Limits of using Signatures for Permutation Independent Boolean Comparison

Janett Mohnke Paul Molitor Sharad Malik

Institut für Informatik
Martin-Luther-Universität
06099 Halle (Saale), Germany

Dep. of Electrical Engineering
Princeton University
Princeton NJ 08544, U.S.A.

Abstract— This paper addresses problems that arise while checking the equivalence of two Boolean functions under arbitrary input permutations. The permutation problem has several applications in the synthesis and verification of combinational logic: It arises in the technology mapping stage of logic synthesis and in logic verification. A popular method to solve it is to compute a signature for each variable that helps to establish a correspondence between the variables. Several researchers have suggested a wide range of signatures that have been used for this purpose. 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. The paper investigates this problem.

I. Introduction

Checking if two Boolean functions are equivalent is a general and important task in logic synthesis and verification. With the development of the reduced ordered binary decision diagram (ROBDD) as a canonical form for representing Boolean functions, this problem reduces to checking if the two canonical forms are the same [2, 1]. However, in several cases in logic synthesis and automatic logic verification the following problem arises: We need to test the equivalence of two functions, but we do not know the correspondence between their inputs. In logic synthesis it occurs during the technology mapping stage, when a match has to be found from a cell library to parts of the technology independent network. This is especially difficult when a large number of inputs is involved. In logic verification we are confronted with it when the correspondence between the inputs of two circuits is not known. That may be the case when using different tools, which have their own naming conventions, at the various stage of synthesis.

Here, 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 is then 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. Several papers have investigated this problem in recent years, for example [6, 3, 7, 4, 11, 10]. Except [10], 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 that 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.

However, this method is not complete. There is *no* signature which can uniquely identify *all* the variables of the investigated benchmark sets (mostly the LGSynth91 and ESPRESSO benchmarks for logic synthesis). On comparing the most successful signatures, we observed that those benchmarks that have variables that could not be uniquely identified have been 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, not just 2 or 3 variables of those benchmarks are not uniquely identified, but about 15 and more, so that the number of possible correspondences is still large (see for example [3, 7]). Furthermore, this seems to be independent of the method used to solve that problem: In [10], 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 [9]. Another observation is that those benchmarks with non-uniquely identified variables are not the benchmarks with the most number of inputs. So, from a statistical point of view, we can conjecture that the quality of the used signatures is not the problem: 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. So, 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 our purpose — the permutation independent comparison of two Boolean functions. The focus of this paper is to break this tie. Our effort is directed towards finding special properties of these variables that cannot be uniquely identified. We will demonstrate that these properties are special symmetries which avoid a unique identification by any signature. We then propose methods to identify these symmetries and break the ties among the unidentified variables.

II. Preliminaries

In the following, let $\mathcal{B}_n$ be the set of completely–specified Boolean functions with n input variables, $X = \{x_1, x_2, \ldots, x_n\}$ and one output. We use X as a sequence of variables. For any $f \in \mathcal{B}_n$, we use the following basic definitions and notations:

A *literal* is a variable, x_i, or its complement, $\bar{x_i}$.

The *cofactor* of f with respect to a literal $x_i(\bar{x_i})$ is the Boolean function obtained by setting $x_i(\bar{x_i})$ to 1 in f and is denoted by f_{x_i} $(f_{\bar{x_i}})$. The function f_{x_i} $(f_{\bar{x_i}})$ is considered as a function with the same number of input variables as the function f, i.e. as a function with n input variables.

A *minterm* is a point (also referred to as a vertex) in the Boolean n-space of inputs.

The *satisfy set* of f is the set of all minterms for which the function value is 1. The *satisfy count* of f, denoted by $|f|$, is the cardinality of this set.

The function f is *symmetric* w.r.t. a subset of input variables, $\tilde{X} \subseteq X$, if f is invariant under all permutations of the variables in $\tilde{X}$.

The set $\tilde{X}$ is a *maximal symmetry group* (a maximal set of symmetric variables) of f if there is no variable $x_i \notin \tilde{X}$ so that f is symmetric w.r.t. to $\tilde{X} \cup x_i$ as well.

Last but not least, let us define the problem that is the subject of this paper:

Let $\mathcal{P}_n$ be the set of all possible permutations on the set of inputs $X = \{x_1, x_2, \ldots, x_n\}$, i.e., one-to-one mappings of X onto itself. $\mathcal{P}_n$ is a group, and we call it the *permutation group* in the sequel.

Let f_1 and f_2 be two Boolean functions of $\mathcal{B}_n$. The *permutation problem, P_π* is defined as follows: Does there exist a permutation $\pi \in \mathcal{P}_n$ such that $f_1(X) = (f_2 \circ \pi)(X)$? If so, we say that f_1 and f_2 are *permutation equivalent.*

III. Signatures

In this section we review the basic ideas behind using signatures for permutation independent Boolean comparison. Since we want to test if two Boolean functions are equivalent independent of the permutation of their inputs, we have to solve the problem of establishing a correspondence between the input variables of these two functions. The basic components of the approach we have used for that are signatures. In this context, a signature can be described as follows:

Definition III.1 *Let U be an ordered set, $(U, \leq)$.*

A mapping $s : \mathcal{B}_n \times X \longrightarrow U$ is a signature function *iff:*

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

Then, 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 $f \in \mathcal{B}_n$ is a description of x_i which provides special information about that 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 on x_j, then the signature of x_i in f must be the same as the signature of x_j in $f \circ \pi$.

A signature may be a value or a vector of values as well as a special function. For example, a simple signature for an input variable x_i of a Boolean function f is the satisfy count of the positive phase cofactor of this function w.r.t. x_i, $|f_{x_i=1}|$. However, there are a number of other signatures that have been used in practice. For information about these signatures see [6, 3, 7, 4, 11].

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 $g \in \mathcal{B}_n$. It only makes sense to establish a correspondence between these two variables, if variable x_i of f has 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 inputs of f and the inputs of any other function g.

What can we say about the practical experiences with using signatures? Let us consider the benchmarks of the LGSynth91 and the ESPRESSO benchmark set. Using the signatures introduced in [7], approx. 92% of all benchmarks have a unique correspondence for their inputs, i.e. all variables can be uniquely identified. (Symmetries as defined in the previous section have been considered. See [7] for details.) For the 8% of benchmarks with aliasing, the number of possibilities for correspondence among inputs ranges from 4 to approx. 10^{23} after applying all signatures.

Considering the complete set of benchmarks, we can make the observation that there is no relationship between the number of input variables of a function and the ability of the signatures to distinguish between all these inputs. The problem seems to be that there are special properties that make it impossible to distinguish between those variables via signatures. Since signatures work well for approx. 92% of the benchmarks, there is no reason to reject the signature approach. What we need is a moderate solution for the other 8% as well. For examples with just a few correspondence possibilities, the obvious solution of enumerating all possibilities works well. But what about the other examples (see for instance [7])? For those, we cannot be satisfied with existing solutions. A further understanding of these cases is the focus of the rest of this paper.

IV. Limits of Signatures

In this section, we discuss some properties of input variables that make it impossible to distinguish between these variables with the help of signatures. These properties are special symmetries that we can generalize as follows:

Consider a group $\mathcal{G} \subseteq \mathcal{P}_n$ of permutations. We say that a Boolean function $f \in \mathcal{B}_n$ is *$\mathcal{G}$–symmetric* if f keeps invariant under all permutations π in $\mathcal{G}$ [5]. 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 $\mathcal{P}_n$, and we say, f is $\mathcal{P}_n$–symmetric.

In order to understand the significance of $\mathcal{G}$–symmetry for the permutation problem, let us consider the following partition of the set of inputs, X into a set of nonempty and disjoint subsets constructed by a $\mathcal{G}$–symmetry:

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

1. Any permutation of $\mathcal{G}$ maps any element of A_i to an element of A_i again:
   For all permutations $\pi \in \mathcal{G}$: $\pi \circ A_i = A_i$ (for all $i = 1, 2, \ldots, k$)
2. There is no finer partition of X that satisfies Condition 1.

That this partition exists for *each* $\mathcal{G}$–symmetry is obvious to see. Furthermore, it is unique and has the following property:

Property IV.1 *Consider any element A_i of partition $\mathcal{A}$, any $\mathcal{G}$–symmetric Boolean function f, and any signature function s. All elements of A_i have the same signature $s(f, \cdots)$.*

Proof:

1. For all $x_k, x_l \in A_i$ there is a permutation π of $\mathcal{G}$ so that $\pi(x_k) = x_l$.

   To see why this must be so, suppose that this is not the case. That implies that x_l is not an element of the subset $\mathcal{G}(x_k)$. However, then $\mathcal{G}(x_k)$ is a *proper* subset of A_i: $\mathcal{G}(x_k) \subset A_i$. So, it is possible to partition A_i into $\mathcal{G}(x_k)$ and $A_i \backslash (\mathcal{G}(x_k))$. That implies that there is a finer partition of the inputs of A_i than A_i itself, and this is a contradiction to the property that partition $\mathcal{A}$ was the finest.

2. Consider any $x_l, x_k \in A_i$, a permutation π of $\mathcal{G}$ such that $\pi(x_k) = x_l$, and any signature function s. As defined, $s(f, x_k)$ is equal to $s(f \circ \pi, \pi(x_k))$. Furthermore, this is equal to $s(f, \pi(x_k))$, because f is $\mathcal{G}$–symmetric.

   So, we have: $s(f, x_k) = s(f, x_l)$ for any $x_k, x_l \in A_i$. □

This result now gives us a possible explanation for the trouble several benchmarks have with the signature approach: It is possible that these benchmarks 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. 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 section.

V. Special Symmetries

We now discuss some special kinds of $\mathcal{G}$–symmetry that 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. These are: hierarchical, group, and rotational symmetries. Here we will not consider the well known symmetry of a Boolean function w.r.t. a subset of its input variables since that is well understood (see Preliminaries). This kind of symmetry is under control already, and is not included in the 8% of benchmarks with aliasing that we consider in this paper (see for example [7]).

A. 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 considered in [7] is the following kind of symmetry:

Definition V.1 *Let $f \in \mathcal{B}_n$ 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 $|X_1| = |X_2| > 1$, X_1 and X_2 are maximal symmetry groups of f (see Preliminaries), and f is $H(X_1, X_2)$–symmetric, where $H(X_1, X_2)$ is equal to 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: $\forall i, j \in \{1, 2, \ldots, k\} : X_i$ is h–symmetric to X_j.*

Consider the following example:

Example V.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 symmetric variables, but there is no symmetry between the variables of these 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.

The definition of h–symmetry indicates that this is a special kind of $\mathcal{G}$–symmetry. So, let us examine the partition $\mathcal{A}$ built by it. It is obvious that all variables of X_1 and X_2 are in one element A_i of partition $\mathcal{A}$ if X_1 and X_2 are h–symmetric subsets of variables. Thus, from Property IV.1, we know that all of these variables have to have the same signatures, i.e., they form an aliasing group. Thus, there is no way to distinguish between them via signatures.

Luckily, there is a solution for this that is based upon our handling of symmetric variables. To understand this, let us consider the algorithm to identify the input variables by signatures. Here, the first step should be to determine all maximal groups of pairwise symmetric variables. This can be done very fast [8], and the advantage is that the signature computations can be restricted to one representative of each maximal symmetry group [7]. Furthermore, pairwise symmetric variables are kept together in aliasing groups in this way. 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 $(X_1 \leftrightarrow Y_2, X_2 \leftrightarrow Y_1)$. And 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 has to be done with these aliasing groups in order to solve the permutation problem, P_π.

Thus, we need an algorithm to decide if two subsets of symmetric variables of a Boolean function f, X_1 and X_2, that have the same signature, are h–symmetric. Our algorithm works as follows: It starts by first determining aliasing groups via signatures. Next, it works on groups like $\{X_1, X_2\}$ that contain subgroups of pairwise symmetric variables. On those, special cofactor computations are made, based on the following fact.

Fact V.1 *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 a Boolean function f iff for all assignments $a_1, a_2 \in \{0, 1\}^k$ to the variables of X_1 and X_2, the following two cofactor functions are equal:*

$$f_{a_1(X_1)a_2(X_2)} = f_{a_2(X_1)a_1(X_2)}.$$

In our special case, X_1 and X_2 are subsets of pairwise *symmetric* variables. So, just the number of 1's in an assignment a_i to the variables of X_1 as well as to those of X_2 is of consequence, i.e., we have to take into considerations exactly $|X_1| + 1$ assignments a_i to the variables. This is easily accomplished by the cofactor operation on the ROBDD of f. The overall complexity of this algorithm is $O(|X_1| \cdot bdd_f + |X_1|^2)$ where bdd_f is the size of the ROBDD of function f: We can take all cofactors simultaneously and have to test $O(|X_1|^2)$ equations.

B. Group and Rotational Symmetries

Two other $\mathcal{G}$–symmetries that appear in practice are group and rotational symmetries. *Group symmetry* (g–symmetry in short) is the term for certain $\mathcal{G}$–symmetries that have one characteristic in common — There are at least two different, nonempty, and disjoint subsets of the set of inputs of a Boolean function f that have the following property: There is a set of permutations on these subsets, so that applying the permutations simultaneously on all subsets does not change function f.

Example V.2 $f = A_0\overline{(x_0 + x_1)} + B_0\overline{(x_2 + x_3)}$

Here, $\{x_0, x_1\}$ and $\{x_2, x_3\}$ are pairs of symmetric variables, but there is no h-symmetry between them because of the 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, this is what we call g-symmetry between the two subsets of inputs, $\{x_0, x_1, x_2, x_3\}$ and $\{A_0, B_0\}$. The partition $\mathcal{A}$ constructed by this $\mathcal{G}$–symmetry (see Section IV.) includes exactly those subsets. 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 result in the formation of aliasing groups (see Property IV.1).

Our practical experiences on the benchmark set show that this kind of symmetry appears relatively often. One well-known example is a 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.

Unfortunately, it seems to be very complicated to detect a g–symmetry in general. Furthermore, we believe that the knowledge of this property is not very helpful — in general the reduction of possible variable correspondences is too small to be very useful.

However, we have made an interesting observation in this regard. The subsets of input variables that result in a g–symmetry are connected with each other in the following sense: If we have identified the variables of one of these sets, then it is possible to identify the variables of the other subsets as well. With that knowledge we have developed a heuristic to distinguish between variables of g–symmetric subsets. We provide a brief outline of the proposed algorithm for this purpose.

Consider the cofactors of a Boolean function that only depend on those variables that have been already uniquely identified by signatures. If these cofactors provide special information about an aliasing variable x_i, then they can be used to identify this variable: The information is independent of the permutation of variables with aliasing. We can consider this to be a new kind of signature. An example of such a signature is the cofactor with respect to all aliasing variables set to 0 accept x_i set to 1. The advantage of using these signatures is that they can be used to break the tie in terms of g–symmetries. This is now illustrated using Example V.2. After signature computation, using a basic set of signatures, the partition of its variables is $\mathcal{A} = \{\{(x_0, x_1), (x_2, x_3)\}, \{A_0, B_0\}\}$. The variables x_0 and x_1 as well as x_2 and x_3 are marked as pairs of symmetric variables. However, we want to break up this partition. So, let us try the following: Just *assume* that the variables of one of these two aliasing groups, say $\{A_0, B_0\}$, are uniquely identified. Then, we can use the signature described above to try to distinguish between x_0 and x_2. (Note, that this would be enough because of the pairwise symmetries between x_0 and x_1 as well as x_2 and x_3.): $g_1 = f_{x_0\bar{x_1}\bar{x_2}\bar{x_3}} = B_0$ and $g_2 = f_{\bar{x_0}\bar{x_1}x_2\bar{x_3}} = A_0$. At this point, the cofactor functions g_1 and g_2 are different. It is now possible to distinguish between x_0 and x_2 under this assumption! However, there is still a problem. This identification is not canonical. It depends on the order of A_0 and B_0 in function f. Because of this, we do this procedure for each possible order of A_0 and B_0, i.e., $f = A_0\overline{(x_0 + x_1)} + B_0\overline{(x_2 + x_3)}$ and $\hat{f} = B_0\overline{(x_0 + x_1)} + A_0\overline{(x_2 + x_3)}$. Thus, we get two different identifications for the variables x_0 and x_2 depending on the order of A_0 and B_0. Finally, we need a canonical way to select one of these two identifications. However, this is easy. Signatures are elements of an ordered set (see definition). So, we can order the input variables of a Boolean function f by their signatures. Furthermore, we can reconstruct f using this new order. Now, for our special case, we have two new orders, i.e. two new functions as well. So, let us choose that order that belongs to the lexicographic smaller of these two functions as the canonical one, and we have a unique solution for our problem.

We have tested this approach for the examples of our benchmark set. The experimental results are very promising (see next section).

Last and least, another kind of $\mathcal{G}$–symmetry is the rotational symmetry: Let $f \in \mathcal{B}_n$ be a Boolean function and $Y := \{y_1, y_2, \ldots, y_k\} \subset X$ a subset of its input variables. f is *rotational symmetric* (r–symmetric) if $k \geq 3$, f is not symmetric in Y, but does not change on applying the following permutation π on Y: $\pi(y_2) = y_1, \pi(y_3) = y_2, \pi(y_k) = y_{k-1}, \pi(y_1) = y_k$ or $\pi(y_1) = y_2, \pi(y_2) = y_3, \pi(y_{k-1}) = y_k, \pi(y_k) = y_1$.

The following simple example illustrates this kind of symmetry:

Example V.3 $f = x_1x_2 + x_2x_3 + x_3x_4 + x_4x_5 + x_5x_1$

Here it is obvious that rotating the variables does not change f, even though there is no pairwise symmetry between the variables.

In our experience r–symmetries do not appear very often in practice.

VI. Experimental Results

We have implemented the ideas presented in the previous sections in the Berkeley *SIS* system in C. To get an overview about the quality of the signatures in [7] we have tested all available benchmarks from the LGSynth91 and ESPRESSO benchmark set for which we were able to construct the ROBDDs, as well as a couple of additional benchmarks (*act1, act2* – the actel1 and actel2 cells from the FPGA manufacturer *Actel*; *mult3* – a 3-bit multiplier). In all there are 243 benchmarks. Of these, only 8% demonstrate aliasing with a range of signatures. These

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

TABLE I
Benchmarks with aliasing

have been listed in Table I. These benchmarks have been the subject of our research for this paper. A description of each circuit (name, number of inputs, outputs and ROBDD nodes) is followed by the number of possible correspondences after using signatures only(*sig*), signatures and the algorithms to determine h–symmetry (*+hsymm*), signatures, the h–symmetry algorithms and the heuristics for g–symmetry (*+grsym*). The last column includes the CPU time in seconds needed to apply all three heuristics on a SUN Sparcstation 10.

As can be observed, the results are very promising. Approx. 26% of all benchmarks with aliasing groups include h–symmetry. For all but two of the rest the *grsym*-algorithm has been successful. These two examples are *ts10* and *t481*. We know, that *ts10* is r–symmetric in 6 variables. Furthermore, we are still investigating the exact nature of the symmetry of the last case, *t481*.

The CPU times are very modest. With a minimal amount of time we could solve the correspondence problem in almost all cases.

VII. Conclusions

This paper presents interesting new insights into the permutation independent Boolean comparison problem. It first examines the limitations of using signatures to tackle this problem by presenting a basic result which identifies exactly what the limitations are. Next, it identifies new kinds of symmetry classes that help in finding a correspondence between the variables of two functions being compared. The CPU times necessary to establish such a unique correspondence is very promising. Thus, in addition to providing theoretical insight, the algorithms presented also have direct practical impact on the complete solution of this problem.

Acknowledgements

This research has been supported in part by DFG grants SFB 124 and Mo645/2-1.

References

[1] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

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

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

[4] 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 30th ACM/IEEE Design Automation Conference*, pages 54–60, 1993.

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

[6] 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.

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

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

[9] I. Pomeranz and S.M. Reddy. personal communication.

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

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