Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking

P. F. Williams[1] A. Biere[2] E. M. Clarke[3] A. Gupta[3]

February, 2000

CMU-CS-00-110

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Dept. Information Technology, Technical University of Denmark
[2]Department of Informatics, University of Karlsruhe
[3]School of Computer Science, Carnegie Mellon University

Abstract

In this paper we show how to do symbolic model checking using Boolean Expression Diagrams (BEDs), a non-canonical representation for Boolean formulas, instead of Binary Decision Diagrams (BDDs), the traditionally used canonical representation. The method is based on standard fixed point algorithms, combined with BDDs and SAT-solvers to perform satisfiability checking. As a result we are able to model check systems for which standard BDD-based methods fail. For example, we model check a 256 bit shift-and-add multiplier and we are able to find a previously undetected bug in the specification of a 16 bit multiplier. As opposed to Bounded Model Checking (BMC) our method is complete in practice.

Our technique is based on a quantification procedure that allows us to eliminate quantifiers in Quantified Boolean Formulas (QBF). The basic step of this procedure is the *up-one* operation for BEDs. In addition we list a number of important optimizations to reduce the number of basic steps. In particular the optimization rule of *quantification-by-substitution* turned out to be very useful: $\exists x : g \wedge (x \Leftrightarrow f) \equiv g[f/x]$. The rule is used (1) during fixed point iterations, (2) for deciding whether an initial set of states is a subset of another set of states, and finally (3) for iterative squaring.

This research is sponsored in part by the Semiconductor Research Corporation (SRC) under agreement through Contract No. 99-TJ-684 and the National Science Foundation (NSF) under Grant Nos. CCR-9505472 and CCR-9803774. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.

Keywords: automatic verification, binary decision diagrams, boolean expression diagrams, SAT procedures, symbolic model checking, temporal logic

1 Introduction

Symbolic model checking has been performed using fixed point iterations for quite some time [10]. The key to the success is the canonical Binary Decision Diagram (BDD) [7] data structure for representing Boolean functions. However, such a representation explodes in size for certain functions. In this paper we show how to do symbolic model checking using Boolean Expression Diagrams (BEDs) [2], a non-canonical representation of Boolean functions. The method is theoretically complete as we only change the representation and not the algorithms. Dropping the canonicity requirement has both advantages and disadvantages: Non-canonical data structures are more succinct than canonical ones – sometimes exponentially more. Determining satisfiability of Boolean functions is easy with canonical data structures, but with non-canonical data structures it is hard. We show how to overcome the disadvantages and exploit of the advantages in symbolic model checking.

As a non-canonical representation, BEDs do not allow for constant time satisfiability checking. Instead we use two different methods for satisfiability checking: (1) SAT-solvers like GRASP [14] and SATO [16], and (2) conversion of BEDs to BDDs. BDDs are canonical and thus satisfiability checking is a constant time operation. We perform symbolic model checking the classical way with fixed point iterations. One of the key elements of our method is the *quantification-by-substitution* rule: $\exists x : g \wedge (x \Leftrightarrow f) \equiv g[f/x]$. The rule is used (1) during fixed point iterations, (2) while deciding whether an initial set of states is a subset of another set of states, and finally (3) while doing iterative squaring.

While complete in the sense that it handles full CTL model checking, our method performs best if the system has few inputs and the transition relation can be written as a conjunction of next-state functions. The reason is that this allows us to fully exploit the *quantification-by-substitution* rule.

Using our method, we can model check a 256 bit shift-and-add multiplier, which requires 256 iterations to reach the fixed point. This should be compared with the 23 bit multipliers that standard BDD methods can handle. In fact, we are able to detect a previously unknown bug in the specification of a 16 bit multiplier. It was generally thought that iterative squaring was of no use in model checking. However, we show that iterative squaring enables us to calculate the reachable set of states for *all* 32 outputs of a 16 bit multiplier faster than without iterative squaring.

Model checking was invented by Clarke, Emerson, and Sistla in the 1980s [12]. Their model checking method required an explicit enumeration of states which limited the size of the systems they could handle. Burch *et al.* [10] showed how to do model checking without enumerating the states. They called this symbolic model checking. The idea is to represent sets of states by characteristic functions. The data structure of Binary Decision Diagrams turns out to a very efficient representation for characteristic functions. The advantages of BDDs are compactness of representation, canonicity, and ease of manipulation. Since the appearance of BDDs, many other related data structures have been proposed. Bryant gives an overview in [8]. One such data structure is the Boolean Expression Diagram. It is a generalization of BDDs. In this paper we will study BEDs for use in symbolic model checking.

Biere, Clarke *et al.* have proposed Bounded Model Checking (BMC) as an alternative method to BDD-based model checking [3, 4, 5]. They unfold the transition relation and look for repeatedly longer and longer counterexamples, and they use SAT-solvers instead of BDDs. BMC is good at finding errors with short counterexamples. The diameter of the system determines the number of unfoldings of the transition relation. Unfortunately, for many examples the diameter cannot be calculated and the estimates are too rough. In such cases BMC reduces to a partial verification method in practice. Our method does not need the computation of the diameter or approximations of it. Thus it turns out to be complete for much more examples.

The work most closely related to ours is by Abdulla, Bjesse and Eén. They consider symbolic reachability analysis using SAT-solvers [1]. For representing Boolean functions they use the Reduced Boolean Circuit data structure which closely resembles our Boolean Expression Diagrams. They perform reachability analysis using a fixed point iteration, and like us they make use of the *quantification-by-substitution* rule. They use Stålmarck's patented method [15] to determine satisfiability of Boolean functions. While related, their method and ours differ in a number of ways: In essence, the basic step in their and our quantification algorithm can be computed by the *up-one* BED-algorithm. Therefore we think BEDs are the most natural representation in this context. We handle full CTL while they concentrate on reachability. In our method the *quantification-by-substitution* rule is extensively used at three different places and not just during fixed point calculation. We have heuristics for choosing different SAT-procedure depending on the expected result of the satisfiability check. Candidates are various SAT solvers or an explicit BED to BDD conversion. We use SAT-solvers if the formula is expected to be satisfiable and either SAT-solvers or an explicit BED to BDD conversion if the formula is expected to be unsatisfiable. In their work Stålmarck's method is the only SAT procedure used. BEDs are always locally reduced and we identify further important simplification rules. Finally we make use of iterative squaring.

This paper is organized as follows. In section 2, we review the BED data structure. In section 3, we show how to do model checking using BEDs. In section 4, we give three applications of the *quantification-by-substitution* rule. In section 5, we deal with the size of BEDs. In section 6, we present the experimental results. Finally in section 7, we conclude.

2 Boolean Expression Diagrams

A Boolean Expression Diagram [2] is a data structure for representing and manipulating Boolean formulas. In this section we review the data structure.

Definition 2.1 (Boolean Expression Diagram). A *Boolean Expression Diagram* (BED) is a directed acyclic graph $G = (V, E)$ with vertex set V and edge set E. The vertex set V contains four types of vertices: terminal, variable, operator, and quantifier vertices.

- A *terminal vertex* v has as attribute a value $val(v) \in \{0, 1\}$.
- A *variable vertex* v has as attributes a Boolean variable $var(v)$, and two sons $low(v), high(v) \in V$.
- An *operator vertex* v has as attributes a binary Boolean operator $op(v)$, and two sons $low(v)$, $high(v) \in V$.
- A *quantifier vertex* v has as attributes a quantifier $quant(v) \in \{\exists, \forall\}$, a Boolean variable $var(v)$, and one son $low(v) \in V$.

The edge set E is defined by

$$\begin{aligned} E &= \big\{(v, low(v)) \bigm| v \in V \text{ and } v \text{ has the } low \text{ attribute } \big\} \\ &\cup \big\{(v, high(v)) \bigm| v \in V \text{ and } v \text{ has the } high \text{ attribute } \big\}. \end{aligned}$$

The relation between a BED and the Boolean function it represents is straightforward. Variable vertices have the semantics as vertices of BDDs and correspond to the *if-then-else* operator $x \to f_1, f_0$ defined as $(x \wedge f_1) \vee (\neg x \wedge f_0)$. Operator vertices correspond to their respective Boolean connectives. Quantifier vertices correspond to the quantification of their associated variable. This leads to the following correspondence between BEDs and Boolean functions:

Definition 2.2. A vertex v in a BED denotes a Boolean function f^v defined recursively as:

- If v is a terminal vertex, then $f^v = val(v)$.
- If v is a variable vertex, then $f^v = var(v) \to f^{high(v)}, f^{low(v)}$.
- If v is an operator vertex, then $f^v = f^{low(v)}\ op(v)\ f^{high(v)}$.
- If v is a quantifier vertex, then $f^v = quant(v)\ var(v)\ :\ f^{low(v)}$.

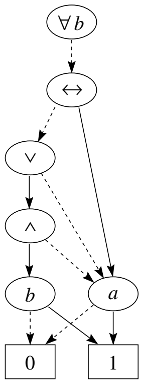

Figure 1: The BED for $\forall b : a \vee (a \wedge b) \Leftrightarrow a$. All edges are directed downwards; the dashed edges being the low ones.

The BED data structure is a representation form for formulas in QBF. If we disallow quantifier vertices, we get a representation form for propositional logic. If we disallow both operator and quantifier vertices, we get a BDD. As an example Figure 1 shows a BED for the formula $\forall b : a \vee (a \wedge b) \Leftrightarrow a$.

There exist algorithms for transforming a BED into a BDD. One such algorithm is *up-one*. It sifts variables one at a time to the root of the BED. Using *up-one* repeatedly to sift all the variables transforms the BED to a BDD. We refer the reader to [2, 13] for a more detailed description of *up-one* and its applications.

3 Model Checking

In this section, we review the standard model checking algorithm. The system to be verified is represented as a Kripke structure. A Kripke structure M is a tuple (S, I, T, ℓ), with a finite set of states S, a set of initial states $I \subset S$, a transition relation $T \subset S \times S$, and a labeling of the states $\ell : S \to \mathcal{P}(\mathcal{A})$ with *atomic propositions* $\mathcal{A}$.

A reactive system consists of a set of states and a set of inputs. The states are encoded as a Boolean vector of state variables $s_1, \ldots, s_n$. The inputs are also encoded as Boolean variables $s_{n+1}, \ldots, s_m$. These together form the state variables of the Kripke structure, $s_1, \ldots, s_m$. The atomic propositions correspond to the state variables. Each state is assumed to be labeled with the variables s_i that are 1 for that state. We use primed variables as next state variables, unprimed variables as current state variables, and we use characteristic functions over the state variables to represent sets. Since the inputs are non-deterministic, they are not constrained by the transition relation. Thus, the transition relation does not contain the primed versions of the input variables.

There are two ways to specify a transition relation in an SMV program: (a) by use of the "TRANS" statement, and (b) by use of the "ASSIGN" statement. In (a) one specifies the transition relation directly as a Boolean expression. In (b) one specifies next-state functions for state variables. Both methods can be used at the same time. We capture this as follows:

$$T(s, s') = t(s, \bar{s}') \wedge \bigwedge_i \tilde{s}'_i \Leftrightarrow f_i(s) \qquad (1)$$

where $\bar{s}$ and $\tilde{s}$ form a partitioning of $s_1, \ldots, s_n$. Here, $t(s, \bar{s}')$ comes from the "TRANS" statements and we call it the relational part, while $\bigwedge_i \tilde{s}'_i \Leftrightarrow f_i(s)$ comes from the "ASSIGN" statements and we call it the functional part. (If a primed variable is restricted by both "TRANS" and "ASSIGN" statements, we place it in the relational part of T.) Our verification method performs best if the transition relation is mainly in functional form.

We use CTL formulas to capture the properties we want to verify. A CTL formula characterizes a set of states, namely the set of states satisfying the formula. This set can be computed by a fixed point iteration. The central part of the fixed point iteration is the computation of relational products.

A relational product between the transition relation T and a set of states R is a new set of states. In a *forward* computation, the new set is the set of states reachable in one step from R. We call it the *Image* of R. In a *backward* computation, the new set is the set of states which in one step can reach a state in R. We call it the *PreImage* of R.

The following formulas show how to compute the image and preimage of R:

$$\begin{aligned} Image_{T,R}(s') &= \exists s \;:\; T(s,s') \wedge R(s) \\ PreImage_{T,R}(s) &= \exists s' \;:\; T(s,s') \wedge R(s') \end{aligned}$$

Note that all variables, including the inputs, are quantified out. For example, the algorithm in Figure 2 computes the characteristic function for the set of states satisfying the CTL formula "**AG** P" (read: always globally P) using backward iteration. It actually computes "$\neg$**EF** $\neg P$", i.e., it computes the set of states from which there exists a path to a state where P does not hold. The complement set then has the property that P holds along all paths.

AG P =
  $R_0 \leftarrow$ characteristic function for
    the set of states *not* satisfying P
  $i \leftarrow 0$
  repeat
    $i \leftarrow i+1$
    $R_{i+1} \leftarrow R_i \vee Preimage_{T,R_i}(s)$
  until $R_{i+1} \Rightarrow R_i$
  return $\neg R_i$

Figure 2: The algorithm for computing "**AG** P" using backward iteration. T is the transition relation for the system.

A Kripke structure $M = (S, I, T, \ell)$ satisfies a specification R if and only if I is a subset of R. In terms of characteristic functions this translates to the implication: $I \Rightarrow R$.

3.1 Quantification

The basic step in our quantification algorithm is to eliminate *one* quantified variable by the following rule:

$$\exists x : f \;\equiv\; f[0/x] \vee f[1/x] \qquad\qquad \forall x : f \;\equiv\; f[0/x] \wedge f[1/x]$$

Note that this basic step can easily be computed by performing a *up-one*(f, x) BED-operation and then replacing the top level variable vertex by an appropriate operator vertex.

In the worst case, while removing a quantifier from a formula, we double the formula size. Since each (pre)image computation involves existential quantification of all m state variables, we risk increasing the formula size by a factor of up to 2^m. In this section we present some syntactical transformations which help us to perform the quantifications efficiently.

The most important transformation is the *quantification-by-substitution* rule. It allows us to replace an existential quantification by a substitution:

$$\exists x : g \wedge (x \Leftrightarrow f) \quad\equiv\quad g[f/x] \tag{2}$$

Our verification method performs best when we can exploit the *quantification-by-substitution* rule. Such cases include systems with few inputs and systems with a transition relation that is mainly in functional form. After performing *quantification-by-substitution*, we quantify the remaining state variables (including inputs) using the rules below.

By applying scope reduction rules to a formula, we can push quantifiers down and thus reduce the potential blowup. The scope reduction rules are the following (shown for negation, conjunction and disjunction):

$$
\begin{array}{rclcrcl}
\exists x : \neg f & \equiv & \neg \forall x : f & \qquad & \forall x : \neg f & \equiv & \neg \exists x : f \\
\exists x : f \vee g & \equiv & (\exists x : f) \vee (\exists x : g) & & \forall x : f \wedge g & \equiv & (\forall x : f) \wedge (\forall x : g) \\
\exists x : f(y) \wedge g(x) & \equiv & f(y) \wedge (\exists x : g(x)) & & \forall x : f(y) \vee g(x) & \equiv & f(y) \vee (\forall x : g(x))
\end{array}
$$

Because BEDs are always reduced, for details see [2, 13], the quantifiers disappear if they are pushed all the way to the terminals.

3.2 Satisfiability Checking

There are two places where we need to determine whether a Boolean formula represented by a BED is satisfiable. First we need to detect that a fixed point has been reached in the computation of the set of states satisfying a CTL formula. Let R_i be the ith approximation to the fixed point. The fixed point has been reached if $R_{i+1} = R_i$. Using characteristic functions, this translates to $R_{i+1} \Leftrightarrow R_i$. However, depending on the CTL operator, the series of approximations will either be monotonically increasing or monotonically decreasing. It is therefore enough to check set inclusion instead of set equivalence. In the increasing case we check if $R_{i+1} \Rightarrow R_i$ is a tautology. In the decreasing case we check if $R_i \Rightarrow R_{i+1}$ is a tautology. Until we reach the fixed point, these formulas will *not* be tautologies. In other words, the negation of the formulas will be satisfiable. SAT-solvers are good at finding a satisfying variable assignment so we use a SAT-solver here.

Second we need to determine whether the initial set of states I is a subset of the set of states R represented by the CTL specification. In particular we have to check $I \Rightarrow R$ for tautology. There are two cases:

- The specification holds. This means that $I \Rightarrow R$ is a tautology. We could use a SAT-solver to prove that the negation of $I \Rightarrow R$ is not satisfiable. However, most SAT-solvers are not very good at proving non-satisfiability. We can also use BDDs. By using the *up-one* algorithm, we convert the BED for $I \Rightarrow R$ to a BDD. In many cases this conversion will *not* blow up in size [2, 13].

- The specification does not hold. A proof will be a variable assignment falsifying $I \Rightarrow R$. Or equivalent, a variable assignment satisfying $\neg(I \Rightarrow R)$. SAT-solvers are good at finding such variable assignments.

Of course, we do not know before hand whether the specification holds. A possibility is to run a SAT-solver and a BED to BDD conversion in parallel.

SAT-solvers like GRASP [14] and SATO [16] expect their input to be a propositional formula in conjunctive normal form (CNF). After the elimination of quantifiers, as described in Section 3.1, we still need to convert BEDs into CNF. For this conversion we use the well known technique of introducing new variables for every non-terminal vertex [3].

4 Applications of Quantification-by-Substitution

4.1 Preimage Computation

Consider the preimage computation in section 3. If the transition relation T is written as in equation (1), then we can apply rule (2) directly for the functional part. This can be done in one traversal of the BED. Figure 3 shows the pseudo-code. The algorithm works in a bottom-up way replacing all variables from the functional part of T with their next-state function. Line 4 does the replacing by performing a Shannon expansion of the variable vertex and inserting the next-state function.

preimage(u) =

if u is a terminal **then return** u

$(l,h) \leftarrow (\text{preimage}(low(u)), \text{preimage}(high(u)))$

if u is a variable vertex with variable from the functional part of T **then**

  return $(f_{var(u)} \wedge h) \vee (\neg f_{var(u)} \wedge l)$

else

  return makenode$(\alpha(u), l, h)$

Figure 3: The *preimage* algorithm computes the preimage of u for the functional part of the transition relation: $T_{func} = \bigwedge_i s'_i \Leftrightarrow f_i(s)$. The BED u is assumed to be quantifier-free. The tag $\alpha(u)$ is short for either $var(u)$ or $op(u)$.

4.2 Set Inclusion

We now describe a preprocessing step simplifying $I \Rightarrow R$, i.e., whether the initial set of states is a subset of the states characterized by the specification. The initial set of states I often has the form:

$$I = \bigwedge_i s_i \Leftrightarrow init_i(s)$$

where $init_i(s)$ is the function describing the initial state for the variable s_i. (Note that not all variables have an initial state specified.) In many cases $init_i(s)$ is either a constant or a very simple function, and we can use this fact to simplify $I \Rightarrow R$. Let I be written $I' \wedge (s_i \Leftrightarrow init_i(s))$ and assume $init_i(s)$ does not depend on variable s_i. Recall that $I \Rightarrow R$ is a tautology if and only if $\forall s_i : I \Rightarrow R$ is a tautology:

$$\begin{aligned} & \forall s_i : I \Rightarrow R \\ = \quad & \forall s_i : \neg (I' \wedge (s_i \Leftrightarrow init_i(s)) \wedge \neg R) \\ = \quad & \neg \exists s_i : I' \wedge (s_i \Leftrightarrow init_i(s)) \wedge \neg R \\ = \quad & \neg (I' \wedge \neg R)[init_i(s)/s_i] \\ = \quad & (I' \Rightarrow R)[init_i(s)/s_i] \end{aligned}$$

The $[init_i(s)/s_i]$ means a substitution of $init_i(s)$ for s_i. This reduces the number of variables and often simplifies the formula.

4.3 Iterative Squaring

Iterative squaring is a technique for reducing the number of iterations needed to reach the fixed point [9]. During reachability analysis we repeatedly square the transition relation:

$$T^2(s,s') = \exists s'' : T(s,s'') \wedge T(s'',s')$$

Assume that T is written as in equation (1). In general there is no way to square T and keep it in this form – the functional part will disappear. However, if we restrict ourselves to transition relations purely in functional form, squaring can be done easily:

$$\begin{aligned} T^2(s,s') &= \exists s'' : T(s,s'') \wedge T(s'',s') \\ &= \exists s'' : \left(\bigwedge_i s''_i \Leftrightarrow f_i(s) \right) \wedge \left(\bigwedge_i s'_i \Leftrightarrow f_i(s'') \right) \\ &= \bigwedge_i s'_i \Leftrightarrow (f_i(s'')[f(s)/s'']) \end{aligned}$$

where $[f(s)/s'']$ is a substitution of function $f_j(s)$ for variable s''_j (for all j). The algorithm is similar to the preimage algorithm in Figure 3.

In this way we can compute $T^{(2^k)}$ in only k steps. $T^{(2^k)}$ is a new transition relation representing all paths in T with a length of *exactly* 2^k. However, it is not possible to represent in functional form the transition relation allowing paths of length *up to* 2^k. As a consequence we cannot combine this form of iterative squaring with, for example, frontier set simplifications.

5 BED Simplifications

As we mentioned in section 3.2, transforming a BED to CNF increases the size of the formula as we introduce a new variable for each BED non-terminal vertex. It is therefore vital to keep the size of the BEDs down.

During the conversion of a BED to a BDD, the size may blow up. Even when the final BDD is small (as for a tautology), the intermediate results might be large. In this section we describe a method of keeping the BEDs small.

Keeping the BEDs reduced, as mentioned above, already gives us size reductions due to, for example, constant propagation. But we can reduce the size of the BEDs even more. This can be achieved by increasing the sharing of vertices and by removing local redundancies. In [13] we describe a set of rewriting rules in detail. Here we will just mention some of the ideas:

- Sharing can be increased by disallowing operator vertices which only differ in the order of their sons; for example $a \wedge b$ and $b \wedge a$. We fix an ordering $<$ of vertices and only create operator vertices with $low < high$.
- Size can be reduced by eliminating all negations below binary operators since for all binary operators op there exists another operator op' with $op'(x,y) = op(\neg x, y)$
- Size can be reduced by not using all 16 binary Boolean operators but only a subset of them. We use the set *nand*, *or*, *left implication*, *right implication*, and *bi-implication*. (For clarity, the BED in Figure 1 has not been reduced to this subset.)
- Size can be reduced by exploiting equivalences like the *absorption laws*, for example $a \vee (a \wedge b) = a$, and *distributive laws*, for example $(a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c)$.

We apply all these rewriting rules each time we create a new operator vertex.

6 Experimental Results

We have constructed a prototype implementation of our proposed model checking method. It performs CTL model checking on SMV programs. As SAT-solver we use SATO. We compare our method with the NUSMV model checker [11] and with Bwolen Yang's modified version of SMV[1], both of which are state-of-the-art in BDD-based model checking. Finally we compare reachability results with FIXIT from Adbulla, Bjesse, and Eèn [1].

6.1 Multiplier

This example comes from the BMC-1.0f distribution[2]. It is a $16 \times 16 \rightarrow 32$ shift-and-add multiplier. The specification is the c6288 combinational multiplier from the ISCAS'85 benchmark series [6]. For each output bit we verify that we cannot reach a state where the shift-and-add multiplier has finished its computation and the output bits of the two multipliers differ.

The multiplier fits into the category of SMV programs that we handle well. The operands are not modeled as inputs. Instead they are modeled as state variables with an unspecified initial state and the

[1] `http://www.cs.cmu.edu/~bwolen`
[2] `http://www.cs.cmu.edu/~modelcheck`

identity function as the next-state function. This lets us use *quantification-by-substitution* for all but the last iteration in the fixed-point calculation. Only in the last iteration do we need to quantify the operands out using the standard quantification methods.

Table 1 shows the runtimes for verifying that the multiplier satisfies the specification. Our BED-based method out-performs both NuSMV and Bwolen Yang's SMV as we are able to model check twice as many outputs as they are. FixIt handles the same number of outputs as our method, however, our method is faster by an order of magnitude.

Bit	BED	NuSMV	Bwolen	FixIt
0	2.2	11	9.4	2.9
1	2.3	23	17	3.1
2	2.9	50	33	3.7
3	3.8	130	71	4.8
4	5.2	290	159	6.6
5	7.0	702	383	11
6	9.2	-	1031	20
7	12	-	-	47
8	16	-	-	150
9	31	-	-	544
10	68	-	-	2078
11	352	-	-	8134
12	2201	-	-	30330

Table 1: Runtimes in seconds for verifying the correctness of a 16 bit multiplier. The BED, NuSMV, and Bwolen Yang's SMV experiments were performed on the same computer. The FixIt experiments are from Abdulla *et al.* using a backward method – computer unknown. A dash "-" indicates that the verification could not be completed with 800 MB of memory.

For the most difficult output in Table 1, the fixed point iteration accounts for only a fraction of the total runtime for our method. It takes less than a minute and almost no memory to calculate the fixed point. By far the most time is spent in proving $I \Rightarrow R$. SAT-solvers gave poor results, so we converted the BED for $I \Rightarrow R$ to a BDD.

We did the experiments in Table 1 without use of iterative squaring to enable fair comparisons. However, iterative squaring speeds up the fixed point calculations. Table 2 shows the runtimes for calculating the fixed points – with and without iterative squaring – for the same model checking problem as above.

Bit	Without I.S.	With I.S.
0	2.1	0.9
5	6.8	1.6
10	14	3.7
15	16	8.3
20	37	12
25	19	8.8
30	> 300	6.4

Table 2: Runtimes in seconds for the fixed point calculation in verifying the correctness of the 16 bit shift-and-add multiplier. Results are shown for computations with and without iterative squaring (I.S.). The space requirements are small, i.e., less than 16 MB.

To see how our method handles erroneous designs, we introduced an error in the specification of the multiplier by negating one of the internal nodes (this is marked as "bug D" in the multiplier file in the BMC distribution). We observe that the fixed points are computed in roughly the same amount of CPU time and memory (both with and without iterative squaring). The difference is when we prove $I \Rightarrow R$. Using BED to BDD conversion as with the correct design, we now get poorer results because $I \Rightarrow R$ is *not* a tautology and the final BDD is not necessarily small. However, using a SAT-solver, we get much better results. In many cases, the SAT-solver is able to find a counterexample almost immediately. We are able to model check the first 19 outputs as well as some of the later outputs of the multiplier using less than 16 MB of memory and one minute of CPU time per output. NuSMV and Bwolen Yang's SMV perform as bad as before.

We were able to find a bug in the "correct" specification of the multiplier for the two most significant outputs. Using iterative squaring and SATO we find these errors in seven and eight seconds, respectively. It turns out that the two outputs have been swapped. The original net-list for c6288 does not contain information about which gates correspond to which multiplier outputs. However, each gate is numbered and the outputs seem to be increasing with the the gate numbers – with the exception of the last pair of outputs. This emphasizes the fact that SAT-bases methods are good at finding bugs in a system.

We constructed shift-and-add multipliers of different sizes and verified that they always terminate, i.e., we checked "**AF** done". The number of iterations needed to reach the fixed point is equal to the size of the multiplier. This lets us test how well our methods handles cases with lots of iterations. Table 3 shows the results. We compare our method with NUSMV and Bwolen Yang's SMV. Our method performs much better as we are both significantly faster and we are able to handle much larger designs. We cannot compare with FIXIT as it does not handle **AF** properties.

Size	BED	NUSMV	Bwolen
16	1.6	2.2	5.2
18	1.8	18	9.1
20	2.0	90	24
22	2.3	472	104
23	2.7	-	253
24	2.8	-	-
32	3.7	-	-
64	17	-	-
128	119	-	-
256	1185	-	-

Table 3: Runtimes in seconds for verifying that shift-and-add multipliers of different sizes always terminate, i.e., we check "**AF** done". The number of iterations to reach the fixed point is equal to the size of the multiplier.

6.2 Barrel Shifter

This example is a barrel shifter from the BMC-1.0f distribution and like the multiplier, it also falls within the category of systems which we handle well. A barrel shifter consists of two register files. The contents of one of the register files is rotated at each step while the other file stays the same. The width of a register is $log\ R$, where R is the size of the register file.

The correctness of the barrel shifter is proven by showing that if two registers from the files have the same contents, then their neighbors are also identical. The left part of Table 4 shows the results. The BED and FIXIT methods are both fast, however, the BED method scales better and thus outperforms FIXIT. NUSMV and Bwolen Yang's SMV are both unable to construct the BDD for the transition relation for all but the smallest examples.

We prove liveness for the barrel shifter by showing that a pair of registers in the files will eventually become equal. The number of iterations for the fixed point calculation is equal to the size of the register file. The right part of Table 4 shows the results. FIXIT cannot handle liveness properties so we cannot compare with it. As in the previous case, NUSMV and Bwolen Yang's SMV can only handle small examples.

7 Conclusion

We have presented a BED-based CTL model checking method based on the classical fixed point iterations. Quantification is often the Achilles heel in CTL fixed point iterations but by using *quantification-by-substitution* we are in some cases able to deal effectively with it. While our method is complete, it performs best on examples with a low number of inputs and where the transition relation is mainly in functional form. In these situations we can fully exploit the *quantification-by-substitution* rule.

Size	BED	NuSMV	Bwolen	FixIt
2	0.1	0.1	1.0	0.1
4	0.3	0.2	2.5	0.1
6	0.4	609	-	0.2
8	0.4	-	-	0.5
10	0.6	-	-	1.1
20	1.9	-	-	14
30	4.0	-	-	52
40	8.0	-	-	231
50	13	-	-	502
60	19	-	-	?
70	30	-	-	?

Size	BED	NuSMV	Bwolen
2	0.2	0.1	1.0
4	0.5	0.2	2.1
6	0.7	521	-
8	0.9	-	-
10	1.2	-	-
20	3.2	-	-
30	5.9	-	-
40	11	-	-
50	18	-	-
60	28	-	-
70	47	-	-

Table 4: Runtimes in seconds for invariant (left) and liveness (right) checking of the barrel shifter example. A question mark indicates that the runtime for FixIt was not reported in [1]. For the BED method we use Sato for checking satisfiability of $I \Rightarrow R$.

We have shown how the *quantification-by-substitution* rule can also help simplify the final set inclusion problem of model checking and help perform efficient iterative squaring. Our proposed method combines SAT-solvers and BED to BDD conversions to perform satisfiability checking. We have presented a set of local rewriting rules which helps to keep the size of the BEDs down.

We have demonstrated our method by model checking large shift-and-add multipliers and barrel shifters, and we obtain results superior to standard BDD-based model checking methods. Furthermore, we were able to find a previously undetected bug in the specification of a 16 bit multiplier.

Future work includes investigating two variable ordering problems. One is the variable ordering when converting the BED for $I \Rightarrow R$ to a BDD. The variable ordering is known to be very important in BDD construction, and since we, in some cases, spend much time on converting $I \Rightarrow R$ to a BDD, our method will benefit from a good variable ordering heuristic. The other problem is the order in which we quantify the variables in the preimage computation. This will be interesting especially in cases where we cannot use the *quantification-by-substitution* rule. Finally we are currently investigating how to extend our method to work well for systems with many inputs.

References

[1] P. A. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on SAT solvers. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2000. (To appear).

[2] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *IEEE Symposium on Logic in Computer Science (LICS)*, July 1997.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[5] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[6] F. Brglez and H. Fujiware. A neutral netlist of 10 combinational benchmarks circuits and a target translator in Fortran. In *Special Session International Symposium on Circuits and Systems (ISCAS)*, 1985.

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

[8] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 236–243, November 1995.

[9] J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[11] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.

[12] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[13] H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions on Computer Aided Design*, July 1999.

[14] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48, 1999.

[15] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. J. Windley, editors, *Proc. Formal Methods in Computer-Aided Design, Second International Conference, FMCAD'98, Palo Alto/CA, USA*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99, November 1998.

[16] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275, Berlin, July 1997. Springer-Verlag.