Complete BDDs for Fast and Efficient Equivalence Checking*

Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

Abstract

Modern proof engines for formal verification, like used in equivalence checking and bounded model checking, are based on multi-engine concepts. This guarantees robustness and high flexibility of the tools. One important technique are Ordered Binary Decision Diagrams (BDDs). Typically, BDDs are used in a reduced form.

In this paper we study complete BDDs, i.e. BDDs that are not fully reduced. Even though complete BDDs require more space counted in the number of nodes, they have significant advantages for applications where a fast decision is needed whether the BDD can be build or not, since the synthesis operations can be simplified. Experiments run on a prototype implementation show that very good improvements in terms of run time can be achieved, i.e. in some cases more than 90% of the CPU time can be saved.

1 Introduction

With larger and larger designs, the verification task becomes the bottleneck in many flows. Simulation alone is not sufficient to guarantee sufficient coverage. Furthermore, too much manual interaction is required for test vector generation and evaluation and analysis of the results. As a very promising alternative formal verification techniques have been proposed. In the meantime they have been successfully applied in equivalence checking and (bounded) model checking (see e.g. [2, 11, 14, 9]). The core algorithms for these methods are so-called proof engines, that make e.g. use of SAT, BDDs, random pattern simulation, and ATPG. Non of the techniques alone is powerful enough to solve all problems. Only by a tight integration of the different concepts robust and flexible engines can be build (see e.g. [5, 14]).

One of the most successful prover techniques are *ordered Binary Decision Diagrams* (BDDs) [4]. BDD packages are typically based on recursive operations that make use of a three operand function commonly known as ITE [3]. For details on the efficient implementation of BDD packages see [3, 12, 16]. Usually, BDD packages are "general" packages that allow all type of Boolean function manipulation and minimization concepts, like dynamic variable ordering. For the application descried above, this is not needed. Typically, variable reordering [15] is always switched off, since it is too time consuming. In case the BDD cannot find a solution fast, it is likely that the problem is difficult for BDDs, and it is better to start another prover engine. For this, we now focus on optimization of the techniques needed for fast equivalence checking. 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. For an overview of the general verification flow of an equivalence checker see [9]. Recently, a restricted BDD package has been proposed in [6], where all operators were substituted by NAND. The resulting BDDs are called NAND-BDDs. Of course, still all binary operations can be used, since NAND is a complete basis Significant reductions of runtime compared to ITE could be observed.

In this paper we present a technique to further improve the CPU time for BDD construction on top of the results of NAND-BDDs. Instead of fully reducing the BDD as originally proposed in [4], we consider complete BDDs (also called pseudo-reduced), i.e. BDDs that share isomorphic sub-graphs, but nodes with both edges pointing to the same node are not removed. Complete BDDs need more nodes on average, but the memory needed per node is smaller, since no index has to be stored. Furthermore, the synthesis operation is sped up, due to cases that have to be checked in reduced BDDs, but become superfluous for complete BDDs. The caching behavior is also improved. Experimental results run on a prototype implementation show that reduction in runtime of more than 90% can be observed.

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

2 Preliminaries

A brief definition of (NAND-)BDDs and a review of the synthesis operation is presented to make the paper self-contained [6].

*This work was supported in part by DFG grant DR 287/8-1.

```
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

```
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

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 does it have redundant nodes. Reduced and ordered BDDs are a canonical representation since for each Boolean function the BDD is uniquely specified.

A BDD is called *complete*, if isomorphic sub-graphs are shared, but each node appears along each path. The complete BDDs considered in the following make use of an ordering restriction.

For functions represented by reduced and ordered BDDs efficient manipulations are possible [4]. These algorithms with slight modifications can be transfered to complete BDDs (see below) In the following, we refer to reduced and ordered BDDs for brevity as BDDs and explicitly mention the completeness property.

To consider complete BDDs is also justified from a theoretical point of view, since the difference to fully reduced BDDs is bound by a linear factor [1]. Furthermore, the main reduction from an asymptotical point of view results from sharing of isomorphic sub-graphs, as it is also carried out in complete BDDs (see [10]). For an in-depth discussion on the size of complete decision diagrams see [13].

2.2 If-Then-Else Operation

A brief description of the typical synthesis operation employed in most BDD software packages is given here. The synthesis of a BDD F depending on some Boolean relation between two existing BDDs G and H is carried out by performing a recursive call on subgraphs. A sketch of the recursive *If-Then-Else* (ITE) algorithm from [3] is given in Figure 1.

The ite function can be considered to be a functionally complete three-input logic gate that implements the expression, $ite = F \cdot G + \overline{F} \cdot H$. Using this relation, the BDD $APPLY$ operation can be implemented with any arbitrary Boolean operation as an argument. The computed table stores previously computed results and the three arguments are pointer values to F, G and H. Therefore, if the synthesis operation has been previously computed, further recursions are unnecessary as the computed result in the cache is simply passed back. The addition of the cache structure to BDD package implementations is well known to significantly reduce runtime in the synthesis of a BDD (see e.g. [8]).

2.3 NAND-BDDs

In the approach considered in [6] 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.

For a detailed discussion of the properties of NAND-based synthesis in comparison to ITE see [6]. The underlying BDDs are reduced in both cases, i.e. only the synthesis algorithm is modified, but not the representation itself. Notice that NAND is sufficient to carry out all Boolean operations, like AND and OR, since NAND is a complete basis.

3 Complete BDDs

In this section the use of complete BDDs in fast equivalence checking is discussed. The implementation is done on top of NAND-BDDs, i.e. all the (positive) properties can be directly transferred.

An implementation based on complete BDDs differs from reduced BDDs in several ways:

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

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

  // 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 3. Synthesis algorithm for complete BDDs

Additional nodes for path completion: All variables appear along each path from the root to the terminal nodes of the BDD. Starting from a fully reduced BDD, the nodes can simply be introduced. The overhead is moderate, i.e. only linear in the number of variables (as has been proven in [1]).

No check for top variable: The synthesis algorithm does not have to check for the top variable, since due to the completeness property the top variables are the same in both functions to be synthesized.

No reduction of nodes with the same successor: During the recursive call of the synthesis algorithm, one out of the two "reduction checks" is skipped.

No storing of index: In standard BDD packages, the index of the variable has to be stored in each node. For complete BDDs it is sufficient to store the ordering only once and not as part of each node. This reduces the BDD node size by approximately 10-20%.

Earlier terminal cases: The completeness property often guarantees earlier termination of the recursive calls.

A sketch of the algorithms on top of NAND-BDDs is shown in Figure 3. The current implementation described here does not use complemented edges (see [3, 12]). This can be 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. This allows for no processing time overhead to be expended for a CEC application that would otherwise be present if a general purpose BDD package were employed. 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 neither *Dynamic Variable Ordering* (DVO) nor a memory management is included. Both are left out due to efficiency reasons, since they are not of interest for CEC, i.e. both are time consuming operations, but do not contribute to the decision whether functions are equivalent.

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 when using multi-engine concepts, 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 re-writing.

Complete BDDs have the following properties:

- They 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 due to the completeness property. Further nodes are generated by the NAND operation instead of ITE. However, memory is also saved by avoiding the index per node, the reference count and the reduction of the computed table size due to two instead of three operands.
- Complete 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 complete BDDs as compared to an ITE and a NAND-BDD realization - both in reduced form - 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 software has been written in *C++*. In order to provide a fair comparison, all packages are implemented in the same environment in that neither package uses a memory manager and all three are implemented without the use of complemented edges. The packages only make use of the simple terminal case and do not consider techniques like case normalization or *ITE_constant* as described in [3]. The implementation of complete BDDs makes use of the techniques of NAND-BDDs.

Benchmarks from ISCAS85 and the combinational part of ISCAS89 are used in the experimental results. For all packages the same static variable ordering using a method similar to that described in [7] is used and a hard upper node limit of 500.000 is used[1]. The only benchmarks reported here are those for which at least one technique obtained a result within this node limit and within 1 CPU hour. Furthermore, we focus on "non-trivial" examples which take longer than 1 CPU second.

In contrast to reduced BDDs, the introduction of new variables on which the function does not depend, can create additional nodes in the representation. But this difference is rather small and has no significant influence on the runtime as is shown in Figure 1 for benchmark *c0432*. Here, *max var* denotes the maximal number of variables, while the circuit has only 36 inputs. The runtime is given in the

[1] Note that the use of a node limit instead of a memory limit is pessimistic for complete BDDs, since the memory per node is smaller.

Table 2. Reduced vs. complete BDDs during construction

name	in	out	ITE		NAND		Complete	
			nodes	time	nodes	time	nodes	time
cs01423	91	79	87987	2.34	117833	1.04	343027	5.36
cs05378	214	228	47583	15.20	52375	12.47	400847	7.81
c0432	36	7	28656	190.81	35591	5.48	49050	0.67
c1908	33	25	162035	264.24	185989	2.43	160208	2.53
c5315	178	123	116416	10.36	186178	1.80	-	-
c0880	60	26	50843	0.35	60326	0.52	154470	2.04
c0499	41	32	136821	264.14	146193	84.67	154842	2.53
c1355	41	32	-	-	-	-	347448	7.96

Table 1. Dependence on number of variables

name	max var	nodes	time
c0432	36	49050	0.67
	40	49058	0.67
	44	49066	0.67
	50	49078	0.67
	100	49178	0.68

last column. Here the "redundant variables" are added at the end of the ordering. The situation can become worse, if they are chosen in between.

In the experiments we measure the CPU time and the number of nodes needed for the BDD construction. The results are given in Table 2. The *name* of the benchmark is given in the first column. The number of inputs and outputs of each circuit are given in column *in* and *out*, respectively. In columns ITE, NAND, and Complete, *nodes* and *time* denotes the number of nodes allocated during the BDD construction and the time needed, respectively. As can be seen, complete BDDs perform on average much better than reduced BDDs. Even compared to NAND-BDDs further significant reductions can be observed (see e.g. *c0499*). The number of nodes in complete BDDs is larger and this can lead to the case that the BDD cannot be build, while the other techniques succeed. This can especially occur for functions with many variables (see *c5315*). On the other hand, due to the fast operation, this "node overflow" is detected very fast and in the case of CEC does not block other techniques. In contrast, if happens that complete BDDs succeed to build the graph in a few seconds, while both other techniques cannot do it within the given time limit. Of course, for fast equivalence checking based on multi-engine solvers the proposed technique has significant advantages, since no time is wasted on a non-promising approach.

5 Conclusions

Complete BDDs have been studied in this paper. It has been shown that the synthesis operation can be simplified and several advantages have been discussed. First experimental results on a prototype implementation have lead to significant runtime reductions for symbolic simulation as it is used in combinational equivalence checking.

It is focus of future work to include the techniques described above in a highly optimized BDD package, like CUDD [16], to see how the numbers transfer. It is obvious that savings can be obtained also in this case, since the package is simplified, but more experimental studies are needed.

References

[1] B. Becker, R. Drechsler, and R. Werchner. On the relation between BDDs and FDDs. *Information and Computation*, 123(2):185–197, 1995.

[2] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conf.*, 1999.

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

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

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

[6] R. Drechsler and M.Thornton. Fast and efficient equivalence checking based on NAND-BDDs. In *IFIP Int'l Conf. on VLSI*, pages 401–405, 2001.

[7] 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.

[8] 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.

[9] A. Kuehlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Design Automation Conf.*, pages 232–237, 2001.

[10] H.-T. Liaw and C.-S. Lin. On the OBDD-representation of general Boolean functions. In *IEEE Trans. on Comp.*, volume 41, pages 661–664, 1992.

[11] J.P. Marques-Silva and K.A. Sakallah. Boolean satisfiability in electronic design automation. In *Design Automation Conf.*, pages 675–680, 2000.

[12] 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.

[13] S. Nagayama, T. Sasao, Y. Igushi, and M. Matsuura. Representations of logic functions using QRMDDs. In *Int'l Symp. on Multi-Valued Logic*, pages 261–267, 2002.

[14] 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.

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

[16] F. Somenzi. Efficient manipulation of decision diagrams. *Software Tools for Technology Transfer*, 3(2):171–181, 2001.