FACTA UNIVERSITATIS (NIŠ)
SER.: ELEC. ENERG. vol. 20. no. 3, December 2007, 415-436.

An Integrated Approach for Combining BDDs and SAT Provers

Rolf Drechsler, Görschwin Fey, and Sebastian Kinder

Abstract: Many formal verification tools today are based on Boolean proof techniques. The two most powerful approaches in this context are Binary Decision Diagrams (BDDs) and methods based on Boolean Satisfiability (SAT). Recent studies have shown that BDDs and SAT are orthogonal, i.e. there exist problems where BDDs work well, while SAT solvers fail and vice versa. Beside this, the techniques are very different in general. E.g. SAT solvers try to find a single solution and BDDs represent all solutions in parallel.

In this paper the first integrated approach is presented that combines BDDs and SAT within a single data structure. This hybrid approach combines the advantages of the two techniques, i.e. multiple solutions can be computed while the memory requirement remains small. Experimental results demonstrate the quality of the approach in comparison to BDDs and SAT solvers.

Keywords: SAT, BDD, Hybrid Data Structure, Symbolic Technique

1 Introduction

Many problems in circuit design can easily be formulated in terms of Boolean variables. E.g. in verification or automatic test pattern generation a satisfying assignment for a Boolean formula has to be determined (see e.g. [1–3]). Several Boolean techniques to solve this problem have been proposed in the past. Among them are simulation based approaches, like random pattern simulation. But with increasing design complexity pure simulation is not sufficient to find solutions in huge search spaces. For this, complete methods based on formal proof techniques have been proposed.

Manuscript received August 15, 2007.

The authors are with the Institute of Computer Science, University of Bremen, Bremen, 28359 Germany (e-mail: [drechsle, fey, kinder]@informatik.uni-bremen.de).

The two most frequently used methods are *Binary Decision Diagrams* (BDDs) and provers for *Boolean Satisfiability* (SAT). Experimental studies have shown that these techniques are orthogonal, i.e. there exist problems where BDDs work well, while SAT solvers fail and vice versa. This trade-off can even be formally proven [4].

BDDs and SAT provers are very different in nature. While BDDs compute all solutions in parallel, they require a large amount of memory. In contrast SAT is very efficient regarding memory consumption, but only gives a single solution. There are many applications where multiple solutions are needed (see e.g. [5, 6]). Motivated by this, many authors tried to combine the best of the two approaches, by applying SAT solvers and BDDs alternatively or iteratively. Even though remarkable results have been obtained, so far none of the approaches considered an integration of the two methods within a single data structure. (A more detailed discussion of related work is given in the next section.)

In this paper we present the first approach that allows to tightly combine BDDs and SAT. Even though the overall principle of the two techniques is very different, there are also some similarities. In both concepts, starting from a Boolean description, the problem is decomposed by assigning a Boolean value to a variable. This has already been observed in [7]. For this, we introduce the concept of *expansion nodes*. The given Boolean problem is initially represented by a single expansion node that is recursively expanded. If this is done in a strict *Depth First Search* (DFS) manner, the resulting algorithm is close to a SAT procedure. But if all operations are carried out symbolically, the algorithm computes a BDD. The relation between the two approaches is discussed in more detail later. Experimental results demonstrate the efficiency of the approach.[1]

The paper is structured as follows: Related work is discussed in Section 2. SAT and BDDs are briefly reviewed in Section 3 to make the paper self-contained. Then, the relation between the two is considered. The new approach is presented in Section 4. In Section 5 experiments are presented. Finally the results are summarized and directions for future work are given.

2 Related Work

In this section we discuss earlier work that is related to our approach.

Streaming BDDs have been proposed to reduce the memory requirements [10]. The idea is to represent a BDD as a bracketed sequence. The sequence can be processed sequentially using limited memory. But this can only be done by giving

[1]Preliminary versions of this paper have already been published in [8, 9].

up canonicity.

In the context of extensions of the classical BDD concept introduced by Bryant [11], some approaches have been presented that make use of different types of functional nodes.

The approach in [12] keeps control of the memory needed for the BDD construction by projecting some parts of the graph to a new terminal node U (=unknown). Instead of completely calculating each subgraph, the calculation may be stopped at a given depth and the complete tree is replaced by the terminal node U. As a result, exactness cannot be recovered afterward.

Nodes to represent the exclusive-or of the children have been introduced in [13]. The purpose of these nodes is to reduce the size of the BDD. Then, probabilistic methods are applied to find a satisfying assignment may take a significant computational effort.

Extended BDDs as proposed in [14] apply existential quantification and universal quantification as edge attributes. By introducing a so-called "structural variable" s, the equality $\exists_s f = f_s + f_{\overline{s}}$ can be exploited to represent the Boolean operation $f+g$ in terms of a node v. This can be seen as follows: Let v be a node and f and g be the Boolean functions represented by its children. Then, v represents the function $sf+\overline{s}g$. Now, assume an incoming edge has the attribute for existential quantification. The function represented by this edge is retrieved as follows:

$$\begin{aligned} \exists_s(sf+\overline{s}g) &= (sf+\overline{s}g)_s + (sf+\overline{s}g)_{\overline{s}} \qquad \text{(as introduced above)} \\ &= f+g \end{aligned}$$

Similarly, universal quantification is used to represent $f \cdot g$. These structural variables allow to control the size of the extended BDD. Again, the problem is to find a satisfying assignment of the resulting extended BDDs.

The same principle was exploited in [15]. By introducing extra nodes at the top level of two BDDs, a Boolean operation is represented. Then, these nodes are moved towards the terminals by exchanging adjacent variables. At the terminals these nodes can be eliminated. In both cases the use of new variables implies that a new level is introduced in the shared BDD structure.

The approach was further extended in [16] for *Boolean Expression Diagrams* (BEDs). Functional nodes that directly represent Boolean operations were introduced. Again, these nodes can be eliminated by swapping adjacent levels in the BED. If a BED is built from a description of a circuit, the size of a BED is similar to the circuit size. All of these approaches are presented as extensions of BDDs. The advantage of using SAT-like algorithms on such a structure has not been considered.

Another recent direction of research are efficient all-solution SAT solvers that do not stop after reaching the first satisfying assignment but calculate all possible satisfying solutions, e.g. [17]. A drawback of these approaches is the potentially large representation of all solutions usually as cubes or as BDDs. In contrast, the hybrid approach proposed here targets applications where not *all* but a set of *good* solutions is needed.

Recently, several techniques have been proposed to combine BDDs and SAT solvers (see e.g. [18–21]), but no real integration is done. Instead, the proof engines are started one after the other, or alternating. By this, good experimental results have often been obtained, demonstrating the potential of an integrated approach.

3 Proof Techniques

In this section we briefly review BDDs and SAT. Then the relation between the two is discussed to provide a better understanding and a motivation for the hybrid approach presented below.

3.1 BDD

As well-known each Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) [11], i.e. a directed acyclic graph where the Shannon with respect to a variable x_i is carried out in each node:

$$f = \overline{x}_i f_{x_i=0} + x_i f_{x_i=1}$$

In the following $f_{x_i=0}$ is called *low-child* and $f_{x_i=1}$ is called *high-child*.

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it does not contain vertices either with isomorphic subgraphs or with both edges pointing to the same node.

Reduced, ordered BDDs are a canonical data structure for Boolean functions and allow efficient manipulations [11]. In the following only reduced, ordered BDDs are considered and for briefness these graphs are called BDDs.

3.2 SAT

Let f be a Boolean function in *Conjunctive Normal Form* (CNF), i.e. in a product-of-sum representation. Then, the problem of *Boolean Satisfiability* (SAT) is to determine an assignment of the variables of f such that f evaluates to 1 or to prove that such an assignment does not exist.

Example 1. *Let* $f = (x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_3)(\overline{x}_2 + x_3)$. *Then* $x_1 = 1$, $x_2 = 1$ *and* $x_3 = 1$ *is a satisfying assignment. The values of* x_1 *and* x_2 *ensure that the first sum becomes* 1, *while* $x_3 = 1$ *ensures this for the remaining sums.*

In many applications, like formal verification and automatic test pattern generation, the problem is initially given in the form of a circuit. This circuit can be transformed to a CNF by a simple transformation. Afterwards, the CNF is solved using a SAT solver.

Recently, several very powerful SAT provers have been developed that make use of e.g. Boolean constraint propagation and clause recording to speed up the proof process [22–24].

Note, in the following SAT also refers to algorithms to solve the Boolean Satisfiability problem.

3.3 Discussion

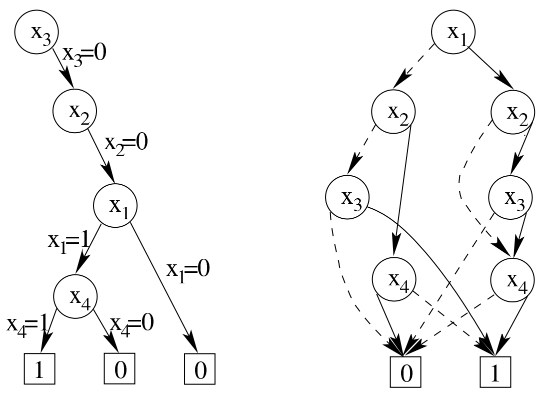
Both techniques have advantages and disadvantages. While BDDs represent all solutions in parallel at the cost of large memory requirements, A SAT solver only provides a single solution, while the memory needed is very low. In [7] the relation between BDDs and SAT has been studied from a theoretical point of view. It has been proven that the BDD corresponds to a complete representation of the SAT backtrack tree, if a fixed variable order is assumed.

As a motivation for the next section, where our approach is described in more detail, an example is given to show the main difference between SAT and BDDs. We will later come back to this example.

Example 2. *Consider a Boolean function* f *over four variables given by:*

$$\begin{aligned} f &= (x_1+x_2+x_3)(x_1+\overline{x}_2+\overline{x}_4)(\overline{x}_1+x_2+x_4) \\ &\quad (\overline{x}_1+\overline{x}_2+x_3)(\overline{x}_1+\overline{x}_2+\overline{x}_3+x_4) \end{aligned}$$

A sketch of the search tree, if the function is processed by a SAT solver is shown in Figure 1(a). The corresponding BDD is given in Figure 1(b). As can be seen, the SAT solver by construction only gives a single solution, while the BDD represents satisfying assignments in parallel at the cost of a larger number of nodes.



(a) SAT search tree (b) BDD

Fig. 1. Different approaches

4 Hybrid Approach

In this section we describe our approach for BDD and SAT integration. First, the overall idea is given. Then the concept of *expansion nodes* is introduced followed by a discussion of expansion heuristics. Finally, we comment on some issues related to an efficient implementation.

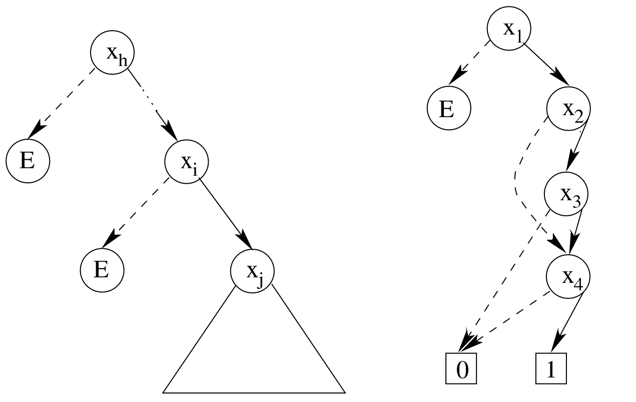
4.1 Basic Idea

In our approach we start the processing by symbolic operations analogously to BDDs. For the operations the ITE operator [25] has been modified. During the starting phase, the constructed graphs are simply BDDs. But when composing BDDs a heuristic is used to decide, which parts of the solution space are explored.

To guarantee that the algorithm is exact, i.e. no solution is missed, a node is introduced where the computation can be resumed. These nodes are called *expansion nodes* in the following. By this, our approach stores all necessary information resulting in a complete proof method.

A sketch of a configuration during the run is shown in Figure 2(a). In this case the upper part is “SAT-like” while the lower part is a complete symbolic representation as it occurs in BDDs. The expansion nodes are denoted by *E*. The decomposition nodes are labeled by variables, these variables occur in the same order on all paths. The order is fixed, i.e. the variables cannot be reordered. In the following we refer to such graphs that allow a smooth transition between SAT and BDDs as *hybrid structure*.

Remark 1. *Several expansion nodes in a hybrid structure may represent the same function. This cannot be detected before completely expanding the node. Thus, a hybrid structure is not a canonical representation of Boolean functions.*



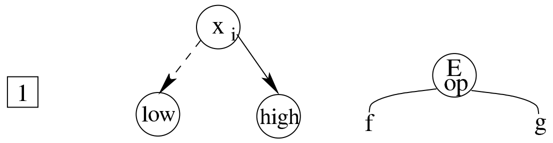
(a) Sketch of the hybrid approach (b) Hybrid representation

Fig. 2. Hybrid approach

4.2 Expansion Nodes

The hybrid approach makes use of three types of nodes (see Figure 3):

(a) Terminal nodes
(b) Decomposition nodes
(c) Expansion nodes



(a) Terminal (b) Decomposition node (c) Expansion node

Fig. 3. Overview over different node types

The first two can also be found in BDDs. Terminal nodes represent the constant functions 0 and 1. In decomposition nodes the Shannon decomposition is carried out.

Expansion nodes are labeled by a Boolean operation op and have two successors f and g, that represent Boolean functions (which are also denoted by f and g for simplicity). The expansion node represents the function $f\ op\ g$.

Example 3. *Consider again the function from Example 2 and Figures 1(a) and 1(b). A possible hybrid structure is shown in Figure 2(b). This one results if the top variable is only decomposed in one direction, while on the other branch an expansion node is placed. As can be seen the structure is more memory efficient. Compared to the BDD five instead of seven nodes are needed. At the same time* *three solutions are represented in contrast to the SAT approach that only returns a single solution.*

This simple example demonstrated that the approach combines the two proof techniques SAT and BDD. A crucial point to address is where to place the expansion nodes. For this, we propose a heuristic in the next section.

4.3 Expansion Heuristics

Inserting expansion nodes at suitable locations is crucial for the approach to work. If too many expansion nodes are inserted, no solutions can be found. Only structures without a path to a terminal will be constructed and the expansion of partial trees will take most of the run time until computing a solution. On the other hand not inserting enough expansion nodes will lead to a memory blow-up as known from BDDs.

In a BDD-based approach the final solutions are computed by composing intermediate BDDs. This is similar for the new approach. The following steps are necessary to retrieve solutions:

(1) Build BDDs for basic functions without any expansion nodes. For example, the clause $(x_1 + x_2 + x_3)$ from Example 2 may be built completely without using expansion nodes.
(2) Compose the function and insert expansion nodes according to a predetermined heuristic.
(3) Select expansion nodes to expand the hybrid structure and obtain (further) solutions.

Which functions are considered as basic functions in step (1) depends on the problem and the input format, e.g. projection functions and cubes were chosen in our experiments. Building BDDs for these basic functions is not necessary for the approach to work, but having the basic functions completely represented improves the performance drastically by reducing the number of necessary expansions.

In the following several heuristics to limit the size of the resulting hybrid structure in step (2) have been evaluated:

(S1) A fast procedure is to directly limit the memory consumption. This limit can be determined efficiently. Once the limit is reached no further decomposition nodes are created, but only expansion nodes. Therefore, prior to performing an expansion the memory limit is increased by a user defined value.
(S2) The second procedure is to limit the number of nodes in a subgraph to a certain threshold. Tracking this limit is computationally more expensive. But allowing more than n nodes in a subgraph guarantees that there is at least one path to a terminal node. I.e. for at least one assignment the function can directly be evaluated.

```
Node∗ DFS(N){
      if(isTerminal(N)) return NULL;
      if(isFuncNode(N)) return N;
      tmp = DFS(Nhigh);
      if(tmp) return tmp;
      tmp = DFS(Nlow);
      return tmp;
}
```

Fig. 4. Depth first traversal

The selection of nodes to expand in step (3) has also been done using different heuristics:

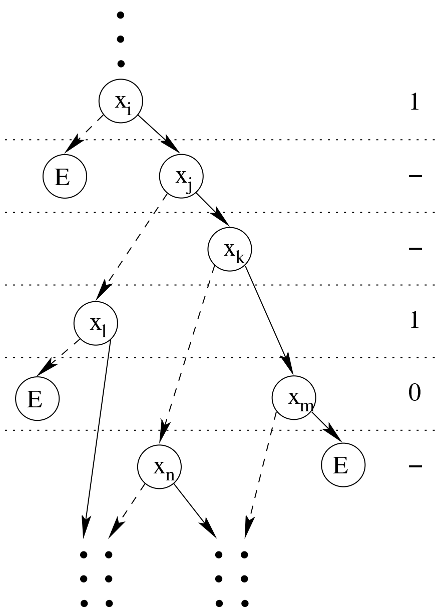


Fig. 5. Cube heuristic

(E1) Randomly

(E2) "DFS-like" (using the algorithm in Figure 4): The hybrid structure is traversed in a depth first manner until an expansion node is reached. This node is selected and then expanded by carrying out the stored operation. I.e., they are expanded with the operation they are labeled with. The same scheme is applied recursively if further selections are necessary.

```
 1  Node∗ applyCube( F, G, op, cube ){
 2    if(terminalCase) return result;
 3    if(computedTableHasEntry( F, G, op ))
 4      return result;
 5    index = topVariable( F, G );
 6    if(cube[index] == 1){
 7      Rlow  = expNode( Flow, Glow, op );
 8      Rhigh = applyCube( Fhigh, Ghigh, op, cube );
 9    } else if(cube[index] == 0){
10      Rlow  = applyCube( Flow, Glow, op, cube );
11      Rhigh = expNode( Fhigh, Ghigh, op );
12    } else if(cube[index] == −){
13      Rlow  = applyCube( Flow, Glow, op );
14      Rhigh = applyCube( Fhigh, Ghigh, op);
15    }
16    if (Rlow == Rhigh) return Rlow;
17    R = findOrAddUniqueTable( index, Rlow, Rhigh );
18    insertComputedTable( F, G, op, R);
19    return R;
20  }
```

Fig. 6. Apply for Heuristic SE3

Alternatively, there is a heuristic which integrates both, a heuristic for composing and a heuristic for expanding the hybrid structure:

(SE3) "Cube-oriented": Exemplarily, a hybrid structure and the corresponding cube are sketched in Figure 5. This cube is defined over the input variables. For the example given in Figure 5 the cube is:

$$\cdots 1 - - 1 0 - \cdots$$

In Figure 6 the algorithm for the composition using this heuristic is presented. In general, this algorithm corresponds to the standard apply algorithm used for BDDs. To obtain an apply algorithm for the heuristic lines 6 to 12 have to be added. An additional function *expNode* (Lines 7 and 11), which inserts expansion nodes into the structure, is implemented. The algorithm in Figure 6 also has a new parameter *cube*. The cube is a sequence of the tokens: '1', '0' and '−'. A '1' in this sequence means that only the high-child of a node is calculated and for the low-child an expansion node is inserted (lines 6 – 8). A '0' indicates that only the low-child is calculated (lines 9 – 11) and a '−' indicates that both children have to be calculated (lines 12 – 15). The expansion part of this heuristic works in a similar way. Now, if such a cube is used to expand nodes in the hybrid structure the cube has to be changed, e.g.:

$$\cdots 1 - - 1 - - \cdots$$

An expansion with this cube would result in the complete calculation of variable x_m (i.e. all reachable nodes of this level within the hybrid structure).

The heuristic to choose a cube depends strongly on the given application. A heuristic is exemplarily shown in Section 5.3.

Heuristic (E2) ensures a moderate growth of the memory needs. Experimental studies showed that the combination of a hard limit on memory consumption (S1) with deterministic DFS (E2) gives the best results, i.e. small run times and a large number of solutions. From a more general point of view this combination of heuristics leads to a SAT-like search tree in the upper part of the hybrid structure which is enriched by a BDD-like lower part. These heuristics are well applicable if there is no information about the search space which has to be explored. In contrast, heuristic (SE3) allows for a targeted exploration of the search space. Hence, this heuristic is applied if there are already information about the search space which is to be explored.

Remark 2. *When using heuristics (S1) and (E2) in combination the search space is traversed similar as with "BDDs at SAT leaves" in [18, 26]. But the proposed hybrid structure is more general in the sense that switching between SAT-like and BDD-like behavior is subject to heuristics.*

Remark 3. *During expansion canonicity is also an issue. When expanding a node, a function that is already represented by another node may be the result. The hybrid structure can be reduced at a computational cost linear in the number of nodes using an algorithm similar to [27]. In our implementation no reduction was carried out to save run time.*

4.4 Implementation

The technique described above has been integrated into the CUDD package [28], where the core data structures are taken from. To store the expansion nodes, the structure for storing nodes has been extended (see line 8 in Figure 7). The structure for the new type is given in lines 11-14.

Table 1. Index of node types (32-bit)

Node type	Index
decomposition nodes	0 - 65532
XOR-node	65533
AND-node	65534
terminal node	65535

```
struct Node {
    HalfWord index;
    HalfWord ref;
    Node *next;
    union {
        Terminal value;
        Children kids;
        ExpNode func;
    }
}
struct ExpNode{
    Node *F;
    Node *G;
}
```

Fig. 7. Modified node structure

In case of an expansion node, also the operation has to be stored. For reasons of efficiency we restrict ourselves to store only operations of type AND and XOR. Negation is realized by complemented edges [25]. All other Boolean operators are mapped accordingly. The information is stored in the index of each node. The complete encoding is given in Table 1, i.e. three indices have a special meaning, while all the remaining ones are used for decomposition variables. A hash table is used for the expansion nodes. Therefore, a particular expansion node with the stored nodes f and g and the operation op exists only once.

5 Applications and Experimental Results

In this section experimental results are presented. First, heuristics (S1), (S2), (E1) and (E2) are evaluated with the well known n-Queens problem which is considered as an example of a combinational problem where BDDs are known to perform poorly on large instances while a large number of solutions is available. In this way, the best combination of heuristics is determined. Afterwards, the combination if heuristics is applied to the synthesis problem of minimizing *EXOR-Sum-Of-Product* (ESOP) representations. This optimization problem is known to be hard. Often, ESOP representations are optimized heuristically. Hence, a framework to estimate the quality of heuristically obtained ESOP representations is introduced in the last section using a problem specific heuristic ((SE3)). In this case the transition from “SAT-like” to “BDD-like” behavior of the hybrid approach is of particular importance.

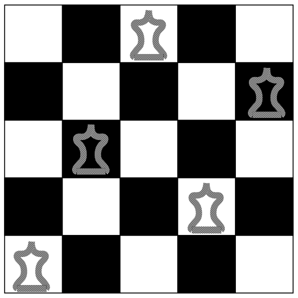


Fig. 8. Solution for the 5-Queens problem

5.1 *n*-Queens

The *n*-Queens problem is a well-known combinational problem. The objective is to place *n* queens on an $n \times n$ board such that no queen can be captured by another one. An example for a solution of the 5-Queens problem is shown in Figure 8. This game problem is encoded using n^2 binary input variables, each one deciding, if a queen is placed on the corresponding field of the chess board or not. Obviously, the constraints are to place one queen per row and column and at most one queen per diagonal.

Table 2. Heuristics to limit the size of the hybrid structure

			Limit for the size			
		BDD	Memory (S1)		Subgraph (S2)	
n	#sol.	sec.	sec.	overhead	sec.	overhead
6	4	0.00	0.00	-	0.01	-
7	40	0.01	0.01	0.00 %	0.03	200.00 %
8	92	0.05	0.06	20.00 %	0.18	260.00 %
9	352	0.37	0.37	0.00 %	1.30	251.35 %
10	724	1.56	1.59	1.92 %	8.20	425.64 %
11	2680	7.81	7.82	0.13 %	62.39	698.84 %
12	14200	48.12	48.54	0.87 %	490.33	918.97 %
13	73712	352.11	353.21	0.31 %	4566.75	1196.97 %

The experiments given here have been carried out on an Intel Pentium 4 processor with 3 GHz and 1 GByte of main memory running Linux. In a first experiment the heuristics to limit the size were considered. For all experiments the limits were loose enough to retrieve all solutions. Therefore the overhead of the heuristics to limit the size can directly be measured in comparison to BDDs. Results are reported in Table 2. Given are the number of solutions for increasing values of *n* and run times in CPU seconds for BDDs and the two heuristics introduced in Section 4.3, respectively. The resource requirements for BDDs increase rapidly and no further solutions beyond $n = 13$ could be retrieved. Also the computational overhead of limiting the size of subgraphs using heuristic (S2) is too large. But directly limiting the memory consumption according to heuristic (S1) does introduce almost no overhead. This heuristic has been used in all remaining experiments to restrict the size.

The performance of heuristics to select nodes for expansion has been investigated in the next experiment. Expansion was carried out until a total memory limit of 750 MB was reached. Due to the expansion of subfunctions more than one solution can be contained in the final representation. The results are shown in Table 3. Up to $n = 13$ all solutions were obtained with both heuristics.

Table 3. Selection of expansion nodes

		Randomly (E1)		DFS (E2)	
n	#var	#sol.	sec.	#sol.	sec.
3	9	0	0.00	0	0.00
4	16	2	0.00	2	0.00
5	25	10	0.00	10	0.00
6	36	4	0.00	4	0.00
7	49	40	0.02	40	0.01
8	64	92	0.06	92	0.06
9	81	352	0.37	352	0.37
10	100	724	2.10	724	1.83
11	121	2680	16.54	2680	10.30
12	144	14200	158.86	14200	73.34
13	169	73712	2062.39	73712	578.54
14	196	0	384.45	56672	1836.93
15	225	0	289.01	33382	1669.50
16	256	0	652.64	20338	2555.35
17	289	0	1366.25	5061	2055.97
18	324	0	693.13	204	2238.79
19	361	0	529.37	1428	3357.97
20	400	0	1923.07	38	1592.94
21	441	0	1957.39	111	1972.60

Then, the random selection performs very poorly. When expanding the last node in a cascade of expansion nodes new decomposition nodes are created. But the next expansion will often occur at an expansion node in a different subgraph. Thus, the previously created decomposition nodes cannot be utilized for the next step.

In contrast the deterministic DFS starts the next expansion where new decomposition nodes have been constructed previously. As a result the new approach yields solutions up to $n = 21$ in a moderate amount of time.

5.2 ESOP Minimization

Compared to a SOP-representation of a function the ESOP-representation can be exponentially smaller. But most algorithms for ESOP minimization only apply local transformations to improve from an initial solution, e.g. [29, 30]. In [31] the problem to compute an ESOP for a given Boolean function f over n variables has been formulated using the Helliwell equation. The Helliwell equation H_f for function f has 3^n input variables, each input variable corresponds to a cube and is 1, iff this cube is chosen for the ESOP of f. A satisfying assignment to H_f determines an ESOP for f and vice versa. The Helliwell equation is defined in Equation 1.

$$\begin{aligned} H_f(c) &= \prod_{a\in\mathbb{B}^n}\Big[(\bigoplus_{b\in\mathbb{D}} c_b) \equiv f(a)\Big] \qquad (1) \\ \mathbb{D} &= \{(d_0,\ldots,d_n)|a_k = d_k \vee a_k = -\} \\ & \quad\; n,k \in \mathbb{N}; 0 \le k \le n; d \in \{0,1,-\}^n \end{aligned}$$

In Equation (1) all assignments a of a function f are iterated. The correct value of $\oplus$-sums of the cubes c_b is enforced by the comparison to $f(a)$. Afterwards, all valid $\oplus$-sums of the cubes are joint to the Helliwell equation $H_f(c)$. For a detailed description we refer to [31].

The hybrid structure was built for the Helliwell equation. By additional constraints the number of cubes was limited to be at most k. The experimental results for applying this method to $f = \bigoplus_{i=1}^{4} x_i$ are shown in Table 4. Given are results for using BDDs, the hybrid structure, and the SAT solver zchaff [23]. We modified the SAT solver zchaff to calculate more than one solution: For each solution a blocking clause is added and the solve process is continued. For the hybrid structure results are reported when different numbers of solutions are calculated: more than 1, more than 10^3 and more than 10^6 solutions, respectively. For different values of k the CPU time in seconds, the memory requirements in kB and the number of nodes in the BDD or the hybrid structure, respectively, are reported (measured on a computer with an Intel Pentium 4 processor with 3 GHz and 1 GByte of main memory). For zchaff the CPU time is given. The number of available solutions is not reported, but grows rapidly. While there are only 38 valid solutions for $k = 4$, there are 564 for $k = 5$ and more than $3.3 \cdot 10^7$ for $k = 10$.

The results show the superiority of the hybrid approach compared to BDDs. For a tightly restricted solution space ($k < 25$) BDDs are feasible. But after that the memory and especially the run time requirements grow prohibitively fast. In contrast the hybrid approach exhibits a rather stable performance as CPU time and memory requirements remain in the same order for all runs. The increased run time for $k = 10, 15$ when calculating more than 10^6 solutions is due to the small number of possible solutions. In this case a large part of the BDD has to be recreated using the expansion technique without retrieving more solutions. In this case BDDs are

Table 4. ESOP minimization

	BDD			hybrid structure									zchaff		
	all solutions			≥ 1 solution			$\geq 10^3$ solutions			$\geq 10^6$ solutions			1 sol.	10^3 sol.	10^6 sol.
k	sec.	kB	#nodes	sec.	kB	#nodes	sec.	kB	#nodes	sec.	kB	#nodes	sec.	sec.	sec.
4	0.55	16433	628	0.50	16449	568	0.53	16466	1108	0.53	16466	1108	<0.01	0.07	0.07
5	0.58	16483	4075	0.53	16450	638	0.60	16534	4729	0.61	16534	4729	<0.01	0.09	0.09
10	1.75	23610	420655	0.47	16450	145	0.70	16728	11597	51.28	19140	155018	<0.01	0.14	-
15	4.96	49270	1428139	0.48	16468	352	0.61	16744	11634	10.17	19420	172422	<0.01	0.11	-
20	53.96	65539	2444782	0.47	16484	112	0.54	16670	7459	1.13	19516	177708	<0.01	0.32	-
25	1945.01	84280	3449866	0.48	16500	490	0.52	16582	5465	0.98	18732	133396	<0.01	0.37	-
30	9985.37	99752	4441463	0.49	16500	495	0.49	16534	2618	0.66	17395	48107	<0.01	0.12	-
35	13900.22	113883	5361182	0.52	16500	544	0.51	16516	878	0.75	16931	21608	<0.01	0.16	-
39	13913.44	123635	5906441	0.44	16500	217	0.45	16516	1241	0.53	16662	5910	<0.01	0.09	-

faster. But usually even calculating a large number of solutions does not degrade the performance of the new approach.

When calculating a single solution, the SAT solver is faster. But even for calculating 10^3 solutions the computation time increases significantly. Finally, when calculating a large number of solutions the added blocking clauses lead to a memory blow-up even for the SAT solver. Using a more sophisticated approach the blocking clauses could be compacted, but only at the expense of CPU time for logic optimization. By this the new approach provides a good compromise between a SAT-based approach and a BDD-based approach.

Nevertheless, ESOP minimization is a very time consuming task. Thus, ESOP representations are often obtained heuristically, but their quality is usually unknown. Therefore, a measure is needed to determine the quality. The application of the hybrid structure to estimate the quality of such representations is presented in the next section.

5.3 Estimating the Quality of ESOP Optimization Results

To estimate the quality of a solution four steps are applied. These steps are briefly outlined below. More details are given afterwards.

(1) Obtain an initial ESOP representation for a Boolean function f.
(2) Construct the Helliwell equation H_f for f.
(3) Derive the neighborhood of the initial solution.
(4) Calculate all solutions of H_f within the neighborhood.

(1) Initial Solution: An initial minimization solution has to be calculated. This is done heuristically. In our framework either Mint [32] or the truth-table of the function were used.

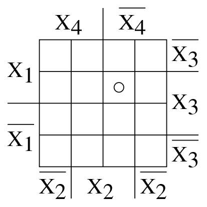
(2) Helliwell Equation: The Helliwell equation is constructed. For details about the Helliwell equation see Equation (1) in Section 5.2.

(3) Neighborhoods To estimate whether a good solution has been found, the neighborhood of the initial solution is considered. Thus, it can be determined whether the initial solution was a local minimum and whether there are local transformations which can improve the solution further.

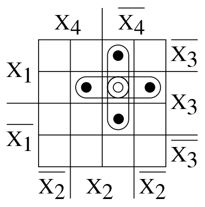
Given a single cube c the 1-neighborhood is determined by all cubes that can be derived by extending c in one dimension or by inverting c with respect to one dimension.

Example 4. *Consider the cube* `c1110` *from the Karnaugh map in Figure 9(a). The set of cubes belonging to the corresponding* 1*-neighborhood is:*

```
c0110  c1010  c1100  c1111
c-110  c1-10  c11-0  c111-
```



(a) Chosen cube



(b) 1-Neighborhood

Fig. 9. Cube and its neighborhood

The Karnaugh map for this neighborhood is shown in Figure 9(b).

This is done for all cubes chosen in the initial solution. Afterwards, all sets of neighboring cubes are joined to retrieve the 1-neighborhood of the solution.

In a similar way the *m*-neighborhood is determined. The *m*-neighborhood is derived by extending a cube *c* in up to *m* dimensions and by inverting *c* with respect to up to *m* dimensions. An upper bound for the number of cubes neighboring a solution is:

$$min\left(\left(\sum_{i=1}^{m} k \cdot 2^i \cdot \binom{n}{i}\right), 3^n\right) \tag{2}$$

In Equation (2), *k* is the number of initially chosen cubes, *m* is the index of the neighborhood and *n* is the number of inputs of the corresponding function. The first part of the upper bound is composed of the product of $\binom{n}{m}$ possibilities to choose the dimension and 2^m options to choose a cube in this dimension. This is done for all *k* chosen cubes. Since the (*m*-1)-neighborhood is a subset of the *m*-neighborhood, all numbers of cubes of previous neighborhoods have to be added up.

There are cubes which may occur in different neighborhoods of cubes from the initial solution. Thus, the formula given is an upper bound. Since the underlying minimization algorithm is exact a minimal solution is obtained if the bound of 3^n allowed cubes is reached.

(4) Calculation To calculate the Helliwell equation the hybrid data structure introduced in Section 4 was used. The hybrid structure using the cube heuristic (SE3) facilitates a targeted exploration of the relevant search space, while irrelevant parts are not calculated at all. Heuristic (SE3) was also chosen because the initial solution and its neighborhood defines the search space to be explored using the hybrid structure. The Helliwell equation was calculated using the initial solution and the hybrid data structure. Afterwards, the hybrid structure was computed using the neighborhood of the initial solution. Circuits from the LGSynth93 benchmark set were chosen to be minimized. In case of a multi output function always the first output was considered. Because of the large search space defined

by the neighborhoods (see Equation (2)), only the 1-neighborhood was calculated. The experiments were terminated by “soft limits” after either 1 hour or 768 MB of memory usage evaluated at particular checkpoints, e.g. after every expansion step. If these limits could not be met, the experiments were aborted without evaluation if they needed more than two hours or consumed more than 1 GB of main memory. The experiments have been carried out on a computer with two DualCore 64-Bit Opteron 2,8GHz CPUs and 32 GB main memory.

Table 5. Quality estimation of ESOP optimizations

			truth-table					mint				
				hybrid technique					hybrid technique			
function	n	#var	k	#sol.	min	sec.	%	k	#sol.	min	sec.	%
cm42a	4	81	4	3	4	6694.55	76.76	4	1	4	26.89	100.00
xor4	4	81	8	144	8	5093.57	58.33	4	23	4	3.69	100.00
cm82a	5	243	4	9	4	5894.74	50.00	3	3	3	2176.66	64.52
squar5	5	243	6	66	3	6610.86	46.00	3	3	3	5718.69	93.55
majority	5	243	10	4	7	5592.06	22.67	5	-	-	ABORTED	-
xor5	5	243	16	1	16	784.82	15.18	5	23	5	2448.27	80.49
rd53	5	243	16	1	16	784.52	15.81	5	79986	5	442.31	80.49
cm138a	6	729	6	1	6	2911.74	20.59	6	1	6	2235.38	56.90
con1	7	2187	9	1	9	931.77	8.85	4	-	-	ABORTED	-
z4ml	7	2187	36	-	-	ABORTED	-	13	86	13	1524.94	18.90
rd73	7	2187	64	-	-	ABORTED	-	7	3	7	1512.90	31.31
sqrt8	8	6561	2	1	2	524.76	25.00	3	1	3	898.90	43.18

In Table 5 computational results are shown. In column *function* the name of the corresponding function is written. In columns *n* and *#var* the number of inputs of the original functions and the number of inputs of the functions representing the Helliwell equation are given. The two big columns *truth-table* and *mint* show the results starting from the initial solution obtained from the truth-table and from Mint, respectively. The number of cubes chosen by the initial solution is presented in columns *k*. The number of chosen cubes of the best solution obtained from the neighborhood is provided in column *min*. The number of solutions calculated is given in the corresponding column *#sol.*. In column *sec.* the run time in seconds is shown. The calculation of the 1-neighborhood could not be finished for most experiments. The percentage of the calculated neighborhood is presented in column *%*.

As can be seen for the experiments starting with an initial solution obtained by Mint the 1-neighborhood was calculated farther. On average 66.93% of these neighborhoods were explored. For the experiments starting with a solution obtained from the truth-table the 1-neighborhoods were only explored to 33.92% on average. That is because most solutions obtained from the truth-table contain more cubes than the ones obtained by Mint resulting in a larger neighborhood according to Equation (2). For smaller functions, e.g. *xor4* and *cm42a*, even the complete 1-neighborhood was calculated using the initial solution from Mint. In these cases it is guaranteed that no better solution can be found by local transformations within the 1-neighborhood. For larger functions, e.g. *xor5*, *rd53* or *cm138a*, the 1-neighborhood was explored to a high percentage. In the explored search space no better than the initial solution was found. In contrast, better solutions were found in the 1-neighborhood for two functions, *majority* and *squar5*, when starting with an initial solution from the truth-table.

In this way, a quality measure for heuristically obtained ESOP representations is facilitated using the hybrid structure in unity with a heuristic that allows for a directed search space exploration.

6 Conclusions and Future Work

We introduced a new approach to handle satisfiability problems. This approach can be seen as an integrated technique using BDDs and SAT solvers and incorporates benefits of both: The memory consumption can be limited while calculating a large number of solutions in a single run. Heuristics have been proposed and evaluated. Experiments show the efficiency of the hybrid technique in contrast to classical approaches.

Future work consists of the introduction of powerful learning techniques as known from the SAT domain to composition and expansion heuristics. The application to formal verification will be examined. Suitable heuristics for such applications have to be developed.

References

[1] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Trans. on CAD*, vol. 15, pp. 1167–1176, 1996.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[3] J. Shi, G. Fey, R. Drechsler, A. Glowatz, F. Hapke, and J. Schlöffel, "PASSAT: Efficient SAT-based test pattern generation for industrial circuits," in *IEEE Annual Symposium on VLSI*, 2005, pp. 212–217.

[4] J. F. Groote and H. Zantema, "Resolution and binary decision diagrams cannot simulate each other polynomially," *Discrete Applied Mathmatics*, vol. 130, no. 2, pp. 157–171, 2003.

[5] G. Fey and R. Drechsler, "Finding good counter-examples to aid design verification," in *ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2003, pp. 51–52.

[6] C. Haubelt, J. Teich, R. Feldmann, and B. Monien, "SAT-based techniques in system synthesis," in *Design, Automation and Test in Europe*, vol. 1, 2003, pp. 11 168–11 169.

[7] S. Reda, R. Drechsler, and A. Orailoglu, "On the relation between SAT and BDDs for equivalence checking," in *Int'l Symp. on Quality Electronic Design*, 2002, pp. 394–399.

[8] R. Drechsler, G. Fey, and S. Kinder, “An integrated approach for combining BDD and SAT provers,” in *VLSI Design Conf.*, 2006, pp. 237–242.

[9] S. Kinder, G. Fey, and R. Drechsler, “Estimating the quality of AND-EXOR optimization results,” *Int’l Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 2007.

[10] S. Minato, “Streaming BDD manipulation,” *IEEE Trans. on Comp.*, vol. 51, no. 5, pp. 474–485, 2002.

[11] R. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[12] D. Ross, K. Butler, R. Kapur, and M. Mercer, “Fast functional evaluation of candidate OBDD variable ordering,” in *European Conf. on Design Automation*, 1991, pp. 4–9.

[13] C. Meinel and H. Sack, “$\oplus$-OBDDs - a BDD Structure for Probabilistic Verification,” in *Workshop on Probabilistic methods in Verification*, 1998, pp. 141–151.

[14] S. Jeong, B. Plessier, G. Hachtel, and F. Somenzi, “Extended BDD’s: Trading of canonicity for structure in verification algorithms,” in *Int’l Conf. on CAD*, 1991, pp. 464–467.

[15] A. Hett, R. Drechsler, and B. Becker, “MORE: Alternative implementation of BDD packages by multi-operand synthesis,” in *European Design Automation Conf.*, 1996, pp. 164–169.

[16] H. Andersen and H. Hulgaard, “Boolean expression diagrams,” in *Logic in Computer Science*, 1997, pp. 88–98.

[17] B. Li, M. Hsiao, and S. Sheng, “A novel SAT all-solutions solver for efficient preimage computation,” in *Design, Automation and Test in Europe*, 2004, pp. 272–277.

[18] A. Gupta, Z. Yang, P. Ashar, and A. Gupta, “SAT-based image computation with application in reachability analysis,” in *Int’l Conf. on Formal Methods in CAD*, ser. LNCS, vol. 1954, 2000, pp. 354–371.

[19] G. Cabodi, S. Nocco, and S. Quer, “SAT-based bounded model checking by means of BDD-based approximate traversals,” in *Design, Automation and Test in Europe*, 2003, pp. 898–903.

[20] S. Safarpour, G. Fey, A. Veneris, and R. Drechsler, “Utilizing don’t care states in SAT-based bounded sequential problems,” in *Great Lakes Symp. VLSI*, 2005, pp. 264–269.

[21] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification,” *IEEE Trans. on CAD*, vol. 21, no. 12, pp. 1377–1394, 2002.

[22] J. Marques-Silva and K. Sakallah, “GRASP – a new search algorithm for satisfiability,” in *Int’l Conf. on CAD*, 1996, pp. 220–227.

[23] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *Design Automation Conf.*, 2001, pp. 530–535.

[24] E. Goldberg and Y. Novikov, “BerkMin: a fast and robust SAT-solver,” in *Design, Automation and Test in Europe*, 2002, pp. 142–149.

[25] K. Brace, R. Rudell, and R. Bryant, “Efficient implementation of a BDD package,” in *Design Automation Conf.*, 1990, pp. 40–45.

[26] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, “Partition-based decision heuristics for image computation using SAT and BDDs,” in *Int’l Conf. on CAD*, 2001, pp. 286–292.

[27] D. Sieling and I. Wegener, “Reduction of BDDs in linear time,” *Information Processing Letters*, vol. 48, no. 3, pp. 139–144, 11 1993.

[28] F. Somenzi, “Efficient manipulation of decision diagrams,” *Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 171–181, 2001.

[29] D. Brand and T. Sasao, “Minimization of AND-EXOR expressions using rewrite rules,” *IEEE Trans. on Comp.*, vol. 42, pp. 568–576, 1993.

[30] A. Mishchenko and M. Perkowski, “Fast heuristic minimization of exclusive-sums-of-products,” in *Int’l Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 2001, pp. 242–250.

[31] M. Perkowski and M. Chrzanowska-Jeske, “An exact algorithm to minimize mixed-radix exclusive sums of products for incompletely specified Boolean functions,” in *Int’l Symp. Circ. and Systems*, 1990, pp. 1652–1655.

[32] T. Kozlowski, E. L. Dagless, and J. M. Saul, “An enhanced algorithm for the minimization of exclusive-or sum-of-products for incompletely specified functions,” in *Int’l Conf. on Comp. Design*, 1995, pp. 244–249.