Approximation and Decomposition of Binary Decision Diagrams*

Kavita Ravi
University of Colorado

Kenneth L. McMillan
Cadence Design Systems

Thomas R. Shiple
Synopsys

Fabio Somenzi
University of Colorado

Abstract

Efficient techniques for the manipulation of Binary Decision Diagrams (BDDs) are key to the success of formal verification tools. Recent advances in reachability analysis and model checking algorithms have emphasized the need for efficient algorithms for the approximation and decomposition of BDDs. In this paper we present a new algorithm for approximation and analyze its performance in comparison with existing techniques. We also introduce a new decomposition algorithm that produces balanced partitions. The effectiveness of our contributions is demonstrated by improved results in reachability analysis for some hard problem instances.

1 Introduction

Symbolic state enumeration techniques based on Binary Decision Diagrams (BDDs [2]) have revolutionized formal verification [8, 4, 17, 1, 14]. They have two key features that make them suitable to the exploration of very large state graphs: They represent sets compactly, and they avoid explicit enumeration in image computation. Given the transition relation of a system, $R(x, y)$, and a set of states, $F(x)$, the set of states reachable in one step from states in F, $T(y)$, is computed by

$$T(y) = \exists_x [R(x, y) \cdot F(x)].$$

This simple formula is at the heart of efficient algorithms for reachability analysis, language containment, and model checking. However, numerous improvements must be applied to the basic idea in order to make it work for realistic problems. The common aim of these improvements is to control the size of the BDDs created and manipulated during state exploration. This has been achieved by keeping the transition relation in partitioned form [28, 3, 10, 22]; by controlling the BDD variable order [12, 24]; by abstracting the system to be verified [16, 13, 15, 7]; or by abandoning pure breadth-first search in favor of more flexible approaches [23, 5, 21, 19].

Abstractions and methods that mix breadth-first and depth-first search rely, sometimes in crucial ways, on operations that approximate and decompose BDDs. In this paper we present new algorithms for these problems, and compare new and existing techniques. Our contributions help high-density traversal [23] achieve significant acceleration over the conventional breadth-first approach.

Minimization and decomposition of boolean functions have been the subject of much research in the last half century. However, early work is mostly inapplicable to decision diagrams; hence, the first relevant results are the *constrain* [8] and *restrict* [9] algorithms. We discuss *restrict* briefly now for two reasons: First, to provide the background required in Section 3, and second, to illustrate an approach to manipulation of BDDs that is common to the algorithms introduced in this paper.

*This work was supported in part by NSF grant MIP-94-22268 and SRC contract 96-DJ-560.

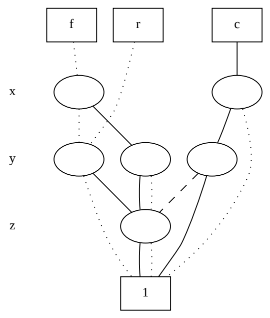

Figure 1: Remapping in *constrain* and *restrict*. Solid lines are *then* arcs. Dashed lines are regular *else* arcs, and dotted lines are complement arcs.

Throughout this paper we assume that a variable order is given—possibly derived dynamically. We use $|f|$ to denote the size of the BDD of function f under the given order; and $\|f\|$ to denote the number of minterms of function f. We only distinguish between a function and its BDD when it is necessary to avoid ambiguity.

The inputs to *restrict* are a BDD to be minimized, f, and the BDD of a *care set*, c. The output is a BDD, $f \Downarrow c$, that agrees with f wherever $c = 1$. The choice of $(f \Downarrow c)(x)$ when $c(x) = 0$ is heuristically made to reduce $|(f \Downarrow c)(x)|$. Consider Figure 1. One child of c is 0; hence the corresponding subgraph of $f = x f_t + x' f_e$ can be changed. If it is replaced by the subgraph of the other child—in the figure, f_e is replaced by f_t—not only the nodes that are only in the replaced child are eliminated, but the parent node becomes redundant as well. The process then recurs on f_t to yield r. (Which in this case is f_e'.) This *remapping* step, which *restrict* recursively applies while it traverses f and c, is its basic optimization technique; it is quite typical of BDD algorithms because it is a local transformation, whose purpose is to increase the sharing of nodes. We shall see that the remapping step plays a significant role in both our approximation and decomposition algorithms.

This paper is organized as follows: Section 2 discusses the problem of approximating BDDs and presents a new algorithm called *remapUnderApprox*. Section 3 is devoted to BDD decomposition. Section 4 presents experimental results obtained by applying the new approximation algorithm to reachability analysis, as well as a detailed comparison of various approximation techniques. Conclusions are in offered in Section 5.

2 Approximation

BDD approximation is the problem of deriving from a given BDD another BDD smaller in size, and whose function is at a low Hamming distance from the input BDD. (That is, differing from the input BDD in a small number of input assignments.) Let $\alpha(f)$ be the BDD produced by the application of approximation algorithm α to the BDD of f. Usually, the function of the approximating BDD is required to be either a subset or a superset of the input function, that is,

$$\alpha(f) \leq f \quad \text{or} \quad \alpha(f) \geq f.$$

For an *underapproximation* algorithm α (such that $\alpha(f) \leq f$), $\neg\alpha(\neg f) \geq f$. Hence, we only discuss underapproximation.

Underapproximation algorithms must trade off the size of the result for the distance from f. The two trivial solutions, 0 and f itself, are seldom useful. A natural way to rank different approximations is by their density, denoted by $\delta(\alpha(f))$ and defined in [23] by:

$$\delta(g) = \frac{\|g\|}{|g|}.$$

High density corresponds to a concise representation, and is therefore desirable. (For overapproximation, we want to maximize the density of the complement of the result.) However, in some circumstances, the number of nodes, or the number of minterms, and not just their ratio, are important.

Two algorithms for underapproximation were proposed in [23]; both are two-pass procedures, where the input is analyzed in the first pass and the result is produced in the second pass by replacing some arcs to internal nodes with arcs to the constant 0.

Heavy-branch subsetting (HB) determines how many minterms are in the function rooted at each internal node, and how many nodes would be eliminated by replacing arcs pointing to it. The second pass then proceeds from the root of the BDD and discards the "light branch" of each node, that is, the child with fewer minterms, until the size of the residual BDD crosses a given threshold. The result is a BDD with a string of nodes at the top, each with one child as the constant 0.

Short-path subsetting (SP) is based on the idea that short paths in a BDD correspond to large implicants of the function and use few nodes. Therefore in its first pass it determines the length of the shortest paths through each node, and then builds the result by discarding nodes with no short paths through them.

Both HB and SP run in time linear in $|f|$: This implies that only a limited study of the BDD is possible. Also, both are controlled by an upper bound on $|\alpha(f)|$, and though they both strive to increase density, they may occasionally decrease $\delta(\alpha(f))$ to reach the target $|\alpha(f)|$. Therefore, in analogy to the concept of *safe minimization* of [11], we introduce the following definition:

Definition 1 *An underapproximation algorithm α is* safe *if $\delta(f) \leq \delta(\alpha(f))$.*

It follows from Definition 1 that a safe overapproximation algorithm increases the density of the complement.

2.1 A Safe Underapproximation Algorithm

We now present a new algorithm for safe underapproximation whose main idea first appeared in [26, 25]. The algorithm is called *remapUnderApprox* (RUA) and differs from HB and SP in two main respects: 1) It computes for each node a lower bound on the increase in density that would follow from its replacement. 2) It can replace a node not just with a constant 0, but also with other subfunctions of f.

remapUnderApprox consists of three passes, as shown in Figure 2. The first pass, *analyze*, consists of a depth-first search of the BDD to be approximated. For each node, the number of minterms in the function rooted at the node, and the number of arcs pointing to it from other nodes in f (referred to as *functionRef* later) are computed. This information is collected in the data structure *info*, which is then updated in the second pass and used in the third pass to build the result; *info* also holds global information on the BDD

```
remapUnderApprox($f$, threshold, quality) {
    info = analyze($f$);
    info = markNodes($f$, threshold, quality, info);
    return(buildResult($f$, info));
}
```

Figure 2: Remapping underapproximation algorithm.

```
markNodes($f$, threshold, quality, info) {
    queue = emptyPriorityQueue();
    enqueue(queue, $f$, level($f$));
    while (queue not empty) {
        if (resultSize(info) $\leq$ threshold) return(info);
        node = dequeueFirst(queue);
        replacement = findReplacement(node,info);
        if (densityRatio(replacement, info) $>$ quality)
            info = updateInfo(info, node, replacement);
        enqueueChildren(queue, node, replacement);
    }
    return(info);
}
```

Figure 3: Node marking procedure.

and its approximation, and other fields for each node that are used by *markNodes* and *buildResult*. In particular, for each node it holds its replacement status, which initially is *do-not-replace*.

2.1.1 Determining Node Replacement

The second step of *remapUnderApprox* is detailed in Figure 3. The BDD of f is traversed in top-down fashion using a priority queue, so that all nodes labeled by variable x are examined before any node labeled by variable y is examined, if x precedes y in the order. When a node is retrieved from the queue, it is tested for replacement. We have implemented three types of replacement. The first type, *remap*, replaces a node by one of its children as in *constrain* [8]: It can be applied if the function rooted at one child of the current node is contained in the function rooted at the other child. (That is the function rooted at the current node is unate in its top variable.) The current node is replaced by the smaller child: If $f = xf_t + x'f_e$ and $f_e \leq f_t$, for instance, then f is replaced by f_e. The second type of replacement, *replace-by-grandchild*, applies when the two children of the current node are labeled by the same variable, and they share either the "then" child or the "else" child. Under these conditions, the shared grandchild can be used to replace the current node: If $f = x(yf_{tt} + y'f_{te}) + x'(yf_{et} + y'f_{ee})$, and $f_{tt} = f_{et}$, then $yf_{tt} \leq f$ and yf_{tt} can replace f. The last and simplest replacement type is *replace-by-0*: it can always be applied.

Procedure *findReplacement* tries the replacements in the above order and selects the first that applies. The chosen replacement is then definitely accepted or rejected based on the impact that it has on the density of the result. The impact is computed by counting exactly the number of minterms that will be lost to the replacement, and by computing a lower bound to the number of nodes that will be saved. *findReplacement* computes these two quantities as detailed in the next section. It then returns the result in "replacement," which also holds the replacement type.

2.1.2 Estimating the Change in Density

The number of minterms lost if a node is removed from a BDD is computed as the product of two quantities: The minterm count of the function rooted at the node (considered as a function of all the variables in f) and the fraction of paths from the root of f to either constant that go through the current node. Procedure *markN-*

```
nodesSaved($g$,info) {
        savings = 0;
        setLocalRef(info,$g$,functionRef(info,$g$));
        queue = emptyPriorityQueue();
        enqueue(queue, $g$, level($g$));
        while (queue not empty) {
                $v$ = dequeueFirst(queue);
                if (localRef(info,$v$) = functionRef(info,$v$)) {
                        savings++;
                        increaseLocalRefOfChildren(info,$v$);
                        enqueueChildren(queue, $v$);
                }
        }
        return(savings);
}
```

Figure 4: Node savings computation. Not shown is the initialization of *localRef* to 0 for all nodes.

odes computes this fraction during the top down traversal, so that for each node it may take into account the replacements for nodes higher in the BDD. The minterm count, on the other hand, is not affected by changes to the upper part of the BDD and therefore is computed during the first pass. The complexity of counting the minterms lost to replacements is linear in $|f|$ if this way of breaking down the computation is used.

The bound on the node savings, on the other hand, is simply computed by finding how many nodes are dominated[1] by the nodes that are eliminated. The eliminated nodes are the current node, for *replace-by-0*; the containing child of the current node for *remap*; either child of the current node for *replace-by-grandchild*. In addition, the current node is always eliminated. In all three cases the procedure of Figure 4 is used to find the dominated nodes: A top-down search of the BDD is started from each node g that is being eliminated. When a node v is retrieved from the queue all its predecessors reachable from g have been processed. Hence, *localRef* gives the number of predecessors of v reachable from g. On the other hand, *functionRef* gives the number of predecessors of v node reachable from f. If these two numbers are the same, then g dominates v. The total number of predecessors returned by *functionRef* is initialized by *analyze* and updated by *updateInfo*. The other important tasks of *updateInfo* are to record the type of replacement chosen, and to update the number of minterms and the estimate of the size of the result.

The nodes that are dominated by nodes that are going to be eliminated clearly contribute to the node savings of a replacement. However, further nodes may be saved due to sharing caused by the replacement in the part of the BDD above the current node. This is why we only have a lower bound on the reduction in size of the BDD.

Function *densityRatio* computes the ratio of the densities with and without the selected replacement and compares it to "quality." If parameter "quality" is greater than or equal to 1, then only replacements that increase the density are accepted. Values smaller or greater than 1 can be used when the result obtained for quality = 1 have too many or too few nodes, respectively.

Once all nodes have been marked with their replacement status, *buildResult* traverses f in depth-first manner and for each node returns the result of applying the selected replacement.

2.1.3 Discussion

The worst-case run time of *remapUnderApprox* is quadratic in $|f|$, due to the total cost of checking for each node the containment of each child in the other, and to the call to *nodesSaved*. In practice, however, the algorithm takes time linear in $|f|$ in most cases.

So far, we have ignored complement arcs for the sake of simplicity. The algorithm as outlined applies also in their presence, with a few modifications. First, the only replacement that can be applied to nodes reachable through paths of different complementation parity is *replace-by-0*. In addition, the estimate of the node savings is no longer guaranteed to be a lower bound if *replace-by-0* is applied to those nodes. Therefore, replacement is only applied to nodes reachable through paths of one complementation parity only. Minterm counts and fractions of paths are counted separately for the positive and negative phase, and the minterms lost to replacements are computed as a weighted sum.

The original *bddUnderApprox* algorithm of [26, 25] differs from *remapUnderApprox* in the following aspects:

- The cost function is a convex combination of the number of minterms and the number of nodes, instead of the ratio of the two numbers.
- Only *replace-by-0* is used. Because of this restriction, it is easy to replace nodes that are reachable through paths of different complementation parity (if complement arcs are used). The resulting algorithm is not safe, because replacement of one such node may cause splitting of a node higher in the BDD, but in most cases it gives denser results than the safe version.

2.2 Compound Algorithms

Given an approximation algorithm $\alpha(f)$, suppose a minimization algorithm $\mu(l,u)$ is also given such that $l \leq \mu(l,u) \leq u$. We say that $\mu(l,u)$ is *safe* [11] if $|\mu(l,u)| \leq |l|$ and $|\mu(l,u)| \leq |u|$. Then

$$\mu(\alpha(f), f)$$

is an approximation algorithm, which is safe if both α and μ are safe. Also, if α_1 and α_2 are approximation algorithms, then

$$\alpha_1(\alpha_2(f))$$

is an approximation algorithm, which is safe if both α_1 and α_2 are safe. These simple rules can be used to trade off CPU time for the increased density of the results. We call the resulting algorithms *compound* approximation algorithms. As an example one can repeatedly apply RUA starting with a quality factor greater than 1 and decreasing it at each iteration until it equals 1. This has the effect of mitigating the greediness of RUA. Other compound algorithms are discussed in Section 4. Methods that are not compound are *simple*.

3 Decomposition

Decomposition is another important approach to reducing the size of large BDDs. Decomposition of BDDs is closely related to finding efficient partitioned representations of a given boolean function. Partitioned representations [20, 3] may be derived in the process of building a BDD or by decomposing a given BDD. The former is easier to obtain when some structural information, such as the network, is provided. Auxiliary variables are introduced while constructing the BDD. They alleviate ordering constraints and reduce the size of the partitions. Alternately, a static analysis of the network may yield good decomposition points.

We concentrate on the problem of functional decomposition—decomposing a large BDD into a set of smaller ones, which can be combined to form the original BDD. Given a BDD, a variable order, and no prior knowledge of the nature of the function,

[1] A node v dominates a node w if all paths from the root of the BDD to w go through v.

decomposition techniques require an analysis of the given BDD. In the simplest case, these may be conjunctive or disjunctive decompositions. The goal is to reduce the shared size (to occupy less storage space) as well as the individual sizes (for easier manipulation) of the decomposed set as compared to the original BDD size.

Prior Work. To form conjunctive factors for a function f such that

$$f = x \cdot f_x + x' \cdot f_{x'}$$

we can create two factors g and h where

$$f = g \cdot h \; ; \; \; g = x + f_{x'} \; ; \; \; h = x' + f_x \tag{1}$$

Each of the 2 factors has a different cofactor set to 1. Disjunctive partitioning, in this approach, is completely symmetric to the conjunctive method and is obtained by setting the different cofactors to 0.

Cabodi *et al.* [6] and Narayan *et al.* [19] propose decompositions based on Equation 1. They pick a suitable set of variables and both produce disjunctive factors by computing cofactors of the function f with respect to all boolean assignments of the chosen variables. They use different cost functions for the choice of variables. The cost functions heuristically reduce the size of the decomposed BDDs. Cabodi *et al.* use a threshold size to derive the decomposed BDDs and Narayan *et al.* use a pre-specified number of partitions. These methods have been tested in reachability analysis where a balanced partition is important for efficient traversal. Cabodi *et al.* also observed that obtaining a disjunctive partition using dense subsets [23] does not yield balanced partitions.

McMillan [18] presents a different approach to decomposition. His method computes a canonical conjunctive decomposition based on exploiting conditional independence between variables. The algorithm performs successive existential abstraction and cofactoring to reduce the size of the decomposed representation. The number of factors produced are equal to the number of variables. The size of the decomposed representation is linear in the number of partitions and the size of the original BDD.

Our Approach. Our decomposition method is closely related to the first approach. In that method, variable x forms a cut in the BDD of f. Creating the factors amounts to setting the THEN children of nodes labeled x to 1 in g and the ELSE children to 1 in h. A more general strategy would be to set the opposite children of each node labeled x to 1 (instead of setting *all* positive or *all* negative cofactors to 1). A further generalization would be to relax the constraint of the choice of nodes—from those labeled x to an arbitrary set of nodes in f. We refer to these nodes as *decomposition points*.

The algorithm is illustrated in Figure 5. Given a set of *decomposition points*, our algorithm constructs the factors bottom up. At the decomposition points, factors are created using Equation 1. At every node above the decomposition point, factors are constructed by combining those of the children as shown below.

$$g = x \cdot g_T + x' \cdot g_E \; ; \; \; h = x \cdot h_T + x' \cdot h_E \qquad \text{or}$$
$$g = x \cdot g_T + x' \cdot h_E \; ; \; \; h = x \cdot h_T + x' \cdot g_E$$

The algorithm maintains a cache for previously computed factors for each node. It also attempts to encourage node sharing by storing nodes created for each factor.

Decomposition Points. The choice of *decomposition points* is crucial to this method. We have experimented with two greedy approaches to pick these *decomposition points*.

Band: The first method is to pick nodes that are low enough in the BDD to achieve substantial reduction in the individual factor sizes. However, if the nodes are picked too low in the BDD, most of the recombination may be destroyed in building the factors. The required "middle band" is determined by using the distance of the node from the constant as a measure. This measure requires one pass of the BDD.

```
decomp ($f$) {
    if ($f$ is 1 or 0) return ($f$, 1);
    if cacheLookup($f$, $g$, $h$) return ($g$, $h$) ;
    $v$ = topVar($f$);
    if ($f$ is decomposition point) return ($v + f_e$ , $v' + f_t$);
    ($g_t$,$h_t$) = decomp ($f_t$);
    ($g_e$,$h_e$) = decomp ($f_e$);
    ($g$, $h$) = combine($g_t$,$h_t$, $g_e$,$h_e$);
    return ($g$, $h$);
}
```

Figure 5: Decomposition algorithm.

Disjoint: Nodes with sufficiently disjoint children yield maximum reduction in individual sizes, yet leave the shared size small. Our second approach attempts to identify such nodes. For every node, the number of nodes shared between its children is measured. A node is chosen as a *decomposition point* when its children have little sharing and are balanced. Determining whether each node is a *decomposition point* requires one pass of the BDD per node. Hence the cost of this measure is quadratic in the number nodes of the BDD. However, in practice, only a fraction of the nodes are sampled for this measure.

We implemented a slightly different version of the algorithms of [6, 19] (referred to as *Cofactor*). Experiments and results of the comparison with our approach are presented in the next section.

4 Experimental Results

The algorithms described in Sections 2 and 3 have been implemented in the CUDD package [27] and tested in two sets of experiments, one using an extension of VIS-1.2 [1], and the other using CUDD's stand-alone test program. The first set of experiments regards the application of BDD approximation techniques to reachability analysis. The RUA algorithm described in Section 2 is applied to "high density" reachability analysis [23] in the same manner as the SP procedure. We experimented with a few circuits and present the results in Table 1.

Table 1 presents best-time results for high-density reachability analysis of four circuits. High density reachability analysis modifies breadth-first search to compute a subset of reachable states at each iteration of the traversal. It can be viewed as a mixed depth-first breadth-first exploration of the state space. At each iteration, image computation is provided with a dense subset that is extracted from the set of new states. This extraction is implemented with one of the approximation techniques. Additionally, the traversal computes a subset of the image of a given set by computing subsets of intermediate products of image computation. Subsetting is used when the size of the intermediate product exceeds some large threshold. Dynamic reordering is always turned on during these experiments. For all the experiments reported (except BFS on `am2910`), both BFS traversal and high density traversal completed with the exact set of reachable states.

Columns 1–3 of Table 1 provides the circuit name, the number of flip-flops and the reachable states of this circuit. The exact traversal times are reported in Column 4. `s1269`, although small in terms of the number of latches (37), took more than 256MB for exact traversal and hence, was run on a SUN Ultra-1 with 1GB of main memory. The main column labeled RUA indicates results where *remapUnderApprox* is used for extracting the subset of the new states as well as the subset of the intermediate products of image computation. Similarly, the column labeled SP refers to *short-path* *subsetting*. The sub-columns report run times for each method, the threshold used and (for RUA) the quality factor. The column labeled PImg reports two numbers: The first is a node limit that triggers approximation of partial products in image computation; the second is the threshold passed to the approximation procedure. The "NA"s in the table indicate that partial image computation was not required as the intermediate sizes always stayed small.

These experiments are the best runs for some parameter settings (threshold, quality, partial image factors). The relative merit of tuning the quality factor or adjusting the threshold cannot be studied in this table. The parameters were determined by trial and error. The speedup afforded by HD with RUA over breadth-first search is very large in two cases out of four. In our experiments, we noticed that when RUA had the fastest run times, the runs always corresponded to small BDD sizes of reachable states (under 50000 nodes). The intermediate sizes were significantly smaller compared to the original runs: RUA is very effective in reducing the sizes of large intermediate products of image computation. The fast run times and low memory occupation also support the conjecture that RUA extracts dense subsets, which is required for high-density reachability analysis to perform well.

In comparison to SP, there is no clear indication as to which method is better. With a 0 threshold, RUA is free to reduce the size of the given set arbitrarily. This works well for `s1269`, `s3330`, and `am2910`. In the case of `s5378opt`, SP is more effective for the given threshold. The differences among the various circuits seem due to the topology of their state graphs in ways that we only partly understand so far. However, we have observed that the method that performs better always has a higher average density of subset of new states. We conducted some experiments using SP for creating subsets of new states but RUA for partial image computation, and the run-times were faster than using SP for both. Using approximation procedures in reachability analysis has compounded effects on the statistics of the runs (reordering, growth of the reached set in number of states and nodes). If a procedure extracts the "wrong" subset of states for image computation, then traversal may progress very slowly or may not complete. Indeed, this is true for some parameter settings. Notice that "wrong" here does not necessarily mean "less dense." However, the fast run times in this table indicate that RUA is effective in spite of the cascaded effects and is practically applicable in efficient symbolic traversal techniques.

It is difficult to conduct a head-to-head comparison of different approximation techniques in the context of reachability analysis, due to the repercussions of any change in one step of the procedure on the successive steps. Therefore we present a second set of experiments, where we apply the approximation techniques to the outputs and next state functions of a collection of circuits. From a total of 7157 functions, we extracted the 336 that had 5000 nodes or more. In Tables 2 and 3 we report the geometric means of the numbers of nodes, the number of minterms, and the densities of several methods, as well as the number of cases in which each technique produced the densest result, either alone (wins) or together with other techniques (ties). Table 2 describes the simple methods and Table 3 describes compound methods. The results of UA are for the non-safe implementation, which on average outperformed the safe one, in spite of decreasing density in 3 of the 336 cases. Simple and compound methods are kept separate so that the relative strengths of SP and RUA can be better appreciated, because C1 never loses to RUA, and C2 never loses to SP.

In these experiments, the thresholds for UA and RUA were set to 0 and the quality factor was kept at 1. In general these were the most favorable values for these experiments. The sizes of the BDDs produced by RUA were used as thresholds for HB and SP. Though there is no guarantee that these thresholds produce the best results, experimentation showed that they did not put SP and HB at a disadvantage with respect to UA and RUA.

Method	nodes	minterms	density	wins	ties
F	14449.4	1.06e+45	7.30e+40	0	2
HB	24.5	3.30e+42	1.35e+41	3	65
SP	41.9	2.48e+44	5.92e+42	6	7
UA	28.3	3.72e+44	1.32e+43	24	78
RUA	30.4	6.04e+44	1.99e+43	219	80

Table 2: Comparison of approximation methods I: Simple methods. F denotes the original function, HB heavy-branch subsetting, SP short-path subsetting, UA bddUnderApprox, and RUA remapUnderApprox.

Method	nodes	minterms	density	wins	ties
C1	30.3	6.14e+44	2.03e+43	125	87
C2	14.7	2.59e+44	1.76e+43	124	85

Table 3: Comparison of approximation methods II: Compound methods. C1 denotes RUA followed by minimization, and C2 denotes SP followed by RUA followed by minimization.

Regarding the compound methods, one should notice that C1 retained more minterms of F than RUA on average, and C2 retained more minterms than SP (in spite of halving the number of nodes). This increase is the effect of the minimization step.

We also ran experiments comparing the decomposition methods described in Section 3 for two-way decomposition. We implemented the approach of [6, 19] choosing as cofactoring variable the one that minimizes the size of the larger of the two cofactors. The cost of estimation of the cofactor sizes is linear in the product of the number of variables and the size of the function.

Table 4 shows results for the 3 methods—the above implementation, and our algorithm with the two different choices of decomposition points. The experimental setup is similar to Tables 2 and 3. There are 2 sets of results in this table—one for BDDs with at least 5000 nodes and the other for BDDs with at least 20000 nodes. Each method produced 2 factors, G and H, whose mean sizes are reported in the table. The column labeled "Shared" reports the mean shared size of G and H. The wins and ties of each method, reported in the last 2 columns, are based on the size of the larger of G and H.

Cofactor yields the best results for a large number of BDDs. *Band* performs a little better than *Disjoint* (additionally is cheaper to compute). One explanation for the results is that *Disjoint* and *Band*, although less restricted than *Cofactor* in the choice of *decomposition points*, perform a local search and make greedy choices. *Cofactor*, on the other hand, benefits from a global search for the smallest factors. For some of the larger BDDs, *Disjoint* does perform better than *Cofactor*, as shown in the lower half of the table. However, since the sample space is small, this result is not conclusive.

Min. Nodes = 5000 ,$\|f\|$ = 11134.5, 279 BDDs					
Method	Shared	G	H	wins	ties
Cofactor	11352.1	7029.7	4906.3	192	4
Disjoint	11990.9	8637.5	8182.8	57	4
Band	13590.4	8244.0	7182.7	26	0
Min. Nodes = 20000, $\|f\|$ = 42872.9, 11 BDDs					
Cofactor	47952.0	26394.3	25813.0	2	1
Disjoint	42877.0	22575.4	22866.5	8	1
Band	52614.4	30728.8	26055.5	0	0

Table 4: Comparison of decomposition methods.

Ckt	FF	States	BFS time	RUA				SP		
				Th	Qual	PImg	Time	Th	PImg	Time
s3330	132	$7.2778e+17$	**3204**	0	1.0	50000/50000	**562**	7000	100000/100000	**1351**
s1269	37	$1.13134e+09$	**52691**	0	0.5	100000/50000	**290**	3000	100000/50000	**525**
s5378opt	121	$2.58061e+17$	**1454**	5000	1.4	NA	**1140**	4000	NA	**575**
am2910	99	$1.16057e+26$	>2 **weeks**	0	1.0	NA	**217**	99	NA	**224**

Table 1: Reachability analysis results using BDD approximations.

5 Conclusions

We have presented new algorithms for BDD approximation and decomposition. Our new approximation algorithm produces significantly denser subsets than existing techniques thanks to a combination of versatile replacement techniques and efficient and accurate estimation of the impact of local transformations. We have also shown how approximation algorithms can be combined among themselves and with minimization procedures to yield even better results. Our new conjunctive decomposition algorithm efficiently balances the size of the conjuncts. In spite of being more general, however, it does not improve on the algorithm of [6, 20]: Further work is required on the selection of the decomposition points. Both the approximation and decomposition techniques have been incorporated in an experimental reachability analysis engine based on high-density traversal. The results are very promising, with substantial increases in speed over conventional breadth-first traversal. More work is required to increase the robustness and performance of the new traversal techniques, but the effectiveness of the algorithms discussed in this paper greatly increases our confidence in their eventual success.

Acknowledgments

We thank Dirk Grunwald for providing us with computing resources.

References

[1] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eigth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 403–407, San Francisco, CA, June 1991.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[5] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive partitionining and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In *Proceedings of the Design Automation Conference*, pages 728–733, Anaheim, CA, June 1997.

[6] G. Cabodi, P. Camurati, and S. Quer. Improved reachability analysis of large finite state machines. In *Proceedings of the International Conference on Computer-Aided Design*, pages 354–360, Santa Clara, CA, November 1996.

[7] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. *IEEE Transactions on Computer-Aided Design*, 15(12):1465–1478, December 1996.

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.

[9] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 126–129, November 1990.

[10] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation parititons. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 299–310, Berlin, 1994. Springer-Verlag. LNCS 818.

[11] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Safe BDD minimization using don't cares. In *Proceedings of the Design Automation Conference*, pages 208–213, Anaheim, CA, June 1997.

[12] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable ordering and selection for FSM traversal. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 476–479, Santa Clara, CA, November 1991.

[13] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[14] R. P. Kurshan. Formal verification in a commercial setting. In *Proceedings of the Design Automation Conference*, pages 258–262, Anaheim, CA, June 1997.

[15] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 76–81, San Jose, CA, November 1996.

[16] D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie-Mellon University, July 1993.

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[18] K. L. McMillan. A conjunctively decomposed boolean representation for symbolic model checking. In R. Alur and T. A. Henzinger, editors, *8th Conference on Computer Aided Verification (CAV'96)*, pages 13–25. Springer-Verlag, Berlin, August 1996. LNCS 1102.

[19] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned ROBDDs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 388–393, November 1997.

[20] A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partition ROBDDs: A compact, canonical and efficiently manipulable representation for boolean functions. In *Proceedings of the International Conference on Computer-Aided Design*, pages 547–554, Santa Clara, CA, November 1996.

[21] A. Pardo and G. D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*. Springer-Verlag, Berlin, 1997. LNCS 1254.

[22] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, Lake Tahoe, CA., May 1995.

[23] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–158, San Jose, CA, November 1995.

[24] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, Santa Clara, CA, November 1993.

[25] T. R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, October 1996. Memorandum No. UCB/ERL M96/76.

[26] T. R. Shiple, R. K. Ranjan, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Deciding state reachability for large FSMs. Technical Report UCB/ERL M97/73, University of California at Berkeley, August 1996.

[27] F. Somenzi. *CUDD: CU Decision Diagram Package*. ftp://vlsi.colorado.edu/pub/.

[28] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDD's. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 130–133, November 1990.