A Signature–Based Approach to Formal Logic Verification

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

vorgelegt der

Mathematisch–Naturwissenschaftlich-Technischen Fakultät
der Martin Luther Universität Halle–Wittenberg

von Frau Janett Mohnke

geb. am: 02. April 1967 in: Luckenwalde

Gutachter:

1. Prof. Dr. Bernd Becker
2. Prof. Dr. Sharad Malik
3. Prof. Dr. Paul Molitor

Halle (Saale), 12. Februar 1999

Zusammenfassung

In der vorliegenden Arbeit untersuchen wir das Problem, für zwei Boolesche Funktionen zu testen, ob es eine Permutation ihrer Eingangsvariablen gibt, unter der diese Funktionen äquivalent sind. Eine Lösung für dieses Problem ist in verschiedenen Gebieten der Synthese und Verifikation kombinatorischer Schaltkreise von Nutzen. In der Logikverifikation findet es dann Verwendung, wenn die genaue Zuordnung der Eingangsvariablen der zu verifizierenden Schaltkreise nicht mehr bekannt ist. Das Problem ist NP–schwer, weshalb auf heuristische Lösungsansätze zurückgegriffen werden muß. Der Lösungsansatz, der in dieser Arbeit vorgestellt wird, berechnet für jede der Eingangsvariablen einer Booleschen Funktion Signaturen. Diese Signaturen sollen helfen, eine für Permutationsäquivalenz gültige Zuordnung der Variablen auszuwählen. Signaturen sind für eine große Anzahl der getesten Beispiele sehr wirkungsvoll. Leider gibt es auch oft Variablen, die nicht eindeutig identifiziert werden können. Interessanterweise gehören zu dieser Menge, für ein gegebenes Beispiel, immer nahezu dieselben Variablen — unabhängig von der verwendeten Signatur. In der vorliegenden Arbeit wird dieses Problem eingehend untersucht. Außerdem zeigen wir, wie die vorgestellten Methoden zur Lösung eines Problems der Verifikation sequentieller Schaltkreise herangezogen werden können, dem Problem, eine Zuordnung der Speicherelemente zweier sequentieller Schaltkreise zu finden, wenn diese nicht mehr bekannt ist, man aber weiß, daß die Zustände beider Schaltkreise mit der gleichen Methode kodiert wurden. Die Effizienz der in dieser Arbeit vorgestellten Methoden wird anhand einer Vielzahl von Experimenten veranschaulicht.

Abstract

We consider the problem of checking the equivalence of two Boolean functions under arbitrary input permutations. This problem has several applications in the synthesis and verification of combinational logic. In logic verification this is needed when the exact correspondence of inputs between the two circuits is not known. The problem is NP–hard, thus recourse is taken to heuristics that work well in practice. The approach presented in this thesis computes signatures for each input variable that will help to establish correspondence of variables. Signatures work well for a large number of the investigated examples. However, for each choice of signature, there remain variables that cannot be uniquely identified. Our research has shown that, for a given example, this set of problematic variables tends to be the same — regardless of the choice of signatures. In this thesis, we investigate this problem. Furthermore, we demonstrate how the introduced techniques can be applied to a problem in sequential logic verification, the problem of establishing the unknown correspondence of the latches (memory elements) of two sequential circuits which have the same state encoding. Experimental results on a large number of examples establish the efficacy of the introduced methods.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For years very large scale integrated circuits and the resulting digital systems have conquered a place in almost all areas of our life — even in security sensitive applications. Complex digital systems control airplanes, have been used in banks and on intensive-care units. Hence, the demand for error-free designs is more important than ever. In addition, economic reasons also underline this demand: the design and production process of present day VLSI-circuits is highly time- and cost-intensive. Moreover, it is nearly impossible to repair integrated circuits. Thus, it is desirable to detect design errors *early* in the design process using computer-aided tools and not just after producing the prototype chip. Circuits have become more complex regarding to their design and the tasks that they are designed for as the level of integration on an integrated circuit (IC or chip) itself has been increasing. While a handful of devices were integrated on the first circuits in the 1960s, circuits with over a million devices can be manufactured nowadays. With that, the probability increases that design errors remain undetected (e.g. the problems with the INTEL-Pentium-processor in 1995). In other words, it is not just more important to get error-free designs, but it also becomes an increasingly difficult task for a team of human designers to carry out a full design without errors. So, the development and improvement of verification tools that are able to prove the correctness of design of present day digital systems has obtained major significance.

Verification is the comparison of two models for consistency. Traditionally, checking the correctness of a system is done by simulation based methods. In such an approach, the designer has to create a complete set of test vectors which represents all possible inputs of the system. Then, the outputs of the design for each of these input vectors must be analyzed in order to guarantee the correctness of this design. It is obvious to see that this process is very CPU-time intensive and thus impractical for larger designs.

As a result, another kind of verification strategies have been becoming popular: strategies that use formal verification techniques. By using these techniques, the correctness of a design for all input combinations can be guaranteed.

Aspects of Formal Logic Verification

Nowadays, a digital system is designed with the help of computer–aided design tools that work at different, almost independent levels in a hierarchical manner. Such a process of designing usually starts with describing the system, which has to be designed, in an abstract model. On this model, an extensive simulation is made. Then it becomes the *golden specification* [20]. Starting with this golden specification, a detailed implementation is derived, passing through different design levels step by step. First a synthesizable behavioral Register-Transfer-Level (RTL) description is derived which describes the block structure behavior of the design. This description is then translated into the structural description describing the combinational logic of the system. From this the transistor netlist is derived which finally leads to the physical layout description.

To be able to catch bugs as early as possible in the above design process, it is important to verify the functionality of the design at *every* level of this process against the golden specification. At first glance, this so called *implementation verification* should play a minor role when computer-aided design tools are used. Those methods should provide *correctness by construction* [23]. However, because of the widespread use of synthesis tools this is not the case. While the synthesis algorithms have guaranteed properties of correctness, their software implementations cannot be guaranteed to be error–free. The same holds for the implementation of the operating system and the data bases. So there is a need of implementation verification methods that support synthesis tools.

This verification proceeds in two phases [20]. In the first phase, a Boolean network is extracted from the actual description (e.g., [7, 35]). Then, in a second phase, this Boolean network is verifyed by some formal verification methods against the original (golden) specification. It is this part of the verification process, that is the focus of this thesis. Moreover, we need to distinguish between the verification of *combinational* and *sequential* circuits. In combinational circuits, the outputs depend only on the current inputs, where in sequential circuits, the outputs depend not only on the current inputs but also on the past sequence of inputs.

The Combinational Permutation Equivalence Problem

In this thesis, we focus on a special problem in *combinational* logic verification. The tools used at the different levels in the design process may have their own naming conventions for the inputs or the outputs of the circuit which has to be designed. Then the input/output correspondence between the different descriptions of the circuit design gets lost. However, before we can verify the equivalence of the two Boolean networks, we need to know this correspondence. The main focus of this thesis is to determine such a lost correspondence for the inputs of two Boolean networks. This problem is $\mathcal{NP}$–hard. Thus, we need to consider techniques that are non-exhaustive and work well in practice. We will introduce such techniques and demonstrate their practical efficacy. Moreover, we shortly explain how they can be used to find a lost correspondence for the outputs of two Boolean networks, although a evaluation of this would be beyond the scope of this thesis.

Note that a solution for the permutation equivalence problem can be used in another part of circuit design as well. It also is important for Boolean matching. Boolean matching is the key operation

in technology mapping. It checks whether an element of a given library can be used to implement a part of a Boolean function. This can be formulated as checking the equivalence between a given Boolean function, called the *target function*, and the set of functions representing a library element. Often this is considered for any permutation of the input variables.

The Latch Correspondence Problem

Furthermore, we demonstrate how the techniques that we have used to handle the *combinational* permutation equivalence problem can be easily applied to a problem arising from *sequential* logic verification.

Verifying general sequential circuits is a very difficult task. The techniques that exist so far for verifying those circuits are not applicable to very large designs. So their use in a practical design methodology is limited. However, some sequential verification problems can be reduced to a combinational verification problem. One of these is the case when the corresponding latches (memory elements) in the two designs that we have to test for equivalence, are identified. Thus it is desirable to have a way to establish this correpondence between the latches of two given sequential circuits. More formally, we define the following problem of sequential logic verification (we call it the latch correspondence problem): given two sequential circuits, does there exist a correspondence between the latches of these two circuits, such that the combinational parts are equivalent using this latch correspondence?

It is easy to see that there is a connection between the combinational permutation equivalence problem and this latch correspondence problem. We show that this is indeed the case, underline the differences between the two problems, and demonstrate how we can apply the techniques of the combinational permutation equivalence problem to the latch correspondence problem.

Contents

After our introduction in **Chapter 1**, we discuss the background of this thesis in **Chapter 2**.

In **Chapter 3** and in **Chapter 4**, we describe the combinational permutation equivalence problem, explain why it is $\mathcal{NP}$–hard, and provide techniques to handle the problem. Moreover, an extensive analysis and evaluation of our practical experiments is made.

Chapter 5 describes, how the techniques used to handle the combinational permutation equivalence problem can be applied to the problem of finding a correspondence between the latches of two sequential circuits.

Finally, we give a summary of the techniques provided in Chapters 3, 4, and 5 and mention some ideas for future projects based on the results of this thesis in **Chapter 6**.

Acknowledgements

I would like to thank my advisor Prof. Paul Molitor for the stimulating discussions, the helpful advice and the constructive critical comments during the work on this thesis. I am particulary indebted to him for his invaluable support, his patience and understanding after the birth of my daughter Jasmin in 1994. This helped me very much to continue the scientific work on my thesis.

I would also like to express my special gratitude to Prof. Sharad Malik for giving me the opportunity to come to Princeton University in 1992. These six months in Princeton laid the foundations of this thesis. I am very thankful to him for his continuing encouragement and for the inumerable and very helpful discussions during all the time after 1992. I not only appreciate very much his patience and ability to explain things clearly but also profited a lot from his experience in presenting scientific results.

I am very grateful to Dr. Michael Weber, who *was* my teacher, for introducing me into the topic of VLSI circuit design and for supporting my research visits at University of Saarland and at Princeton University. I also thank Dr. Michael Weber, who *is* the manager of DResearch Digital Media Systems GmbH, for giving me the permission to use the equipment of our company to print out my thesis.

I thank Prof. Günther Hotz from University of Saarland for inviting me to his chair at University of Saarland and for his support during this time.

Also I would like to thank the International Research & Exchange Board IREX for the scholarship. I am especially grateful to Beate Dafeldecker from IREX for her friendly help and support in organizing the research visit in Princeton before, during and after my stay there.

I thank Prof. Tsutomu Sasao for the structural description of benchmark *t481* which helped me to figure out the characteristic of this circuit.

I would like to thank all my friends and colleagues from my time at Humboldt University in Berlin, University of Saarland in Saarbrücken, Princeton University, and Martin–Luther University in Halle who helped me with enlightening discussions, with encouragement and with their friendship. Especially, I would like to thank Dr. Frank Bauernöppel, Laura Heinrich-Litan, Steve Chun-yao Huang, Petra Ludt-Vogelgesang, Dirk Möller, Ralf Oelschlägel, Ines Peters, Dr. Klaus Peters, Anke Remus, Dr. Christoph Scholl, and Wolfgang Vogelgesang.

Mein besonderer Dank gilt meinen Eltern, Monika und Gerhard Lochert, für eine wunderbare Kindheit als eine der Quellen meiner Kraft und für die Hilfe und Ermutigung auf meinem Weg zur Vollendung dieser Arbeit.

Last but not least I would like to thank my family for their love and their patience. I thank my daugther Jasmin for beeing there, and I thank my husband Klaus for the hours of proof reading and for providing me with enough chocolate during the last time of completing this thesis. Thank you for beeing an inexhaustible source of motivation and encouragement!

Chapter 2

Background

In this chapter we review some basic notation and definitions that will be used in the sequel. In addition, we discuss tractable and intractable problems — one of the basic aspects of not just circuit design but also computer science. Finally, we discuss a state–of–the–art data structure — the reduced ordered binary decision diagram [6] — and justify our use of it as the basic data structure for our algorithms. For more information we refer to [23].

2.1 Notation

We mainly use the notations introduced in [23]. We use braces (i.e., $\{\}$) to denote *unordered* sets and parentheses (i.e., $[\,]$) to denote *ordered* sets. For example, an unordered set of three elements is denoted by $S = \{a, b, c\}$. Vectors of elements are ordered sets. They are denoted by lowercase bold characters. The *cardinality* of a set is the number of its elements. It is denoted by $||$. Given a set S, a *partition* of S is a set of disjoint subsets of S whose union is S. For example, $P_S = \{\{b\}, \{a, c\}\}$ is a partition for our example set. Set membership of an element is denoted by $\in$, set inclusion by $\subset$ or $\subseteq$. The symbol $\forall$ is the *universal quantifier*, the symbol $\exists$ the *existential quantifier*. Implication is denoted by $\Longrightarrow$ and co–implication by $\Longleftrightarrow$. The symbol : means *such that*.

The *Cartesian product* of two sets X and Y, denoted by $X \times Y$, is the set of all ordered pairs (x, y), such that $x \in X$ and $y \in Y$. A *relation* R between two sets X and Y is a subset of $X \times Y$. We write xRy when $x \in X$, $y \in Y$ and $(x, y) \in R$. An *equivalence relation* is a subset R of $X \times X$ which is *reflexive* (i.e., $(x, x) \in R$), *symmetric* (i.e., $(x, y) \in R \Longrightarrow (y, x) \in R$), and *transitive* (i.e., $(x, y) \in R$ and $(y, z) \in R \Longrightarrow (x, z) \in R$). A *partial order* is a relation between X and itself that is reflexive, anti–symmetric (i.e., $(x, y) \in R$ and $(y, x) \in R \Longrightarrow x = y$) and transitive.

A *function (or map)* between two sets X and Y is a relation having the property that each element of X appears as the first element in one and only one pair of the relation. A function between two sets X and Y is denoted by $f : X \longrightarrow Y$. The sets X and Y are called the *domain* and the *co–domain* of the function, respectively.

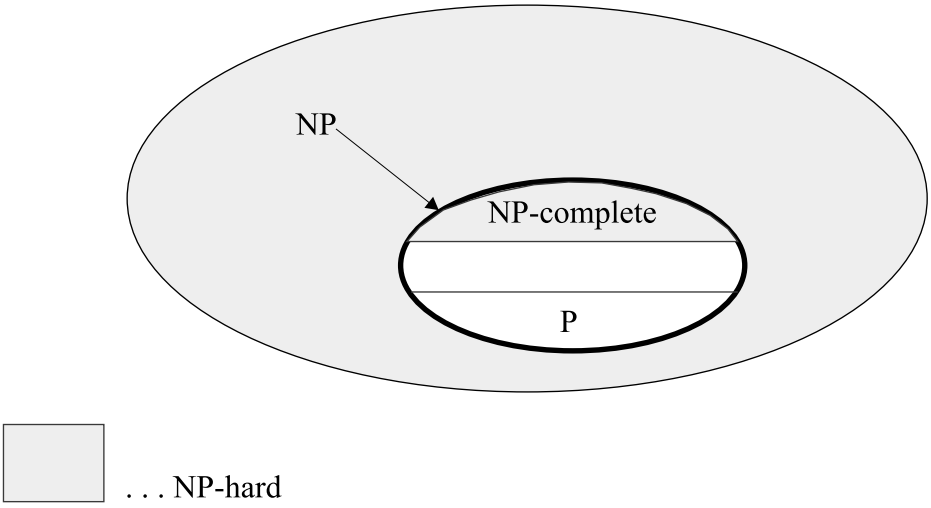

Figure 2.1: The Relation between $\mathcal{P}$ and $\mathcal{NP}$

2.2 The Problem of Complexity

Most problems of computer–aided design tasks for digital circuits are discrete in nature. In other words, it is necessary to solve combinatorial decision and optimization problems. Optimization problems can be reduced to sequences of decision problems. So from now on let us concentrate on decision problems.

A *decision problem* is a problem with a binary–valued solution, i.e., TRUE or FALSE. For instance, such a problem is the formal verification of digital circuits, i.e., the question whether the implementation and the specification of a digital circuit describe the same function.

Some of these decision problems can be solved by algorithms with polynomial complexity, i.e., the number of elementary operations which are repeated in the algorithm is polynomial in the size of some input to the algorithm. This class of problems is known as $\mathcal{P}$ or the class of *tractable* problems [23]. Unfortunatly, it covers only a small part of the problems in the synthesis and optimization of digital circuits. Then, there is another class of problems that could be solved by polynomial algorithms on non–deterministic machines. These are machines which can start with a guess before performing a deterministic algorithm. We call this class $\mathcal{NP}$. Obviously $\mathcal{P} \subseteq \mathcal{NP}$. The question of whether $\mathcal{P} = \mathcal{NP}$ is still unsolved.

However, there is a class of problems for which it has been shown, that if any of these problems can be solved with a polynomial algorithm, then $\mathcal{P} = \mathcal{NP}$. This class of problems is called $\mathcal{NP}$–hard, and the subclass of these problems which is also in $\mathcal{NP}$ is called $\mathcal{NP}$–complete. Figure 2.1 shows the relations among them.

The named property of $\mathcal{NP}$–hard problems implies that if a polynomial algorithm for one of these problems could be found then many other problems for which no polynomial algorithm has been known so far could be solved by polynomial algorithms as well. In other words, it is very unlikely

that there *are* polynomial algorithms for deterministic machines. That is, why these problems are called *intractable*.

Unfortunately, most of the problems that have to be solved in computer–aided design of microelectronic circuits belong to these class of intractable problems. So it is necessary to think about alternative solution possibilities for $\mathcal{NP}$–hard problems. The basic solution idea is as follows: if it is not possible to find an *exact* solution for a problem in reasonable time, i.e., if there is no polynomial algorithm to solve this problem, then try to find polynomial algorithms which are not guaranteed to find the exact solution for all problem instances but are able to provide good approximations to the exact solution for practical applications. Since these kind of algorithms work with heuristics, i.e., problem–solving techniques which are developed based on experiences, they are called *heuristic* algorithms. In other words, for $\mathcal{NP}$–hard problems the effort of research is to find heuristic algorithms that are expected to have polynomial complexity with small exponents and provide good solutions for a lot of practical problem instances.

2.3 Boolean Functions

First, let us consider the *Boolean n–space*. This is the multi–dimensional space spanned by n binary–valued Boolean variables and is denoted by $B^n = \{0,1\}^n$. A point in this space is referred to as a *minterm* and is denoted $c_1 \ldots c_n$ with $c_i \in \{0,1\}$.

A *completely–specified Boolean function* is a mapping between two Boolean spaces. A Boolean function with n input and m output variables is a mapping $f : B^n \longrightarrow B^m$. An *incompletely–specified Boolean function* is defined over a subset of B^n. The minterms where the function is not defined are called *don't care* conditions. If we consider multiple–output functions, i.e., $m > 1$, the *don't care* components may differ for each output of the function. Therefore, incompletely–specified Boolean functions are represented as $f : B^n \longrightarrow \{0,1,*\}^m$, where $*$ represents a *don't care* condition. For each output of f, we can divide its domain into three subsets: the *off–set* includes all minterms for that the function value is 0, the *on–set* (also referred to as the *satisfy set*) those minterms for that the function value is 1, and finally, the *dc–set* contains those minterms for that the function value is $*$. A completely–specified Boolean function can also be described as the set of its on–set minterms.

The Boolean n–space can be graphically represented as a hypercube. Here, a point in B^n is represented by a binary–valued vector of dimension n. Now, when the binary input variables of a Boolean function f are associated with the components of B^n, a point in this Boolean space can be identified by the values of the corresponding variables. A *literal* is a variable x_i or its complement $\bar{x_i}$, and a product of n literals denotes a point (also referred to as a *vertex*) in B^n. Figure 2.2 (a) shows this cube for the three–dimensional Boolean space, B^3 with the three Boolean variables x_1, x_2, and x_3. In this cube of Figure 2.2(a) we can put the description of any Boolean function f with three input variables and one output variable. Let us do this for the example function $f = x_1x_2 + \bar{x_1}x_3$. Let the black dots indicate those points of the Boolean three–space that belongs to the on–set, and

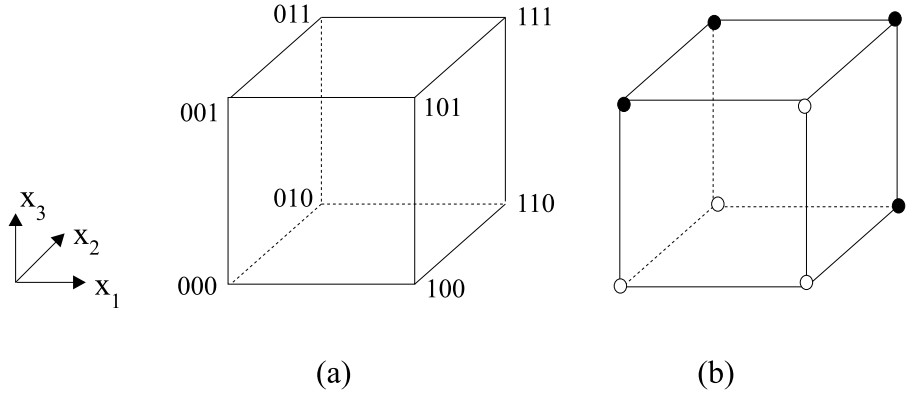

Figure 2.2: The Boolean 3–Space (a) and the Description of $f(x_1, x_2, x_3) = x_1x_2 + \bar{x_1}x_3$ (b)

white dots the points of the off–set of function f, then the graphical description of function f in a cube looks as described in Figure 2.2 (b). The generalization for incompletely–specified Boolean functions and/or functions with more than one output variable is straightforward: use a separate cube for each output, and denote those points of the domain that belongs to the dc–set with an extra sign like $\times$.

The *distance (or Hamming distance)* between two vertices v_1 and v_2 in the Boolean n–space is the number of components that their coordinates differ on. Considering for instance two points of the Boolean three–space, $v_1 = 000$ and $v_2 = 011$, we see that the distance between these two points is 2.

In the following, we will consider completely–specified Boolean functions. We denote the set of all those completely–specified Boolean functions with n input variables and m output variables by $\mathcal{B}_{n,m}$, the set of input variables by $X = [x_1, x_2, \ldots, x_n]$, and the set of output variables by $Y = [y_1, y_2, \ldots, y_m]$. Note that we use these sets as *ordered* sets of variables.

For the sake of simplicity, we will mostly consider a completely–specified Boolean function with n input variables and one output variable. For any of these $f \in \mathcal{B}_{n,1}$, we use the following basic definitions and notations.

The *lexicographical order relation* for Boolean functions is defined as follows: the Boolean function f is lexicographical smaller than a Boolean function $g \in \mathcal{B}_{n,1}$ ($f <_L g$) iff $f(c_1, \ldots, c_n) = 0$ and $g(c_1, \ldots, c_n) = 1$ and the binary vector $(c_1, \ldots, c_n)$ is an encoding of the smallest integer, such that $f(c_1, \ldots, c_n) \neq g(c_1, \ldots, c_n)$[8].

The *satisfy count* of function f is the number of on–set minterms of function f and is denoted as follows:

$$|f| = |\{c_1 \ldots c_n \in B^n : f(c_1, \ldots, c_n) = 1\}|.$$

Definition 2.1 *[23] The* **cofactor** *of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to x_i is $f_{x_i}(x_1, x_2, \ldots, x_i, \ldots, x_n) = f(x_1, x_2, \ldots, 1, \ldots, x_n)$. The* **cofactor** *of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to $\bar{x_i}$ is $f_{\bar{x_i}}(x_1, x_2, \ldots, x_i, \ldots, x_n) = f(x_1, x_2, \ldots, 0, \ldots, x_n)$.*

The cofactor functions f_{x_i} and $f_{\bar{x}_i}$ are considered as functions with the same number of input variables as the function f, i.e., as functions with n input variables.

The logical composition of cofactors of a Boolean function f is denoted as a *subfunction* of f.

The *essential variables* of a Boolean function f are those variables from that f depends on, i.e., x_i is an essential variable of function f iff $f_{x_i} \neq f_{\bar{x}_i}$.

Definition 2.2 *[23] A function* $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *is (positive/negative)* **unate in variable** x_i *if* $f_{x_i}(\mathbf{x}) \geq f_{\bar{x}_i}(\mathbf{x})$ *(*$f_{x_i}(\mathbf{x}) \leq f_{\bar{x}_i}(\mathbf{x})$*) for all possible assignments to the other variables* x_j *with* $j \neq i, j = 1, 2, \ldots, n$*. Otherwise it is* **binate** *in that variable.*

A function is (positive/negative) **unate** *if it is (positive/negative) unate in all essential variables. Otherwise it is binate.*

Besides the cofactor function, there are three other functions that will be used with respect to a Boolean function f: the existential abstraction, the universal abstraction, and the Boolean difference.

Definition 2.3 *[23] The* **existential abstraction (or smoothing)** *of function* $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *with respect to a variable* x_i *is* $\exists_{x_i} f := f_{x_i} + f_{\bar{x}_i}$.

A minterm $c_1 \ldots c_i \ldots c_n$ belongs to the on–set of $\exists_{x_i} f$ if there exists an assignment to x_i which satisfies f. That is, $f(c_1, \ldots, 0, \ldots, c_n) = 1$ or $f(c_1, \ldots, 1, \ldots, c_n) = 1$. In other words, the existential abstraction represents those minterms for which the function is true for at least one assignment to x_i.

Definition 2.4 *[23] The* **universal abstraction (or consensus)** *of function* $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *with respect to a variable* x_i *is* $\forall_{x_i} f := f_{x_i} \cdot f_{\bar{x}_i}$.

A minterm $c_1 \ldots c_i \ldots c_n$ belongs to the on–set of $\forall_{x_i} f$ if all assignments to x_i satisfy f. That is, both $f(c_1, \ldots, 0, \ldots, c_n) = 1$ and $f(c_1, \ldots, 1, \ldots, c_n) = 1$. So the universal abstraction of a function are those minterms for which the function is true for all assignments to x_i.

Definition 2.5 *[23] The* **Boolean difference** *of a function* $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *with respect to a variable* x_i *is* $\frac{\partial f}{\partial x_i} := f_{x_i} \oplus f_{\bar{x}_i}$. $\oplus$ *is the exclusive-or operator.*

The Boolean difference represents the minterms for which f changes when variable x_i changes, i.e., it represents the minterms for which x_i is observable at f. When it is zero, then the function does not depend on x_i.

Now let us define a special subset of Boolean functions from $\mathcal{B}_{n,n}$. We denote with $\mathcal{P}_n \subset \mathcal{B}_{n,n}$ the set of all possible permutations on the set of input variables $X = [x_1, x_2, \ldots, x_n]$, i.e., one-to-one mappings of X onto itself. $\mathcal{P}_n$ is a group [17], and we call it *permutation group* in the sequel.

Let $f \in \mathcal{B}_{n,1}$ be a Boolean function and $\pi \in \mathcal{P}_n$ be a permutation of the input variables in X. Then we define:

$$f \circ \pi(x_1, \ldots, x_i, \ldots, x_n) = f(x_{\pi(1)}, \ldots, x_{\pi(i)}, \ldots, x_{\pi(n)}).$$

Often, we will consider a permutation $\pi \in \mathcal{P}_n$ as a map $\pi : X \longrightarrow X$ defined with $\pi(x_i) = x_{\pi(i)}$. Note that then $\pi_1 \circ \pi_2(x_i) = \pi_2(\pi_1(x_i))$ according to this definition of $\pi_1, \pi_2 \in \mathcal{P}_n$. Furthermore, we describe with

$$\pi(x_1, x_2, \ldots, x_n) = (\pi(x_1), \pi(x_2), \ldots, \pi(x_n))$$

the complete permutation π and denote by

$$\mathbf{1}(x_1, x_2, \ldots, x_n) = (x_1, x_2, \ldots, x_n)$$

the identity.

2.4 Reduced Ordered Binary Decision Diagrams

There are different ways to represent Boolean functions. We are especially interested in the representation by binary decision diagrams.

A *binary decision diagram* (BDD) represents a set of binary–valued decisions resulting in an overall decision which can be TRUE or FALSE. They are represented by trees or rooted, directed, and acyclic graphs, where the decisions are associated with the vertices. A special kind of BDDs are *ordered* BDDs which were introduced by Randal E. Bryant in 1986 [6].

Definition 2.6 *[23] An* **OBDD** *is a rooted directed graph with vertex set V. Each non–leaf vertex has as attributes a pointer $index(v) \in \{0, 1, \ldots, n-1\}$ to an input variable in the set $\{x_1, x_2, \ldots, x_n\}$, and two children $low(v)$ and $high(v) \in V$. A leaf vertex v has as attribute a value $value(v) \in \{0, 1\}$.*

For any vertex pair $\{v, low(v)\}$ (and $\{v, high(v)\}$) such that no vertex is a leaf,
$index(v) < index(low(v))$ (and $index(v) < index(high(v))$).

The restriction on the variable ordering guarantees that the graph is acyclic. Considering a vertex v and its two children $low(v)$ and $high(v)$, we also call $low(v)$ the 0–branch and $high(v)$ the 1–branch of vertex v. We associate a Boolean function with an OBDD as follows.

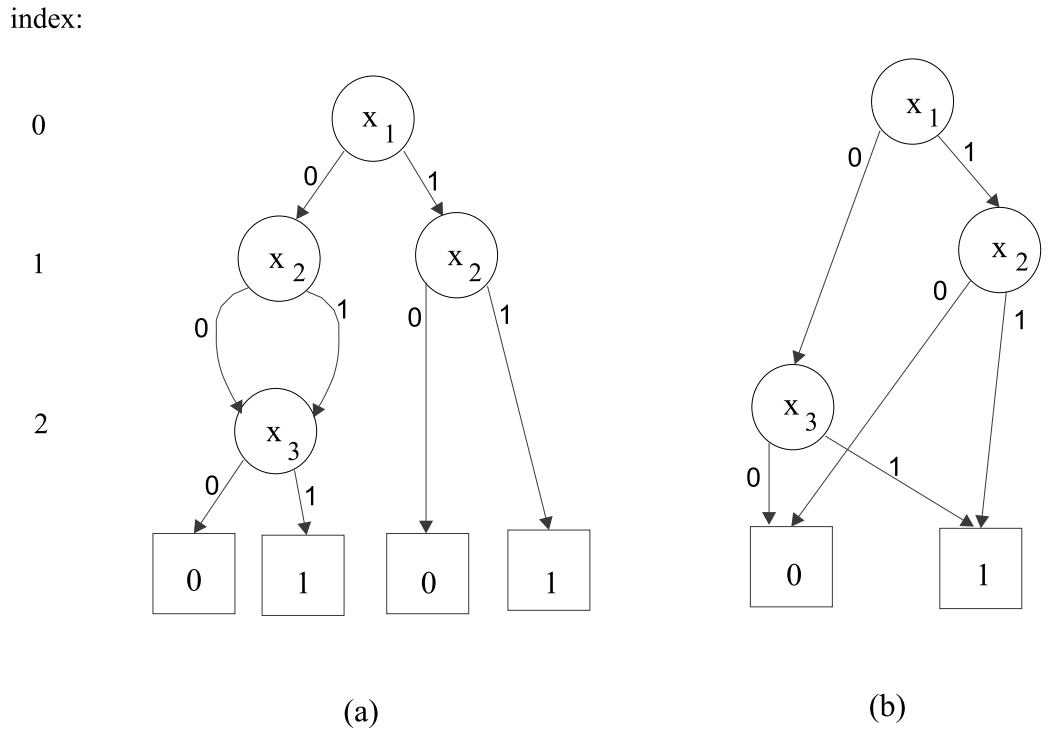

Figure 2.3: Ordered Binary Decision Diagrams for $f(x_1, x_2, x_3) = x_1x_2 + \bar{x_1}x_3$

Definition 2.7 *[23] An* OBDD *with root vertex v denotes a function f^v such that: If v is a leaf with $value(v) = 1$, then $f^v = 1$. If v is a leaf with $value(v) = 0$, then $f^v = 0$. If v is not a leaf and $index(v) = l$, where l is associated with variable x_i, then $f^v = \bar{x_i} \cdot f^{low(v)} + x_i \cdot f^{high(v)}$.*

Figure 2.3 shows two OBDDs of function $f = x_1x_2 + \bar{x_1}x_3$ with the variable ordering x_1, x_2, x_3. I.e., vertex index 0 is associated with variable x_1, index 1 with variable x_2, and index 2 with variable x_3. Changing the variable ordering in an OBDD means to change the relation from the indices to the variables, thus the variables in the ROBDD need to be reordered.

OBDDs have more practical applications than other kinds of BDDs because of two properties. First, OBDDs can be transformed into a unique form for representing a Boolean function $f \in \mathcal{B}_{n,1}$ by reduction of isomorphic subgraphs and redundant vertices.

Definition 2.8 *[23] An* OBDD *is said to be a* **reduced** OBDD *(or* **ROBDD***) if it contains no vertex v with $low(v) = high(v)$, nor any pair $\{u, v\}$ such that the subgraphs rooted in v and in u are isomorphic.*

The OBDD in Figure 2.3 (b) is an ROBDD for function $f = x_1x_2 + \bar{x_1}x_3$. Note that the ROBDD of any Boolean function is unique only with respect to a certain variable ordering, i.e., with resepct to a certain relation of the indices of the vertices of the ROBDD to the variables of the function. In [6], Randal E. Bryant proved that ROBDDs are unique forms for representing Boolean functions.

For this reason, they are especially suited for any problem in the computer–aided design of digital circuits for which equivalence checks between Boolean functions are necessary. Here, equivalence checking reduces to checking if the two unique representations of the two functions are the same.

The other property which has made ROBDDs popular in the circuit design area is that operations on ROBDDs can be done in polynomial time of their size, i.e., the number of vertices [6]. However, it would be a false conclusion that ROBDDs can be used to efficiently solve intractable problems. Unfortunatly, the size of OBDDs may strongly depend on the ordering of the variables. For example, the size of OBDDs for the adder function is very sensitive with respect to the selected variable ordering [23]. While it is polynomial in the number of input variables in the best case, it is exponential in the worst case. Other functions, like the arithmetic multiplier functions, do not have an OBDD representation with polynomial size. They have exponential OBDDs regardless of the selected variable ordering [6].

However, for a lot of common and reasonable practical examples of logic functions, a variable ordering can be found such that the size of their OBDDs is tractable. Many researchers have investigated heuristics to find optimal variable orderings (e.g., [3, 11, 12, 22, 32]). ROBDD packages have been developed and improved that allow an efficient manipulation of Boolean functions with the help of ROBDDs (e.g., [4, 18, 19, 24]). Alltogether, ROBDDs have developed to become the state–of–the–art data structure for representing Boolean functions and have been successfully used in many applications. The development of the ROBDD has led to a big factor in the progression of research in synthesis and verification of digital systems during the last 10 years.

Chapter 3

The Combinational Permutation Equivalence Problem

The background for the combinational permutation equivalence problem is the following task: we need to test the equivalence of two Boolean functions, but we do not know the correspondence between their inputs. In formal logic verification this may be the case while using different tools at various stages of the design process which have their own naming conventions (see Introduction). Here, often ROBDDs are used to check the equivalence of Boolean functions. For our task, the problem is that we cannot use the ROBDD representations directly to check the equivalence. We have to establish a correspondence between the input variables of the two functions before we can apply this method. The most direct way to do this, is to try each possible correspondence. However, it is clear that this cannot be a practical one: for two Boolean functions with n input variables there are $n!$ possible correspondences between the inputs of these two functions. In **Section 3.1**, it is explained why this problem is $\mathcal{NP}$-hard. So we need techniques that use heuristic methods. Several papers have investigated the problem in recent years, for example, [9, 10, 21, 25, 29, 31, 33]. Except [29, 31], all have developed ideas for the ROBDD data structure and basically used the following method: derive signatures for each input variable of a function to uniquely identify this variable.

What is the basic idea behind using signatures? A signature is a description of an input variable which is independent of the permutation of the inputs of a Boolean function f. So, it can be used to identify this variable independent of permutation, i.e., any possible correspondence between the input variables of *two* functions is restricted to a correspondence between variables with the same signature. So, if each variable of a function f had a *unique* signature, then there would be at most one possible correspondence to the variables of any other function. That is why the quality of any signature is characterized by its ability to be a unique identification of a variable and, of course, by its ability to be computed fast. The signatures that have been introduced in the cited papers differ in terms of the quality. Nevertheless, we can say that this concept, in general, is a promising one and successful in a large number of practical cases. In **Section 3.2**, we demonstrate this approach in detail and introduce special signatures that are developed by us, mainly presented in [25]. In

Section 3.3, the utility of these signatures is demontrated on a large set of benchmarks and their quality in comparison to the signatures developed in related papers is discussed.

3.1 Problem Description

Let us describe the actual problem as follows.

Definition 3.1 *Let f and g be two Boolean functions of $\mathcal{B}_{n,1}$ defined over the set of variables $X = [x_1, x_2, \ldots x_n]$.*

The combinational permutation equivalence problem*, P_π, is defined as follows: does there exist a permutation $\pi \in \mathcal{P}_n$ such that*

$$f(X) \equiv (g \circ \pi)(X)$$

is a tautology?

In addition to directly answering this question, the permutation π which establishes the equivalence must also be provided (in the case when the answer is in the affirmative). This does not have to be unique, as illustrated by the following example.

Example 3.1

$$\begin{aligned} X &= [x_1, x_2, x_3, x_4] \\ f &= x_1 x_3(\bar{x}_2 + x_4) + x_2 x_4 \\ g &= x_1 x_2(\bar{x}_3 + x_4) + x_3 x_4 \\ \pi_1(X) &= (x_1, x_3, x_2, x_4) \\ \pi_2(X) &= (x_3, x_1, x_2, x_4) \end{aligned}$$

In this example the functions f and g are obviously *permutation equivalent* with respect to the permutation π_1 of X. However, the permutation π_2 establishes a correspondence for equivalence between the inputs of f and g as well.

Depending on the application, either any such π or all such π may be needed.

This permutation equivalence problem (also referred to as the Boolean matching problem in literature [23]) is intractable, i.e., it is $\mathcal{NP}$–hard, because the complement of the tautology problem belongs to the $\mathcal{NP}$–complete class of problems [15]. In other words, we need to find efficient heuristics in order to be able to handle the permutation equivalence problem.

3.2 Signatures

Let us recall the problem. We want to check if two Boolean functions are equivalent independent of the permutation of their inputs. Therefore, we need to establish a correspondence between the input variables of these two functions. Signatures are the basic components of the approach which we use to handle this problem.

3.2.1 Definition

In this context, a signature can be described as follows:

Definition 3.2 *Let U be an ordered set, $(U, \leq)$.*
A mapping $s : \mathcal{B}_{n,1} \times X \longrightarrow U$ is a **signature function** *iff:*

$$\forall f \in \mathcal{B}_{n,1} \forall \pi \in \mathcal{P}_n \text{ and } \forall x_i, x_j \in X : \pi(x_i) = x_j \Longrightarrow s(f, x_i) = s(f \circ \pi, x_j).$$

We call $s(f, x_i)$ a **signature** *for the input variable x_i of function f.*

That means that a signature for an input variable x_i of a Boolean function $f \in \mathcal{B}_{n,1}$ is a description of x_i which provides special information about this variable in terms of f. Furthermore, it is very important that this information is independent of any permutation of the inputs of f, i.e., if a permutation π maps the variable x_i to x_j, then the signature of x_i in f must be the same as the signature of x_j in $f \circ \pi$.

Why the property of a signature to be an element of an ordered set is useful, we will demonstrate in the following section. Note, that this property is not a necessary one, but it makes things easier and the variety of signature functions that can be developed increases.

In the following, let $\mathcal{S}_n$ be the set of all signature functions for the input variables of a Boolean function $f \in \mathcal{B}_{n,1}$.

3.2.2 Solution Paradigm

We can use a signature to identify an input variable x_i independent of permutation and to establish a correspondence between this variable x_i of f with a variable x_j of any other Boolean function $g \in \mathcal{B}_{n,1}$. In order to establish a correspondence between these two variables, variable x_i of f must have the same signature as variable x_j of g.

The main idea of this approach is clear: if we are able to compute a unique signature for each input variable of f, then the correspondence problem is solved – there is only one or no possible correspondence for permutation equivalence of function f with *any* other function g. If we find for each variable of f a variable of g which has the same signature, then we have established a correspondence. Otherwise, we know immediately that these two functions are not permutation equivalent. The main problem that arises in this paradigm is when more than one variable of a

function f has the same signature, so that it is not possible to distinguish between these variables, i.e., there is no *unique* correspondence that can be established with the inputs of any other function. We call a group of such variables an *aliasing group*. Suppose there is just one aliasing group of inputs of a function f after applying certain signatures. If the size of this group is k, then there are still $k!$ correspondence possibilities to test between the input variables of f and the input variables of any other function g.

However, before we go into details of how signatures are generated and what the practical experiences of using signatures are, let us illustrate this solution paradigm on an example of two Boolean functions, f and g with four input variables, $X = [x_1, x_2, x_3, x_4]$.

First, we compute a signature $s \in \mathcal{S}_4$ for each of the variables $x_1, \ldots, x_4$ with respect to f. In this context, a signature for a variable x_i is a value or a list of values which provides special information about x_i in terms of f. Remember, since we want to use those signatures to identify each input variable independent of the permutation of all input variables in the function, each signature for a variable should be independent of the permutation of the other input variables.

Let us assume a signature is the assignment of an integer to each variable. For purpose of this example, let us assume that

$$\begin{aligned} s(f, x_1) &= 3, \\ s(f, x_2) &= 2, \\ s(f, x_3) &= 1, \\ s(f, x_4) &= 2. \end{aligned}$$

The signature list for function f, $\mathcal{L}(f)$, is an ordered list of the signatures of the variables in X for f. Thus,

$$\mathcal{L}(f) = [3, 2, 1, 2].$$

Now, given the Boolean function g we compute $\mathcal{L}(g)$. Since the signature is a permutation independent property of the variable with respect to a function, a necessary condition for f and g to be permutation equivalent is that their signature lists have the same elements. This can be easily checked by sorting and comparing the two lists. If the lists do not contain the same elements, the inequality of the two functions is already established. Note that in this way, the lists of signatures can be considered as detailed output variable filters which were introduced by Lai, Sastry and Pedram in [21]. In other words, the sorted lists of signatures for the input variables could be used to identify the *output* variables of Boolean functions with more than one output independent of permutation. However, this evaluation is out of the scope of this thesis. So let us come back to P_π.

Let us assume that in our example the lists contain the same elements and let

$$\mathcal{L}(g) = [2, 2, 3, 1].$$

Based on the signatures, we can directly establish that any permutation π for which the two functions could be permutation equivalent, i.e., $f = g \circ \pi$, has to satisfy $\pi(x_3) = x_1$ and $\pi(x_4) = x_3$.

Thus, the correspondence of the variables has been partially established. However, $\pi(x_1)$ could either be x_2 or x_4 and $\pi(x_2)$ is then the remaining variable. In this case aliasing has occurred since x_1 and x_2 have the same signature (in g).

There are two reasons for the existence of these aliasing groups: either the used signature function was not strong enough to distinguish between the two variables or the variables have some special properties that make it impossible to distinguish between them using signatures. In the first case, we can use another signature function. This is the topic of **Section 3.2.3**. The second case is more complicated: we need to find out that there is such a property (so that we do not apply one signature after the other without any hope of success), and we need to apply a non–exhaustive technique to handle those cases. In **Chapter 4**, we investigate this problem and show some ways to overcome these limits of using signatures.

However, first let us come back to those cases where signatures have been proven to be very helpful. Given signatures, they may be used in two possible ways. They may be directly used to establish the correspondence, as suggested in the above example, or they may be used to establish a possibly unique (in the absence of aliasing) ordering of variables for P_π. Assuming there is no aliasing, the variables can be sorted uniquely using the signature as a key, because signatures are elements of an ordered set (see the definition). This establishes a unique order for the variables, and this can be used for renaming the variables in a unique way or for constructing an ROBDD using this order. A method to restructure an ROBDD from a given variable ordering to any other variable ordering is proposed in [37]. This ROBDD will be a permutation independent unique representation for this function. In the case where there is aliasing, instead of a single unique order of variables, there will be a set of possible orders corresponding to all possible sorts of the list of variables. Note that this set is unique for a given signature function. However, the cardinality of this set depends on the amount of aliasing that occurs.

3.2.3 Special Signatures

In the following we introduce three kinds of signatures for an input variable x_i of a Boolean function $f \in \mathcal{B}_{n,1}$ for that we investigated to be very successful in practical experiences.

3.2.3.1 Satisfy Count Signatures

The satisfy set of a Boolean function $g \in \mathcal{B}_{n,1}$ is the number of vertices of the Boolean n–space for which the function value is 1. Then, the satisfy count of g is the number of minterms in the satisfy set of this function:

$$|g| = |\{\mathbf{x} \in \{0,1\}^n : g(\mathbf{x}) = 1\}|.$$

This number does not dependent on any permutation of the input variables of g. Let us examine this fact on the example of the simple multiplexer function $f(x_1, x_2, x_3) = x_1x_2 + \bar{x}_1x_3$. Consider Figure 3.1 (a). This figure shows the cube representation of the multiplexer function again. Black dots indicate minterms where this function is equal to 1, white dots indicate the minterms where

the function is equal to 0. Computing the satisfy count of function f in the cube means to count the number of black dots. Obviously, this counting process is permutation independent: we are just interested in the *number* of black dots and not in their locations.

Furthermore, it is easy to compute: assuming the availability of g in an ROBDD it can be done in time linear in the size of this ROBDD [6].

Now, if we consider as function g any function which provides special information of a variable x_i in f, we get the first class of signatures. We call them *satisfy count signatures*.

One of those subfunctions of f is the positive phase cofactor of f with respect to an input variable x_i:

$$g(x_1, x_2, \ldots, x_n) = f_{x_i}(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n).$$

The satisfy count of this function is a simple but very powerful signature for variable x_i in function f (see the experimental results of this chapter). Moreover, it can be computed on the ROBDD of the original function f without even computing f_{x_i} explicitly. This counting is done exactly in the same manner as computing the satisfy count of a function f as suggested by Bryant [6], except that contributions along the paths with $x_i = 0$ are ignored. We call this signature the **cofactor satisfy count signature**. Aliasing occurs when two variables have the same satisfy count for the positive phase cofactors. Note, that this signature has been independently used in other papers as well. For example, in [21] Lai, Sastry and Pedram have used this as well as one of their input variable filters.

Suppose, we use the cofactor signature to distinguish between the input variables of a Boolean function and aliasing occurs for some of these variables. What other subfunctions g can provide new information for a variable x_i in function f?

There are a few of candidates for g:

- the cofactor function with respect to the negative phase of x_i:

$$f_{\bar{x}_i}(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n),$$

- the existential abstraction of f with respect to x_i:

$$\exists_{x_i} f = f_{x_i} + f_{\bar{x}_i},$$

- the universal abstraction of f with respect to x_i:

$$\forall_{x_i} f = f_{x_i} \cdot f_{\bar{x}_i},$$

  as well as

- the Boolean difference of f with respect to x_i:

$$\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\bar{x}_i}.$$

However, examining the information which is provided by the satisfy counts of these functions, we observe the following.

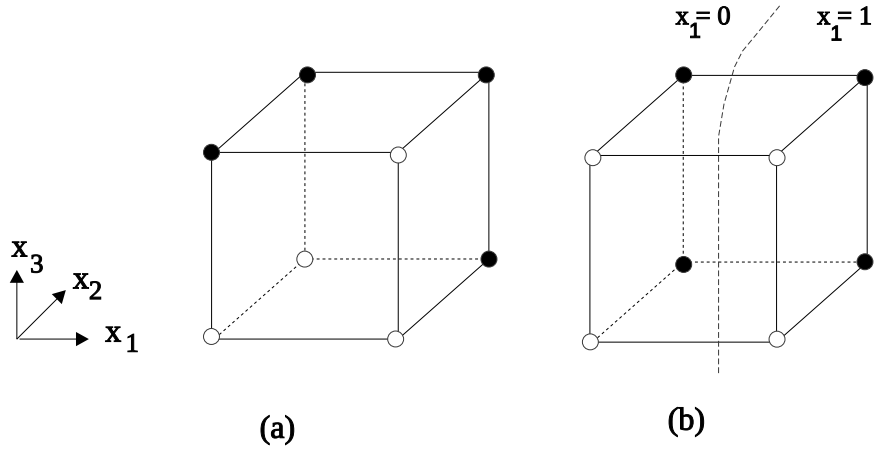

Figure 3.1: Cube Representation of $f(x_1, x_2, x_3) = x_1x_2 + \bar{x}_1x_3$ (a) and $f_{x_1}(x_1, x_2, x_3) = x_2$ (b)

The satisfy count for the negative phase cofactor does not provide any additional information based on the following property.

Property 3.1

$$2 \cdot |f| = |f_{x_i}| + |f_{\bar{x_i}}|$$

Proof: Consider the satisfy count of the Boolean function f.

$$\begin{aligned} |f| &= |x_i \cdot f_{x_i} + \bar{x_i} \cdot f_{\bar{x_i}}| \\ &= |x_i \cdot f_{x_i}| + |\bar{x_i} \cdot f_{\bar{x_i}}| - |x_i \cdot f_{x_i} \cdot \bar{x_i} \cdot f_{\bar{x_i}}| \\ &= |x_i \cdot f_{x_i}| + |\bar{x_i} \cdot f_{\bar{x_i}}| - 0 \\ &= \frac{1}{2} \cdot |f_{x_i}| + \frac{1}{2} \cdot |f_{\bar{x_i}}|, \end{aligned}$$

because f_{x_i} (and $f_{\bar{x_i}}$) is considered as Boolean function in n instead of $n-1$ input variables, i.e., while with $|x_i \cdot f_{x_i}|$ (and $|\bar{x_i} \cdot f_{\bar{x_i}}|$) just the positive (and negative) half of the Boolean n-space with respect to x_i is considered, $|f_{x_i}|$ (and $|f_{\bar{x_i}}|$) includes the complete Boolean n-space. Here, each function value of f_{x_i} (and $f_{\bar{x_i}}$) in the negative half of the Boolean n–space with respect to x_i is equal to the corresponding function value in the positive half.

To illustrate this, let us consider an example. Figure 3.1 shows the cube representations of function $f = x_1x_2 + \bar{x_1}x_3$ and its cofactor function with respect to x_1, $f_{x_1} = x_2$. In this figure it is easy to see that the cofactor function f_{x_1} is constructed by mapping the positive half of the Boolean three–space with respect to x_1 to the negative half. □

From Property 3.1 it follows that if for two input variables x_i and x_j of f $|f_{x_i}| = |f_{x_j}|$, then $|f_{\bar{x_i}}| = |f_{\bar{x_j}}|$.

Considering the satisfy counts of the other three subfunctions of f, namely the existential abstraction, the universal abstraction, and the Boolean difference, we can observe similiar relationships.

If one of them has been used already no further advantage is to be gained by using the other two. For example, if we include the satisfy count of the existential abstraction, then, as demonstrated by the following two properties, the satisfy counts for the other two do not contribute any additional information.

Property 3.2

$$2 \cdot |f| = |\exists_{x_i} f| + |\forall_{x_i} f|$$

Proof: Let us consider the satisfy count of the existential abstraction of function f with respect to x_i.

$$\begin{aligned} |\exists_{x_i} f| &= |f_{x_i} + f_{\bar{x}_i}| \\ &= |f_{x_i}| + |f_{\bar{x}_i}| - |f_{x_i} \cdot f_{\bar{x}_i}| \\ &= 2 \cdot |f| - |\forall_{x_i} f| \end{aligned}$$

This gives us the relationship among the satisfy counts of function f, and the existential and universal abstraction of function f with respect to input variable x_i. □

Property 3.3

$$2 \cdot |f| = 2 \cdot |\exists_{x_i} f| - |\frac{\partial f}{\partial x_i}|$$

Proof: First, let us consider the existential abstraction of function f with respect to x_i again. Here, we can establish the following relationship among existential abstraction, universal abstraction, and Boolean difference of f with respect to x_i.

$$\begin{aligned} \exists_{x_i} f &= f_{x_i} + f_{\bar{x}_i} \\ &= f_{x_i} \cdot (f_{\bar{x}_i} + \bar{f}_{\bar{x}_i}) + f_{\bar{x}_i} \cdot (\bar{f}_{x_i} + f_{x_i}) \\ &= f_{x_i} \cdot f_{\bar{x}_i} + f_{x_i} \cdot \bar{f}_{\bar{x}_i} + f_{\bar{x}_i} \cdot \bar{f}_{x_i} + f_{\bar{x}_i} \cdot f_{x_i} \\ &= \forall_{x_i} f + \frac{\partial f}{\partial x_i} + \forall_{x_i} f \\ &= \forall_{x_i} f + \frac{\partial f}{\partial x_i} \end{aligned}$$

Now, we can establish the relationship among the satisfy counts of function f, its existential abstraction, and its Boolean difference with respect to x_i.

$$\begin{aligned} |\exists_{x_i} f| &= |\forall_{x_i} f + \frac{\partial f}{\partial x_i}| \\ &= |\forall_{x_i} f| + |\frac{\partial f}{\partial x_i}| - |\forall_{x_i} f \cdot \frac{\partial f}{\partial x_i}| \\ &= |\forall_{x_i} f| + |\frac{\partial f}{\partial x_i}| - 0 \\ &= 2 \cdot |f| - |\exists_{x_i} f| + |\frac{\partial f}{\partial x_i}| \end{aligned}$$

P_π	$\lvert f_{x_i}\rvert$	$\lvert \exists_{x_i} f\rvert$
$i=1$	4	6
$i=2$	6	6
$i=3$	6	6

Table 3.1: Satisfy Count Signatures for $f(x_1, x_2, x_3) = x_1x_2 + \bar{x}_1x_3$

This gives us the desired relationship. □

Using Property 3.2 and Property 3.3, we know that if $|\exists_{x_i} f| = |\exists_{x_j} f|$, then $|\forall_{x_i} f| = |\forall_{x_j} f|$ as well as $|\frac{\partial f}{\partial x_i}| = |\frac{\partial f}{\partial x_j}|$ for any two variables x_i and x_j.

It is interesting to compare the information that the phase cofactors and the existential abstraction (equivalently the universal abstraction or the Boolean difference) convey. The phase cofactors provide information as to what is happening in each half subspace corresponding to $x_i = 0$ and $x_i = 1$. The existential abstraction provides information as to the relationship across the two half subspaces. Between the two of them, they tell the complete story as to what is happening in the two halves, as well as how the two halves interact with each other.

However, there are several functions for that we cannot avoid aliasing using the satisfy count of the positive phase cofactor and that of the existential abstraction (referred to as **existential abstraction satisfy count signature**). Consider the simple example function in Table 3.1. For this multiplexer function, $f = x_1x_2 + \bar{x}_1x_3$, we cannot break the tie for x_2 and x_3 using the cofactor signature as well as the existential abstraction signature.

Also the disadvantage of using $|\exists_{x_i} f|$, $|\forall_{x_i} f|$, or $|\frac{\partial f}{\partial x_i}|$ as a signature is that this will need to construct the ROBDD for the function $\exists_{x_i} f$, $\forall_{x_i} f$, or $\frac{\partial f}{\partial x_i}$ first, which may take time quadratic in the size of the ROBDD of f [6].

We now examine what other information about a variable can be extracted from the ROBDD that is independent of the permutation of all input variables.

3.2.3.2 Breakup Signatures

First, let us consider the satisfy count of a Boolean function $g \in \mathcal{B}_{n,1}$ again. The idea is to break it into $n+1$ special, permutation independent components:

$$s = [|g^0|, |g^1|, \ldots, |g^n|],$$

where $|g^k|$ is the number of minterms in the satisfy set of g that have distance k from the origin $\mathcal{O} = [x_0 = 0, x_1 = 0, \ldots, x_n = 0]$ of the Boolean n-space ($k = 0, 1, \ldots, n$). This is a good characteristic of g and it is independent of the permutation of its input variables. So we can use it to define several new signature functions. We call this kind of signatures *breakup signatures*.

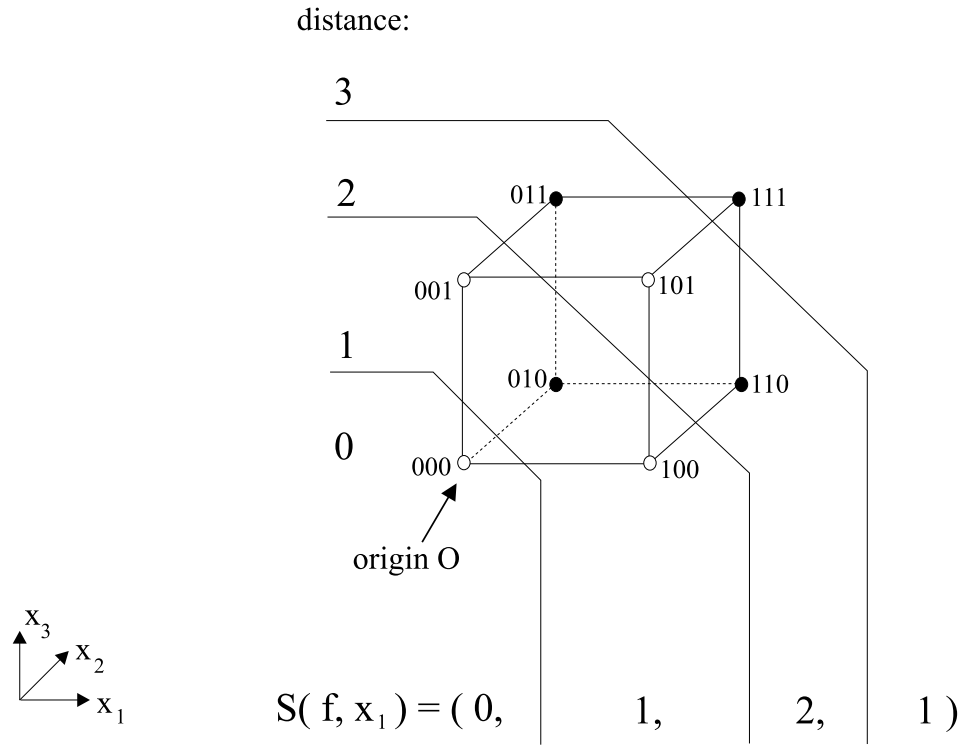

Figure 3.2: Breakup Signature for $f_{x_1}(x_1, x_2, x_3) = x_2$ with $O = [0, 0, 0]$

First, we can use special subfunctions of f like f_{x_i} or $\exists_{x_i} f$ — similiar to the satisfy count signatures. However, we also can use any other vertex than the origin of the Boolean n–space which may be dependent on x_i but independent of the permutation of the other input variables in f in order to characterize the different levels: for example, the vertex $\mathcal{O} = [x_1 = 0, x_2 = 0, \ldots, x_i = 1, \ldots, x_n = 0]$.

We have investigated the first way with success (see the experimental results in Section 3.3), and have used the cofactor function to create the **cofactor breakup signature** and the existential abstraction to create the **existential abstraction breakup signature**.

An example for a cofactor breakup signature is shown in Figure 3.2. Again, we consider the function $f = x_1x_2 + \bar{x}_1x_3$ and the variable x_1. This time, Figure 3.2 shows a cube representation for the positive phase cofactor of f with respect to x_1, $f_{x_1} = x_2$. Again, black dots indicate vertices where this function is equal to 1, white dots indicate the vertices where the function is equal to 0. Using the origin of this cube $[x_1 = 0, x_2 = 0, x_3 = 0]$ and the distance, the vertices can be separated in 4 different levels as shown in the figure. Then, the kth component of the actual breakup signature is the number of black dots in the kth level (which is determined by distance k from the origin). In this figure, the necessary permutation independence is easy to see: the origin is permutation independent because each of the three variables is equal to 0, the distance is permutation independent because it groups all vertices with a constant number of variables equal to 1 in it together independent of the order of these variables themselves, and the counting step itself is permutation independent because just the number of black dots (or minterms of the satisfy

set of f_{x_1}) in a level is interesting, not their places in this cube level. Similarly, we can compute the breakup signature for the variables x_2 and x_3:

$$\begin{aligned} s(f, x_1) &= (0, 1, 2, 1), \\ s(f, x_2) &= (0, 2, 3, 1), \\ s(f, x_3) &= (1, 2, 2, 1). \end{aligned}$$

For our example function, the cofactor breakup signature is more powerful than the cofactor signature and the existential abstraction signature: it helps to distinguish between the variables x_2 and x_3 as well.

The fact that the breakup signature is more powerful than the satisfy count signatures, can be examined by another consideration. In [21] the authors use a simple method for analytically comparing the effectiveness of signatures (they call them filters). Given a signature function $s : \mathcal{B}_{n,1} \times X \longrightarrow U$, a good and simple measure of the effectiveness of a signature function is the cardinality of the co–domain of s, $|s(B_{n,1} \times X)|$. In other words, the more different values a signature may have, the larger is the probability that it may uniquely identify a function. We examine this further for the signatures we have developed.

The satisfy count of a Boolean function f with n input variables may assume 2^n different values. Now, let us examine the breakup signature for any function with n input variables. The value of the kth signature component ($k = 0, \ldots, n$) may be $0, 1, \ldots, l$ where l equals the number of vertices in the Boolean n–space with distance k from the origin. Thus, there are $l + 1$ different possibilities for this component and $l = \binom{n}{k}$. Finally, the number of different breakup signatures that we can get for a function with n input variables is $\prod_{k=0}^{n}[\binom{n}{k} + 1]$. With Property 3.4 we immediately see that the breakup signature for any Boolean function with n input variables is more powerful than satisfy count signatures of it.

Property 3.4

$$2^n < \prod_{k=0}^{n}[\binom{n}{k} + 1]$$

Proof: It is $\binom{n}{k} + 1 \geq 2$ for each $k = 0, 1, \ldots, n$. From this fact it follows that

$$\prod_{k=0}^{n}[\binom{n}{k} + 1] \geq 2^{n+1} > 2^n,$$

and we are done. □

In the last part of describing breakup signatures, we give an idea of how to compute them for a given function g and a vertex $\mathcal{O} = [c_1 \dots c_i \dots c_n]$ of the Boolean n–space on an ROBDD G for g. We use a similiar counting technique as that used to compute the satisfy count of a Boolean function [6]. The difference is that we need to compute a vector of values instead of a single value. We call this vector *br_sig_vec* and create it in a bottom-up procedure on the ROBDD G of g.

The theoretical background of this calculation can be explained as follows. At each vertex v of the ROBDD G, the algorithm computes the breakup signature of the subfunction which is represented by the subROBDD with root vertex v. Say this function is $g' := g^v$. If v is a non–terminal node, then we assume that g' depends on $n' = n - j$ variables, where $j \in \{0, 1, \dots, n-1\}$ is the index of node v. If v is a terminal node, then g' is independent of any variable, i.e., $n' = 0$. For each vertex, the value vector *br_sig_vec* has $n' + 1$ entries.

The number of *all* minterms in the Boolean n'–space that have distance $k = 1, 2 \dots, (n'-1)$ from the actual origin is

$$\binom{n'}{k} = \binom{n'-1}{k} + \binom{n'-1}{k-1}.$$

This equation can be used as follows. Consider g', the subfunction which is represented by the subROBDD of G with root vertex v again. Let us consider the terminal case. If $g' = 1$ or $g' = 0$, then there is exactly one entry in *br_sig_vec*, and this is $|g'^0| = 1$ for $g' = 1$ and $|g'^0| = 0$ for $g' = 0$. Thus we guarantee that only the minterms of the on–set are counted. In the non–terminal case let x_i be the variable which is represented by the root vertex v. Then we compute the kth entry of *br_sig_vec* of function g' $(k = 0, 1, \dots, n')$ as follows:

1. if $c_i = 0$ in the origin $\mathcal{O}$:

$$|g'^k| = \begin{cases} |g'^0_{\bar{x}_i}| & : \quad k = 0 \\ |g'^{n'-1}_{x_i}| & : \quad k = n' \\ |g'^k_{\bar{x}_i}| + |g'^{k-1}_{x_i}| & : \quad k = 1 \dots n'-1 \end{cases}$$

2. if $c_i = 1$:

$$|g'^k| = \begin{cases} |g'^0_{x_i}| & : \quad k = 0 \\ |g'^{n'-1}_{\bar{x}_i}| & : \quad k = n' \\ |g'^{k-1}_{\bar{x}_i}| + |g'^k_{x_i}| & : \quad k = 1 \dots n'-1 \end{cases}$$

This means regarding to the computation on the ROBDD: the kth entry of the vector *br_sig_vec* of any non-terminal vertex v is computed by adding two special entries of the value vectors of its children — one entry from the 0-branch child and one from the 1–branch child. Which ones are added depends on the phase in which the variable x_i represented by v occurs in the origin $\mathcal{O} = [c_1 \dots c_i \dots c_n]$. The recursion terminates at the leaves where the value vector of the leaf vertex with value 1 contains the single entry 1 and the leaf vertex with value 0 contains the single entry 0. In this way, we make sure that only the minterms of the ON–set are counted in *br_sig_vec*.

The complete algorithm, called *breakup_sig*, is described in pseudo–code in Figure 3.3. In this algorithm, *G.root* is the node which occurs at the top of the ROBDD G. *G.root.high* and *G.root.low*

```
int*
function breakup_sig ( G , o, l )
BDD      G;   co: actual ROBDD
int*     o;   co: point in Bool. n-space
int      l;   co: level of recursion
begin
  if ( br_sig_vec of G.root computed )
   return br_sig_vec;
  if ( G = bdd_zero ) return 0;
  if ( G = bdd_one ) return 1;
  br_sig_vec_h ← breakup_sig (G.high, o, l + 1);
  br_sig_vec_l ← breakup_sig (G.low, o, l + 1);
  create a new value vector br_sig_vec for G.root;
  root ← G.root.index;
  low ← G.low.root.index;
  high ← G.high.root.index;
  co: ....index ∈ {0, 1, ..., n − 1}
  if ( o[root] = 0 ) {
    br_sig_vec[0] ← br_sig_vec_l[0];
    br_sig_vec[n − root] ← br_sig_vec_h[n − high];
  }
  else {
    br_sig_vec[0] ← br_sig_vec_h[0];
    br_sig_vec[n − root] ← br_sig_vec_l[n − low];
  }
  for i from i=1 to n − root − 1 do
   if ( o[root] = 0 )
    br_sig_vec[i] ←
      cal_sig_val (br_sig_vec_l, root, low, i) + cal_sig_val (br_sig_vec_h, root, high, i−1);
   else
    br_sig_vec[i] ←
      cal_sig_val (br_sig_vec_l, root, low, i−1) + cal_sig_val (br_sig_vec_h, root, high, i);
  if ( (l = 0) && (root ≠ 0) ) co: !! see comment in the text
   complete br_sig_vec to |g^{k,o}|, k = 0, ..., n;
  return br_sig_vec;
end breakup_sig;
```

Algorithm *breakup_sig*

Figure 3.3: Pseudo–Code for *breakup_sig(G, o, l)*

are the subBDDs of G describing the 1–branch and the 0–branch of $G.root$, respectively, o is the vertex which we compute the distances from and l indicates the level of the recursion beginning with 0 for the top level.

The algorithm *breakup_sig* works as follows. It considers the function represented by the ROBDD G with root index $G.root.index$ as a function in $(n - G.root.index)$ variables and calculates the values $br\_sig\_vec[0] = |g^{0,o}|, \ldots, br\_sig\_vec[n - G.root.index] = |g^{n-G.root.index,o}|$. In other words, the vector *br_sig_vec* which is created by *breakup_sig* includes the breakup signature of the subfunction

```
int
function cal_sig_val ( br_sig_vec , r, child_i, ni )
int*  br_sig_vec;   co: value vector of the actual child
int   r,            co: node index of the root
      child_i,      co: node index of the actual child
      ni;           co: to compute actual br_sig_vec[ni]
begin
  if ( child_i = r + 1 )
   co: no gap to consider
   return br_sig_vec[ni];
  l ← child_i − r − 1;
  co: l is the number of levels between
        root node and child node
  v ← 0;
  co: v is the return value
  for i from i= min(l, ni) to 0 by −1 do {
    if ( ni − i > |br_sig_vec| )
     co: there are no minterms with distance ni − i and larger
     break;
    v ← v + (l choose i) · br_sig_vec[ni − i];
    co: (l choose i) is the number of minterms with i "1"'s in the l levels
        br_sig_vec[ni − i] is the number of minterms with distance ni − i
        in the subfunction of the actual child
  }
  return v;
end br_sig_vec;
```

Algorithm *cal_sig_val*

Figure 3.4: Pseudo–Code for *cal_sig_val(br_sig_vec, r, child_i, ni)*

represented by the ROBDD of the actual root vertex (which need not be the root vertex of the function). For leaf vertices, it returns 1 for the vertex marked with value 1, and 0 for the vertex marked with value 0. For a non–leaf root vertex the value vector is computed using the vectors of its children and taking care of the phase in which the variable represented by this actual vertex occurs in o.

If there exists a gap between the root index and the index of the actual child of G, the number of points added to a value $br\_sig\_vec[i]$ by this child is not a single value of the vector of the child but a combination of different values. This happens since we need to consider all possible value combinations of the variables filling the gap. Let l be the number of levels between the root and the child index, let *br_sig_vec* be the value vector of the actual child, and let g' be the subfunction of function g repesented by the actual child. The function g' depends on n' variables.

The number of minterms with distance k in g' considered as subfunction in $n'+l$ instead of n' variables is equal to:

$$\sum_{i=0}^{\min(k,l)} (\binom{l}{i} \cdot br\_sig\_vec[k-i]).$$

Consider the ith term of this sum. Here, $\binom{l}{i}$ is the number of minterms with i components set to 1 in the l levels. Multiplying this number with the number of minterms with distance $k-i$ in g' considered as subfunction with n' variables provides one part of the complete number of minterms of g' with distance k considered as subfunction in $n'+l$ variables. Adding all of such possibilities for distance k provides the complete number of ON–set minterms of g' considered in $n'+l$ variables. This is carried out by procedure *cal_sig_val* shown in Figure 3.4 and can be done in time $O(l)$. Thus *cal_sig_val* has worst-case complexity $O(n)$.

If the root index is not 0, then the idea of algorithm *cal_sig_val* has to be used at the top level of *breakup_sig* as well to assign the value vector of the root vertex to the desired signature (see the if–statement commented with '!!' in the pseudo-code of Figure 3.3). The computation of a vector *br_sig_vec* of one vertex is $O(n^2)$, thus the complexity of *breakup_sig* is $O(|V| \cdot n^2)$, where $|V|$ is the number of vertices in G. However, in average cases it is more like $O(|V| \cdot n)$ since the worst case runtime for *cal_sig_val* occurs seldom.

For a given Boolean function f and an input variable x_i of f, there are mainly two functions that are useful for computing a breakup signature for this input variable with respect to f. That is the positve phase cofactor of f with respect to x_i and the existential abstraction (see Section 3.2.3). Here, the use of the cofactor function is preferred because it is not necessary to create the ROBDD of a cofactor function f_{x_i} before computing the breakup signature. Instead of computing the vector *br_sig_vec* of each vertex which is associated with x_i by using the vectors *br_sig_vec* of both children, we only have to assign the vector *br_sig_vec* of the 1–branch BDD using the idea of procedure *cal_sig_val*.

The complexity of this procedure is $O(|V| \cdot n^2)$, where $|V|$ is the number of vertices in the ROBDD of g. We see that the higher effectiveness of the breakup signatures in comparison to that one of the satisfy count signatures must be paid with a higher complexity of the procedure which computes the breakup signature. (The complexity to compute the satisfy count signatures is linear in the number of ROBDD vertices of the actual function.) So it will probably be useful to apply satisfy count signatures before applying breakup signatures. Further details are provided in Section 3.3.

3.2.3.3 Function Signatures

The last category of signatures for input variables that we want to introduce here are *function signatures*. This kind of signatures does not represent special values or vectors of values, like the satisfy count signatures and the breakup signatures, but special Boolean functions or vectors of Boolean functions. These signatures can be applied to try to distinguish between input variables of a Boolean function f after some variables have been uniquely identified. Thus we can explicitly use

the fact that we can uniquely identify some input variables by permutation independent information computed on f. In other words, function signatures are special Boolean subfunctions of f that depend on those variables only that have been uniquely identified by other signatures before (like the satisfy count signature or the breakup signature).

How does this work? Let us consider a Boolean function $g \in \mathcal{B}_{n,1}$ with the sequel of input variables $X = [x_1, x_2, \ldots, x_n]$, and suppose that we have applied the introduced signatures to the variables and that there are k variables with aliasing. For the sake of simplicity say, that these are the variables $x_1, x_2, \ldots, x_k$. That means, $x_1, x_2, \ldots, x_k$ are the variables which we still have to identify uniquely. As described, we use special subfunctions of g that only depend on the variables $x_{k+1}, x_{k+2}, \ldots, x_n$ for this purpose. These are the variables that have had a unique description by the other signatures already. Now, if we use these signatures to order the variables $x_{k+1}, x_{k+2}, \ldots, x_n$, then we get a unique and permutation independent order of these $n-k$ variables. Let $\pi_{can} \in P_{n-k}$ be the permutation which constructs this unique order, and $g' \in \mathcal{B}_{n-k,1}$ be a subfunction of g which is constructed by permutation independent operations on g and only depends on the variables $x_{k+1}, x_{k+2}, \ldots, x_n$. Then, the Boolean function

$$\hat{g} = g' \circ \pi_{can}$$

is obviously a permutation independent information for function g.

Now, let us apply this in order to define function signatures for an input variable x_i in a Boolean function $f \in \mathcal{B}_{n,1}$. We need to construct Boolean functions that have the properties of the function $\hat{g}$ and in addition can provide some special information about x_i with respect to function f. The idea, which immediately suggests itself is to use cofactor functions and existential abstraction functions (as well as the universal abstraction and the Boolean difference) once again. This time, we need to construct cofactor (or similar) functions evaluated to all variables $x_1, x_2, \ldots, x_k$ that still have aliasing. The so constructed subfunctions of f only depend on the variables $x_{k+1}, x_{k+2}, \ldots, x_n$. So, each of them is a typical function g' as described above.

Let us consider the cofactor functions evaluated with respect to the variables $x_1, x_2, \ldots, x_k$. Let x_i be one of these variables, then the two cofactor functions

$$f_{x_1 x_2 \ldots \bar{x_i} \ldots x_k} \qquad \text{as well as} \qquad f_{\bar{x_1}\bar{x_2}\ldots x_i \ldots \bar{x_k}}$$

represent two function signatures for the variable x_i. Similarly, we can apply this idea to the existential abstraction and to other special subfunctions.

Moreover, we also can use other masks besides *000...010...0* and *111...101...1* used here. Similar to the unique variables of a function f we can uniquely identify each aliasing group by the signatures of its variables. Thus it is not necessary to set all aliasing variables but the actual x_i to the same value. What we need to do, is to guarantee that all variables of the *same* aliasing group are set to the same value, and to keep track of which aliasing group is set to which value. So, with more aliasing groups we have more distinct masks that we can create in this way.

This kind of signature functions seems to construct the strongest well–defined signatures that we can describe for an input variable. Furthermore, it gives us new information about an input variable which is completely different from the one we get using the other two kinds of signature functions. The complexity of its computation depends on the two steps that are necessary to create a function signature. The first step, to compute, for instance, the ROBDD of the cofactor function, is linear in the size of the ROBDD of f. The second step, to reorder the input variables in this function with respect to the unique order, is dominated by the actual reordering algorithm. We used the algorithm introduced in [37] with very good practical results (see Section 3.3).

Finally, it is necessary to point out the following. One property of a signature for an input variable of a Boolean function is that it is an element of an ordered set. This is important to be able to use signatures for the creation of a unique permutation independent variable ordering. So, we need an order relation for the function signature to be able to order the variables. Here we simply use the lexicographical order relation for Boolean functions. To compare two Boolean functions f and g lexicographically using their ROBDDs needs time linear in the number of essential variables of these functions. This has been done as follows. We need to find the first point in the Boolean n-space, i.e., the one with the lowest order in the ordering of these points, such that it is contained in the ON–set of one function and in the OFF–set of the other. So we start at the root vertex in the ROBDDs of f and g and keep walking left down both ROBDDs (i.e., taking the 0–branch) at each variable unless pointers for both ROBDDs are the same. In this case, we branch right (i.e., along the 1–branch). When this terminates, one of the pointers will be the leaf vertex with value 1 and the other will be the leaf vertex with value 0. This gives us the required ordering among the functions. Note that if $f = g$, then these two functions are represented by the same ROBDD and we are done at the root vertex of this ROBDD. Since we are doing constant work at each level, this procedure is linear in the number of essential variables of function f and g.

3.3 Experimental Results

We implemented the ideas presented in this chapter in the Berkeley SIS–system, release 1.3 [34]. Here, we used the UCB–BDD–package of the system. All experiments were done on a SUN Sparcstation 10 with 64 MB RAM.

To test the quality of our signatures, we used all available benchmarks from the LGSynth91 [1] and ESPRESSO [5] benchmark set for which we were able to construct the ROBDDs, as well as a couple of additional benchmarks (*act1, act2* – the actel 1 and actel 2 cells from the FPGA manufacturer Actel; *mult3* — a 3-bit-multiplier). In all there are 243 benchmarks.

On these benchmarks, we used the in Section 3.2 introduced signatures to distinguish between their input variables step by step. Starting by considering all input variables as one aliasing group, we organized the refinement process as follows. We applied a signature function to the variables of each aliasing group, sorted these variables by their actual signatures, and separated the variables with different signatures from each other. If there still was aliasing after this process, we applied the next signature function.

	%unique	%cpu–time
co_sig	39%	14.3%
p–symmetry	24%	8.4%
co_br_sig	24%	62.1%
ex_sig	1%	0.1%
ex_br_sig	2%	1.5%
fctn_sig	2%	13.6%
$\sum$	92%	100.0%

Table 3.2: The Quality of Signatures in P_π

Signature functions were applied in the following order:

1. *co_sig* : cofactor satisfy count signature
2. *co_br_sig* : cofactor breakup signature
3. *ex_sig* : existential abstraction satisfy count signature
4. *ex_br_sig* : existential abstraction breakup signature
5. *fctn_sig* : cofactor function signatures with mask *000 ... 1... 0* and mask *111 ... 0... 1*. Note that the application of other than these two masks was not successful in our experiments.

Here, we use those signatures first for which no ROBDD–constructions are necessary (namely the first two signatures). Note again that at each step a signature function was applied for the variables with aliasing only, i.e., once an input variable has been uniquely identified, no further work needs to be done for this variable. Furthermore, most of the benchmarks have had more than one output variable. Since we assume that the correspondence between the output variables of two circuits is known, we can use all output functions of a benchmark sequentially starting with the first one in the description.

Partial symmetries as known from literature appear relatively often in Boolean functions [28]. If two input variables are partial symmetric, then there is no signature function which can distinguish between these two variables. This will be discussed in detail in the next chapter (see Section 4.2). However, partial symmetry can be detected easily using the methods introduced in [28]. Once they have been detected it is enough to compute the signatures for one representative of a maximal group of symmetric input variables. Our experiments have shown that it is the best with respect to CPU–time to compute these symmetries between applying the cofactor signature and the cofactor breakup signature.

Now let us come to our practical experiences with using signatures. Table 3.2 shows the percentage of benchmark circuits which have had additionally a unique identification for their input variables after applying a signature function (second column). Furthermore, it includes the part of CPU–time which was necessary to apply this signature function (third column).

Using all the signature functions introduced in this chapter and the test for partial symmetries (*p_symmetry*), approx. 92% of all 243 tested benchmarks have a unique identification for their input variables. Here, approx. 24% of the benchmarks contain partial symmetric variables.

Appendix A includes tables that provide details for this result. In these tables, each benchmark circuit is listed with its constellation of aliasing groups after applying all signature functions, and with the CPU–time which was necessary to exercise the complete procedure of computing signatures and distinguishing input variables using these signatures.

The CPU–times are very promising. Even for our worst case in terms of CPU–time, which is *C5315* with 178 input and 123 output variables, and 21194 ROBDD nodes, the procedure needs only approx. 10 minutes to determine a unique identification (see Appendix A). In most of the other cases, the CPU–time is in the order of a few seconds. The cofactor satisfy count signature (*co_sig*) is the most efficient signature function: it is able to identify the input variables of approx. 39% of all benchmarks uniquely and needs just 14.3% of the CPU–time to do this. The cofactor breakup signature breaks the tie for 24% of the benchmarks, but it takes over the half of the complete CPU–time. We tested other orders of applying signature functions as well (e.g., use *ex_sig* before using *co_br_sig*), with the following result: the demonstrated order is the one which needs the least amount of CPU–time. Avoiding ROBDD–constructions is more important than prefering the signature functions with the best time complexity. The reason for the relatively low CPU–time which was necessary to apply the existential abstraction satisfy count and breakup signature function is that the number of benchmarks that these signatures were applied on is relatively small in comparision to the complete set of benchmarks. Finally, we also observed that function signatures have to be used last — for unique input identification of 2% of the benchmarks, 13.6% of the total CPU–time was necessary.

These results show that the introduced approach to handle the permutation equivalence problem seems to be very promising. To compare it with related work on the permutation equivalence problem, let us discuss some of the key pieces of work here. For years this problem has been worked on by several other authors. In 1990, F. Mailhot and G. De Micheli described a new algorithm for Boolean matching which uses tautology checking based on Shannon decomposition [14]. Boolean matching is the key operation in technology mapping. It checks whether an element of a given library can be used to implement a part of a Boolean function. This can be formulated as checking the equivalence between a given Boolean function, called the *target function*, and the set of functions representing a library element. Often, this is considered for any permutation of the input variables. This provides another application for our permutation equivalence problem.

In their Boolean matching algorithm F. Mailhot and G. De Micheli use the symmetry and unateness property of input variables of a Boolean function as follows:

1. *Any input permutation must associate each unate (binate) variable in the target function to an unate (binate) variable in the function of the library element [14].*
2. *Variables or groups of variables that are interchangable (i.e., partial symmetric) in the target function must be interchangable in the function of the library element [14].*

Examining this in our context, they have used the property of an input variable of a Boolean function to be unate or binate as a signature, as well as the property of an input variable to belong to a set of partial symmetric variables (may be of size one). So, only variables belonging to symmetry sets of the same size can correspond to each other for equivalence. Obviously, these two properties may be useful for circuits with just a few number of input variables.

In 1992/93, several authors independently developed improvements for the Boolean matching algorithm (e.g., [9, 10, 21, 25, 33]). The common feature of these approaches is, that all have used ROBDDs for their computations and introduced signature–based methods to speed up the matching process. Most of the signatures for a Boolean function $f \in \mathcal{B}_{n,1}$ with respect to an input variable x_i that were introduced in the mentioned papers are based on analyzing the ON–set of f with respect to x_i in certain ways. Moreover, the property of x_i to be an essential variable or not as well as the property of x_i to be unate or binate was used. The signatures proposed in [25] were introduced in this thesis. One advantage of our methods for handling the permutation equivalence problem in comparison to the other signature–based approaches is, that we present powerful signature functions that can be computed needing just one ROBDD of the actual Boolean function (the cofactor satisfy count signature and the cofactor breakup signature function). Since these computations do not depend on the actual variable ordering of the ROBDD, we can apply the techniques for signature computation outlined in this thesis as long as we can construct an ROBDD for the function with any variable ordering. Furthermore, it is advantagous with respect to the CPU–time, necessary to compute possible correspondences, to avoid as many ROBDD manipulations as possible, as our experiments have shown.

Another approach, not based on the use of ROBDDs, was presented by I. Pomeranz and S. M. Reddy. In [29, 31], the authors provide a method for handling the permutation independent Boolean equivalence problem where the input correspondence as well as the output correspondence between two function $f, g \in \mathcal{B}_{n,m}$ is not known. This method is based on another data structure, namely circuits described at the gate level and works based on the following observation:

Let us consider an input pattern of function f with N 1's that yields in an output pattern with M 1's. The correspondence between the input and output variables of f and g is unknown. So this pattern can correspond to any input pattern of g with N 1's that yield in an output pattern with M 1's.

Using this consideration, the authors partition the input/output pattern of function f and g into subsets with the same number of 1's in the input part and the output part of the pattern, respectively. If f and g are permutation equivalent, then these partitions must be the same (modulo the order of the 1's in the pattern). Furthermore, a couple of signatures for each input variable can be defined using the subsets of this partition: say, $S(N, M)$ is the subset of input/output pattern with N 1's in the input part and M 1's in the output part of the patterns, then a signature for an input variable with respect to subset S is the number of patterns in S in which x_i assumes the value 1. (Note, that this works similar for output variables.) With the help of these signatures the authors try then to distinguish among the input variables in a similar way as we described it in Section 3.2.2.

	circuit			number of correspondences
name	#i	#o	#n	
CM150	21	1	64	$\approx 10^7$
CM151	12	2	32	216
act2	8	1	12	4
addm4	9	8	225	16
cordic	23	2	86	4
dist	8	5	135	16
ex4	128	28	896	4
i3	132	6	134	$\approx 10^{23}$
lal	26	19	123	24
misg	56	23	109	5184
mlp4	8	8	141	16
mult3	6	6	44	8
mux	21	1	88	$\approx 10^7$
mux_cl	11	1	18	216
ryy6	16	1	27	4
sao2	10	4	123	16
t481	16	1	80	331776
ts10	22	16	271	720
term1	34	10	616	$\approx 10^8$

Table 3.3: Benchmarks with Aliasing after Signature Computation

The advantage of this method is that it can be used on very large circuits for that no ROBDD description is possible. However, most of the actual Boolean matching and formal verification tools work with ROBDDs, and when this is the case, we think that the ROBDD–based methods should be prefered because they can be applied directly. Moreover, the worst-case complexity of several of the ROBDD–based approaches is less than the complexity of this approach. This complexity depends strongly on the number of the input/output patterns that are used to distinguish among the variables. Let us explain this further. Since it is not possible to use all pattern combinations for the matching process (that would require the complete truth table description of the circuit), the authors restrict the number of patterns that are used to distinguish among the variables by bounding N. This is the number of 1's in the input part of the actual considered set of patterns. So, an upper bound for the number of considered input/output patterns is given by $O(\binom{n}{N})$, where n is the number of inputs of the circuit. However, this is a polynomial of degree N in the number of input variables, n. In their experiments, the authors use $0 \leq N \leq 3$ and $n-3 \leq N \leq n$, i.e., they use all pattern with input combinations up to three 1's or three 0's. Thus, they have a set of $O(n^3)$ input patterns for which simulation must be done in order to determine the corresponding output pattern. This simulation is linear in the size of the circuit description. Then, they also have to handle a set of $O(n^3)$ input/output patterns for the purpose to separate the variables.

Moreover, even in the set of benchmarks that the authors used for their experiments, there were 4 circuits for that all patterns with up to 4 1's and 0's, respectively, were necessary to get good results. The quality of the results with respect to the correspondence possibilities between the input variables is comparable with our results after applying the first two categories of signature functions as presented in Section 3.2.3, although the authors handle the more general problem of simultaneous input *and* output matching. Another advantage is, that it can be directly extended to handle incompletly–specified functions [31]. Unfortunatly, the practical efficiency of the methods introduced by I. Pomeranz and S. M. Reddy cannot directly be compared with that of our methods since the authors do not provide CPU–times for their experiments.

Now let us consider the practical experiences with the signatures developed in this thesis again. For the 8% of benchmarks with aliasing, the number of possibilities for correspondence between input variables of two functions ranges from 4 to approx. 10^{23} after applying all signature functions. Table 3.3 lists these benchmarks. In this table a description of each circuit (name, number of inputs, outputs, and ROBDD nodes) is followed by the number of possible correspondences after using signatures. For these circuits, the problem seems to be that there are special properties that make it impossible to distinguish between the variables with aliasing with the help of signatures. Here, the obvious solution of enumerating all remaining correspondence possibilities to handle P_π may be acceptable for examples with a very small number of those possibilities. However, a further understanding of the other cases is necessary. This is the focus of the next chapter.

Chapter 4

Limits of Using Signatures

The method to handle the permutation equivalence problem introduced in the previous chapter is not complete. There is *no* signature function which can uniquely identify *all* the variables of the investigated benchmark sets. Comparing the most successful signature functions, we observed that those benchmarks that have variables that could not be uniquely identified are always the same over the different signatures. In other words, there is a nearly constant set of benchmarks for which signatures could not help to solve the permutation problem. Unfortunately, there are not just 2 or 3 variables of those benchmarks that are not uniquely identified, but about 15 and more, such that the number of possible correspondences is still large. Furthermore, this seems to be independent of the method used to solve that problem: in [29, 31], a totally different method has been used, based on another data structure – circuits described on the gate level. However, even here the same group of benchmarks causes problems [31]. Another observation is that those benchmarks with non-uniquely identified variables are not the benchmarks with the most number of inputs. From a statistical point of view, we can conjecture that the quality of the used signatures is not the problem for practical applications: we did not find a relationship between the number of input variables of a function and the ability of the signatures to distinguish between all these inputs. Hence, it seems a likely supposition that the variables that cannot be distinguished by the signatures have special properties that make it *impossible* to distinguish between them for the permutation independent comparison of two Boolean functions.

In this chapter, we discuss a property of input variables that make it impossible to distinguish between variables with aliasing with the help of signatures. This property is $\mathcal{G}$–symmetry. In **Section 4.1**, we introduce $\mathcal{G}$–symmetry and explain why $\mathcal{G}$–symmetry avoids a unique identification of input variables with the help of signatures. In the **Sections 4.2–4.4** , we discuss some special kinds of $\mathcal{G}$–symmetry that often appear in practice. In **Section 4.5** we discuss our experimental results on the 8% of benchmarks with aliasing. Finally, in **Section 4.6** the variety of $\mathcal{G}$–symmetry in general is discussed. Parts of these results are presented in [26].

4.1 The Property of $\mathcal{G}$–Symmetry

$\mathcal{G}$–symmetry can be defined as follows:

Definition 4.1 *Consider a group $\mathcal{G} \subseteq \mathcal{P}_n$ of permutations. A Boolean function $f \in \mathcal{B}_{n,1}$ is* $\mathcal{G}$–**symmetric** *if f keeps invariant under all permutations π in $\mathcal{G}$.*

$\mathcal{G}$–symmetry was defined similar by Hotz in 1974 [17]. The simplest example for $\mathcal{G}$–symmetry is a Boolean function f that is symmetric in all input variables. Here, $\mathcal{G}$ is equal to the permutation group $\mathcal{P}_n$, and we say, f is $\mathcal{P}_n$ – symmetric.

By definition, the group of permutations $\mathcal{G}$ may also be the group which contains the identity $\mathbf{1}$ only. So we can consider those Boolean functions with no $\mathcal{G}$–symmetry as functions that are $\mathcal{G}$–symmetric with respect to the set $\mathcal{G} = \{\mathbf{1}\}$. In other words, we can formulate the following property:

Property 4.1 *For each Boolean function $f \in \mathcal{B}_{n,1}$ the set $\mathcal{G} \subseteq \mathcal{P}_n$ of permutations such that f is symmetric with respect to all permutations in $\mathcal{G}$ forms a group.*

Proof: Let $\pi_1 \in \mathcal{G}$ and $\pi_2 \in \mathcal{G}$ be two permutations of $\mathcal{G}$. We need to prove that the permutation $\pi_1 \circ \pi_2 \in \mathcal{G}$:

$$f \circ (\pi_1 \circ \pi_2) = (f \circ \pi_1) \circ \pi_2 = f \circ \pi_2 = f$$

In other words, the permutation $\pi_1 \circ \pi_2$ keeps the function f invariant, thus $\pi_1 \circ \pi_2 \in \mathcal{G}$. □

In order to understand the significance of $\mathcal{G}$–symmetry for the permutation equivalence problem, let us consider the relation $R_{\mathcal{G}}$ on the set of input variables, $X = [x_1, x_2, \ldots, x_n]$, which is defined by a group of permutations $\mathcal{G} \subseteq \mathcal{P}_n$. Let $x_i \in X$ and $x_j \in X$ be two input variables:

$$x_i R_{\mathcal{G}} x_j \Longleftrightarrow \exists \pi \in \mathcal{G} : \pi(x_i) = x_j.$$

Property 4.2 *The relation $R_{\mathcal{G}}$ is an equivalence relation.*

Proof: This follows immediately from the property that $\mathcal{G}$ is a group.□

Let us denote with $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ the partition of the input variables in X which corresponds to a relation $R_{\mathcal{G}}$. This partition is well–defined with respect to a $\mathcal{G}$–symmetry, and it follows from Property 4.1 and Property 4.2:

Fact 4.1 *For each Boolean function $f \in \mathcal{B}_{n,1}$ and its set of input variables X there is a well–defined partition $\mathcal{A}$ of the input variables of f.*

Often it is enough to consider this partition $\mathcal{A}$ as an *unordered* set of subsets of the inputs. However, sometimes it is necessary to look at it as an *ordered* set.

Property 4.3 *Given a Boolean function $f \in \mathcal{B}_{n,1}$ and the well–defined partition of its input variables, $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$, there is a well–defined and* ordered *partition of the input variables with respect to f,*

$$\mathcal{A}^f = \{A_1^f, A_2^f, \ldots, A_k^f\},$$

which is constructed as follows:

1. *$\mathcal{A}^f$ considered as* unordered *set of subsets is equal to partition $\mathcal{A}$.*
2. *The ordering step works as follows:*
   (a) *At first, we order the subsets by their size, such that $|A_i^f| \leq |A_{i+1}^f|$ for all $i = 1, 2, \ldots, k$.*
   (b) *If some subsets have the same size, then we use the elements of them to establish an ordering, such that the result looks as follows: if $|A_i^f| = |A_{i+1}^f|$ for any $i \in \{1, 2, \ldots, k\}$, then $j < l$ for $x_j \in A_i^f$ with $j = \min\{h : x_h \in A_i^f\}$ and $x_l \in A_{i+1}^f$ with $l = \min\{h : x_h \in A_{i+1}^f\}$.*

Proof: The so constructed partition $\mathcal{A}^f$ of the input variables of a Boolean function f contains the same elements as the original partition $\mathcal{A}$. So it is well–defined as an *unordered* set of subsets. Furthermore, the ordering process is well–defined: Step 1.a orders those subsets uniquely that have different sizes. Step 1.b orders the subsets with the same size uniquely. Note, that this second step depends on the order of the input variables of the actual function f. In other words, the order of the subsets in partition $\mathcal{A}^f$ is just unique with respect to the Boolean function f which has been considered. □

Now we consider signatures again. What can we say about the relationship between signatures and $\mathcal{G}$–symmetry? There are two properties that complete our picture about signatures. The first property is very important and can be formulated without further considerations:

Property 4.4 *Let $\mathcal{G} \subseteq P_n$ be the group of permutations of X which constructs partition $\mathcal{A}$. Consider any element A_l of $\mathcal{A}$, any $\mathcal{G}$–symmetric Boolean function f, and any signature function $s \in \mathcal{S}_n$. For any $x_i, x_j \in A_l : s(f, x_i) = s(f, x_j)$.*

Proof: For all $x_i, x_j \in A_l$ there is a permutation $\pi \in \mathcal{G}$ such that $\pi(x_i) = x_j$ (see definition of partition $\mathcal{A}$). Consider any $x_i, x_j \in A_l$, a permutation $\pi \in \mathcal{G}$ such that $\pi(x_i) = x_j$, and any signature function s. As defined, $s(f, x_i) = s(f \circ \pi, \pi(x_i))$. This is equal to $s(f, \pi(x_i))$ since f is $\mathcal{G}$–symmetric. Thus, $s(f, x_i) = s(f, x_j)$ for any $x_i, x_j \in A_l$. □

This result now gives us a possible explanation for the trouble several benchmarks have with the signature approach: it is possible that the benchmarks with aliasing include $\mathcal{G}$–symmetric functions.

Then *there is no unique description by signatures* for the input variables of these functions. Thus, it is futile to try and distinguish all the variables with additional signatures. The immediate follow-up question is then: what do we do in this case? Our response to this is that we do not really need to uniquely identify the variables in most cases. Perhaps we can achieve our end goal of establishing permutation independent function equivalence by identifying the variables involved in the $\mathcal{G}$–symmetry and exploring the exact nature of the $\mathcal{G}$–symmetry. This is further explored in the next sections.

Before we discuss it, let us develop the second property of signatures with respect to $\mathcal{G}$–symmetry. For this, we need to think about a universal signature function.

Definition 4.2 *A* **universal signature function** $u : \mathcal{B}_{n,1} \times X \longrightarrow U$ *is a signature function which has the following property. Given a Boolean function* $f \in \mathcal{B}_{n,1}$ *and the partition of its input variables,* $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$*:*

$$\forall x_i, x_j \in X : \text{ if } x_i \in A_l, x_j \in A_h \text{ with } l \neq h \Longrightarrow u(f, x_i) \neq u(f, x_j).$$

If two input variables x_i and x_j are not in the same subset of partition $\mathcal{A}$ of a Boolean function $f \in \mathcal{B}_{n,1}$, then the universal signature function must provide different signatures for these two variables.

Such a universal signature function has the property to dominate any other signature function $s \in \mathcal{S}_n$ in the following sense:

$$\forall f \in \mathcal{B}_{n,1} \forall x_i, x_j \in X : \quad \text{if } \exists s \in \mathcal{S}_n : s(f, x_i) \neq s(f, x_j) \Rightarrow u(f, x_i) \neq u(f, x_j).$$

In other words, if there is any signature function which can distinguish between two input variables of a Boolean function, then the universal signatures of these two variables must be able to distinguish between them as well. This property follows immediately from Property 4.4 and the definition of a universal signature function. If two input variables x_i and x_j of a Boolean function $f \in \mathcal{B}_{n,1}$ have different signatures for any signature function $s \in \mathcal{S}_n$, then they have to be in different subsets of the partition $\mathcal{A}$ of the input variables of f (see Property 4.4). That is why the universal signatures of these two variables must be different as well (see Definition 4.2). In this sense, we can say that a universal signature function is the strongest signature function which can be constructed.

Theorem 4.1 *There is a universal signature function* $\mathcal{U} : \mathcal{B}_{n,1} \times X \longrightarrow U$.

Proof: At first we construct a candidate for a universal signature function. Then we show, that this candidate is a signature function and that it is universal.

1. Construction:

   Let us consider a Boolean function $f \in \mathcal{B}_{n,1}$. From Fact 4.1, it follows that there is a well–defined partition of the input variables of function f. Let us construct this partition:

   $$\mathcal{A} = \{A_1, A_2, \ldots, A_k\} = \{\{x_1^1, \ldots, x_{l_1}^1\}, \{x_1^2, \ldots, x_{l_2}^2\}, \ldots, \{x_1^k, \ldots, x_{l_k}^k\}\}.$$

   Now, we select the lexicographical smallest of all functions that can be constructed by permuting the input variables of f:

   $$f_{\min} = \min\{g \in \mathcal{B}_{n,1} : g = f \circ \pi \text{ with } \pi \in \mathcal{P}_n\}.$$

   This function, $f_{\min}$ has a unique partition $\mathcal{A}'$ of its input variables as well.

   What do we know about the relationship between function f and function $f_{\min}$ and the partition of their input variables? We know, that $f_{\min} = f \circ \hat{\pi}$ for a permutation $\hat{\pi} \in \mathcal{P}_n$. In other words, these two functions are permutation equivalent. From this fact, it follows that the partition $\mathcal{A}'$ has the same structure as partition $\mathcal{A}$:

   $$\mathcal{A}' = \{A_1', A_2', \ldots, A_k'\} = \{\{y_1^1, \ldots, y_{l_1}^1\}, \{y_1^2, \ldots, y_{l_2}^2\}, \ldots, \{y_1^k, \ldots, y_{l_k}^k\}\}.$$

   Furthermore, there is a 1–1–mapping $\Phi : \mathcal{A} \longrightarrow \mathcal{A}'$ which maps the subsets of partition $\mathcal{A}$ to the subsets of partition $\mathcal{A}'$. Φ depends on the permutation $\hat{\pi}$ which we use to construct $f_{\min}$ and is defined as follows:

   $$\Phi(A_i) = \hat{\pi}(A_i) = A_j'$$

   for all $i = 1, \ldots, k$ and the corresponding $j \in \{1, 2, \ldots, k\}$.

   Furthermore, there is a well–defined and ordered partition of the input variables of $f_{\min}$:

   $$\mathcal{A}^{f_{\min}} = \{A_1^{f_{\min}}, A_2^{f_{\min}}, \ldots, A_k^{f_{\min}}\},$$

   which is constructed by ordering the subsets of partition A' (see Property 4.3). With the help of this partition we are able to construct our candidate for a universal signature function. Let $x_i \in X$ be an input variable of f. Then,

   $$\mathcal{U}(f, x_i) = j \text{ with } \hat{\pi}(x_i) \in A_j^{f_{\min}},$$

   where $\hat{\pi}$ is a permutation of $\mathcal{P}_n$ which constructs the lexicographical smallest function, $f_{\min} = f \circ \hat{\pi}$.

2. Proof of correctness:

   We need to prove that $\mathcal{U}$ is a universal signature function. This is done in three parts:

   (a) $\mathcal{U}$ is a well–defined mapping from $\mathcal{B}_{n,1} \times X$ into an ordered set $(U, \leq)$.

   Proof: $\mathcal{U}$ maps the set $\mathcal{B}_{n,1}$ of all Boolean functions and their input variables X into the set of indices of the subsets of the ordered partition $\mathcal{A}^{f_{\min}}$, which is equal to the set of integers from $\{1, 2, \ldots, k\}$ (k is the number of subsets in partition $\mathcal{A}$). In other words, $\mathcal{U}$ maps into the set of non–negative integers. This is an ordered set.

   In order to prove that $\mathcal{U}$ is a *well–defined* mapping, we need to consider the case that there are more than one permutations $\hat{\pi} \in \mathcal{P}_n$ that construct the lexicographical smallest function $f_{\min} = f \circ \hat{\pi}$.

   Suppose, there are two of those permutations, π_1 and π_2. What can we say about these two permutations? We know that

   $$f_{\min} = f \circ \pi_1 = f \circ \pi_2.$$

   So it follows that $f = f \circ \pi_1 \circ \pi_2^{-1}$, i.e., the permutation $\pi_1 \circ \pi_2^{-1}$ keeps the Boolean function f invariant. Now consider any input variable x_i of function f. By definition of partition $\mathcal{A}$, it follows that the two variables x_i and $\pi_2^{-1}(\pi_1(x_i))$ are in the same subset of $\mathcal{A}$.

   Let us see what happens on applying the permutation π_2 to these two variables. Remember that this permutation can be used to construct partition $\mathcal{A}'$ from partition $\mathcal{A}$. We immediately deduce that the variables $\pi_2(x_i)$ and $\pi_1(x_i)$ are in the same subset of partition $\mathcal{A}'$. However, these are the two variables that x_i is mapped on using the permutation π_1 and π_2, respectively. So, we finally know that the input variable x_i is mapped into the same subset of partition $\mathcal{A}'$, independent of the permutation which we use to construct this partition. □

   (b) $\mathcal{U}$ is a signature function.

   Proof: Let us consider the function $f_{\min}$. This function is information about the Boolean function f which is independent of any permutation of the input variables of f since we use *all* $\pi \in \mathcal{P}_n$ for its construction. Also the partition $\mathcal{A}^{f_{\min}}$ is permutation independent information for f. From this, it follows that the information we get for an input variable x_i using $\mathcal{U}$ is independent of any permutation of the inputs. □

   (c) $\mathcal{U}$ is universal.

   Proof: Let us consider the mapping $\Phi : \mathcal{A} \longrightarrow \mathcal{A}'$ again, which maps the subsets of the partition of the inputs of function f to those subsets of the partition of the inputs of function $f_{\min}$. From the construction of this mapping, it follows that if two input variables x_i and x_j of function f are in different subsets of partition $\mathcal{A}$, then $\hat{\pi}(x_i)$ and $\hat{\pi}(x_j)$ are in different subsets of the partition $\mathcal{A}'$ of the lexicographical smallest function $f_{\min}$ too. Remember, the permutation $\hat{\pi}$ is a permutation of the input variables of function f, which constructs the lexicographical smallest of all those functions: $f_{\min} = f \circ \hat{\pi}$.

Now, if we order $\mathcal{A}'$ as proposed in Property 4.3, then we get the well–defined and ordered partition $\mathcal{A}^{f_{\min}}$ for the function $f_{\min}$, and it follows for any Boolean function $f \in \mathcal{B}_{n,1}$ and for all input variables $x_i, x_j \in X$:

$$x_i \in A_l \text{ and } x_j \in A_h \text{ with } l \neq h \text{ iff } \hat{\pi}(x_i) \in A_p^{f_{\min}} \text{ and } \hat{\pi}(x_j) \in A_q^{f_{\min}} \text{ with } p \neq q.$$

This is the same as: for all $x_i, x_j \in X$:

$$x_i \in A_l \text{ and } x_j \in A_h \text{ with } l \neq h \text{ iff } \mathcal{U}(f, x_i) \neq \mathcal{U}(f, x_j).$$

In other words, the signature function $\mathcal{U}$ is universal. □ □

The signature function $\mathcal{U}$ demonstrates the property of a universal signature function to dominate all other signature functions very well. The partition $\mathcal{A}$, which is the basic component for the construction of $\mathcal{U}$, is nothing else but the partition of the input variables which we try to construct with the help of our practical signatures. Moreover, this partition is sorted independent of permutation by those signatures. Now, if the universal signature for an input variable is the index of such a set of the partition, then one thing is obvious: if there is a practical signature which is different for two variables, then these two variables will be in two different subsets of the partition. Also the universal signatures of these two variables will be different.

Why does it help to know that there is a universal signature function? It cannot be used in practice since its construction would be computationally too intensive. However, it is useful to have such a universal signature function for theoretical investigations.

In Section 3.2.3 we introduced a method for analytically comparing the effectiveness of a signature function [21]. Remember, given a signature function $s : \mathcal{B}_n \times X \longrightarrow U$, a measure of the effectiveness of a signature function is the cardinality of the co–domain of s, which is $|U|$. However, as we can see here, a small co–domain does not automatically imply that the signature function is not efficient. It depends on all properties of a signature function. In the case of the universal signature function, it is: $|U| \leq n$, where n is the number of input variables of the actual function. However, as we have proven, the universal signature function is the most powerful signature function at all.

At the moment, we are interested in signatures in relationship to $\mathcal{G}$–symmetry. With the help of the universal signature function U we can prove, that if any two input variables of a Boolean function have the same signature $\mathcal{U}$, then there is a permutation $\pi \in \mathcal{P}_n \setminus \mathbf{1}$ of the input variables of this function, such that $f = f \circ \pi$. This permutation π of the input variables keeps the function f invariant, i.e., the function f is $\mathcal{G}$–symmetric with respect to a group of permutations $\mathcal{G} \neq \{\mathbf{1}\}$. This property can be formulated as follows.

Property 4.5 *Let f be any Boolean function of $\mathcal{B}_{n,1}$.*
Let $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ be the partition of the input variables of f.

$$\forall x_i, x_j \in X : \mathcal{U}(f, x_i) = \mathcal{U}(f, x_j) \Longrightarrow x_i, x_j \in A_l \quad for \quad some \quad l \in 1, 2, \ldots, k.$$

Proof: This follows immediately from the definition of $\mathcal{U}$. □

The property ensures that we do not need to think about other problems with signatures identifying input variables for permutation independent function equivalence. It demonstrates that $\mathcal{G}$–symmetry is indeed the only handicap for signatures to be able to uniquely identify an input variable of a Boolean function independent of permutation. Thus, we now need to focus on $\mathcal{G}$–symmetries.

Specifically, we now discuss some special kinds of $\mathcal{G}$–symmetry which often appear in practice. Of course, this cannot be a complete enumeration of possible $\mathcal{G}$–symmetries. We discovered these cases in our quest to understand why signatures were proving to be inadequate for permutation independent Boolean comparison in some cases.

4.2 Partial Symmetries

At first, let us consider the most common case. A Boolean function $f \in \mathcal{B}_{n,1}$ is symmetric with respect to a subset of input variables $\hat{X} \subseteq X$ if f is invariant under all permutations $\pi \in \mathcal{P}_k$ of the input variables in $\hat{X} \subseteq X$, where k is the number of input variables in $\hat{X}$. We say, that f is *partial symmetric* with respect to $\hat{X}$. Furthermore, the set $\hat{X}$ is a *maximal symmetry group* (a maximal set of symmetric variables) of f if f is symmetric with respect to $\hat{X}$, and there is no variable $x_i \notin \hat{X}$ such that f is symmetric with respect to $\hat{X} \cup x_i$ as well. This is the simplest kind of $\mathcal{G}$–symmetry, and it is well–known.

In the partition $\mathcal{A}$ of the inputs of a partial symmetric Boolean function f all input variables of the subset $\hat{X}$ are in one subset.

However, these symmetries are easy to detect and to handle.

Fact 4.2 *A Boolean function f is partial symmetric with respect to the variables x_i and x_j iff $f_{x_i\bar{x}_j} = f_{\bar{x}_i x_j}$ [28].*

With the help of this fact we are able to test partial symmetry, since it is an equivalence relation [28]. Furthermore, different methods to improve this basic symmetry test have been developed. We use the methods introduced in [28], while one of the methods, introduced there, is to use simple signatures as proposed in Section 3.2.3.1.

After the detection of partial symmetries we are done, since each correspondence between symmetric variables of two Boolean functions is fine for the purpose to test permutation equivalence.

Finally, let us explain an algorithm to identify the input variables by signatures that considers these symmtries. With our knowledge about symmetric variables, we see that one of the first steps of this procedure is to determine all maximal groups of pairwise symmetric variables. As we know, this can be done fast, and the advantage is that the signature computations can be restricted to one representative of each maximal symmetry group. In order to combine the used methods of symmetry detection [28] with the different signature functions as introduced in Section 3.2.3, we determine the maximal symmetry groups after applying the cofactor satisfy count signature function (see Section 3.2.3 and Section 3.3).

4.3 Hierarchical Symmetries

Investigations on our benchmark set have shown that for several examples the reason for the existence of aliasing groups after computation of all signatures introduced in Section 3.2.3 is the following kind of symmetry.

Definition 4.3 *Let $f \in \mathcal{B}_{n,1}$ be a Boolean function with the input variables $X = [x_1, x_2, \ldots, x_n]$. Let $X_1, X_2 \subset X$ be two subsets of X.*

X_1 and X_2 are **hierarchical symmetric (h–symmetric)** *iff*

1. $|X_1| = |X_2| > 1$
2. *X_1 and X_2 are maximal symmetry groups of f. (see Section 4.2)*
3. *f is $H(X_1, X_2)$–symmetric, where $H(X_1, X_2)$ is the subgroup of the permutation group $\mathcal{P}_n$ generated by the following set of permutations:*

$$\{\pi \in \mathcal{P}_n | \pi(X_1) = X_2 \text{ and } \pi(X_2) = X_1\}.$$

   I.e., f keeps invariant under any exchanging of the variables of X_1 with those of X_2.

A group of subsets of X, $\{X_1, X_2, \ldots, X_k\}$ is **h-symmetric** *iff:*

$$\forall i, j \in \{1, 2, \ldots, k\} : X_i \text{ is h–symmetric to } X_j.$$

Let us consider the following example:

Example 4.1 $f = \overline{(x_1 + x_2)} + \overline{(x_3 + x_4)} + \overline{(x_5 + x_6)}$

Here, $\{x_1, x_2\}$, $\{x_3, x_4\}$ and $\{x_5, x_6\}$ are pairs of partial symmetric variables, but there is no partial symmetry between two variables of different pairs. However, it is easy to see, that exchanging any two of these three pairs keeps the function f invariant. This simple example illustrates h–symmetry.

Property 4.6 *H–symmetry is an equivalence relation on the partition of the set of input variables of a Boolean function f in its maximal symmetry groups.*

Proof: The symmetry and reflexivity is obviously to see. For transitivity we have to show the following. Let X_1, X_2, and X_3 be three disjoint sets of symmetric variables. If X_1 is h–symmetric to X_2 and X_2 is h–symmetric to X_3, then X_1 and X_3 are h–symmetric as well. Therefore, let us go through all points of the definition of h–symmetry:

1. $|X_1| = |X_2| = |X_3| > 1$
2. true by assumption
3. We know, that exchanging X_1 with X_2 as well as exchanging X_2 and X_3 does not change the function f. So, let us do the following. First, exchange the variables of X_1 with those of X_2. After that, exchange the variables of X_2 with those of X_3. The resulting function is f, and what we have done is to exchange the variables of X_1 with those of X_3 using the variables of X_2. This implies that X_1 with X_3 does not change function f. □

The definition of h–symmetry indicates that this is a special kind of $\mathcal{G}$–symmetry. Let us examine the partition $\mathcal{A}$ of the input variables constructed by h–symmetry.

Property 4.7 *If two subsets of input variables, X_1 and X_2 are h–symmetric, then all input variables of X_1 and X_2 are in one element A_i of partition $\mathcal{A}$.*

Proof: This follows from the definition of partition $\mathcal{A}$. □

Thus, from Property 4.4, we know that all of these variables have to have the same signatures, i.e., they form an aliasing group. In other words, there is no way to distinguish between them via signatures.

Luckily, there is a solution for this which is based upon our handling of partial symmetric variables. To understand this, let us consider the algorithm to identify the input variables by signatures. Here, we first determine all maximal groups of pairwise symmetric variables (see previous section). In this way, pairwise symmetric variables are kept together in aliasing groups. Now, let us consider two h–symmetric subsets, X_1 and X_2, of input variables of function f again. As we know, they form an aliasing group: $\{X_1 \cup X_2\}$. A correspondence between these variables and the variables of an aliasing group of any other function g is possible if the variables of the other group have the same signatures, and this aliasing group has the same structure, i.e., $\{Y_1 \cup Y_2\}$ with Y_1 and Y_2 are maximal symmetry groups and $|X_1| = |Y_1|$. Then there are two possible correspondences between these groups: $(X_1 \leftrightarrow Y_1, X_2 \leftrightarrow Y_2)$ as well as $(X_1 \leftrightarrow Y_2, X_2 \leftrightarrow Y_1)$. Because of h–symmetry both of these correspondences are acceptable for our purpose. In other words, our remaining task in terms of h–symmetry is to *detect* this kind of symmetry. That is sufficient to decide that no further work needs to be done with these aliasing groups in order to solve the permutation problem, P_π.

So let us try and see what we have to do. First, we want to answer the following question. Let f be a Boolean function with n input variables, $X = [x_1, x_2, \ldots, x_n]$, and X_1 and X_2 be two disjoint subsets of the set of inputs, $X_1, X_2 \subset X$. When is it possible to exchange X_1 and X_2 in the function f without changing f itself?

Of course, a necessary condition is that the number of variables in X_1 and X_2 must be the same. Otherwise you could not completely exchange one subset with the other. However, this is not sufficient. Supposing that $|X_1| = |X_2| = k$ let us see what *sufficient* condition exists for the exchangeability of X_1 and X_2 in f.

Let $f_{a_1(X_1)a_2(X_2)}$ be the cofactor of f where the variables of X_1 are set to $a_1 \in \{0,1\}^k$ and the variables of X_2 are set to $a_2 \in \{0,1\}^k$. Now we are able to formulate our condition.

Fact 4.3 *Exchanging two different, ordered subsets of variables, $X_1 = [x_1^1, \ldots, x_k^1]$ and $X_2 = [x_1^2, \ldots, x_k^2]$, i.e., exchanging x_i^1 with x_i^2 for all $i = 1, 2, \ldots, k$, does not change the function f iff for all assignments $a_1, a_2 \in \{0,1\}^k$: $f_{a_1(X_1)a_2(X_2)} = f_{a_2(X_1)a_1(X_2)}$.*

Note that a well–known special case of this property is $k = 1$, i.e., the question if two variables, x_i and x_j are exchangeable in a Boolean function f without changing f. In this case, we call x_i and x_j a pair of *symmetric variables*, and we know that this symmetry can be shown using Fact 4.2.

In other words, Fact 4.3 is only a generalization of the well–known symmetry test for symmetric variables to a test for the exchangeability of groups of variables.

Let us consider an example:

Example 4.2 $f = \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1x_2\bar{x}_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3x_4 + x_1x_2\bar{x}_3\bar{x}_4$

This function f is a Boolean function over the set of input variables $X = [x_1, x_2, x_3, x_4]$. There is no pair of symmetric variables in it. We can prove it using Fact 4.2. However, two subsets of X, $X_1 = \{x_1, x_2\}$ and $X_2 = \{x_3, x_4\}$ can be exchanged without changing f. We can prove that fact having a look at the equation of Example 4.2. For a formal proof using Fact 4.3 it is necessary to check the six different cofactor equations that can be constructed by the different assignments a_1 and a_2 to X_1 and X_2:

a_1	a_2	cofactor equation
00	01	$f_{\bar{x}_1\bar{x}_2\bar{x}_3x_4} = f_{\bar{x}_1x_2\bar{x}_3\bar{x}_4} = 1$
00	10	$f_{\bar{x}_1\bar{x}_2x_3\bar{x}_4} = f_{x_1\bar{x}_2\bar{x}_3\bar{x}_4} = 0$
00	11	$f_{\bar{x}_1\bar{x}_2x_3x_4} = f_{x_1x_2\bar{x}_3\bar{x}_4} = 1$
01	10	$f_{\bar{x}_1x_2x_3\bar{x}_4} = f_{x_1\bar{x}_2\bar{x}_3x_4} = 0$
01	11	$f_{\bar{x}_1x_2x_3x_4} = f_{x_1x_2\bar{x}_3x_4} = 0$
10	11	$f_{x_1\bar{x}_2x_3x_4} = f_{x_1x_2x_3\bar{x}_4} = 1$

Note that we do not consider assignments with $a_1 = a_2$.

However, having a closer look at f we see that while exchanging x_1 with x_3 and x_2 with x_4 keeps f invariant, exchanging x_1 with x_4 and x_2 with x_3 generates another function: $\hat{f} = x_1\bar{x}_2\bar{x}_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + x_1x_2\bar{x}_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3x_4$. This function is not equal to f, although we also have exchanged the variables of X_1 with those of X_2.

Example 4.2 clarifies the following. In general there are two major problems that prevent us to use Fact 4.3 in practice. The first one is its complexity. For two variable groups of size k there are $(2^{2k-1} - 2^{k-1})$ tests necessary to check their exchangeability in the prescribed manner. There are 2^k different vectors of constant values in $\{0,1\}^k$. In the first step, the selected vector a_1, say $00\ldots0$, has to be checked with all possible vectors a_2 except the same as a_1. That gives $2^k - 1$ different possibilities. Once that has been done, we can select another vector a_1, say $00\ldots01$, and start again with selecting the vector a_2. However, we do not have to use $a_2 = 00\ldots0$ again, because the equation

$$f_{X_1 \leftarrow 00\ldots0 X_2 \leftarrow 0\ldots01} = f_{X_1 \leftarrow 0\ldots01 X_2 \leftarrow 00\ldots0}$$

has been tested in Step 1 already. Similarly, there are only $2^k - 2$ equations to test in the second step, $2^k - 3$ in the third step, and so on. All in all, there are $\sum_{i=1}^{2^k-1} i = [2^k \cdot (2^k - 1)]/2$ different cofactor equations that have to be tested for our purpose.

The second problem is that Fact 4.3 only holds for exactly one exchange of the variables of X_1 with those of X_2, namely x_i^1 with x_i^2 for all $i = 1, 2, \ldots, k$. However, as shown in Example 4.2, there are other possibilities to exchange the variables, as well. Those possibilities are not covered by the cofactor tests of Fact 4.3. That implies that if we want to know if none of the possible exchanges of the variables of X_1 with those of X_2 changes f, then it is necessary to apply a series of cofactor tests similar to that formulated in Fact 4.3 to *each* of the $k!$ exchanging possibilities.

Considering these two problems we know that it is not possible to use Fact 4.3 for a practical solution of the general problem: there would be $\Theta(k! \cdot 2^k)$ cofactor tests necessary to check the exchangability of two sets of variables with size k.

However, let us consider our special case – the test, whether X_1 and X_2 are *h–symmetric.*

Fortunately, there is one more property given on X_1 and X_2 that makes it comfortable for us to use Fact 4.3 in this case. *This property is the partial symmetry of the variables of X_1 and X_2, respectively.*

Let us see how we can use this in the detection of h–symmetries. A Boolean function $f \in \mathcal{B}_{n,1}$ is symmetric with respect to a subset of input variables $X_i \subseteq X$ iff f keeps invariant under all permutations of the variables in X_i. Furthermore, since each vector $a \in \{0,1\}^k$ with exactly l one's in it is a permutation of any other vector with l one's in it, it is equivalent to say: f is symmetric with respect to X_i if f only depends on the number of one's in the input assignment to X_i [38]. We will call this number of one's in an input assignment to a set of variables, X_i the *weight* of that input assignment with respect to X_i.

Now we can directly conclude the following two facts about the cofactor of f with respect to a set of symmetric variables.

Fact 4.4 *Let V_l be the set of all assignments of constant values to the set of symmetric variables, X_i with the weight l. Let $a_1, a_2 \in V_l$. Then $f_{a_1} = f_{a_2}$.*

This gives us the number of different cofactors with respect to a set X_i of partial symmetric variables that can be constructed.

Fact 4.5 *Consider the set of all cofactors of f that can be constructed with respect to the set X_i of partial symmetric variables. The maximal cardinality of this set is $|X_i| + 1 = k + 1$. (One cofactor for every weight.)*

With Fact 4.4 and 4.5 we can go back to our two problems checking the exchangeability of two sets of variables in a function. We will see that these two problems are solved for h–symmetry.

First let us consider the complexity of the cofactor test again. Because of Fact 4.4 and 4.5 it can be modified as follows: instead of all 2^k possible assignments for the variables of X_1 and X_2, respectively, we just have to consider $k+1$ assignments of X_1 as well as of X_2, one of every possible weight. In other words, we do not have to test all combinations of the 2^k *assignments* but only all combinations of the $k + 1$ possible *weights*.

Fact 4.6 *Exchanging two different subsets of partial symmetric variables, X_1 and X_2 does not change the function f iff for each set of assignments V_{l_1} and V_{l_2} of $\{0,1\}^k$ to the variables of X_1 and X_2 with weight l_1 and l_2 and $l_1 \neq l_2$:*

$$f_{V_{l_1}(X_1)V_{l_2}(X_2)} = f_{V_{l_2}(X_1)V_{l_1}(X_2)}.$$

Note, that we have to know that X_1 and X_2 are sets of partial symmetric variables before using Fact 4.6 for our purpose.

What about the weight combinations $0/0$, $1/1, \ldots, k/k$? Obviously, we do not have to test the combinations $0/0$ and k/k, i.e., all variables of X_1 and X_2 set to 0 and 1, respectively, since an exchange of the two variable groups with these assignments does not change the function value of f. However, the same holds for all other combinations with the same weight. The reason of this is the partial symmetry of the variable groups. In order to test the combination i/i with $i \in \{1, 2, \ldots, k-1\}$, we have to select an assignment with weight i for the variables of X_1 as well as for the variables of X_2. Since we can select *any* assignment with weight i, let us take the *same* assignment for X_2 as for X_1. Now we have the same situation for the weight combination i/i as for the combinations $0/0$ and k/k – an exchange of the variable groups with these weights cannot change the function value of f.

Suppose the symmetry of the variables of our two groups X_1 and X_2 is tested before, we get the moderate number of $(k^2 + k)/2$ tests necessary to check the h–symmetry of X_1 and X_2: k choices

for the assignment to X_2 in the first step, $k-1$ in the second, and so on. Dealing with ROBDDs we know that these tests need only constant time on the given cofactors. The cofactor construction itself needs time linear in the number of ROBDD nodes of f. So, in all, we need time $O(k^2|V|)$ where $|V|$ denotes the number of nodes in the ROBDD of f and k the size of the two variable groups we want to test.

Let us now discuss the second problem with the cofactor test. As we know, if X_1 and X_2 are different groups of partial symmetric variables, then f only depends on the weight of the inputs of X_1 and X_2, but does not depend on the permutation of those variables in f. That is why it is enough to do the cofactor tests of Fact 4.6 with respect to one possible exchange of X_1 with X_2 in order to get the information about the exchangeability of X_1 and X_2 regarding to all other exchanges as well. In other words, in the case of partial symmetry groups it is possible to exchange the variables of the groups in either all or no combinations without changing the function f.

Now we can start with a description of the complete algorithm to determine the h-symmetry groups of a Boolean function $f \in \mathcal{B}_{n,1}$.

First, let us review on Example 4.1: $f = \overline{(x_1 + x_2)} + \overline{(x_3 + x_4)} + \overline{(x_5 + x_6)}$. As we know, there is h–symmetry between certain variable groups in this example. This can be proved as follows:

1. Consider the groups of symmetric variables. In the case of this example there are three groups with size 2: $X_1 = \{x_1, x_2\}$, $X_2 = \{x_3, x_4\}$, and $X_3 = \{x_5, x_6\}$.

2. To prove the h–symmetry of these groups it is enough to check X_1 with X_2, and X_2 with X_3 (see Property 4.6). Three cofactor test are necessary in both cases: weight combination 0/1, 0/2, and 1/2. For X_1 and X_2 these are:

$$\begin{aligned} f_{\bar{x}_1\bar{x}_2\bar{x}_3x_4} &= f_{\bar{x}_1x_2\bar{x}_3\bar{x}_4} = 1 \\ f_{\bar{x}_1\bar{x}_2x_3x_4} &= f_{x_1x_2\bar{x}_3\bar{x}_4} = 1 \\ f_{\bar{x}_1x_2x_3x_4} &= f_{x_1x_2\bar{x}_3x_4} = \overline{(x_5 + x_6)}. \end{aligned}$$

   Similarly this has to be done to test the exchangeability of X_2 and X_3.

Now, let us come to our algorithm. Given a Boolean function $f \in \mathcal{B}_{n,1}$ we can determine the h–symmetries of these variables in two steps:

Step 1: Create the list of all candidates for h–symmetry.

A *candidate for h–symmetry* is a set of same–sized partial symmetry groups of variables of f, except those of size one. Note, that we construct only maximal candidates. So, there is no h–symmetry possible between different candidates.

Recall Example 4.1, here we have exactly one candidate, *viz.* $\{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}\}$.

Step 2: For each candidate do: use Fact 4.6 and Property 4.6 to establish the h–symmetry groups.

The second step of our algorithm, Step 2 has been described already on Example 4.1. The generalization is straightforward. Note that a candidate for h–symmetry could include sets of input variables that belong to different h–symmetries. If this case has to be taken into account, then we need to test the exchangeability of *each* pair of variable sets in the candidate. In other words, we cannot use the property of *one* h–symmetry to be an equivalence relation.

Let us discuss Step 1. What is the best way to determine the list of candidates? One way is the following: first compute all groups of partial symmetries of X with respect to function f [28], then sort these groups by their size. A candidate is the set of all symmetry groups with the same size, except the one including all groups of size one.

However, this kind of selection of candidates is not the best – any cofactor test (and so any cofactor construction) that can be avoided is a bonus point with respect to CPU time and storage requirements. Nevertheless, we already *know* a better way for this. Property 4.4 tells us that the variables of any two partial symmetry groups must have the same signature to be a candidate for h–symmetry. That is why the signature computation is an efficient pre–processing for the selection of candidates for h–symmetry: only aliasing groups are candidates, furthermore only those groups that consist of groups of partial symmetric variables. In this way we are likely to get a smaller set of candidates, such that we can avoid cofactor constructions. Furthermore, the probability that there are variable groups in one candidate that belong to different h–symmetries is less, because the partition of the input variables by signatures is more qualified than just determining same–sized groups of partial symmetric variable groups.

Thus to create the list of candidates for h–symmetry, we use all the signatures introduced in Section 3.2.3 first. If there are aliasing groups, then we select those groups that consist of partial symmetric variable groups, as candidates for h–symmetry.

4.4 Group Symmetries

Now we change our focus to the following kind of symmetry.

Definition 4.4 *Let $f \in \mathcal{B}_{n,1}$ be a Boolean function with n input variables $X = [x_1, x_2, \ldots, x_n]$.*
Let $X_1, X_2, \ldots, X_k \subset X$ be $k > 1$ non-empty and pairwise disjoint subsets of X.
Let $\pi_i \in \mathcal{P}_{|X_i|} \setminus \mathbf{1}$ be a permutation on the input variables of X_i, for all $i = 1, 2, \ldots, k$.
The k groups of input variables are **group symmetric (g–symmetric)** *iff*

1. *$|X_i| > 1$ for all $i \in \{1, 2, \ldots, k\}$*
2. *There is a set $[\pi_1, \pi_2, \ldots, \pi_k]$ of permutations of the variables in $X_1, X_2, \ldots, X_k$, such that applying the permutations π_i to X_i simultaneously for all $i = 1, 2, \ldots, k$ does not change the function f.*

Note, that manipulating just one or a couple of variable subsets X_i may change the function f. The following examples will help to clarify the definition.

Example 4.3 $f = a_0\overline{(x_0 + x_1)} + b_0\overline{(x_2 + x_3)}$

Here, $\{x_0, x_1\}$ and $\{x_2, x_3\}$ are pairs of partial symmetric variables, but there is no h-symmetry between them because of the existence of the input variables a_0 and b_0. However, exchanging $\{x_0, x_1\}$ and $\{x_2, x_3\}$ AND a_0 and b_0 keeps the function invariant. So, there is what we call g–symmetry between the two subsets of input variables, $\{x_0, x_1, x_2, x_3\}$ and $\{a_0, b_0\}$.

Group symmetries are a special kind of $\mathcal{G}$–symmetry, too. So let us consider the partition $\mathcal{A}$ of the n input variables constructed by this $\mathcal{G}$–symmetry.

Property 4.8 *If k subsets of input variables are g–symmetric, then each of these subsets form a subset of the partition $\mathcal{A}$ of the inputs given by this special $\mathcal{G}$–symmetry.*

Proof: This follows from the definition of partition $\mathcal{A}$. □

In our example, $\mathcal{A}$ is as follows:

$$\mathcal{A} = \{\{x_0, x_1, x_2, x_3\}, \{a_0, b_0\}\}.$$

Again, we have a case where signatures will be unable to distinguish between the variables and thus results in the formation of aliasing groups (see Property 4.4).

Our practical experiences on the benchmark set show that this kind of $\mathcal{G}$–symmetry appears relatively often. One well–known example is an n–bit multiplier:

$$x_n x_{n-1} \ldots x_{\frac{n}{2}+1} * x_{\frac{n}{2}} \ldots x_2 x_1.$$

Exchanging $x_{\frac{n}{2}+i}$ and x_i for each $i = 1, \ldots, \frac{n}{2}$ simultaneously keeps the function invariant. The partition $\mathcal{A}$ of these variables is:

$$\mathcal{A} = \{i = 1, 2, \ldots, \frac{n}{2} : \{x_i, x_{\frac{n}{2}+i}\}\}.$$

So there are $2^{\frac{n}{2}}$ possible variable correspondences for an n-bit multiplier after signature computation. Taking the g–symmetry of the n–bit multiplier function into account, this number would decrease by the half, i.e., instead of $2^{\frac{n}{2}}$ possible variable correspondences there would be $2^{\frac{n}{2}-1}$, which is still too large in general.

Moreover, it seems to be very complicated to detect a g–symmetry in general. Here, one major problem is that not *all* possible permutations of the input variables in $X_1, X_2, \ldots, X_k$ have to change function f (see the definition of group symmetry). Thus, to be able to handle general g–symmetries we need a way to detect g–symmetric groups of input variables *and* a way to select those permutations of these variables that keep the g–symmetric function invariant. Considering the very general property of g–symmetry, this looks like a complicated task.

In looking for possible ways to handle g–symmetry we made the following interesting observation. The subsets of input variables that result in a g–symmetry are often connected with each other in the following sense: if we have identified the variables of one of these subsets, then it is possible to identify the variables of the other subsets as well. With this knowledge we have developed a heuristic to distinguish between variables of g–symmetric subsets.

This heuristic works as follows. Given a Boolean function $f \in \mathcal{B}_{n,1}$ let us consider the following situation. There is a partial, permutation independent order of the n input variables of f constructed by using the signatures introduced in Section 3.2.3. Furthermore, the groups of partial symmetric input variables (see Section 4.2) are identified as well as the aliasing groups with h-symmetric groups of input variables (see Section 4.3). Nevertheless, there are still unidentified groups of aliasing variables. For the sake of simplicity, say these groups are the first k in the permutation independent order of variables:

$$\mathcal{A} = \{A_1, A_2, \ldots, A_k, \ldots\}$$

In this situation, we assume that all k groups of aliasing variables are connected by one g–symmetry and use the observation that the groups of input variables that result in a g–symmetry are connected with each other in the special sense just mentioned.

Now, we just *assume* that the input variables of one of these aliasing groups, say A_1, are uniquely identified. Under this assumption, we apply a couple of function signatures (see Section 3.2.3) to each variable of the other $k-1$ aliasing groups, i.e., we construct function signatures that depend not only on those input variables that can be uniquely identified but also on those of aliasing group A_1.

The basic idea of this heuristic is that we may be able to find a unique function signature for each of the input variables in A_2 to A_k in the case of g–symmetry, because of the connection among all k aliasing groups of such a g-symmetry. If this is the case, we can uniquely identify each input variable in the aliasing groups A_2 to A_k. Our practical experiences have shown that often this is indeed the case.

Let us consider an example.

Example 4.4 $f(a_0, a_1, a_2, x_0, x_1, x_2) = a_0\overline{(x_0 + x_1)} + a_1\overline{(x_1 + x_2)} + a_2\overline{(x_2 + x_0)}$

This function f is g–symmetric with respect to the variable groups $\{a_0, a_1, a_2\}$ and $\{x_0, x_1, x_2\}$. Let us see how the heuristics works. We pick up one of these two aliasing groups, say $\{a_0, a_1, a_2\}$, and assume that we can uniquely identify these three variables. Under this assumption we can apply a function signature to the other three variables which may depend on a_0, a_1, and a_2:

$$f_{x_0\bar{x_1}\bar{x_2}} = a_1$$
$$f_{\bar{x_0}x_1\bar{x_2}} = a_2$$
$$f_{\bar{x_0}\bar{x_1}x_2} = a_0$$

Order of $\{a_0, a_1, a_2\}$	Implicit Order of $\{x_0, x_1, x_2\}$
(a_0, a_1, a_2)	(x_2, x_0, x_1)
(a_0, a_2, a_1)	(x_2, x_1, x_0)
(a_1, a_0, a_2)	(x_0, x_2, x_1)
(a_1, a_2, a_0)	(x_0, x_1, x_2)
(a_2, a_0, a_1)	(x_1, x_2, x_0)
(a_2, a_1, a_0)	(x_1, x_0, x_2)

Table 4.1: Group Symmetry

And indeed, the variables x_0, x_1, and x_2 can be uniquely identified with these signatures. Furthermore, we can use the order relation for function signatures to get a unique order of the variables x_0, x_1, and x_2. Applying the lexicographically order relation for Boolean functions the unique order of the three variables is (x_2, x_0, x_1).

However, this unique description is not permutation independent. It depends on the permutation of the input variables in A_1, which is in the case of Example 4.4 (a_0, a_1, a_2). So we need a way to make the information permutation independent. Therefore, we consider the permutation π of the input variables of f which results into the actual unique order of these variables given by the signatures (assuming that there is a unique ordering). In our example, this permutation π is equal to

$$\pi(a_0, a_1, a_2, x_0, x_1, x_2) = (a_0, a_1, a_2, x_2, x_0, x_1).$$

With the help of π, we construct the Boolean function $\hat{f} = f \circ \pi$. This function is stored in a set $\hat{F}$. Now, the same procedure is carried out for each possible order of the input variables in A_1. In the left column of Table 4.1 these orders are listed for the set of variables $A_1 = \{a_0, a_1, a_2\}$ of Example 4.4. Each of these orders implies an order for the variables $\{x_0, x_1, x_2\}$, which are listed in the right column of the table. As described for the first variable order in Table 4.1 the function $\hat{f}$ is computed and stored in $\hat{F}$.

When this has been done for all possible orders of the variables in aliasing set A_1, then the lexicographical smallest of all Boolean functions in $\hat{F}$ is selected. The variable order which belongs to this function and the function itself are the permutation independent information for function f.

Note, that we can also use any of the other sets of aliasing variables, $A_2, \ldots, A_k$. We decided to use one of those sets with minimal size, since this reduces the number of steps necessary to construct the set $\hat{F}$ of Boolean functions.

This kind of $\mathcal{G}$–symmetry has been independently investigated in [30] as well. In this paper, the authors introduce methods to determine maximal sets of partial symmetric input variables of a circuit without ROBDDs, mention that there are other than partial symmetries among input variables and develop an idea to find candidates for this kind of symmetric input variables. This idea is similar to that what we have used for our group–symmetry heuristics. In a later work,

	circuit			number of correspondences			cpu (in sec.) for	
name	#i	#o	#n	*sig*	*+hsym*	*+grsym*	*sig*	*all*
CM150	21	1	64	$\approx 10^7$	$\approx 10^7$	1	0.8	4.9
CM151	12	2	32	216	216	1	0.2	0.5
act2	8	1	12	4	4	1	0.0	0.0
addm4	9	8	225	16	16	1	0.6	1.0
cordic	23	2	86	4	1	1	0.4	0.4
dist	8	5	135	16	16	1	0.3	0.7
ex4	128	28	896	4	4	1	27.4	40.8
i3	132	6	134	$\approx 10^{23}$	1	1	48.2	55.0
lal	26	19	123	24	1	1	0.2	0.2
misg	56	23	109	5184	216	1	1.6	36.4
mlp4	8	8	141	16	16	1	0.3	0.8
mult3	6	6	44	8	8	1	0.1	0.2
mux	21	1	88	$\approx 10^7$	$\approx 10^7$	1	0.9	4.9
mux_cl	11	1	18	216	216	1	0.1	1.4
ryy6	16	1	27	4	1	1	0.1	0.1
sao2	10	4	123	16	16	1	0.2	0.6
t481	16	1	80	331776	331776	331776	0.6	0.6
ts10	22	16	271	720	720	720	2.1	2.2
term1	34	10	616	$\approx 10^8$	$\approx 10^8$	1	12.7	36.0

Table 4.2: Benchmarks with $\mathcal{G}$–Symmetries

the authors consider the permutation equivalence problem. We briefly described their method to handle the permutation problem in Section 3.3. In this paper, they mention that it is advantageous to detect as many of that what we call group–symmetric variables as possible before attempting to solve the permutation equivalence problem. However, they do not apply the idea introduced in [30] to do this.

We have implemented *our* approach under the assumption that there is only one $\mathcal{G}$–symmetry between the aliasing groups and tested it for the examples of our benchmark set.

4.5 Experimental Results

Similar to our experiments with signatures, we implemented the ideas presented in this chapter in the Berkeley SIS–system, release 1.3 [34]. All experiments were done on a SUN Sparcstation 10 with 64 MB RAM.

Table 4.2 lists the 8% of benchmarks with aliasing after signature computation which are the subject of our research regarding to $\mathcal{G}$–symmetries. In this table, a description of each circuit (name, number of inputs, outputs, and ROBDD nodes) is followed by the number of possible correspondences after

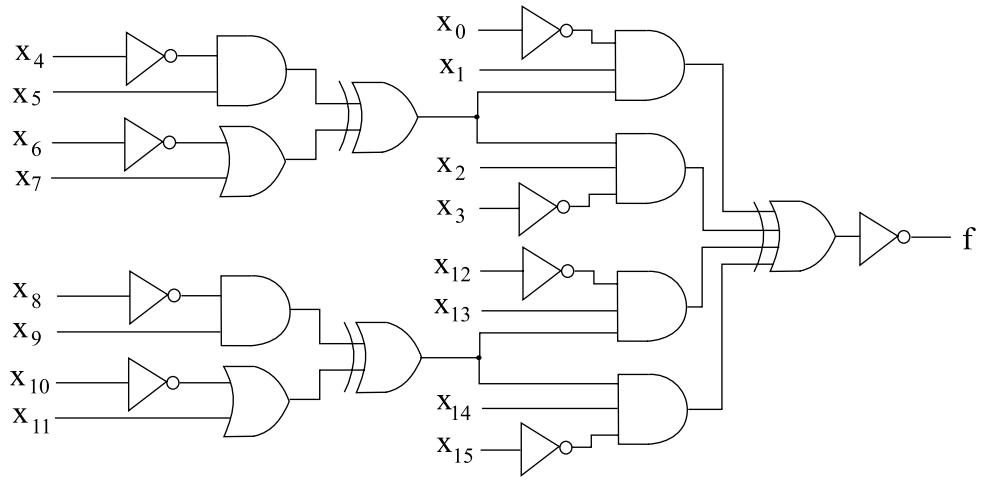

Figure 4.1: Description of Benchmark Circuit *t481*

using signatures only (*sig*), signatures and the algorithm to determine h–symmetry (+*hsym*), and finally, signatures, the h–symmetry algorithm and the heuristic for g–symmetries (+*grsym*). The last two columns include the CPU–time in seconds needed to apply the signatures and to apply the signatures as well as all three heuristics on a SUN Sparcstation 10.

As can be observed, the results are promising. Approx. 20% of all benchmarks with aliasing groups include h–symmetry. Furthermore, in all but one of these cases (namely *misg*) this is the only reason for the existence of aliasing groups. The CPU–time necessary for the detection of h–symmetry after signature computation is very promising. In many cases where h–symmetry could not be detected there is no measurable CPU–time (i.e., 0.0 sec. in the table). Here, the reason is that there are no classes of partial symmetries in the aliasing groups, so testing this necessary condition is enough to establish the fact that there is no h–symmetry. Thus, no cofactor computation is necessary in these cases. However, even in our worst case with respect to the CPU–time, namely *i3*, it only takes 6.8 seconds to establish all h-symmetries. The reason for this fact is that the variable groups that are tested within h–symmetry are rather small (i.e., size 2, 3, or 4), so that the quadratic factor of the complexity ($O(k^2|V|)$, see Section 4.3) is not too expensive. Note that for *i3* the results regarding to the correspondence possibilities is especially impressive, too. After signature computation there are still approx. 10^{23} correspondence possibilities for the input variables of *i3* left, but all come from a permutation of h–symmetric variable groups.

For all but two of the rest the *grsym*–algorithm was successful. These two examples are *ts10* and *t481*. Our investigations have shown that the aliasing variables of both examples are involved in a special symmetry as well. Unfortunately, we have not been able to automate the detection for these cases.

Benchmark *ts10* includes a group of six aliasing variables with rotational symmetry. We discuss rotational symmetry in the next section.

The other example is the benchmark circuit *t481*. Figure 4.1 shows a description of it. This example

has 16 input variables, $X = [x_0, x_1, \ldots, x_{15}]$, and one output. By using the signatures introduced in this thesis the set of inputs is divided into the partition

$$\mathcal{A} = \{\{x_4, x_7, x_8, x_{11}\}, \{x_5, x_6, x_9, x_{10}\}, \{x_0, x_3, x_{12}, x_{15}\}, \{x_1, x_2, x_{13}, x_{14}\}\}.$$

Considering the structure of *t481* we can observe that there is indeed a group symmetry between the four groups of aliasing variables in partition $\mathcal{A}$: for instance, applying the permutation

$$\pi(x_0, \ldots, x_{15}) = (x_{12}, x_{13}, x_{14}, x_{15}, x_8, x_9, x_{10}, x_{11}, x_4, x_5, x_6, x_7, x_0, x_1, x_2, x_3)$$

does not change the function f. However, non of the four groups of aliasing variables can be used to identify *all* other aliasing variables uniquely as described in Section 4.4. In other words, our heuristic to assume that the variables of one of these aliasing groups can be uniquely identified and then to try to distinghuish between the others does not work for *t481*. The reason for this fact may be that there are permutations of the inputs of *t481* that keep this function invariant but do not involve *all* aliasing groups. For example, exchanging the variable x_0 with x_3 and the variable x_1 with x_2 does not change *t481* although just the last two aliasing groups are involved in this permutation. In other words, the four aliasing groups of *t481* are not *completely* connected with each other, thus the heuristics introduced here cannot help to distinguish between the variables completely. The characteristic of circuit *t481* underlines once more the complexity of general $\mathcal{G}$–symmetry.

4.6 The Variety of $\mathcal{G}$–Symmetry

We introduced three kinds of $\mathcal{G}$–symmetry that cover a wide range of symmetries appearing in practice. For these symmetries we could provide efficiently working solution paradigms for the permutation equivalence problem P_π.

Nevertheless, it seems to be not possible to find a general solution paradigm which allows us to handle $\mathcal{G}$–symmetry efficiently because of the generality of this function property. Each kind of $\mathcal{G}$–symmetry seems to demand a special procedure to handle it. Although we feel that the $\mathcal{G}$–symmetries introduced in this chapter are those mostly appearing in practice, other symmetries may appear as well.

Here is one of these $\mathcal{G}$–symmetries for that we have even an example in our benchmark set (This is *ts10*. See the previous section). Consider the following function:

$$f = x_1x_2 + x_2x_3 + x_3x_4 + x_4x_5 + x_5x_1.$$

It is obvious that rotating the five variables in function f (i.e., applying the permutation $\pi = (x_5, x_1, x_2, x_3, x_4)$ to the inputs of f) does not change this function. So it is a kind of $\mathcal{G}$–symmetry, too. All five input variables of the Boolean function f will form one aliasing group that cannot be refined by using signatures. Furthermore, there are no partial, no hierarchical, as well as no group symmetries between these input variables. We call this kind of symmetry *rotational symmetry* and define it as follows.

Definition 4.5 *Let $f \in \mathcal{B}_{n,1}$ be a Boolean function.*
Let $Y = [y_1, y_2, \ldots, y_k] \subseteq X$ be a subset of the input variables of f.
Let $\pi = (y_k, y_1, y_2, \ldots, y_{k-1})$ be a permutation of $\mathcal{P}_k$.
f is **rotational symmetric (r–symmetric)** *iff*

1. *$k \geq 3$,*
2. *f is not partial symmetric in Y, and*
3. *f does not change on applying the permutations π to the variables of Y.*

Note that also the permutation π^{-1} keeps f invariant as well as applying π more than once, since the set $\mathcal{G} \subset \mathcal{P}_n$ which constructs an r–symmetry is a group. Similar to group symmetry, r–symmetry is neither easy to detect nor easy to handle. Furthermore, the heuristic applied for the group symmetries, (i.e., assume one of the input variables of Y has been uniquely identified by a signature, then try to use function signatures to distinguish between the others as well) does not work very successful for rotational symmetry, as our experiments have shown. Rotational symmetry illustrates the difficulties that may appear in general with $\mathcal{G}$–symmetries.

However, partial symmetries, hierarchical symmetries, and group symmetries seem to be the most common $\mathcal{G}$–symmetries appearing in practice. So, the algorithms presented for handling these symmetries have direct practical impact on the complete solution of the permutation problem.

Chapter 5

The Latch Correspondence Problem

In this chapter, we show that the signature–based methods used for solving the permutation equivalence problem can be easily applied to a problem in *sequential* logic verification. This is the problem of establishing the unknown correspondence for the latch variables (memory elements) of two sequential circuits that have the same encoding of their states. If a correspondence of the latches in the two circuits can be established, then the verification problem reduces to a combinational equivalence check on the combinational circuit defined by the latch boundaries. There are several cases in which the correspondence may exist but is unknown. For example, the names for the latches used in a specification may be different from those used in the implementation due to modifications made by synthesis tools. The combinational equivalence check on the primary output and next state functions can be done using ROBDDs. However, without the correspondence of latches we cannot check if the corresponding ROBDDs of the combinational parts of the two sequential circuits are the same. Thus, we need to establish a correspondence first, similar to the combinational permutation equivalence problem.

The application of signature based methods for this problem is straightforward: derive a signature for each latch variable in order to uniquely identify this latch. In [36], the authors have used this method to identify corresponding latch variables in order to be able to simplify the product machine traversal for sequential verification. This is especially useful when the state encoding of the two machines that have to be verified are identical, but the state variable correspondence is not known. Here, they have used signatures for input variables known from literature [9] to do this and have made the observation that sometimes the task still remains too complex. We think that the ideas presented in this thesis are able to improve upon this, since we do not just combine different input signatures but also develop novel signatures which are especially suited for the latch equivalence problem. Thus, these new signatures give better results, i.e., the set of pairs of latch variables which are candidates for correspondence are determined more precisely.

We start with a problem description in **Section 5.1**. In **Section 5.2**, we describe the differences between establishing unknown input correspondence and establishing unknown latch correspondence, introduce special signature functions for latch variables that can be easily computed on ROBDDs, and explain a solution paradigm which uses these signature functions on an example. Then, we

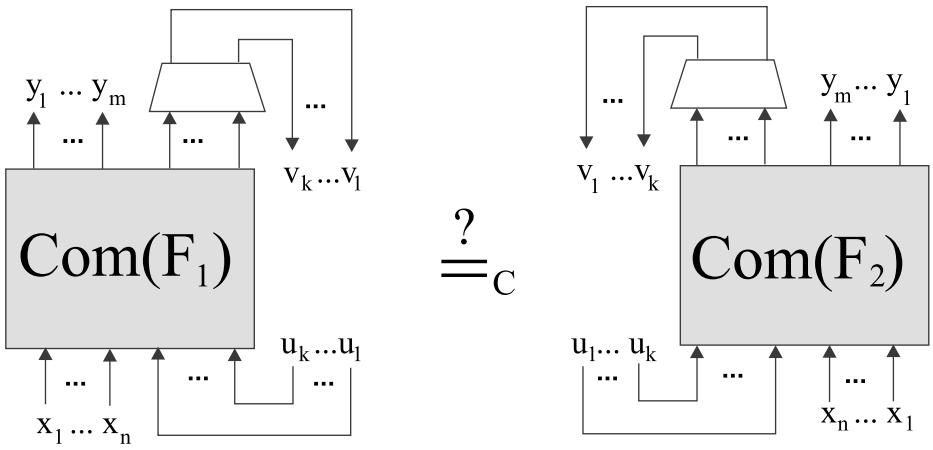

Figure 5.1: The Latch Equivalence Problem

demonstrate the utility of this approach by experimental results in **Section 5.3** and discuss the influence of symmetries in **Section 5.4**. The main results of this chapter are published in [27].

5.1 Problem Description

Let $\mathcal{F}_{n,m,k}$ be the set of all synchronous sequential circuits with n primary input variables, $X = [x_1, x_2, \ldots, x_n]$, m primary output variables, $Y = [y_1, y_2, \ldots, y_m]$, and k latch variables, $L = [l_1, l_2, \ldots, l_k]$. We can consider just the combinational part of a sequential circuit $F \in \mathcal{F}_{n,m,k}$. Therefore, let $U = [u_1, u_2, \ldots, u_k]$ be the set of latches considered as input variables of F, and let $V = [v_1, v_2, \ldots, v_k]$ be the set of latches considered as output variables of F. Then we get the combinational part of F, $Com(F)$ as a Boolean function with $n + k$ input and $m + k$ output variables and can define the *latch permutation equivalence problem.*

Definition 5.1 *The* latch permutation equivalence problem*, L_π (also refered to as the latch correspondence problem) is defined as follows:*

Let be F_1 and $F_2 \in \mathcal{F}_{n,m,k}$. Does there exist a correspondence C between the latch variables of F_1 and F_2, such that the two synchronous sequential circuits have their combinational parts equivalent using this correspondence:

$$Com(F_1) =_C Com(F_2)?$$

This is illustrated in Figure 5.1. Here, we assume that the correspondences between the primary inputs and outputs of F_1 and F_2 is known. The similarity of this problem to the combinational permutation equivalence problem is obvious. We need to find a unique and permutation independent description for each latch variable. The difference is that each latch variable has to be considered as input *and* as output variable. So let us modify the approach used to handle the combinational problem.

5.2 Signatures

In Chapter 3, we define a *signature* as a description of an input variable that is independent of any permutation of all inputs of a Boolean function $f \in \mathcal{B}_{n,1}$. Thus, it can help to solve the combinational permutation equivalence problem, P_π, as follows. With the help of signatures each input variable can be identified independent of permutation, i.e., any possible correspondence between the input variables of two Boolean functions is restricted to a correspondence between variables with the same signature. So, if each variable of a Boolean function f has a unique signature, then there is at most one possible correspondence to the variables of any other Boolean function.

This concept can be used to attack the *latch correspondence problem*, L_π. Here we need to test whether there exist a correspondence between the *latches* of two synchronous sequential circuits, such that the combinational logic used to define them is equivalent under this correspondence.

Let us characterize this new problem and underline the differences from P_π. We assume that the correspondence between the primary inputs of the two circuits is known. So we have the first piece of additional information in comparison to P_π. Similar to P_π, we also assume that we know the correspondence between the primary outputs. These outputs are Boolean functions that depend on the primary input variables as well as the input variables due to the latches. So, we can use the primary outputs to compute signatures for the latch input variables. This is done in exactly the same manner as for the combinational problem P_π. In the following, we will call those signatures *input signatures* and make some remarks regarding them in **Section 5.2.2**. However, latches appear not only as input variables, but also as output variables. So, the most important difference between the combinational problem P_π and the sequential problem L_π is that we can use *latch output signatures* in addition to identify the latches independent of permutation. A latch output signature is a description of a latch, considered as an output variable, that is independent of the permutation of the latches as output as well as input variables. We will explain this in more detail in **Section 5.2.3**.

In **Section 5.2.4**, we demonstrate how we establish a unique possible correspondence for the latches for one of the benchmarks (see Section 5.3), namely *ex4.slif*. A general solution paradigm is given in the next section.

5.2.1 Solution Paradigm

Let us start with describing the solution paradigm on an example of two circuits, F_1 and F_2, with four latch variables $[l_1, l_2, l_3, l_4]$. Let the latches considered as input variables be the variables $[u_1, u_2, u_3, u_4]$, and the latches considered as outputs be the variables $[v_1, v_2, v_3, v_4]$.

First, we look at the latches as input variables and compute an input signature $s_I(F_1, u_i)$ and $s_I(F_2, u_i)$ for $i = 1, 2, 3, 4$ with respect to the primary outputs of F_1 and F_2 respectively. This is done in exactly the same manner as described for the combinational permutation problem P_π in

Section 3.2.2. So let us suppose that we have two lists of input signatures that contain the same elements:

$$\mathcal{L}_I(F_1) = [2, 1, 3, 2] \qquad \text{and} \qquad \mathcal{L}_I(F_2) = [1, 2, 2, 3].$$

Now, based on these signatures we can establish directly that any correspondence between the latches of F_1 and F_2 has to identify latch l_2 of F_1 with latch l_1 of F_2, and latch l_3 with latch l_4. Thus, a correspondence between the latches of F_1 and F_2 has been partially established. In this case, there is a possible correspondence between the latches of F_1 and F_2 for latch equivalence. However, both latch variables, l_1 and l_4 of F_1 could correspond to l_2 as well as to l_3 of F_2. In other words, aliasing occurs between the latch variables l_1 and l_4 of F_1 and between the latch variables l_2 and l_3 of F_2.

Since latch variables are not just input variables but also output variables, we have a second possibility to distinguish between the latches with aliasing. We can consider them as output variables and use latch output signatures to try to uniquely identify them. In our example, we have to do this for the latch output variables v_1 and v_4 of F_1, and for the variables v_2 and v_3 of F_2. So let us assume that

$$\mathcal{L}_O(F_1, [v_1, v_4]) = [4, 5] \qquad \text{and} \qquad \mathcal{L}_O(F_2, [v_2, v_3]) = [5, 4].$$

Now we are done, since we are able to establish a unique correspondence between the latch variables l_1 and l_4 of F_1 and the variables l_2 and l_3 of F_2 as well: latch l_1 of F_1 has to correspond to latch l_3 of F_2, and latch l_4 of F_1 to latch l_2 of F_2.

This gives us the only possible unique correspondence between the latch variables. Furthermore, if we require that signatures have to be elements of an ordered set, we can establish a unique permutation independent order of the latches. In the case of our example, that would be the order $[l_2, l_1, l_4, l_3]$ for the latches of F_1, and the order $[l_1, l_3, l_2, l_4]$ for the latches of F_2.

This example demonstrates that the possibility to consider latch variables as inputs as well as outputs improves our chances to get a unique correspondence between the latches for a possible latch permutation equivalence of two circuits. Of course, it is not guaranteed that such a unique possible correspondence will be obtained. It strongly depends on the input and latch output signatures that we use and on the characteristics of the latches. Thus we focus on the following questions in the rest of this chapter. What input and latch output signatures can we use? What special properties of the problem do we need to take care of? And, what special properties of the latches may cause problems for our signature approach?

5.2.2 Input Signatures

Only some modifications need to be made when considering signatures for input variables as introduced in Section 3.2.3 for use in L_π. Let us consider $F \in \mathcal{F}_{n,m,k}$ again. It has m primary output

functions, $[y_1, \ldots, y_m]$, that we can uniquely identify (see Section 5.1). These output functions depend on the primary input variables $x_1, \ldots, x_n$ and on the latch input variables $u_1, \ldots, u_k$. So we can use them step by step, starting with y_1 up to y_m, to compute input signatures for the variables $u_1, \ldots, u_k$.

Let us describe this on an example. Consider F with two input variables, x_1 and x_2, two outputs, y_1 and y_2, and four latches, l_1, l_2, l_3, and l_4. Then, the two output functions y_1 and y_2 are functions that depend on the set of input variables $[x_1, x_2, u_1, u_2, u_3, u_4]$. Now, let us use the cofactor satisfy count signature function in order to try to separate u_1, u_2, u_3 and u_4. In a first step, we compute this signature using output function y_1. Let us assume we get the partition $[\{u_1, u_3\}, u_4, u_2]$. This means that we can uniquely identify the latch input variables u_2 and u_4 and thus the latches l_2 and l_4. Furthermore, we get a unique partial order of the latches: $[\{l_1, l_3\}, l_4, l_2]$. Now, we can use output function y_2 in a second step to try to distinguish between the latch input variables u_1 and u_3 as well.

Note, that we *cannot* use the latch output variables, $v_1, \ldots, v_k$, for that purpose, since *these* outputs are not uniquely identified as yet.

5.2.3 Latch Output Signatures

Similar to an input signature, we can define a signature for a latch output variable v_i of $F \in \mathcal{F}_{n,m,k}$.

A *signature for a latch output variable* is a value, a vector of values, or a function that provides special information about this latch output variable. This information has to be independent of any permutation of the latch variables of F. In general, such a latch considered as an output depends not only on the *primary* inputs of this sequential circuit, but also on the *latches* considered as input variables. That is why a latch output signature has to be independent of the permutation of these *latch* input variables as well. Furthermore, similar to an input signature, it has to be an element of an ordered set.

Now let us develop latch output signatures that could be useful. Therefore, let us consider the example circuit $F \in \mathcal{F}_{n,m,k}$ with two input variables, x_1 and x_2, two outputs, y_1 and y_2, and four latches, l_1, l_2, l_3 and l_4, again. Suppose, we could not distinguish between the latches l_1 and l_3 considering them as input variables u_1 and u_3. Thus, we still have the partial order of the latches, $[\{l_1, l_3\}, l_4, l_2]$ (see previous subsection). Now, let us consider these latches as *output* variables of F, v_1 and v_3. These output variables represent Boolean functions, that depend on all primary input variables and on the latches considered as input variables.

For simpler notation let us denote the considered latch output variable function as f. We can apply the following kinds of latch output signatures to it.

5.2.3.1 Simple Output Signatures

Signature functions that can be directly developed by applying input signature functions as introduced in Section 3.2.3 to latch output variables are called *simple signature functions* in the

following.

Such a signature function could be for instance:

1. the satisfy count of the output function, $|f|$,
2. the vector of the cofactor satisfy count signatures of the input variables of f sorted by the size of the satisfy counts,

$$sort(|f_{x_1}|, |f_{x_2}|, \ldots, |f_{u_1}|, \ldots, |f_{u_k}|),$$

3. the breakup signature with respect to function f and origin $\mathcal{O} = [0, 0, \ldots, 0]$,

$$[|f^0|, |f^1|, \ldots, |f^{n+k}|].$$

These signature functions satisfy all necessary properties since they are output signatures for f (i.e., for v_1 and v_3, respectively) that are also independent of the permutation of the latch input variables, u_1, u_2, u_3, and u_4.

If these simple output signatures do not break the tie, we have to apply some stronger latch output signatures.

5.2.3.2 Function Signatures for Latch Outputs

Here we use the fact that we can uniquely identify the primary input variables of the circuit. In our example, these are x_1 and x_2. So, any subfunction of a latch output variable f that is independent of the *latch* input variables is a latch output signature. There are several possibilities for this kind of subfunction. Let us consider our example F with the two primary input variables, x_1 and x_2, and the four latch variables considered as inputs, u_1, u_2, u_3, and u_4, again. Subfunctions of a primary output function f of this example that only depend on x_1 and x_2 are for instance $f_{u_1 u_2 u_3 u_4}$, $f_{\bar{u_1}\bar{u_2}\bar{u_3}\bar{u_4}}$, $\forall_{u_1 u_2 u_3 u_4} f$, and $\exists_{u_1 u_2 u_3 u_4} f$.

Such a function contains special information about f, and it is independent of the permutation of the latch output variables as well as of the latch input variables of F. We call this kind of signature a *function output signature*.

In the example, we need to compute function output signatures for the latch output variables v_1 and v_3. Let us consider v_1 and denote the function computed by it as f again.

Furthermore, we can extend the idea of the function signature using exactly the idea of constructing the function signatures for input variables (see Section 3.2.3). For each latch which is uniquely identified at this point, the corresponding latch input variable can be uniquely identified as well. In our example, this is the case for the latches l_2 and l_4. So, subfunctions of v_1 and v_3 that depend on the primary inputs x_1 and x_2 *and* on the latch input variables u_2 and u_4 are latch output signatures — with one minor restriction: we need to reorder the latch input variables u_2 and u_4 independent

of permutation in these subfunctions. Therefore, let us use the order of the latches established by previously used signatures. In the case of our example, the permutation which reorders the input variables of such a subfunction would be: $\pi = (x_1, x_2, u_4, u_2)$ (see the latch order of our example in the previous section). Such a reordered subfunction of v_1 and v_3 has all the properties to be a latch output signature, and we can again try to distinguish between v_1 and v_3. This process can be iterated as long as we can uniquely identify at least one more latch. That is why we call this extended function signature the *iterating function signature*. Our practical experiments have shown that this is a very powerful signature.

5.2.3.3 Canonical Order Signature

There is another strong latch output signature. We call it the *canonical order signature*. For that we use the methods introduced to handle the combinational permutation equivalence problem. Remember, these methods can be used to establish a canonical and permutation independent ordering of the input variables of a Boolean function.

Let us consider the Boolean function f which represents a latch output variable function of F again. On the input variables of this function, the methods used in Chapter 3 and in Chapter 4 can be applied in order to find a canonical permutation independent variable ordering. Suppose $\pi \in \mathcal{P}_{n+k}$ is a permutation of the latch input variables of f which constructs this canonical order, then the canonical order signature is the following function:

$$f^{can} = f \circ \pi.$$

This function is independent of the permutation of *any* input variable of f. Thus it is a latch output signature. Note that finding this canonical ordering can be restricted to the latch input variables because the primary input variables of f are uniquely identified by assumption.

5.2.4 An Example

In this section, we illustrate our solution paradigm with benchmark *ex4.slif* from the LGSynth91 benchmark set [1]. We will not use all signature functions here. However, the general paradigm of finding a unique permutation independent order of the latch variables, as described in the previous sections, will become clear.

Benchmark *ex4.slif* is the description of a sequential circuit with 6 input variables, $v_0, v_1, \ldots, v_5$, 9 output variables, $v20.14, v20.15, \ldots, v20.22$, one clock variable, and 14 latches. Let us call these latches $l_1, l_2, \ldots, l_{14}$ in the order of their appearance in the benchmark description. For more details please see the benchmark description in the LGSynth91 set of benchmarks.

We begin with computing a *simple output signature*: the satisfy count, $|v_i|$ of the latch variable l_i considered as output v_i of the actual circuit. Here, we get the following results:

$$\begin{array}{ll} |v_1| = 0 & |v_2| = 288 \\ |v_3| = 128 & |v_4| = 128 \\ |v_5| = 192 & |v_6| = 64 \\ |v_7| = 128 & |v_8| = 128 \\ |v_9| = 96 & |v_{10}| = 128 \\ |v_{11}| = 128 & |v_{12}| = 192 \\ |v_{13}| = 64 & |v_{14}| = 128. \end{array}$$

Using these signatures we get a partial, permutation independent order of the latches:

$$l_1 \quad \{l_6, l_{13}\} \quad l_9 \quad \{l_3, l_4, l_7, l_8, l_{10}, l_{11}, l_{14}\} \quad \{l_5, l_{12}\} \quad l_2.$$

As you can see, there are three aliasing groups of latches, one of size 7 and two of size 2, for which we have to do further computations. We now consider these latches as input variables, u_i, and use an *input signature* to try to distinguish between these latches. For doing that, we take one primary output function, $v20.i$, after the other and compute the selected input signature with respect to this output function.

Let us take the *cofactor satisfy count signature* as an input signature and start with primary output function $v20.14$. We get the following results for the three aliasing groups:

*group*1	*group*2	*group*3
$\lvert v20.14_{u_6}\rvert = 256$	$\ldots_{u_3} = 0$	$\ldots_{u_5} = 128$
$\ldots_{u_{13}} = 256$	$\ldots_{u_4} = 0$	$\ldots_{u_{12}} = 128$
	$\ldots_{u_7} = 0$	
	$\ldots_{u_8} = 256$	
	$\ldots_{u_{10}} = 0$	
	$\ldots_{u_{11}} = 0$	
	$\ldots_{u_{14}} = 256$	

Based on this we see that the latches of aliasing group 1 and group 3 cannot be distinguished using the cofactor satisfy count signature with respect to primary output function $v20.14$. However, we can split up group 2 in the subgroups $\{l_3, l_4, l_7, l_{10}, l_{11}\}$ and $\{l_8, l_{14}\}$. So, we get a finer partial order for the latches:

$$l_1 \quad \{l_6, l_{13}\} \quad l_9 \quad \{l_3, l_4, l_7, l_{10}, l_{11}\} \quad \{l_8, l_{14}\} \quad \{l_5, l_{12}\} \quad l_2.$$

At this point we have four aliasing groups of latch variables, namely three of size 2 and one of size 5. Now, we can continue with computing the cofactor satisfy count signatures with respect to primary output function $v20.15$, analyzing the new situation with respect to those signatures (i.e., is there a finer partition of the latches?), and so on for all primary outputs — until there is a unique order of the latches.

However, even after using all the primary output variables, we still have just a partial order of the latches:

$$l_1 \quad \{l_6, l_{13}\} \quad l_9 \quad \{l_4, l_{11}\} \quad l_7 \quad l_3 \quad l_{10} \quad l_8 \quad l_{14} \quad \{l_5, l_{12}\} \quad l_2.$$

Now, there are three aliasing groups of size 2. So, let us try and see how the more complex output signatures work. At first, we will use a vector of *function signatures*. This works as follows. We consider the latch variables l_i of the three aliasing groups as output variables v_i again. Then we compute restrictions of such an output variable v_i (output function f^{v_i}), that are independent of the latch input variables. For the purpose of this example, let us take the following two functions:

$$(f^{v_i}_{u_1 u_2 \ldots u_{14}}, f^{v_i}_{\bar{u_1} \bar{u_2} \ldots \bar{u_{14}}}).$$

This is a vector of function signatures for each latch variable of the three remaining aliasing groups of our benchmark circuit. Unfortunately, there is no difference between the function signatures of l_6 and l_{13}, l_4 and l_{11}, and l_5 and l_{12}, respectively. (In our experiments we use six different function signatures.) Applying the *canonical order signature* now, helps to distinguish between latch l_4 and l_{11}, and we get the following partial order for the latches:

$$l_1 \quad \{l_6, l_{13}\} \quad l_9 \quad l_{11} \quad l_4 \quad l_7 \quad l_3 \quad l_{10} \quad l_8 \quad l_{14} \quad \{l_5, l_{12}\} \quad l_2.$$

Let us try the *iterating function signature* next. Here, we use the same functions as for the function signature described above, but with one important difference. The restrictions of an output function f^{v_i} that we compute, is not independent of *all* latch input variables, but only of those that are still in aliasing groups. So, a vector of those restricted functions for latch variable l_i of one of our aliasing groups is:

$$(f^{v_i}_{u_5 u_6 u_{12} u_{13}}, f^{v_i}_{\bar{u_5} \bar{u_6} \bar{u_{12}} \bar{u_{13}}}).$$

However, we need to reorder the unique latch input variables in the restricted functions in order to get an iterating function signature (see Section 5.2). The new order is the permutation independent and unique suborder that we get by the established order of our latch variables:

$$u_1 \quad u_9 \quad u_{11} \quad u_4 \quad u_7 \quad u_3 \quad u_{10} \quad u_8 \quad u_{14} \quad u_2.$$

By reordering the latch input variables in the two functions described above, we get an iterating function signature for latch variable l_i:

$$(g^{v_i}_{u_5 u_6 u_{12} u_{13}}, g^{v_i}_{\bar{u_5} \bar{u_6} \bar{u_{12}} \bar{u_{13}}}),$$

and as our experiments have shown, we finally can establish a unique permutation independent order of all latch variables with the help of these output signatures:

$$l_1 \quad l_6 \quad l_{13} \quad l_9 \quad l_{11} \quad l_4 \quad l_7 \quad l_3 \quad l_{10} \quad l_8 \quad l_{14} \quad l_5 \quad l_{12} \quad l_2.$$

5.3 Experimental Results

We implemented the signatures and ideas presented in the previous sections in the Berkeley SIS–system, release 1.3, in C [34]. To get an understanding about the quality of the signatures we tested a set of 97 benchmarks. These are all *fsmexamples* and all *smexamples* from the LGSynth91 benchmark set for which we could construct the ROBDDs [1]. The experiments were conducted on a SUN Sparcstation 10 with 64 MByte RAM.

The first experiment conducted was to determine the best signature order. There are several input and latch output signatures that we can use to get a unique possible correspondence of the latches. The best order of these signatures is the one with the smallest CPU–time required to obtain unique correspondence. We tried the following orders:

- Use all input signatures first and then all latch output signatures.
- Use all latch output signatures first and then all input signatures.
- Use all those signatures first, for which no exhaustive ROBDD constructions are necessary – do this with input priority and with latch output priority.

We observed that using the latch output signatures first seems to be the better choice. Thus, we decided to use the following order for further investigations.

First compute three simple latch output signatures on each latch output variable f:

- $|f|$,
- $sort(|f_{x_1}|, \ldots, |f_{u_k}|)$,
- breakup signature for f.

Then use some input signatures introduced in Section 3.2.3. These are the satisfy count signatures and the breakup signatures. And finally use the more qualified latch output signatures: function signature, canonical order signature, iterating function signature.

For the *fsmexamples* of the LGSynth91 benchmark set, the signature procedure could establish a unique possible correspondence for all latches of all benchmarks in less than 2 seconds. Here, using the simple latch output signatures was enough, except for benchmark *shiftreg.kiss2*. For this example it was necessary to use the cofactor satisfy count signature function for input variables as well. So, let us concentrate on considering the results of our investigations for the *smexamples*. Table 5.1 presents these results. The first 5 columns include the benchmark characteristics (name, number of inputs, outputs, and latches as well as the number of ROBDD nodes). The next 4 columns show the number of possible correspondences of the latches after using the simple latch output signatures (*so-sigs*), then after using the input signatures in addition (+*i-sigs*), next after using

name	#i	#o	#l	#bdd	# of correspondences with				cpu time
					so-sigs	+i-sigs	+fc-sig (+can-sig)	+itf-sig	(in sec.)
clmA	382	82	33	2211	1				91.2
clmB	382	369	33	1998	1				95.3
daio	2	3	4	21	2	2	1		0.0
ex2	3	3	19	406	$2^4 \cdot 4! \cdot 6!$	$2^7 \cdot 3!$	256	1	10.1
ex3	3	3	10	143	6	1			0.2
ex4	7	10	14	252	$4 \cdot 7!$	8	8(4)	1	2.8
ex5	3	3	9	128	4	1			0.2
ex6	5	8	9	162	1				0.2
ex7	3	3	10	159	4	1			0.2
MinMax4	7	4	12	523	4!	1			0.2
MultiplierB_16	17	1	30	124	$14! \cdot 15!$	$13! \cdot 14!$	1		3.1
MultiplierB_32	32	1	62	249	1				25.4
s1196.bench	14	14	18	2822	1				0.2
s1238	15	15	18	2840	1				0.2
s1423.bench	17	5	74	13657	$4 \cdot 3! \cdot 4!$	$3! \cdot 4!$	6	1	216.0
s1488.bench	8	19	6	489	1				0.0
s1494.bench	8	19	6	484	1				0.0
s208.1.bench	10	1	8	71	1				0.1
s208	12	3	8	82	1				0.1
s27.bench	4	1	3	12	1				0.0
s298.bench	3	6	14	118	2	2	1		0.2
s344.bench	9	11	15	180	$3! \cdot 4!$	4!	1		0.7
s349.bench	9	11	15	178	$3! \cdot 4!$	4!	1		0.7
s382.bench	3	6	21	176	2	2	1		0.3
s386.bench	7	7	6	142	1				0.0
s400.bench	3	6	21	176	2	2	1		0.3
s420.1.bench	18	1	16	203	1				0.7
s444.bench	3	6	21	191	2	2	1		0.3
s510.bench	19	7	6	185	1				0.0
s526.bench	3	6	21	169	2	2	1		0.7
s526n	4	7	21	164	2	2	1		0.7
s641.bench	35	23	19	777	4	1			0.7
s713.bench	35	23	19	777	4	1			0.6
s820.bench	18	19	5	309	1				0.0
s832.bench	18	19	5	309	1				0.0
s838.1.bench	34	1	32	659	1				4.4
s838	36	3	32	323	1				4.6
s953	17	24	29	508	1				0.3
sbc	40	56	28	1689	2	1			0.7

Table 5.1: The Quality of Signatures in L_π

function and canonical order signature (+*fc-sig (+can_sig)*), and finally after using the iterating function signature (+*itf-sig*). The last column includes the CPU time in seconds needed to establish a unique possible correspondence by using these signatures functions. The time 0.0 seconds means that there was no measurable cpu–time. In Section 5.2.4, we demonstrated this process on the

specific benchmark *ex4.slif*.

The results are very promising. We could establish a unique possible correspondence for all latches of each benchmark of our actual set. For about 49% of the benchmarks it was enough to use simple output signatures in order to uniquely identify each of the latches. Applying input signatures helps to solve the problem for a further 18%. And finally, the function signature and the more exhaustive iterating function signature establish a unique possible correspondence for the latches for 33% of all benchmarks. Note, that the canonical order signature could uniquely identify exactly one latch output, that is for *ex4.slif*. Note also that the CPU times are very modest.

5.4 Symmetries in Latch Equivalence

However, applying signatures to solve the latch equivalence problem does not guarantee a complete solution. Similar to the combinational permutation problem, P_π, this approach will fail if any kind of latch symmetry appears in a circuit. Moreover, those symmetries are likely to appear in practice. For example, circuits generated from high level descriptions are likely to have many symmetric (in fact equivalent) state variables. In this case, signatures cannot help (see Chapter 4). Nevertheless, we can extend the signature approach by considering latch symmetry and applying methods used in Chapter 4.

Two latches l_1 and l_2 of $F \in \mathcal{F}_{n,m,k}$ are *symmetric* iff their variables u_1 and u_2 are input symmetric with respect to all primary outputs $y_1, \ldots, y_m$, and with respect to the latch outputs $v_3, \ldots, v_k$. Furthermore, the two functions $v_1(\ldots, u_1, u_2, \ldots)$, and $v_2(\ldots, u_2, u_1, \ldots)$ have to be equal. If this is the case, then the two latches l_1 and l_2 can be exchanged in F without changing the circuit function. It is obvious that this symmetry between two latches can be easily tested by using the known tools in order to test the symmetry of the latch input variables [28]. ROBDDs are used to compare the two latch output functions. It is possible that other kinds of latch symmetry can be defined and handled in a similar way.

Chapter 6

Conclusion

In this thesis, an approach to handle the combinational permutation independent Boolean comparison problem is presented, i.e., the problem whether two Boolean functions f and g are equivalent independent of the permutation of their input variables. The approach concentrates on establishing a possible correspondence for equivalence between the input variables of f and g. This problem is $\mathcal{NP}$–hard. So heuristic solutions are necessary. The approach introduced here uses signatures for an input variable of a Boolean function f to handle the problem. In this context, a signature is a special information about an input variable of a Boolean function f which is independent of the permutation of all input variables of f. The data structure which is used to represent Boolean functions is the reduced ordered binary decision diagram (ROBDD).

The following two observations could be made:

1. *Signatures are especially well–suited to uniquely identify input variables independent of permutation.*

   Three classes of signature functions are presented which were proven to work very efficiently. In fact, in all cases were it was possible to uniquely identify an input variable in our large set of benchmarks these signature functions were able to do it. Furthermore, in order to compute several of these signature functions just one ROBDD describing the Boolean function is needed, and of course, the computation does not depend on the variable ordering of the ROBDD. Thus as long as we can construct an ROBDD for the function with any variable ordering, we can apply these signature functions. This is a very modest requirement, since an inability to construct the ROBDD for any variable ordering would preclude verification with the help of ROBDDs even if you knew the variable correspondence.

2. *The only limitation for signatures to be able to uniquely identify input variables are special symmetries, that we call $\mathcal{G}$–symmetries.*

   At first the limitations of using signatures to tackle the combinational permutation equivalence problem are examined by presenting basic results which identify exactly what these limitations are. The property of $\mathcal{G}$–symmetry of Boolean functions is investigated, and a

universal signature function is introduced which dominates all other signature functions in the following sense: if there is any signature function which can distinguish between two input variables of a Boolean function, then the universal signatures of these two variables must be different as well. In this sense, the universal signature function is the strongest signature function which can be constructed. Then the existence of a universal signature could be proven and so a central theoretical result of this thesis: if any two input variables of a Boolean function f have the same universal signature, then there is a permutation $\pi \in \mathcal{P}_n \setminus \mathbf{1}$ of the input variables of f such that applying this permutation to the inputs of f does not change this function. In other words, the only limitation for signatures to be able to uniquely identify input variables are $\mathcal{G}$–symmetries.

Next, new kinds of symmetry classes (i.e., special $\mathcal{G}$–symmetries) are identified that help in finding a correspondence between the input variables of two Boolean functions being compared.

For our large set of benchmark circuits, the CPU–times necessary to establish such a unique correspondence are very promising. Thus, in addition to providing theoretical insight, the algorithms presented have direct practical impact for the complete solution of the permutation independent Boolean comparison problem.

For years, the permutation equivalence problem has been worked on by several other authors as well [9, 10, 21, 25, 29, 31, 33]. Considering all the approaches for handling the permutation equivalence problem that were proposed in the last few years we can say the following. It is not possible to say that one of these approaches presents the best method to handle the problem. Each of these methods will work well for a special class of practical circuits. However, except for the approach presented in this thesis, none of them takes general kinds of symmetry (that we call $\mathcal{G}$–symmetry) into account. That may be acceptable for application in technology mapping when just a small number of inputs is involved. There, it may be feasible to try all correspondence possibilities established after applying different signature functions or try using the method introduced in [31]. However, when permutation independent Boolean comparison has to be used for functions with a large number of inputs, as in formal logic verification, it is definitely necessary to take $\mathcal{G}$–symmetries into consideration. This can be underlined by the theoretical investigations with respect to the limits of signatures and by the experiments on the large set of benchmarks discussed in this thesis. So, applying the methods that use the knowlegde about $\mathcal{G}$–symmetries significantly sets the work of this thesis apart from the other approaches.

Furthermore, it is shown that these methods can be used in order to handle other problems of circuit design and verification as well. Here, it is demonstrated how the methods can be easily extended and applied to handle one problem in sequential logic verification. That is the problem of establishing the unknown correspondence between the latch variables of two sequential circuits with the same state encoding. We call it the latch permutation equivalence problem. A solution of this problem can be used to verify the combinational equivalence of two sequential logic circuits that have the same state encoding, but the correspondence between the latch variables is not known.

Experiments have shown that as long as there are no latch symmetries, signatures can be used to establish a unique possible correspondence between the latches. The CPU times necessary to establish such a unique correspondence are very promising.

Moreover, this method can be easily extended to identify equivalent latches in a circuit (see Section 5.4). Thus we believe that it is especially suited to be added in sequential verification tools that use product machine traversals. Here, computations can be made more efficient by exploiting combinationally equivalent state variables (see for example, [13, 36]).

In our opinion, the methods presented here to tackle the latch correspondence problem can easily be integrated with existing verification methods and so significantly improve the ability of these techniques to handle sequential circuits with the same state encoding. This has direct practical impact in sequential logic verification because it enlarges the class of sequential circuits for which verification is feasible (see also [13]).

Finally, there are several ideas for future projects related to the work reported in this thesis.

The thesis shows that it is extremely difficult to handle $\mathcal{G}$–symmetry in general. Furthermore, it demonstrates the practical importance of $\mathcal{G}$–symmetry of Boolean functions for logic synthesis and verification. So it would be useful to investigate this further.

However, there are other important fields of investigation. One of these is to check other data structures, i.e., other kinds of decision diagrams (for example OKFDDs [2]), circuit descriptions on the gate level, etc., for their ability to be used in applying these signature–based methods for solving the permutation problem. The advantage of this is that it would enlarge the class of combinational and sequential circuits for that the permutation problem can be efficiently solved.

Another important area is the application of the signature–based methods to incompletely–specified Boolean functions. With the extension of these methods to functions with DC's, several improvements in different areas of circuit design and verification would be possible. In technology mapping, it would enlarge the degrees of freedom for efficient mappings of Boolean functions to a certain library. Some preliminary investigations on this issue were made in [16].

Moreover, it would be interesting to extend the approach to handle the combinational equivalence problem for unknown input *and* unkown output correspondence of two Boolean functions with more then one output. Here, the application of signature functions to the output variables of a Boolean function is straightforward: satisfy count signatures and breakup signatures could also be used to identify output functions. Furthermore, the permutation independently ordered signature vector of the input variables of an output function could be used to identify this output function independent of permutation. The influence of this extension regarding to $\mathcal{G}$–symmetry and the heuristics to handle special kinds of this symmetry has to be investigated.

Appendix A

Benchmark Descriptions

In this section we provide results for the 243 benchmarks from the LGSynth91 benchmark set (Tables A.1–A.2) and the ESPRESSO benchmark set (Tables A.3–A.5) as well as three additional benchmarks (Table A.6: *act1* and *act2* — the actel1 and actel2 cells from the FPGA manufacturer *Actel*; *mult3* — a 3-bit multiplier). These are all benchmarks for that we were able to construct the ROBDDs. Each table is constructed as follows. The first 4 columns contain a description of each benchmark circuit: name, number of input and output variables, and number of ROBDD nodes. In Column 5, the partition of the input variables after signature computation (see Section 3.3) is shown (*(x,y)* indicates that there are x variable groups of size y in this partition. Note that classes of partial symmetric variables are represented by one variable of this group only.). The last column contains the CPU–time in seconds which was needed to compute all signatures and to construct the partition of the input variables using these signatures on a SUN Sparcstation 10 with 64 MB RAM. A time *0.0 sec.* means that there was no measurable CPU–time. Please see Section 3.3 for detailed comments relating to these results.

name	#i	#o	#n	groups: (subsets, size)	cpu (in sec.)
5xp1	7	10	66	(7,1)	0.0
9sym	9	1	26	(1,1)	0.0
9symml	9	1	26	(1,1)	0.0
ADDERFDS	33	17	458	(16,1)	0.1
C17	5	2	10	(5,1)	0.0
C1355	41	32	33195	(41,1)	31.4
C432	36	7	31179	(36,1)	6.4
C499	41	32	33195	(41,1)	29.0
C1908	33	25	12713	(33,1)	2.7
C880	60	26	7889	(57,1)	5.3
C5315	178	123	21194	(176,1)	631.3
CM138	6	8	39	(5,1)	0.0
CM150	21	1	64	(3,1),(3,4),(1,6)	0.8
CM151	12	2	32	(3,1),(3,3)	0.2
mux_cl	11	1	18	(2,1),(3,3)	0.1
CM162	14	5	63	(13,1)	0.0
CM163	16	5	49	(13,1)	0.0
CM42	4	10	23	(4,1)	0.0
CM82	5	3	17	(2,1)	0.0
CM85	11	3	42	(11,1)	0.0
DES	256	245	7257	(256,1)	20.6
PARITYFDS	16	1	18	(1,1)	0.0
alu2_cl	10	6	187	(10,1)	0.0
alu4_cl	14	8	1168	(14,1)	0.1
alupla	25	5	2266	(25,1)	0.7
apex1	45	45	4519	(45,1)	1.8
apex2	39	3	2948	(34,1)	1.3
apex3	54	50	10826	(54,1)	7.0
apex4	9	19	985	(9,1)	0.1
apex5	117	88	2092	(111,1)	3.6
apex6	135	99	1623	(134,1)	39.3
apex7	49	37	568	(49,1)	4.2
b12	15	9	89	(15,1)	0.0
b1	3	4	8	(2,1)	0.0
b9	41	21	177	(36,1)	0.1
bw	5	28	104	(5,1)	0.0
c8	28	18	156	(28,1)	0.0
cc	21	20	77	(21,1)	0.0
cht	47	36	135	(47,1)	0.4
clip	9	5	227	(9,1)	0.0
cmb	16	4	37	(3,1)	0.0
comp	32	3	147	(32,1)	0.0
con1	7	2	18	(7,1)	0.0
cordic	23	2	86	(8,1),(2,2)	0.4
count	35	16	249	(34,1)	0.1
cps	24	109	1456	(21,1)	0.7
cu	14	11	68	(14,1)	0.1
dalu	75	16	4576	(74,1)	64.0
decod	5	16	39	(5,1)	0.0
duke2	22	29	599	(22,1)	0.1
e64	65	65	1761	(65,1)	0.7
ex1010	10	10	1074	(10,1)	0.1
ex4	128	28	896	(66,1),(2,2)	27.4

Table A.1: LGSynth91 Benchmarks, Part I

name	#i	#o	#n	groups: (subsets, size)	cpu (in sec.)
ex5	8	63	362	(8,1)	0.0
example2	85	66	759	(85,1)	3.9
f51m	8	8	69	(8,1)	0.0
frg1	28	3	186	(27,1)	0.1
frg2	143	139	3750	(143,1)	30.5
inc	7	9	96	(7,1)	0.0
i1	25	16	62	(17,1)	0.0
i2	201	1	1587	(21,1)	135.7
i3	132	6	134	(1,15),(1,14),(37,1)	48.2
i4	192	6	350	(110,1)	0.8
i5	133	66	963	(133,1)	1.4
i6	138	67	417	(138,1)	1.2
i7	199	67	505	(199,1)	528.6
i8	133	81	2551	(133,1)	3.2
i9	88	63	2393	(88,1)	1.7
k2	45	45	1616	(45,1)	1.4
lal	26	19	123	(17,1),(1,4)	0.2
misex1	8	7	48	(8,1)	0.0
misex2	25	18	151	(20,1)	0.0
misex3	14	14	2721	(14,1)	0.4
misex3c	14	14	483	(14,1)	0.1
mux	21	1	88	(3,1),(3,4),(1,6)	0.9
traffic_cl	5	1	9	(2,1)	0.0
pair	173	137	18523	(173,1)	64.2
pcle_cl	19	9	80	(19,1)	0.0
pcler8_cl	27	17	141	(27,1)	0.0
pdc	16	40	1196	(16,1)	0.2
pm1	16	13	48	(11,1)	0.0
rd53	5	3	18	(1,1)	0.0
rd73	7	3	32	(1,1)	0.0
rd84	8	4	43	(1,1)	0.01
rot	135	107	10225	(130,1)	29.0
sao2	10	4	123	(2,1),(4,2)	0.2
seq	41	35	5639	(39,1)	3.7
sct	19	15	136	(19,1)	0.0
spla	16	46	1121	(16,1)	0.2
squar5	5	8	40	(5,1)	0.0
t481	16	1	80	(4,4)	0.6
table3	14	14	1962	(14,1)	0.2
table5	17	15	1568	(17,1)	0.2
tcon	17	16	26	(17,1)	0.0
term1	34	10	616	(14,1),(2,2),(3,5)	12.7
too_large	38	3	4404	(34,1)	1.3
ttt2	24	21	168	(24,1)	0.0
unreg	36	16	136	(36,1)	0.1
vda	17	39	1256	(17,1)	0.2
vg2	25	8	391	(23,1)	0.6
x1	51	35	1212	(50,1)	0.8
x2	10	7	40	(9,1)	0.0
x3	135	99	998	(134,1)	10.6
x4	94	71	758	(93,1)	1.1
xor5	5	1	7	(1,1)	0.0
z4ml	7	4	38	(3,1)	0.0

Table A.2: LGSynth91 Benchmarks, Part II

name	#i	#o	#n	groups: (subsets, size)	cpu (in sec.)
accpla	50	69	3795	(50,1)	1.2
al2	16	47	132	(15,1)	0.0
alcom	15	38	100	(14,1)	0.0
alu1	12	8	37	(12,1)	0.0
alu2	10	8	99	(10,1)	0.0
alu3	10	8	131	(10,1)	0.0
amd	14	24	345	(14,1)	0.1
apla	10	12	137	(10,1)	0.0
b10	15	11	590	(15,1)	0.1
b11	8	31	70	(8,1)	0.0
b12	15	9	89	(15,1)	0.0
b2	16	17	4059	(15,1)	0.5
b3	32	20	1237	(32,1)	0.3
b4	33	23	425	(33,1)	0.1
b7	8	31	70	(8,1)	0.0
b9	16	5	126	(16,1)	0.4
bc0	26	11	4254	(21,1)	0.7
bca	26	46	1591	(16,1)	0.2
bcb	26	39	1387	(16,1)	0.2
bcc	26	45	1130	(16,1)	0.2
bcd	26	38	928	(16,1)	0.2
br1	12	8	102	(11,1)	0.0
br2	12	8	100	(12,1)	0.0
chkn	29	7	643	(25,1)	0.3
clpl	11	5	37	(10,1)	0.0
cps	24	109	1456	(21,1)	0.7
dc1	4	7	25	(4,1)	0.0
dc2	8	7	94	(8,1)	0.0
dekoder	4	7	25	(4,1)	0.0
dk17	10	11	112	(10,1)	0.0
dk27	9	9	69	(9,1)	0.0
dk48	15	17	177	(15,1)	0.0
ex4	128	28	896	(66,1),(2,2)	27.6
ex5	8	63	362	(8,1)	0.0
ex7	16	5	126	(16,1)	0.4
exep	30	63	1249	(29,1)	0.2
exp	8	18	197	(8,1)	0.0
exps	8	38	522	(8,1)	0.0
gary	15	11	718	(15,1)	0.1
ibm	48	17	888	(48,1)	1.3
in0	15	11	670	(15,1)	0.1
in1	16	17	4059	(15,1)	0.5
in2	19	10	1384	(19,1)	0.2
in3	35	29	665	(34,1)	0.2
in4	32	20	1140	(32,1)	0.2
in5	24	14	680	(24,1)	0.1
in6	33	23	419	(33,1)	0.1
in7	26	10	234	(26,1)	0.0
inc	7	9	96	(7,1)	0.0
intb	15	7	751	(15,1)	0.2
jbp	36	57	577	(36,1)	0.3

Table A.3: ESPRESSO Benchmarks, Part I

name	#i	#o	#n	groups: (subsets, size)	cpu (in sec.)
lin.rom	7	36	414	(7,1)	0.0
luc	8	27	152	(8,1)	0.0
m1	6	12	49	(6,1)	0.0
m2	8	16	135	(8,1)	0.0
m3	8	16	152	(8,1)	0.0
m4	8	16	230	(8,1)	0.0
mainpla	27	54	9892	(26,1)	1.8
mark1	20	31	224	(20,1)	0.1
max1024	10	6	482	(10,1)	0.1
max128	7	24	169	(7,1)	0.0
max46	9	1	84	(9,1)	0.0
max512	9	6	169	(9,1)	0.0
misg	56	23	109	(31,1),(3,3),(1,4)	1.6
mish	94	43	130	(53,1)	0.4
misj	35	14	58	(16,1)	0.0
mp2d	14	14	76	(13,1)	0.0
newapla	12	10	61	(11,1)	0.0
newapla1	12	7	31	(11,1)	0.0
newapla2	6	7	18	(5,1)	0.0
newbyte	5	8	25	(5,1)	0.0
newcond	11	2	48	(10,1)	0.0
newcpla1	9	16	130	(9,1)	0.0
newcpla2	7	10	70	(7,1)	0.0
newcwp	4	5	12	(4,1)	0.0
newill	8	1	19	(8,1)	0.0
newtag	8	1	12	(5,1)	0.0
newtpla	15	5	84	(13,1)	0.0
newtpla1	10	2	21	(6,1)	0.0
newtpla2	10	4	39	(10,1)	0.0
newxcpla1	9	23	105	(9,1)	0.0
opa	17	69	461	(15,1)	0.1
p82	5	14	63	(5,1)	0.0
pdc	16	40	1196	(16,1)	0.2
pope	6	48	240	(6,1)	0.0
prom1	9	40	1859	(9,1)	0.2
prom2	9	21	974	(9,1)	0.1
risc	8	31	71	(8,1)	0.0
ryy6	16	1	27	(5,1),(2,2)	0.1
sex	9	14	63	(9,1)	0.0
shift	19	16	63	(19,1)	0.1
signet	39	8	8539	(38,1)	2.8
soar	83	94	982	(77,1)	1.4
spla	16	46	1121	(16,1)	0.2
sqn	7	3	57	(7,1)	0.0
t1	21	23	216	(21,1)	0.0
t2	17	16	189	(17,1)	0.0
t3	12	8	115	(11,1)	0.0
t4	12	8	91	(12,1)	0.0
ti	47	72	4461	(43,1)	1.4
tms	8	16	158	(8,1)	0.0
ts10	22	16	271	(16,1),(1,6)	2.1
vg2	25	8	267	(23,1)	0.6

Table A.4: ESPRESSO Benchmarks, Part II

name	#i	#o	#n	groups: (subsets, size)	cpu (in sec.)
vtx1	27	6	520	(24,1)	0.6
wim	4	7	24	(4,1)	0.0
x1dn	27	6	520	(24,1)	0.6
x2dn	82	56	266	(62,1)	0.5
x6dn	39	5	6243	(35,1)	2.0
x7dn	66	15	2020	(61,1)	1.6
x9dn	27	7	595	(24,1)	0.8
xparc	41	73	5319	(38,1)	2.0
Z5xp1	7	10	96	(7,1)	0.0
Z9sym	9	1	26	(1,1)	0.0
add6	12	7	55	(6,1)	0.0
addm4	9	8	225	(1,1),(4,2)	0.6
adr4	8	5	44	(4,1)	0.0
bcd	4	4	20	(4,1)	0.0
co14	14	1	28	(1,1)	0.0
dist	8	5	135	(4,2)	0.3
f51m	8	8	73	(8,1)	0.0
l8err	8	8	104	(8,1)	0.0
life	9	1	27	(2,1)	0.0
log8mod	8	5	76	(7,1)	0.0
m181	15	9	89	(15,1)	0.0
mlp4	8	8	141	(4,2)	0.3
radd	8	5	39	(4,1)	0.0
rckl	32	7	196	(32,1)	0.0
rd53	5	3	18	(1,1)	0.0
rd73	7	3	32	(1,1)	0.0
root	8	5	100	(7,1)	0.0
sqr6	6	12	70	(6,1)	0.0
sym10	10	1	32	(1,1)	0.0
tial	14	8	1273	(14,1)	0.2
z4	7	4	48	(3,1)	0.0

Table A.5: ESPRESSO Benchmarks, Part III

name	#i	#o	#n	groups: (subsets, size)	cpu (in sec.)
act1	8	1	15	(7,1)	0.0
act2	8	1	12	(2,1), (2,2)	0.0
mult3	6	6	44	(3,2)	0.1

Table A.6: Other Benchmarks

Bibliography

[1] *Combinational and Sequential Logic Benchmark Suite*. International Workshop on Logic Synthesis, 1991.

[2] B.Becker, R.Drechsler, and M.Theobald. OKFDDs versus OBDDs and OFDDs. In *Proccedings of ICALP LNCS94*, pages 475–486, 1995.

[3] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *Proceedings of the IWLS*, May 1995.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the DAC*, pages 40–45, June 1990.

[5] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli, editors. *Logic minimization algorithms for VLSI synthesis*. Kluwer Academic Publishers, 1992.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[7] R. E. Bryant. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. In *Proceedings of the ICCAD*, pages 350–353, November 1991.

[8] J. B. Burch and D. E. Long. Efficient Boolean function matching. In *Proceedings of the ICCAD*, pages 408–411, November 1992.

[9] D. I. Cheng and M. Marek Sadowska. Verifying equivalence of functions with unknown input correspondence. In *Proceedings of EDAC*, pages 81–85, February 1993.

[10] E.M. Clarke, K.L.McMillan, X.Zhao, M. Fujita, and J.Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proceedings of the DAC*, pages 54–60, 1993.

[11] R. Drechsler, N. Drechsler, and W. Günther. Fast Exact Minimization of BDDs. In *Proceedings of the DAC*, pages 200–205, 1998.

[12] R. Drechsler, N. Göckel, and B. Becker. Learning Heuristics for OBDD Minimization by Evolutionary Algorithms. In *Proceedings of PPSN (Parallel Problem Solving from Nature), LNCS 1141*, pages 730–739, 1996.

[13] C.A.J.van Eijk and J.A.G. Jess. Detection of equivalent state variables in finite state machine verification. In *Proceedings of the IWLS*, May 1995.

[14] F.Mailhot and G.D.Micheli. Technology mapping using Boolean matching and don't care sets. In *Proceedings of the EDAC*, pages 212–216, February 1990.

[15] M. Garey and D. Johnson. *Computers and Intractability.* W. Freeman, New York, 1979.

[16] G.Helmer. *Anwendung von Signaturen bei der Verifikation unvollständig spezifizierter Boolescher Schaltungen.* Diplomarbeit, Martin–Luther–Universität Halle, 1997.

[17] G.Hotz. *Schaltungstheorie.* De Gruyter Lehrbuch, Walter De Gruyter, 1974.

[18] A. Hett, R. Drechsler, and B. Becker. MORE: An alternative implementation of BDD packages by multi-operand synthesis. In *Proceedings of the EURODAC*, 1996.

[19] A. Hett, R. Drechsler, and B. Becker. Fast and efficient construction of BDDs. In *Proceedings of the ED& TC*, 1997.

[20] J. Jain, A. Narayan, M. Fujita, and A. L. Sangiovanni-Vincentelli. Formal verification of combinational circuits. In *Proceedings of the 10th International Conference on VLSI Design*, pages 218–225, January 1997.

[21] Y.-T. Lai, S. Sastry, and M. Pedram. Boolean matching using binary decision diagrams with applications to logic synthesis and verification. In *Proceedings of the ICCD'92*, pages 452–458, October 1992.

[22] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the ICCAD*, pages 6–9, November 1988.

[23] G. De Micheli. *Synthesis and optimization of digital circuits.* McGraw-Hill International Editions, London, 1994.

[24] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. In *Proceedings of the DAC*, pages 52–57, June 1990.

[25] J. Mohnke and S. Malik. Permutation and phase independent Boolean comparison. *INTEGRATION - the VLSI journal 16*, pages 109–129, 1993.

[26] J. Mohnke, P. Molitor, and S. Malik. Limits of using signatures for permutation independent Boolean comparison. In *Proceedings of ASP-DAC*, August 1995.

[27] J. Mohnke, P. Molitor, and S. Malik. Establishing latch correspondence for sequential circuits using distinguishing signatures. In *Proceedings of MWSCAS*, August 1997.

[28] D. Möller, J. Mohnke, and M. Weber. Detection of symmetry of Boolean functions represented by ROBDDs. In *Proceedings of the ICCAD*, pages 680–684, November 1993.

[29] I. Pomeranz and S.M. Reddy. On diagnosis and correction of design errors. In *Proceedings of the ICCAD*, pages 500–507, November 1993.

[30] I. Pomeranz and S.M. Reddy. On determining symmetries in inputs of logic circuits. In *IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 13 , No. 11*, pages 1428–1434, November 1994.

[31] I. Pomeranz and S.M. Reddy. Simultaneous input and output matching for combinational logic circuits. In *University of Iowa, Department of ECE, Technical Report 8-2-1994*, 1994.

[32] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the ICCAD*, pages 408–411, November 1992.

[33] U. Schlichtmann, F. Brglez, and P. Schneider. Efficient Boolean matching based on unique variable ordering. In *Proceedings of the IWLS*, May 1993.

[34] E. Sentovich, K. Singh, L. Lavagno, Ch. Moon, R. Murgai, A. Saldanha, H.Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. *Department of EECS, UC Berkeley*, May 1992.

[35] K.J. Singh and P.A. Subrahmanyam. Extracting RTL models from transistor netlists. In *Proceedings of the ICCAD*, pages 11–17, November 1995.

[36] S.Quer, G.Cabodi, P.Camurati, L.Lavagno, E.M.Sentovich, and R.K.Brayton. Incremental FSM Re-encoding for Simplifying Verification by Symbolic Traversal. In *Proceedings of the IWLS*, May 1995.

[37] K. H. Wang, T. T. Hwang, and C. Chen. Restructuring binary decision diagrams based on functional equivalence. In *Proceedings of EDAC*, pages 261–265, February 1993.

[38] I. Wegener. *The Complexity of Boolean Functions*. John Wiley & Sons Ltd., and B.G. Teubner, Stuttgart (Wiley–Teubner Series in Computer Science), 1987.

Erklärung

Ich versichere hiermit an Eides Statt, die vorliegende Arbeit selbständig, ohne unerlaubte Hilfsmittel und nur mit Hilfe der angegebenen Literatur angefertigt zu haben.

Es ist das erste Mal, daß ich mich um einen Doktorgrad bewerbe.

Halle (Saale), den 3. Oktober 1998

Lebenslauf

1. Angaben zur Person

Name:	Janett Mohnke, geb. Lochert
Geburtsdatum:	2. April 1967
Geburtsort:	Luckenwalde
Wohnort:	Welsestraße 91 13057 Berlin
Familienstand:	verheiratet seit dem 2. September 1988 1 Tochter

2. Ausbildung/beruflicher Werdegang

1973 — 1983	Polytechnische Oberschule in Frankfurt (Oder) Abschlußprädikat "Auszeichnung"
1983 — 1986	Berufsausbildung mit Abitur zum Facharbeiter für Datenverarbeitung in den Datenverarbeitungszentren Cottbus und Frankfurt (Oder)
Juli 1986	Abitur, Prädikat "Auszeichnung" Facharbeiterprüfung, Prädikat "Auszeichnung"
1986	2monatige Arbeit als Programmierer im Datenverarbeitungszentrum Frankfurt (Oder)
1986 — 1991	Informatikstudium an der Humboldt–Universität zu Berlin
Juli 1991	Diplom in Informatik, Prädikat "Sehr Gut"
1991 — 1992	8monatiger Forschungsaufenthalt an der Universität des Saarlandes in Saarbrücken am Lehrstuhl von Prof. Dr. Günther Hotz (SFB 124)
1992	6monatiger Forschungsaufenthalt an der Princeton University, N.J., U.S.A. bei Prof. Dr. Sharad Malik mit einem Stipendium von IREX
1992 — 1994	Arbeit als wissenschaftlicher Mitarbeiter an der Humboldt-Universität zu Berlin im Rahmen des BMFT-Projektes 01 IS 102 (OMSI)
1994 — 1995	Erziehungsurlaub
1995 — 1997	Arbeit als wissenschaftlicher Mitarbeiter an der Martin-Luther-Universität in Halle (Saale) im Rahmen des DFG-Projektes Mo645/2-1
seit 1997	Software-Entwickler bei DResearch Digital Media Systems GmbH in Berlin

Halle (Saale), den 3. Oktober 1998