Fast and Efficient Equivalence Checking based on NAND-BDDs

Rolf Drechsler

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg, Germany
drechsle@informatik.uni-freiburg.de

Mitch Thornton

Elec. and Computer Engineering
Mississippi State University
Mississippi State, Mississippi
mitch@ece.msstate.edu

Abstract

Ordered Binary Decision Diagrams (BDDs) are a data structure commonly used for the representation and manipulation of Boolean functions as used in VLSI CAD applications. BDDs are used in many equivalence checking tools due to their canonicity. Typically, BDD packages are based on ITE synthesis operations. By restricting the ITE-based packages to a single operation (in this example, the Boolean NAND) simplification of the implementation of the software often results in a speedup of the BDD construction process. Experiments show that significant improvements in terms of runtime can be achieved. In some cases more than a 95% increase in runtime improvement is noted.

1 Introduction

Decision Diagrams (DDs) are often used in VLSI CAD systems for efficient representation and manipulation of Boolean functions. The most popular data structures are *ordered Binary Decision Diagrams* (BDDs) [2, 3]. They have been used in various applications due to their canonical representation and ease of manipulation. Particularly in the case of formal verification, BDDs have been integrated in almost all tools used to date [4, 9]. These BDD packages are based on recursive operations that make use of a three operand function commonly known as ITE [1]. For details on the efficient implementation of BDD packages see [1, 8, 11].

Most BDD packages allow for many types of synthesis operations such as AND and OR; however, variable substitution and quantification operations which are used extensively in sequential equivalence checking are also included. On the contrary, for *Combinational Equivalence Checking* (CEC), it is only necessary to determine whether two given circuit implementations realize the same Boolean function. Many of the operations included in standard packages are not used for CEC applications. The check for combinational function equivalence needs to be performed very fast since, if CEC is applied to large designs, it is often the case that several million comparisons are carried out. In such comparisons no dynamic variable ordering (as proposed in [10]) is invoked since this often can become very time consuming and slows down the overall CEC verification process. For an overview of the general verification flow of an equivalence checker see [7].

A BDD package is presented that is tuned for fast and efficient CEC. Instead of using the three-operand ITE operation, the basic synthesis algorithm is the two-operand Boolean NAND. This simplifies the implementation of the software and also has advantages regarding the hit rate of the computed table. The package does not free nodes once they are allocated and also does not support dynamic variable reordering. The implementation is kept as simple as possible to allow a very fast operation. Experiments show that significant improvement over ITE based packages can be observed. Up to a 95% reduction in runtime, while the memory consumption as measured in the number of nodes allocated only increases slightly.

This paper is structured as follows: In Section 2 BDDs are defined and the ITE operator is briefly reviewed. NAND-BDDs are introduced in Section 3 and differences with respect to the ITE operator are discussed. Experiments are presented in Section 4. Finally, the results are summarized.

2 Preliminaries

A brief definition of BDDs and a review of the ITE operation is presented to provide context for the presentation of the NAND based implementation as described here.

```
ite(F,G,H) {
        if (terminal case) return result;
        if (computed-table entry (F,G,H) exists) return result;

        let x_i be the top variable of {F,G,H};

        THEN  = ite(F_{x_i}, G_{x_i}, H_{x_i}) ;
        ELSE  = ite(F_{x̄_i}, G_{x̄_i}, H_{x̄_i}) ;

        if (THEN == ELSE) return THEN;

        // Find or create a new node with variable v and sons THEN and ELSE
        R = Find_or_add_unique_table(x_i,THEN,ELSE);

        // Store computation and result in computed table
        Insert_computed_table({F,G,H},R);

        return R;
}
```

Figure 1: ITE-algorithm

2.1 Binary Decision Diagrams

As is well-known a Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) which is a directed acyclic graph where a Shannon decomposition

$$f = \overline{x}_i f_{x_i=0} + x_i f_{x_i=1} \quad (1 \leq i \leq n)$$

is carried out in each node.

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal node and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it does not contain isomorphic subgraphs nor a vertex with both exiting edges pointing to the same node. Reduced and ordered BDDs are a canonical representation since for each Boolean function the BDD is uniquely specified.

For functions represented by reduced and ordered BDDs efficient manipulations are possible [2]. In the following, only reduced and ordered BDDs are considered and for brevity these graphs are called BDDs.

2.2 If-Then-Else Operation

The typical synthesis operation employed in most BDD software packages is given a brief overview here. The synthesis of two BDDs is carried out by performing a recursive call on subgraphs. A sketch of the recursive *If-Then-Else* (ITE) algorithm from [1] is given in Figure 1.

3 NAND-BDDs

In the approach considered here the synthesis algorithm is restricted to one operation only, the Boolean NAND. The resulting algorithm is shown in Figure 2. As can be seen, the overall flow is exactly the same as for the ITE algorithm, however, only two instead of three operands are required. This improves the hit rate of the computed table and also reduces its' size.

It is noted that a restriction to other operators such as the Boolean NOR can also be used analogously. In addition, the use of a few operators is possible using different computed tables for each of them. For simplicity, the prototype software described here is restricted to use only the NAND function.

It is well known that the NAND operation $(\overline{f \cdot g})$ is sufficient to realize all possible Boolean functions of 2 variables. For completeness the list of all possible operations is given in Table 1. For readability of the table the negation operation is still allowed since it can easily be mapped to

$$\overline{a} = NAND(a, a) = NAND(a, 1).$$

The current implementation described here does not use complemented edges (see [1, 8]). This can be

```
NAND(F,G) {
    if (terminal case) return result;
    if (computed-table entry (F,G) exists) return result;

    let x_i be the top variable of {F,G};

    THEN  = NAND(F_{x_i}, G_{x_i}) ;
    ELSE  = NAND(F_{x̄_i}, G_{x̄_i}) ;

    if (THEN == ELSE) return THEN;

    // Find or create a new node with variable v and sons THEN and ELSE
    R = Find_or_add_unique_table(x_i,THEN,ELSE);

    // Store computation and result in computed table
    Insert_computed_table({F,G},R);

    return R;
}
```

Figure 2: NAND-algorithm

integrated directly and would likely lead to a further reduction of runtime and memory requirements.

The realization of the package described here is based on the principle that all operations that are not relevant for the computation of the BDD are avoided. Due to this approach, the technique described here is not a "full" BDD package since other features are missing. In particular, it is noted that the following features are not included.

Dynamic Variable Ordering (DVO): Variable re-ordering is a very effective technique for reducing the number of nodes in BDDs. In terms of fast CEC applications, DVO can become very time consuming. By avoiding this feature the implementation is simplified significantly and additional runtime is not expended in an attempt to reduce the size of a BDD.

Memory management: Nodes are only allocated and are never freed during a comparison run. Therefore, no garbage collection is carried out. The advantage is that the package is very memory efficient since no *reference count* [1] needs to be stored. The number of nodes that are created is a fixed quantity which, if exceeded, simply causes the package to return with a "insufficient memory" error.

The lack of inclusion of these features actually has the advantage that the package only performs operations that are relevant for constructing a BDD as fast as possible and within specified memory limits. In practice it is better to get a fast result so that the BDD approach to CEC does not give a solution when the maximum allowable node count is exceeded rather than wasting an excessive amount of runtime that could be better used by other CEC techniques based on principles such as SAT-solvers or term rewriting.

3.1 ITE vs. NAND

In this section the main differences between NAND-BDDs and standard implementations based on ITE operations are compared.

- NAND-BDDs are easier to implement. This results from the simplicity of the synthesis operation and the fact that DVO and memory management is not supported. This also results in simplification of debugging the code.
- Usually more nodes are allocated since the mapping to NAND only results in more synthesis operation calls; however, memory is also saved by avoiding the reference count function and the reduction of the computed table size due to two instead of three operands.

Table 1: Realization of operators by NAND

function	name	expression	NAND-call
0000	0	0	0
0001	AND	$f \cdot g$	$NAND(NAND(f,g),1)$
0010	$f > g$	$f \cdot \overline{g}$	$NAND(NAND(f,\overline{g}),1)$
0011	f	f	f
0100	$f < g$	$\overline{f} \cdot g$	$NAND(NAND(\overline{f},g),1)$
0101	g	g	g
0110	XOR	$f \oplus g$	$NAND(NAND(a,\overline{b}),NAND(\overline{a},b))$
0111	OR	$f + g$	$NAND(\overline{f},\overline{g})$
1000	NOR	$\overline{f+g}$	$NAND(NAND(\overline{f},\overline{g}),1)$
1001	$XNOR$	$\overline{f \oplus g}$	$NAND(NAND(NAND(a,\overline{b}),NAND(\overline{a},b)),1)$
1010	NOT	$\overline{g}$	$\overline{g}$
1011	$f \geq g$	$f + \overline{g}$	$NAND(\overline{f},g)$
1100	NOT	$\overline{f}$	$\overline{f}$
1101	$f \leq g$	$\overline{f} + g$	$NAND(f,\overline{g})$
1110	$NAND$	$\overline{f \cdot g}$	$NAND(f,g)$
1111	1	1	1

- Only two operands are used in the synthesis operation. This improves the hit-rate in the computed table. Since the computed table is the key to fast BDD algorithms [6], this has a direct impact on the overall performance of the BDD software.
- NAND-BDDs are useful for fast CEC and in this sense they are not a "full" BDD package since operations that are important for other applications such as quantification, DVO and garbage collection are not realized as efficiently as in other packages.

4 Experimental Results

In this section experimental results are given that show the behavior of NAND-BDDs as compared to an ITE realization using well known benchmark examples. The experimental results were carried out using a *SUN Ultra 1* with 256 MBytes. All times are given in units of CPU seconds.

The prototype of the software has been written in $C++$. In order to provide a fair comparison, the ITE and NAND-BDD packages are implemented in the same environment in that both packages do not use a memory manager and both are implemented without the use of complemented edges. Both packages only make use of the simple terminal case and do not consider techniques like case normalization or *ITE_constant* as described in [1].

Benchmarks from ISCAS85 and the combinational part of ISCAS89 are used in the experimental results as presented in Table 2. For both packages the same static variable ordering using a method similar to that described in [5] is used and a hard upper node limit of 250.000 is used. The only benchmarks reported here are those for which a result was obtained within this node limit and within 1 CPU hour. Furthermore, we focus on "non-trivial" examples which take longer than 1 CPU second. The benchmark function *c0880* has been included since it was one of the few where ITE showed better performance regarding runtime. For all other larger examples, NAND-BDDs outperformed ITE.

The results are given in Table 2. The *name* of the benchmark is given in the first column. In columns ITE and NAND, *nodes* and *time* denotes the number of nodes allocated during the BDD construction and the time needed, respectively. The last two columns of the table give the relative improvement for the number of nodes and the runtime needed. As can be seen, the number of nodes is never more than 50% larger for NAND-BDDs as compared to the ITE based implementation while the runtime in some cases can be reduced by more than 99% (see *c1908*). During CEC the runtime is usually more critical and the maximum amount of memory that is used can easily be controlled by the hard limit on the number of nodes set by the user.

Table 2: ITE vs. NAND for BDD construction

name	ITE		NAND		improvement	
	nodes	time	nodes	time	nodes	time
cs01423	87987	2.34	117833	1.04	1.3392	0.4444
cs05378	47583	15.20	52375	12.47	1.1007	0.8269
c0432	28656	190.81	35591	5.48	1.2420	0.0287
c1908	162035	264.24	185989	2.43	1.1478	0.0091
c5315	116416	10.36	186178	1.80	1.5992	0.1737
c0880	50843	0.35	60326	0.52	1.1865	1.4857
c0499	136821	264.14	146193	84.67	1.0684	0.3205

5 Conclusions

A technique for the implementation of a BDD package that finds application in fast combinational logic equivalence checking is presented. The prototype package discussed here only makes use of one synthesis operation; the NAND instead of the ITE based one. Due to the simplicity of the implementation discussed here, the experimental outcomes have resulted in significant computer runtime speedup in terms of BDD construction time.

It is focus of future work to include the use of complemented edges, since this would allow for further simplification of the synthesis calls and it can also be expected that the number of nodes will decrease more than the ITE based packages, since negation is used intensively during the construction process.

Acknowledgments

This work was motivated by the talks of Geert Janssen and Andreas Kühlmann at the Dagstuhl Seminar *Computer Aided Design and Test - BDDs versus SAT* in February 2001.

References

[1] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

[2] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[3] R.E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM, Comp. Surveys*, 24:293–318, 1992.

[4] J.R. Burch and V. Singhal. Tight integration of combinational verification methods. In *Int'l Conf. on CAD*, pages 570–576, 1998.

[5] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 38–41, 1993.

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

[7] A. Kuehlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Design Automation Conf.*, 2001.

[8] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. In *Design Automation Conf.*, pages 52–57, 1990.

[9] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Int'l Conf. on Comp. Design*, pages 459–464, 2000.

[10] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

[11] F. Somenzi. Efficient manipulation of decision diagrams. *Software Tools for Technology Transfer*, 2001.