUT VARIABLE GROUP FORMATION
for (j=0; j<nInGps+1; j++)
{
  if ((InGp[j].size+1>1) && (InGp[j].new==0))
  {
    for (x=0; x<nInGps+1; x++)
      InGp[x].new = 0;
    tmpcutmem = (int *)malloc((InGp[j].size+1)*sizeof(int));
    tmplibmem = (int *)malloc((InGp[j].size+1)*sizeof(int));
    tmpsize = InGp[j].size;
    for (k=0; k<tmpsize+1; k++)
      {
        tmpcutmem[k] = InGp[j].cut_mem[k];
        tmplibmem[k] = InGp[j].lib_mem[k];
      }
    free(InGp[j].cut_mem);
    free(InGp[j].lib_mem);
    InGp[j].cut_mem = (int *)malloc(sizeof(int));
    InGp[j].size = 0;
    InGp[j].cut_mem[0] = tmpcutmem[0];
for(k=1;k<tmpsize+1;k++)
{
    assigned=0;
    if((CutInMem[tmpcutmem[k]].PRSig==
        CutInMem[InGp[j].cut_mem[0]].PRSig) &&
        (CutInMem[tmpcutmem[k]].PWRSig==
            CutInMem[InGp[j].cut_mem[0]].PWRSig) &&
        (isGRSumSigEq(CutInMem,CutInMem,tmpcutmem[k],
            InGp[j].cut_mem[0],nOutGps+1,GR_SUM)==1) &&
        (isIpRDSigEq(CutInMem, CutInMem,
            tmpcutmem[k], InGp[j].cut_mem[0],
            (model2->num_inputs+model2->num_latches),
            ROW_DIFF)==1) &&
        (CutInMem[tmpcutmem[k]].NWRSig==
            CutInMem[InGp[j].cut_mem[0]].NWRSig) &&
        (CutInMem[tmpcutmem[k]].NRSig==
            CutInMem[InGp[j].cut_mem[0]].NRSig))
        BuildGroup(&InGp, tmpcutmem[k], j, NEWMEM);
    else
    {
        for(x=0;x<nInGps+1;x++)
        {
            if((InGp[x].new==1) &&
                (isGRSumSigEq(CutInMem, CutInMem,
                    tmpcutmem[k], InGp[x].cut_mem[0],
                    nOutGps+1,GR_SUM)==1) &&
                (CutInMem[tmpcutmem[k]].PRSig==
                    CutInMem[InGp[x].cut_mem[0]].PRSig) &&
                (CutInMem[tmpcutmem[k]].PWRSig==
                    CutInMem[InGp[x].cut_mem[0]].PWRSig) &&
                (isIpRDSigEq(CutInMem, CutInMem,
                    tmpcutmem[k], InGp[x].cut_mem[0],
                    (model2->num_inputs+model2->num_latches),
                    ROW_DIFF)==1) &&
                (CutInMem[tmpcutmem[k]].NWRSig==
                    CutInMem[InGp[x].cut_mem[0]].NWRSig) &&
                (CutInMem[tmpcutmem[k]].NRSig==
                    CutInMem[InGp[x].cut_mem[0]].NRSig))
                BuildGroup(&InGp, tmpcutmem[k], x, NEWMEM);
            assigned=1;
            break;
        }
    if(assigned==0)
    {
        nInGps++;
        BuildGroup(&InGp, tmpcutmem[k], nInGps,
            NEWGRP);
        InGp[nInGps].new=1;
    }
}
for(x=0;x<nInGs+1;x++)
{
    if((x==j)||(InGp[x].new==1))
    {
        InGp[x].lib_mem=(int *)
        malloc((InGp[x].size+1)*sizeof(int));
        tmplibsize=0;
        for(k=0;k<tmplibsize+1;k++)
        {
            if((LibInMem[tmplibmem[k]].PRSig==
            CutInMem[InGp[x].cut_mem[0]].PRSig) &&
            (LibInMem[tmplibmem[k]].PWRSig==
            CutInMem[InGp[x].cut_mem[0]].PWRSig) &&
            (isGRSumSigEq(LibInMem,CutInMem,
            tmplibmem[k],InGp[x].cut_mem[0],nOutGs+1,
            GR_SUM)==1) &&
            (isIpRDSigEq(LibInMem,CutInMem,
            tmplibmem[k],InGp[x].cut_mem[0],
            (model2->num_inputs+model2->num_latches),
            ROW_DIFF)==1) &&
            (LibInMem[tmplibmem[k]].NWRSig==
            CutInMem[InGp[x].cut_mem[0]].NWRSig) &&
            (LibInMem[tmplibmem[k]].NRSig==
            CutInMem[InGp[x].cut_mem[0]].NRSig))
            {
                InGp[x].lib_mem[tmplibsize]=tmplibmem[k];
                tmplibsize++;
            }
        }  
        free(tmpcutmem);
        free(tmplibmem);
    }
}
//End of INPUT VARIABLE GROUP FORMATION

if(GR_SUM==1){
    for(i=0; i<(model2->num_inputs+model2->num_latches);i++)
    {
        free(CutInMem[i].PGRSig);
        free(CutInMem[i].NGRSig);
        free(LibInMem[i].PGRSig);
        free(LibInMem[i].NGRSig);
    }
free(LibInMem[i].NGRSig);
}
}

if(ROW_DIFF==1) {
    for(i=0;i<(model2->num_inputs+
             model2->num_latches);i++)
        {
            free(CutInMem[i].PRDSig[0]);
            free(CutInMem[i].PRDSig[1]);
            free(CutInMem[i].NRDSig[0]);
            free(CutInMem[i].NRDSig[1]);
            free(LibInMem[i].PRDSig[0]);
            free(LibInMem[i].PRDSig[1]);
            free(LibInMem[i].NRDSig[0]);
            free(LibInMem[i].NRDSig[1]);
        }
}

if(done==1)
    break;

BinVec = realloc(BinVec, (nInGps+1)*sizeof(int));
for(i=NumOfGroupsTemp;i<nInGps+1;i++)
    BinVec[i] = 0;

/* Compute and print the time taken by each loop in which either a new input or output variable group is formed */
rc=gettimeofday(&tot5, NULL);
if(((nInGps+1)>(NumOfGroupsTemp+1))&&
   ((nOutGps+1)>(NumOfOPGroupsTemp+1)))
    fprintf(testfile,"CUMULATIVE TIME FOR TESTCYCLE
            %d =%.6lf - both changed
",TestCycle,
            ((tot5.tv_sec+(tot5.tv_usec/1000000.0))-
            (tot4.tv_sec+(tot4.tv_usec/1000000.0))));
else if((nInGps+1)>(NumOfGroupsTemp+1))
    fprintf(testfile,"CUMULATIVE TIME FOR TESTCYCLE
            %d =%.6lf - %d changed\n",TestCycle,
            ((tot5.tv_sec+(tot5.tv_usec/1000000.0))-
            (tot4.tv_sec+(tot4.tv_usec/1000000.0))),1);
else if((nOutGps+1)>(NumOfOPGroupsTemp+1))
    fprintf(testfile,"CUMULATIVE TIME FOR TESTCYCLE
            %d =%.6lf -%d changed\n",TestCycle,
            ((tot5.tv_sec+(tot5.tv_usec/1000000.0))-
            (tot4.tv_sec+(tot4.tv_usec/1000000.0))),0);
} //LOOP ENDS
/Compute and display the total time taken and the group sizes
rc=gettimeofday(&tot2, NULL);
fprintf(testfile,"Printing FINAL I/P Groups info for
ROW_DIFF=%d GR_SUM=%d WR_SUM=%d\n",ROW_DIFF,GR_SUM,WR_SUM);
fprintf(testfile,"TOTAL TIME =%.6lf\n",
(tot2.tv_sec+(tot2.tv_usec/1000000.0))-
(tot1.tv_sec+(tot1.tv_usec/1000000.0)));
for(i=0; i<nInGps+1; i++)
{
    if((InGp[i].size+ 1)>1){
        fprintf(testfile,"Size=%d\n", InGp[i].size+1);
        for(j=0; j<InGp[i].size+1; j++)
            fprintf(testfile,"%d, ",InGp[i].cut_mem[j]);
        fprintf(testfile,"\n");
    }
}
fprintf(testfile,"\n");
fprintf(testfile,"Printing FINAL O/P Groups info for
COL_DIFF=%d GC_SUM=%d\n",COL_DIFF,GC_SUM);
fprintf(testfile,"TOTAL TIME =%.6lf\n",
(tot2.tv_sec+(tot2.tv_usec/1000000.0))-
(tot1.tv_sec+(tot1.tv_usec/1000000.0)));
for(i=0; i<nOutGps+1; i++)
{
    if((OutGp[i].size+ 1)>1){
        fprintf(testfile,"Size=%d\n", OutGp[i].size+1);
        for(j=0; j<OutGp[i].size+1; j++)
            fprintf(testfile,"%d, ",OutGp[i].cut_mem[j]);
        fprintf(testfile,"\n");
    }
}
fprintf(testfile,"\n");
fclose(testfile);
free (pcutcurrent);
free (ncutcurrent);
free (plibcurrent);
free (nlibcurrent);
free(CutInMem);
free(LibInMem);
free(CutOutMem);
free(LibOutMem);
free(OutGp);
free(InGp);
for(x=0; x<(model->num_inputs+model->num_latches); x++)
{
    free(CutOutVector[POS][x]);
    free(CutOutVector[NEG][x]);
    free(LibOutVector[POS][x]);
    free(LibOutVector[NEG][x]);
}

aiger_reset (model);
aiger_reset (model2);
return res;

}//MAIN METHOD END
Bibliography


