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.

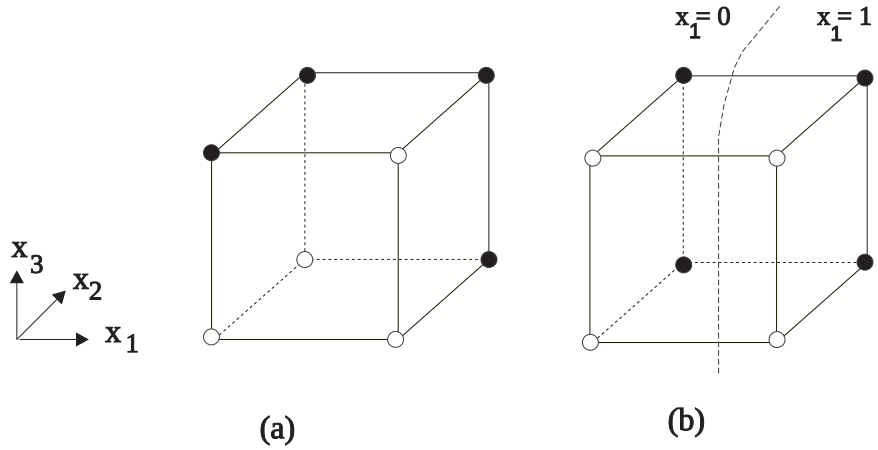

Figure 3.1: Cube Representation of $f(x_1, x_2, x_3) = x_1x_2 + \bar{x}_1x_3$ (a) and $f_{x_1}(x_1, x_2, x_3) = x_2$ (b)

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*.

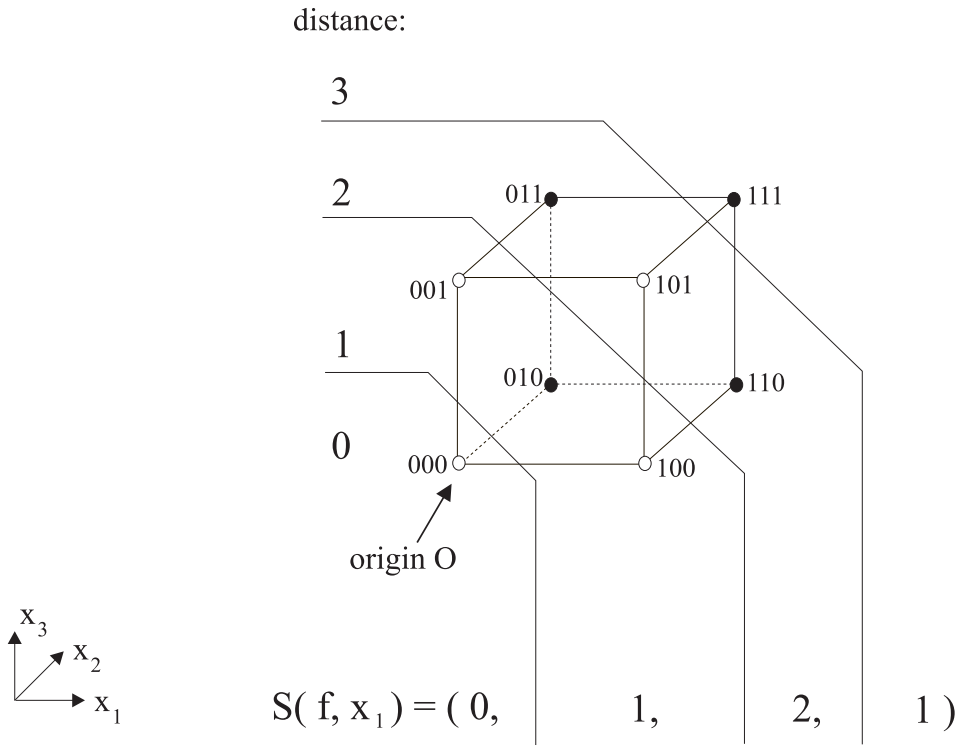

Figure 3.2: Breakup Signature for $f_{x_1}(x_1, x_2, x_3) = x_2$ with $O = [0, 0, 0]$

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.