Exploiting Structural Similarities in a BDD-based Verification Method

C.A.J. van Eijk and G.L.J.M. Janssen

Eindhoven University of Technology, Department of Electrical Engineering
Design Automation Section, P.O. Box 513, 5600MB Eindhoven, The Netherlands

Abstract. A major challenge in the area of hardware verification is to devise methods that can handle circuits of practical size. This paper intends to show how the applicability of combinational circuit verification tools based on binary decision diagrams (BDDs) can be greatly improved. The introduction of dynamic variable ordering techniques already makes these tools more robust; a designer no longer needs to worry about a good initial variable order. In addition, we present a novel approach combining BDDs with a technique that exploits structural similarities of the circuits under comparison. We explain how these similarities can be detected and put to effective use in the verification process. Benchmark results show that the proposed method significantly extends the range of circuits that can be verified using BDDs.

1 Introduction

The times when researchers in the CAD field could sit in their ivory tower thinking up neat solutions for theoretical problems belong to the past. Nowadays, industry and other funding organisations require projects to come up with useful and practical results. Whereas only 10 years ago, a hardware designer would be surprised when an automatic tool could prove his LSI circuit (a couple of gates and a few flipflops) correct, today, the same designer expects tools that handle his 10,000 gates VLSI circuit. Clearly, the biggest problem faced in the area of hardware verification is that of scale.

In this paper we show how a successful technique for proving equivalence of combinational circuits can be extended to greatly enhance its applicability. Our past work has concentrated on the use of BDDs to represent the circuits' functional behaviour. However, it is a known fact that for many practical circuits no reasonably sized BDDs exist no matter what variable order we choose. Several approaches that try to circumvent this intricate problem have been proposed [1][9][11]. Often, the canonicity requirement for BDDs is dropped; then, of course, the equivalence check becomes harder. It is also possible to introduce extra variables with the intention of obtaining smaller BDDs. We feel that those methods are either not general enough for our purposes, or lack convincingly strong results. The method we propose in this paper is modelled after [6][14], and still uses regular BDDs. It works fully automatically and does not impose any special requirements on the design process.

This paper is organised as follows. In section 2 we momentarily divert into the rather novel technique of dynamic variable ordering. We explain the main idea and show how it can be incorporated in a BDD package. The experimental results presented at the section's end serve both as evidence for the usability of the technique and as a stimulus for further investigations. Section 3 introduces our ideas on using structural information of the circuits at hand to aid in the verification process. We show that with a minimum of extra effort, the available BDD routines can be used in a more clever way to establish equivalence. The results of our first experiments are very encouraging.

2 Dynamic Variable Ordering

In this section, we briefly summarize the issues involved in applying a dynamic variable ordering technique, i.e., changing the order of the variables during BDD construction. We assume that the reader is already familiar with the basic concepts of BDDs [8] and the popular way to implement them [5]. Dynamic variable ordering is a very important addition to a BDD package, because it relieves the user of the burden of specifying a good order a priori, i.e., before the BDDs are constructed. We can define a good order as one that permits the function to be represented by a polynomially sized BDD. For gate level descriptions, several static ordering algorithms have been proposed [10][12] and shown to be successful for many circuits. However, it turns out that dynamic variable ordering often substantially improves on the intermediate and final BDD sizes. In particular when BDDs are used in the area of verification, it is our opinion that dynamic variable ordering becomes a mandatory prerequisite for successfully handling large circuits automatically.

2.1 Basic Principles

It is a well-known fact that the size of a BDD representation for a given boolean function may drastically change when a different variable order is adopted. Therefore, it is very important that a good variable order is used. Because generally it is difficult to predict a good order before the BDDs are actually constructed, it is necessary to use a technique that searches a good order during the construction of BDDs. The problem with changing the order dynamically is that one has to maintain canonicity. In the implementation of a BDD package, canonicity is achieved through a so-called unique table of BDD nodes: a node is identified by its pointer. A new node is only then created when it is not yet present in the unique table; otherwise the pointer stored in the table is returned. It would be very inefficient to construct entirely new BDDs for every different order that is tried. Therefore, dynamic variable ordering is based on a succession of local modifications, each of which can easily be made to preserve canonicity. A natural local modification is the swapping of two consecutive variables, as is illustrated in Fig. 1.

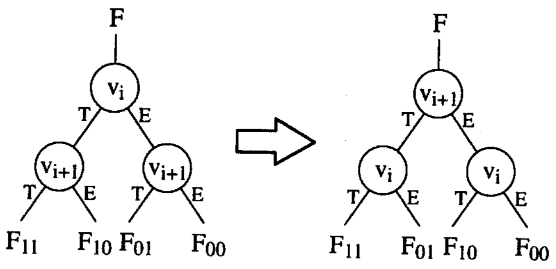


Fig. 1. The effect of a variable swap on a BDD

By repeatedly swapping adjacent variables every variable order can be generated. However, for practical purposes a full exploration of the 'variable orders space' cannot be tolerated; a simple local-search approach with limited hill-climbing is chosen instead. This has become known as the sifting algorithm [16]. In this approach, each variable is tried at all possible positions in the order while the ranks of the other variables remain the same. This search for the best position of that one variable may still lead to the construction of unacceptably large intermediate BDDs, hence the search is aborted as soon as the increase in the total number of nodes exceeds a given limit (we allow an increase of at most 5%). It is easy to see that putting one variable at its best position takes $\mathcal{O}(\#vars \cdot \#nodes)$ time. It makes sense to treat the variables in order of frequency: the variable with the most occurrences in the BDD is sifted first. The rationale is that by changing the position of this variable the largest gain, i.e., decrease in BDD size, may be achieved.

2.2 Implementation Issues

The integration of dynamic variable ordering into a BDD package requires two decisions, namely *where* and *when* the algorithm is applied. Rudell argues that it must be invoked inside the recursive ITE routine, because this is the major source for new nodes. However, dynamic variable ordering violates the invariants of the ITE routine. Since ITE recursively visits the nodes of its three BDD arguments in a depth-first fashion, starting at the top nodes, it is not allowed to swap any variables that have ranks smaller than the node currently under consideration. Dynamic variable ordering may be partly applied to the rest of the variables. This idea has not been persued. Instead, we decided to allow dynamic ordering only to take place outside recursive ITE calls, but potentially after every top-level call.

The question of when to apply dynamic ordering is not an easy one; on the one hand, dynamic variable ordering is a very useful, and for some applications even vital, feature of a BDD package, but on the other hand it takes $\mathcal{O}(\#vars^2 \cdot \#nodes)$ time for a single invocation, and therefore should not be called upon too liberally, especially when many variables are involved. Clearly, there are a number of conflicting interests:

- Dynamic variable ordering should be done as soon as the current order is found to be rather poor.
- The quality of an order can only be assessed relative to the functions that are momentarily represented. There is usually no way to predict the sequence of future BDD operations on the current functions.
- When the initial (or some intermediate) order is good, then the next call to dynamic ordering should be postponed as long as possible.
- When the functions to be represented are such that no good order exists, dynamic ordering should be refrained from completely.

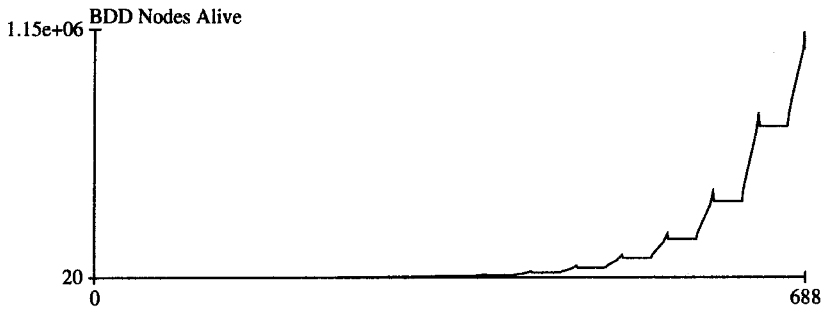
Rudell uses an absolute bound on the total number of BDD nodes to trigger the ordering. After each reordering, it is reset to twice the then existing number of nodes. Our solution is to fix the total time spent in one call to some reasonable constant, say 10 (cpu) seconds, and to use both a relative and an absolute threshold to trigger the variable ordering. The absolute threshold criterion we use is the same as described above. The relative threshold is introduced to be able to anticipate sharp increases in BDD size as a function of the number of top-level ITE calls. Dynamic ordering is triggered when the increase exceeds a factor of 2. This value is chosen because the majority of BDD operations takes two operands, and empirical evidence shows that the size of the result is of the order of the sum of the sizes of the operands; worst-case it would be the product.

The effect of dynamic variable ordering is illustrated in Fig. 2 for a 16-bits rotator circuit with 16-bits data input and output, and a 4-bits control input (20 BDD variables). Mark the difference in scale of the vertical axes: without dynamic variable ordering more than a million BDD nodes are required to compute the rotator's outputs; with dynamic ordering, some 4000 are needed. The ragged behaviour of the graphs is due to the particular way the circuit is described as a table, see Fig. 3. The program that constructs the BDDs handles a table row by row which causes intermediate results that are directly freed again, hence the peaks and plateaus in the figure (careful examination shows that there are precisely 16 of them).

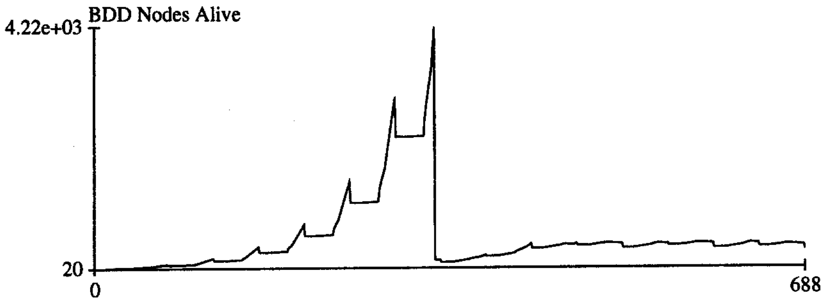
Dynamic variable ordering requires easy access to all BDD nodes with the same variable label. Hence unique tables need to be maintained per variable instead of having one large table shared by all of them. As a consequence and clear disadvantage, memory for those hash tables gets fragmented. On the positive side, it allows for a finer control over the amount of memory allocated for each table. In fact we do this dynamically, i.e., when more nodes for a particular variable are needed the table is extended, and when enough nodes of that variable are freed again the table is shrunk. This approach turns out to be both beneficial with regard to run time and memory usage.

2.3 Experimental Results

We have implemented the algorithm for dynamic variable ordering in the BDD package that is developed by one of the authors. It is based on the work of Karl Brace reported in [5] and papers by Richard Rudell [16][17]. Apart from



(a) without dynamic ordering; exponential increase in BDD size



(b) with dynamic ordering invoked at ITE call 329

Fig. 2. The number of nodes as function of the number of top-level ITE calls for a 16-bit rotator

```
behaviour rot16 (input In[15:0]; input C[3:0]; output Out[15:0])
{
  table {C ::                     Out;
       0 :: In[15:0]                ;
       1 :: In[14:0] || In[15:15];
       2 :: In[13:0] || In[15:14];
       3 :: In[12:0] || In[15:13];
       4 :: In[11:0] || In[15:12];
       5 :: In[10:0] || In[15:11];
       6 :: In[ 9:0] || In[15:10];
       7 :: In[ 8:0] || In[15: 9];
       8 :: In[ 7:0] || In[15: 8];
       9 :: In[ 6:0] || In[15: 7];
      10 :: In[ 5:0] || In[15: 6];
      11 :: In[ 4:0] || In[15: 5];
      12 :: In[ 3:0] || In[15: 4];
      13 :: In[ 2:0] || In[15: 3];
      14 :: In[ 1:0] || In[15: 2];
      15 :: In[ 0:0] || In[15: 1];
  }
}
```

Fig. 3. Description of the 16-bit rotator circuit

the usual logical operations on BDDs, it includes a rich set of meta routines (e.g. quantification with respect to a set of variables, composition, conversion to sum-of-cubes), routines for statistics (e.g. size, number of minterms), support for development of new operations, and routines to visualize BDDs (e.g. for X-Windows).

Table 1 indicates the effect of dynamic variable ordering on some typical benchmarks, all taken from [13] except for the rotator. #nodes is the size of the final (shared) BDDs for all output functions. The run time is in seconds on a HP9000/735 workstation. The 'good' and 'bad' orders are obtained manually, we don't claim them to be the best, resp. worst; 'dynamic' means dynamic variable ordering is on during BDD construction; 'bad' is taken as initial order, and at the end dynamic ordering is applied exhaustively until no more gain is obtained. The results for min_max include BDDs for both the regular outputs and the next-state functions.

With our implementation, we achieve results comparable with Rudell's [15]. As our experiments point out, there is no such thing as "one medicine cures all". Some tuning of the dynamic variable parameters may often give better results.

Table 1. Experimental results of dynamic variable ordering

Circuit	Good		Bad		Dynamic	
	Nodes	Secs	Nodes	Secs	Nodes	Secs
16-bits rotator	81	<1	1081328	45	81	<1
8-bits adder	36	<1	751	<1	36	<1
16-bits adder	76	<1	196575	12.7	123	<1
32-bits adder	156	<1	>1000000	29	452	1.9
8-bits min_max	893	<1	75377	4.7	892	1.7
16-bits min_max	3305	<1	>1000000	45.2	3303	8.9
32-bits min_max	12545	1.4	>1000000	33	33971	48.5
8-bits multiplier	9084	2.1	16697	2.9	8958	13.3
10-bits multiplier	72916	24.5	159278	32.8	72204	204.8
12-bits multiplier	598463	238.6	1513070	360.5	560216	1012.3

3 Exploiting Structural Similarities

In many design environments, formal verification is applied to establish the equivalence of two circuit descriptions. Typically, one description has been derived from the other by one or more design steps. Many of these steps have little effect on the global structure of a circuit. Therefore, it is likely that there exist structural similarities between the two circuits to be verified, i.e., not only the outputs are functionally equivalent, but also some of the internal signals. For example, when the correctness of a technology mapping step is verified, both circuits have a similar global structure, and when a designer wants to check the correctness of some manual modifications, only small parts of the circuit may have changed. Intuitively, it should be easier to prove the equivalence of two circuits which have a similar structure, than to prove the equivalence of two circuits with a totally different structure. However, this is typically not true for BDD-based verification tools; the efficiency of these tools is almost completely determined by the compactness of the BDD representations for the outputs of the circuits. Therefore, the efficiency depends on the *type* of circuit that is verified and not on the actual differences between the circuits that are compared.

One of the first verification methods to use structural equivalences was presented in [3]. However, this method may lead to so-called *false negatives*, i.e., the method may fail to prove the equivalence of two equivalent circuits. A more advanced method was described in [4]. This method still suffers from false negatives, although the authors suggest how BDDs can be used to avoid this problem. Recently, two verification methods were presented that exploit equivalences between internal signals in combination with a test generator [6][14]. These methods do not suffer from false negatives, and the presented benchmark results demonstrate the efficacy of using structural similarities.

In this section, it is shown how a BDD-based verification method can be extended to exploit structural equivalences of the circuits that are compared. This improves the method's ability to deal with circuits that are structurally similar, even if these circuits cannot be represented efficiently by BDDs. First, it is described how equivalent internal signals can be used to improve the verification method; the ideas used are similar to some of the ideas presented in [4]. Then, it is shown how these equivalences can be detected during the verification of the circuits.

3.1 Using Structural Equivalences

Suppose we want to verify that two outputs f and g are functionally equivalent, i.e., we want to establish the equivalence of the corresponding functions $F : B^n \rightarrow B$ and $G : B^n \rightarrow B$, where n denotes the number of primary inputs. This means we have to prove that for every input vector $\underline{i} \in B^n$,

$$F(\underline{i}) \oplus G(\underline{i}) = 0 \ . \tag{1}$$

Now assume that the fanin cones of f and g both contain a signal with the function $H : B^n \rightarrow B$. Then, this function can be used to decompose F and G, i.e., there are functions $F_\mathrm{d} : B^{n+1} \rightarrow B$ and $G_\mathrm{d} : B^{n+1} \rightarrow B$, such that:

$$\begin{aligned} F(\underline{i}) &= F_\mathrm{d}(\underline{i}, H(\underline{i})) \ , \\ G(\underline{i}) &= G_\mathrm{d}(\underline{i}, H(\underline{i})) \ . \end{aligned} \tag{2}$$

In that case, (1) can also be written as follows:

$$(\exists v \in B : (F_\mathrm{d}(\underline{i}, v) \oplus G_\mathrm{d}(\underline{i}, v)) \wedge (v = H(\underline{i}))) \ . \tag{3}$$

This formula indicates how a structural equivalence can be used to decompose the calculation of the difference. First, the difference of the two functions is calculated while the common subfunction H is represented by an extra variable. If this difference is zero, the two functions are equivalent. However, if it is not zero, it cannot be decided directly that the functions are not equivalent; this decision can only be made if the difference expressed in the original variables differs from zero. Otherwise, there must apparently exist variable assignments for which the calculated difference does not equal zero, but these assignments may not be valid because the variables representing subfunctions cannot be assigned values independently. To avoid these false negatives, it is necessary to replace the introduced variable by the function it represents.

A verification method can repeatedly use structural equivalences to compare two circuits. The main advantage of this approach is that the BDD representations remain compact if sufficient equivalences exist between the circuits. The functions of the signals are not only expressed in terms of the primary inputs, but also in terms of variables representing common internal signals. Because the functional dependencies between these variables are not taken into account while constructing BDDs, this approach leads to smaller BDDs, even though the number of variables is increased. The main disadvantage is of course that the equivalence check now requires substitutions, possibly resulting in a large BDD representation for the difference of two signals. However, in case the two circuits not only have some internal signals in common, but also use these signals in a similar manner, the difference may be relatively small, or it may even be zero, in which case there is no need for substitutions at all.

The two circuits in Fig. 4 are used to illustrate how the verification method uses equivalent internal signals.

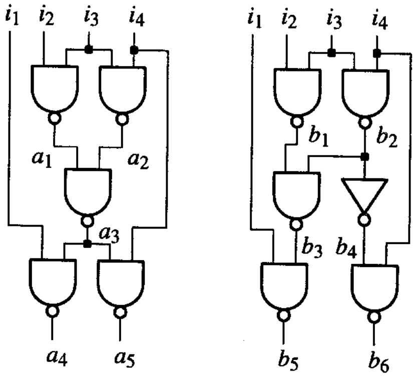


Fig. 4. An example of two similar circuits

First, it is verified whether the outputs a_4 and b_5 are equal. Obviously, this should be easy, because the parts of the circuits which define these outputs, are not only functionally but also structurally identical. During the verification, it is detected that the signals a_1 and b_1, and a_2 and b_2 are equal, and therefore, two new variables v_1 and v_2 are introduced to represent the functions of these signals.

Then, it is detected that a_3 and b_3 are equal, because for both, the function $\neg(v_1 \wedge v_2)$ is computed. The variable v_3 is introduced for this function. Similarly, the function $\neg(i_1 \wedge v_3)$ is calculated for both a_4 and b_5, and the equivalence of the outputs is established. This shows that the proposed method can handle structural equivalence without any problem.

The verification of the outputs a_5 and b_6 is slightly more difficult. If the same equivalences as detected earlier are used, the function $\neg(i_4 \wedge v_3)$ is computed for output a_5, and the function $\neg(i_4 \wedge \neg v_2)$ for output b_6. The difference $a_5 \oplus b_6$ is $i_4 \wedge (v_3 \oplus \neg v_2)$. Because this is not equal to zero and still depends on variables representing internal signals, the function $\neg(v_1 \wedge v_2)$ is substituted for variable v_3. Then, the difference becomes $i_4 \wedge v_2 \wedge \neg v_1$. This is still not equal to zero, and therefore, also v_2 is replaced by the function it represents. This results in the difference $i_4 \wedge \neg(i_3 \wedge i_4) \wedge \neg v_1$, which can also be written as $i_4 \wedge \neg i_3 \wedge \neg v_1$. After the replacement of v_1 by its function, this becomes $i_4 \wedge \neg i_3 \wedge i_3 \wedge i_2$, which equals zero. This proves that the outputs a_5 and b_6 are equivalent.

3.2 Detecting Structural Equivalences

We will now address the problem of detecting structural equivalences of two circuits. A naive approach is to compare every signal calculated in one circuit with all signals calculated in the other circuit. However, this is not very efficient, since every equivalence check may require a series of substitutions; especially the comparison of non-equivalent signals can be quite expensive, both in terms of run time and memory usage. Of course, it is also possible to detect only those equivalences for which no substitutions are required. In that case, it is not guaranteed that all equivalences are found, because it is only tested if two signals have exactly the same BDD representation. However, this is not an effective approach. If an equivalence is not detected because the difference of the BDDs is apparently not equal to zero, it is very likely that all other equivalences in the fanout cone of these signals also remain undetected. In fact, it is very likely that the difference becomes larger while it propagates towards the outputs. Therefore, we introduce an extra preprocessing step to detect which signals are most likely to be equivalent.

An effective preprocessing step is to calculate a signature for every signal by simulating both circuits for a limited number of randomly generated input vectors. Because two signals cannot be equivalent if they have different signatures, such a signature can be used as a coarse check for the equivalence of two signals. This technique does not exclude any equivalences, and therefore does not suffer from the same disadvantage as selecting signals on basis of the calculated BDD representation. However, there are circuits for which random input vectors are not very effective in distinguishing signals. In that case, many inequivalent internal signals will be compared, possibly requiring many substitution operations. The effects of this problem can be diminished by exploiting the result of the equivalence check. This result is namely the set of all input vectors that distinguish the two signals under comparison. Therefore, it is possible to randomly select vectors from this set, and perform extra signature calculations to update the lists of potentially equivalent signals. This way, the negative outcome of an equivalence check is also used to check all other potential equivalences.

Structural equivalences can only be used effectively if they are detected as soon as possible; otherwise, the BDDs for the outputs are constructed before any extra variables are introduced and the method cannot perform better than a conventional BDD-based method. Therefore, it is necessary to calculate signals from both circuits simultaneously and to compare potentially equivalent signals as soon as possible. This means that the calculation order has to group potentially equivalent signals together as much as possible. To do this, we adopt the following strategy. The calculation order is determined by maintaining a so-called 'ready list' of signals. This list contains all signals for which potential equivalences exist and for which all predecessors with potential equivalences have already been calculated; initially, it contains the primary inputs of both circuits. From this list, the signal with the lowest topological level is selected, together with all signals that are potentially equivalent to it. These signals are calculated together with the parts of the corresponding fanin cones that have not yet been calculated. Then, the equivalence of these signals is actually checked, and the ready list is updated.

3.3 Implementation Issues

We have implemented the presented verification method in C++ using the BDD package mentioned in Sect. 2. Dynamic variable ordering is used to order the variables; variables representing internal signals are created at the front of the order. A signature calculation is used to partition the signals of both circuits into groups of signals that are likely to be equivalent. Signatures are calculated by simulating both circuits simultaneously for 32 randomly generated input vectors. These signatures are used to create the initial partition. Then, the partition is refined by calculating new signatures. This is repeated until the partition does not change during five successive runs. Under the assumption of perfect hashing, the calculation of these potential equivalences takes $\mathcal{O}(C \cdot n)$ time and $\mathcal{O}(C)$ space, where C denotes the size of both circuits that are compared, and n is the number of times the partition is refined.

The notion of structural equivalence is slightly extended to allow for signals that are functionally equivalent modulo complementation. Every signal is assigned a 'sign' that is determined by the value of that signal for a randomly selected input vector. If a signal has a negative sign, its function is complemented before we check for equivalences. If we detect structural equivalences and decide to introduce a new variable, the function of a signal with a negative sign is replaced by a negated variable; this has no further consequences for the equivalence check. To limit the number of extra variables introduced by the method, a new variable is created only if the size of the BDD for the corresponding function exceeds a given threshold. In our implementation, we use a threshold of 32 nodes. If it is decided not to introduce a new variable, the BDDs of the equivalent signals are still unified by selecting the smallest BDD as the representation for all these signals.

3.4 Experimental Results

To evaluate the efficiency of the presented verification method, we use it to verify the correctness of some circuits that have been synthesized with the logic synthesis systems EUCLID [2] and SIS, a system developed at University of California, Berkeley. All tests in this section are performed on a HP9000/735 workstation with a memory limit of 100 Mb. As benchmarks, we use instances of the min_max circuit and the unsigned bit multiplier, which are both described in [13]. The notations mM[n] and mult[n] are respectively used to refer to the n-bit instances of these circuits. Because our verification method only deals with combinational circuits, every latch in the min_max circuit is modeled by an extra input and output.

Table 2 shows the performance of a conventional BDD-based verification tool on some selected instances of the benchmarks. As these results illustrate, the min_max circuit is an 'easy' example, which can be represented compactly with BDDs and thus scales well. The multiplier circuit on the other hand is very difficult to verify; the required amounts of memory and run time grow rapidly with increasing bit width. With a memory limit of 100 Mb, the applicability of this tool is restricted to instances with $n \leq 14$.

Table 2. Benchmark results of a conventional BDD-based verification tool

Circuits	CPU time (s)	BDD mem. (kb)
mM[16], mM[16]sim	1.7	340
mM[32], mM[32]sim	13.9	699
mM[48], mM[48]sim	24.0	1301
mM[64], mM[64]sim	29.6	2112
mult[8], mult[8]sim	14.3	741
mult[10], mult[10]sim	168.7	5036
mult[12], mult[12]sim	1725.4	43806

As a first experiment for the presented verification method, we test if all constants in the original circuit descriptions are propagated correctly by EUCLID. The resulting circuits are denoted by the suffix 'sim'. The results of these experiments are given in Table 3. The first two columns show the run times and the amount of memory required for storing the BDDs. The last column shows the number of variables representing equivalent signals. As the results demonstrate, the proposed verification method performs significantly better than the conventional one on these simple examples. This clearly illustrates that the efficiency of the method does not depend solely on the type of circuit that is verified, but also on the actual differences between the two circuits.

In the second set of experiments, we resynthesize the benchmarks with SIS. After the usual optimizations such as constant propagation, the circuit is partially collapsed and factored. This results in circuits that are harder to verify, because large parts of the original structure are modified. The suffix 'res' is used

Table 3. Experimental results for simple benchmarks

Circuits	CPU time (s)	BDD mem. (kb)	Eq. used
mM[16], mM[16]sim	0.6	395	41
mM[32], mM[32]sim	1.1	455	66
mM[48], mM[48]sim	8.9	661	144
mM[64], mM[64]sim	6.4	592	137
mult[8], mult[8]sim	0.8	379	56
mult[16], mult[16]sim	9.7	594	296
mult[24], mult[24]sim	23.6	1422	765
mult[32], mult[32]sim	33.8	2813	1407

to indicate that a circuit is synthesized with this method. The results are shown in Table 4. As these results show, it still pays off to exploit the structural equivalences, even though the verification of the larger instances of the multiplier cannot be completed; for mult[24]res and mult[32]res only 20 and 17 outputs are respectively verified successfully. This is not caused by the absence of structural equivalences, but by the size of the intermediate results during a sequence of substitutions.

Table 4. Experimental results for resynthesized circuits

Circuits	CPU time (s)	BDD mem. (kb)	Eq. used
mM[16], mM[16]res	2.4	385	15
mM[32], mM[32]res	11.2	686	43
mM[48], mM[48]res	11.4	565	32
mM[64], mM[64]res	17.9	1452	72
mult[8], mult[8]res	1.4	353	32
mult[16], mult[16]res	92.6	2134	183
mult[24], mult[24]res	—	—	—
mult[32], mult[32]res	—	—	—

In the third set of experiments, we use EUCLID to map the benchmarks onto another technology, namely a complete library of (3,3)-AOI cells [2]. Because the functions implemented by these cells are larger than the expressions in the original descriptions, the synthesis strategy also involves partial collapsing and factoring of the circuits, which modifies the original structure. The remapped circuits are denoted by the suffix 'map'. The results of these experiments are shown in Table 5. When compared to the results of the previous experiments, it can clearly be observed that the circuits resulting from these synthesis steps are easier to verify, because they have been collapsed less strongly. In this case, 46 and 62 outputs are respectively verified successfully of mult[24]res and mult[32]res.

In the fourth set of experiments, the EUCLID system is used to decrease the maximum delay of the circuits resulting from the previous experiment. This

Table 5. Experimental results for remapped circuits

Circuits	CPU time (s)	BDD mem. (kb)	Eq. used
mM[16], mM[16]map	1.5	387	16
mM[32], mM[32]map	2.9	454	63
mM[48], mM[48]map	7.0	556	54
mM[64], mM[64]map	10.8	638	74
mult[8], mult[8]map	0.8	367	39
mult[16], mult[16]map	17.2	640	201
mult[24], mult[24]map	—	—	—
mult[32], mult[32]map	—	—	—

means that segments of the critical paths in the circuits are restructured in order to improve the delay of these paths. For the min_max circuits, a speedup of 30% is obtained, and for the multipliers a speedup of 15%. The circuits that are synthesized with this approach are given the suffix 'fast'. The results are given in Table 6. Mark that in this case, the results are not verified against the original descriptions, but against the results of the previous experiments.

Table 6. Experimental results for circuits resynthesized for speed

Circuits	CPU time (s)	BDD mem. (kb)	Eq. used
mM[16]map, mM[16]fast	0.4	404	27
mM[32]map, mM[32]fast	0.9	436	46
mM[48]map, mM[48]fast	5.9	489	99
mM[64]map, mM[64]fast	6.8	540	133
mult[8]map, mult[8]fast	0.5	373	41
mult[16]map, mult[16]fast	7.0	498	203
mult[24]map, mult[24]fast	21.2	1288	505
mult[32]map, mult[32]fast	28.1	2575	926

In order to compare our results with [14], we verify some of the more difficult ISCAS benchmarks [7] against the same circuits after redundancy removal [18]. We also verify mult[16] against the ISCAS benchmark c6288, which is a 16-multiplier with a similar structure. The results are shown in table 7. Although it is generally difficult to compare run times measured on different machines, the results indicate that our verification method is at least an order of magnitude faster than the method presented in [14] on the selected benchmarks. For the other ISCAS benchmarks, the differences are relatively smaller. The verification of the 16-bit instance of the multiplier against the ISCAS benchmark c6288 is performed very efficiently, because both circuits have the same architecture, and therefore, many structural equivalences exist.

To measure the effect of dynamic variable ordering on the results, we have also performed some tests with dynamic variable ordering turned off. The re-

Table 7. Experimental results for some ISCAS benchmarks

Circuits	CPU time (s)	BDD mem. (kb)	Eq. used
c3540, c3540nr	4.5	429	94
c5315, c5315nr	13.5	505	115
c7552, c7552nr	46.7	1104	145
mult[16], c6288	24.7	703	201
mult[16], c6288nr	15.2	621	200

sults of these experiments are given in Table 8. As these results show, dynamic variable ordering has a significant impact on the memory usage for the more difficult tests. It succeeds in finding good variable orders to compactly represent intermediate results of a series of substitutions. For the easier examples, the introduction of new variables for intermediate signals ensures that the number of BDD nodes does not grow very fast in case of many structural equivalences. Therefore, dynamic variable ordering cannot improve much in these cases.

Table 8. Influence of dynamic variable ordering

Circuits	Dyn. order.		No dyn. order.	
	Time (s)	Mem. (kb)	Time (s)	Mem. (kb)
mM[64], mM[64]sim	6.4	592	2.6	653
mult[32], mult[32]sim	33.8	2813	20.4	3092
mM[64], mM[64]res	17.9	1452	8.6	1520
mult[16], mult[16]res	92.6	2134	42.5	5802

4 Conclusions and Future Work

In this paper, we have discussed two techniques that enhance the ability of BDD-based verification methods to handle larger circuits. The technique of dynamic variable ordering has been briefly discussed, because it virtually removes the need to worry about a reasonable initial variable order. Therefore, it makes the verification method more robust. We have shown how this technique can be incorporated in a BDD package in a way that is fully transparant to an application of the package. The penalty is an increase in run time, estimated at a factor 4 on average.

We have also shown how a BDD-based verification method can be extended with a technique to exploit structural similarities of the circuits under comparison. These similarities are detected fully automatically. Experimental results demonstrate that the new technique significantly extends the ability of BDD-based verification methods to deal with circuits that are structurally similar. The successful verification of various multipliers shows that it is possible to handle circuits that cannot be represented compactly with BDDs. Our current research focuses on finding a more effective technique to perform the required substitutions, because we believe that the size of the intermediate results during a sequence of substitutions is the bottleneck in our current method. We also intend to incorporate our method in an industrial design system.

Acknowledgements

We kindly acknowledge IBM Corporation, Yorktown, for making their BSN design system available to us. We would also like to thank the anonymous reviewers for their valuable comments, and Harm Arts for his help with synthesizing the benchmarks.

References

1. Ashar, P., Ghosh, A., Devadas, S.: Boolean Satisfiability and Equivalence Checking using General Binary Decision Diagrams. Proc. ICCD-91, pp. 259–264, 1991
2. Berkelaar, M.R.C.M., Theeuwen, J.F.M.: Logic Synthesis with Emphasis on Area-Power-Delay Trade-Off. Journal of Semicustom ICs, September 1991, pp. 37–42
3. Berman, C.L.: On Logic Comparison. Proc. DAC-81, pp. 854–861, 1981
4. Berman, C.L., Trevillyan, L.H.: Functional Comparison of Logic Designs for VLSI Circuits. Proc. ICCAD-89, pp. 456–459, 1989
5. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient Implementation of a BDD package. Proc. DAC-90, pp. 40–45, 1990
6. Brand, D.: Verification of Large Synthesized Designs. Proc. ICCAD-93, pp. 534–537, 1993
7. Brglez, F., Fujiwara, H.: A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN. Special session on the 1985 IEEE Int. Symposium on Circuits and Systems, 1985
8. Bryant, R.E.: Graph Based Algorithms for Boolean Function Representation. IEEE Transactions on Computers, vol. C-35 no. 8, pp. 677–691, August 1986
9. Burch, J.R.: Using BDDs to Verify Multipliers. Proc. DAC-91, pp. 408–412, 1991
10. Fujita, M., Fujisawa, H., Matsunaga, Y.: Variable Ordering Algorithms for Ordered Binary Decision Diagrams and Their Evaluation. IEEE Transactions on CAD, vol. 12 no. 1, pp. 6–12, January 1993
11. Jain, J., et al.: IBDDs: an Efficient Functional Representation for Digital Circuits. Proc. EDAC-92, pp. 440–446, 1992
12. Jeong, S.-W., et al.: Variable Ordering and Selection for FSM Traversal. Proc. ICCAD-91, pp. 476–479, 1991
13. Kropf, T.: Benchmark-Circuits for Hardware-Verification. February 1994
14. Kunz, W.: HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning. Proc. ICCAD-93, pp. 538–543, 1993
15. Mets, A.A.: Dynamic Variable Ordering for BDD Minimization. Student report Eindhoven University, Department of Electrical Engineering, January 1994
16. Rudell, R.: Dynamic Variable Ordering for Ordered Binary Decision Diagrams. Workshop Notes Int. Workshop on Logic Synthesis, 1993

17. Rudell, R.: Dynamic Variable Ordering for Ordered Binary Decision Diagrams. Proc. ICCAD-93, pp. 42–47, 1993
18. Tromp, G.J., van de Goor, A.J.: Logic Synthesis of 100-percent Testable Logic Networks. Proc. ICCD-91, pp. 428–431, 1991