# Symbolic Reachability Analysis Based on SAT-Solvers

Parosh Aziz Abdulla[1], Per Bjesse[2], and Niklas Eén[2]

[1] Uppsala University and Prover Technology, Sweden
parosh@docs.uu.se
[2] Chalmers University of Technology, Sweden
{bjesse,een}@cs.chalmers.se

**Abstract.** The introduction of symbolic model checking using Binary Decision Diagrams (BDDs) has led to a substantial extension of the class of systems that can be algorithmically verified. Although BDDs have played a crucial role in this success, they have some well-known drawbacks, such as requiring an externally supplied variable ordering and causing space blowups in certain applications. In a parallel development, SAT-solving procedures, such as Stålmarck's method or the Davis-Putnam procedure, have been used successfully in verifying very large industrial systems. These efforts have recently attracted the attention of the model checking community resulting in the notion of *bounded model checking*. In this paper, we show how to adapt standard algorithms for symbolic reachability analysis to work with SAT-solvers. The key element of our contribution is the combination of an algorithm that removes quantifiers over propositional variables and a simple representation that allows reuse of subformulas. The result will in principle allow many existing BDD-based algorithms to work with SAT-solvers. We show that even with our relatively simple techniques it is possible to verify systems that are known to be hard for BDD-based model checkers.

## 1 Introduction

In recent years *model checking* [CES86,QS82] has been widely used for algorithmic verification of finite-state systems such as hardware circuits and communication protocols. In model checking, the specification of the system is formulated as a temporal logical formula, while the implementation is described as a finite-state transition system. Early model-checking algorithms suffered from *state explosion*, as the size of the state space grows exponentially with the number of components in the system. One way to reduce state explosion is to use *symbolic model checking* [BCMD92,McM93], where the transition relation is coded symbolically as a boolean expression, rather than explicitly as the edges of a graph. Symbolic model checking achieved its major breakthrough after the introduction of *Binary Decision Diagrams* (BDDs) [Bry86] as a data structure for representing boolean expressions in the model checking procedure. An important property of BDDs is that they are canonical. This allows for substantial

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 411–425, 2000.

sub-expression sharing, often resulting in a compact representation. In addition, canonicity implies that satisfiability and validity of boolean expressions can be checked in constant time. However, the restrictions imposed by canonicity can in some cases lead to a space blowup, making memory a bottleneck in the application of BDD-based algorithms. There are examples of functions, for example multiplication, which do not allow sub-exponential BDD representations. Furthermore, the size of a BDD is dependent on the variable ordering which in many cases is hard to optimize, both automatically and by hand. BDD-based methods can typically handle systems with hundreds of boolean variables.

A related approach is to use satisfiability solvers, such as implementations of Stålmarck's method [Stå] and the Davis-Putnam procedure [Zha97]. These methods have already been used successfully for verifying industrial systems [SS00,Bor97,Bor98,SS90,GvVK95]. SAT-solvers enjoy several properties which make them attractive as a complement to BDDs in symbolic model checking. For instance, their performance is less sensitive to the size of the formulas, and they can in some cases handle propositional formulas with thousands of variables. Furthermore, SAT-solvers do not suffer from space explosion, and do not require an external variable ordering to be supplied. Finally, satisfiability solving is an NP-complete problem, whereas BDD-construction solves a #P-complete problem [Pap94] as it is possible to determine the number of models of a BDD in polynomial time. #P-complete problems are widely believed to be harder than NP-complete problems.

The aim of this work is to exploit the strength of SAT-solving procedures in order to increase the class of systems amenable to verification via the traditional symbolic methods. We consider modifications of two standard algorithms—forward and backward reachability analysis—where formulas are used to characterize sets of reachable states [Bje99]. In these algorithms we replace BDDs by satisfiability checkers such as the PROVER implementation of Stålmarck's method [Stå] or SATO [Zha97]. We also use a data structure which we call *Reduced Boolean Circuits* (RBCs) to represent formulas. RBCs avoid unnecessarily large representations through the reuse of subformulas, and allow for efficient storage and manipulation of formulas. The only operation of the reachability algorithms that does not carry over straightforwardly to this representation is quantification over propositional variables. Therefore, we provide a simple procedure for the removal of quantifiers, which gives adequate performance for the examples we have tried so far.

We have implemented a tool FIXIT [Eén99] based on our approach, and carried out a number of experiments. The performance of the tool indicates that even though we use simple techniques, our method can perform well in comparison to existing ones.

**Related Work.** *Bounded Model Checking* (BMC) [BCC+99,BCCZ99,BCRZ99] is the first approach in the literature to perform model checking using SAT-solvers. To check reachability, the BMC procedure searches for counterexamples (paths to undesirable states) by "unrolling" the transition relation $k$ steps. The unrolling is described by a (quantifier-free) formula which characterizes the set

of feasible paths through the transition relation with lengths smaller than or equal to $k$. The search can be terminated when the value of $k$ is equal to the *diameter* of the system—the maximum length of all shortest path between states in the system. Although the diameter can be specified by a logical formula, its satisfiability is usually hard to check, making BMC incomplete in practice. Furthermore, for "deep" transition systems, formulas characterizing the set of reachable states may be much smaller than those characterizing witness paths. Since our method is based on encodings of sets of states, it may in some cases cope with systems which BMC fails to analyze as it generates formulas that are too large.

Our representation of formulas is closely related to *Binary Expression Diagrams* (BEDs) [AH97,HWA97]. In fact there are straightforward linear space translations back and forth between the representations. Consequently, RBCs share the good properties of BEDs, such as being exponentially more succinct than BDDs [AH97]. The main difference between our approach and the use of BEDs is the way in which satisfiability checking and existential quantification is handled. In [AH97], satisfiability of BEDs is checked through a translation to equivalent BDDs. Although many simplifications are performed at the BED level, converting to BDDs during a fixpoint iteration could cause degeneration into a standard BDD-based fixpoint iteration. In contrast, we check satisfiability by mapping RBCs back to formulas which are then fed to external SAT-solvers. In fact, the use of SAT-solvers can also be applied to BEDs, but this does not seem to have been explored so far. Furthermore, in the BED approach, existential quantification is either handled by introducing explicit quantification vertices, or by a special transformation that rewrites the representation into a form where naive expansion can be applied. We use a similar algorithm that also applies an extra inlining rule. The inlining rule is particularly effective in the case of backward reachability analysis, as it is always applicable to the generated formulas. To our knowledge, no results have been reported in the literature on applications of BEDs in symbolic model checking. We would like to emphasize that we view RBCs as a relatively simple representation of formulas, and not as a major contribution of this work.

## 2 Preliminaries

We verify systems described as synchronous circuits constructed from elementary combinational gates and unit delays—a simple, yet popular, model of computation. The unit delays are controlled by a global clock, and we place no restriction on the inputs to a circuit. The environment is free to behave in any fashion.

We define the *state-holding elements* of a circuit to be the primary inputs and the contents of the delays, and define a *valuation* to be an assignment of boolean values to the state-holding elements. The behaviour of a circuit is modelled as a state-transition graph where (1) each valuation is a state; (2) the initial states comprise all states that agree with the initial values of the delays; and (3) there

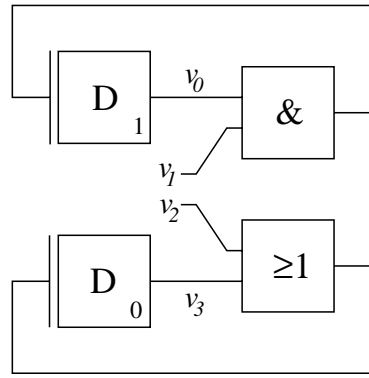


**Fig. 1.** A simple circuit built from combinational gates and delays

is a transition between two states if the circuit can move between the source state and the destination state in one clock cycle.

We construct a symbolic encoding of the transition graph in the standard manner. We assign every state-holding element a propositional state variable $v_i$, and make two copies of the set of state variables, $s = \{v_0, v_1, \ldots, v_k\}$ and $s' = \{v'_0, v'_1, \ldots, v'_k\}$. Given a circuit we can now generate two *characteristic formulas*. The first of the characteristic formulas, $Init(s) = \bigwedge_i v_i \leftrightarrow \phi_i$, defines the initial values of the state-holding elements. The second characteristic formula, $Tr(s, s') = \bigwedge_i v'_i \leftrightarrow \psi_i(s)$, defines the next-state values of state-holding elements in terms of the current-state values.

*Example 1.* The following formulas characterize the circuit in Figure 1:

$$Init = (v_0 \leftrightarrow \top) \wedge (v_3 \leftrightarrow \bot)$$
$$Tr = (v'_0 \leftrightarrow (v_0 \wedge v_1)) \ \wedge \ (v'_3 \leftrightarrow (v_2 \vee v_3))$$

We investigate the underlying state-transition graph by applying operations at the formula level. In doing so we make use of the following three facts. First, the relation between any points in a given circuit can be expressed as a propositional formula over the state-holding variables. Second, we can represent any set $S$ of transition-graph states by a formula that is satisfied exactly by the states in $S$. Third, we can lift all standard set-level operations to operations on formulas (for example, set inclusion corresponds to formula-level implication and set nonemptiness checking to satisfiability solving, respectively).

## 3 Reachability Analysis

Given the characteristic formulas of a circuit and a formula $Bad(s)$, we define the *reachability problem* as that of checking whether it is possible to reach a state that satisfies $Bad(s)$ from an initial state. As an example, in the case of

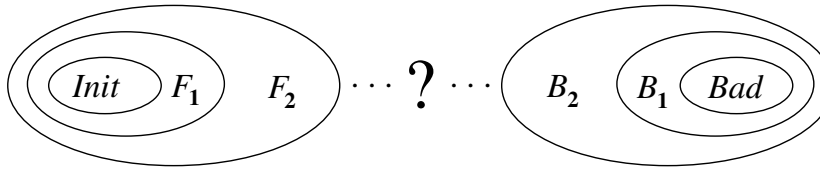


**Fig. 2.** The intuition behind the reachability algorithms

the circuit in Figure 1, we might be interested in whether the circuit could reach a state where the two delay elements output the same value (or equivalently, where the formula $v_0 \leftrightarrow v_3$ is satisfiable). We adapt two standard algorithms for performing reachability analysis. In *forward reachability* we compute a sequence of formulas $F_i(s)$ that characterize the set of states that the initial states can reach in $i$ steps:

$$F_0(s) = Init$$
$$F_{i+1}(s') = toProp(\exists s.\ Tr(s, s') \wedge F_i(s)))$$

Each computation of $F_{i+1}$ gives rise to a *Quantified Boolean Formula* (QBF), which we translate back to a pure propositional formula using an operation *toProp* (defined in in Section 5). We terminate the sequence generation if either (1) $F_n(s) \wedge Bad(s)$ is satisfiable: this means that a bad state is reachable; hence we answer the reachability problem positively; or (2) $\bigvee_{k=0}^{n} F_k(s) \rightarrow \bigvee_{k=0}^{n-1} F_k(s)$ holds: this implies that we have reached a fixpoint without encountering a bad state; consequently the answer to the reachability question is negative.

In *backward reachability* we instead compute a sequence of formulas $B_i(s)$ that characterize the set of states that can reach a bad state in $i$ steps:

$$B_0(s) = Bad$$
$$B_{i+1}(s) = toProp(\exists s'.\ Tr(s, s') \wedge B_i(s')))$$

In a similar manner to forward reachability, we terminate the sequence generation if either (1) $B_n(s) \wedge Init(s)$ is satisfiable, or (2) $\bigvee_{k=0}^{n} B_k(s) \rightarrow \bigvee_{k=0}^{n-1} B_k(s)$ holds.

Figure 2 shows the intuition behind the algorithms. We remark that the two reachability methods can be combined by alternating between the computation of $F_{i+1}$ and $B_{i+1}$. The generation can be terminated when either a fixpoint is reached in some direction, or when $F_n$ and $B_n$ intersect. However, we do not make use of hybrid analyses in this paper.

We need to address three nontrivial issues in an implementation of the adapted reachability algorithms. First, we must avoid the generation of unnecessarily large formula characterizations of the sets $F_i$ and $B_i$—formulas are not a canonical representation. Second, we must define the operation *toProp* in such a way that it translates quantified boolean formulas to propositional logic without

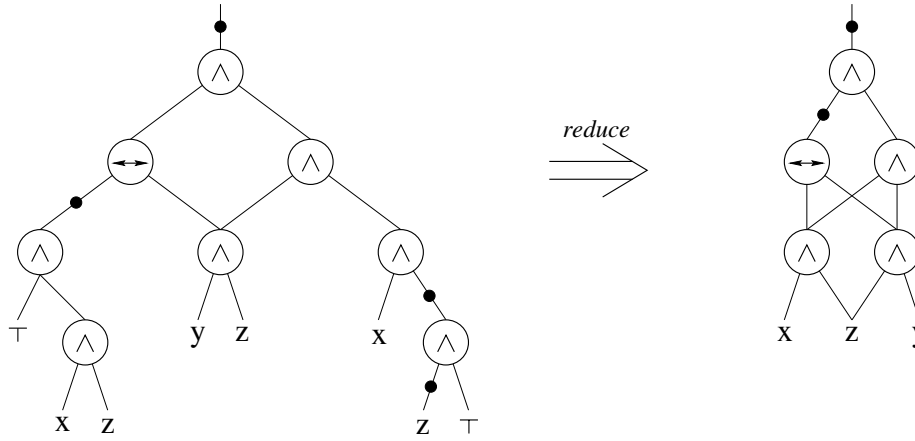


**Fig. 3.** A non-reduced *Boolean Circuit* and its reduced form

needlessly generating exponential results. Third, we must interface efficiently to external satisfiability solvers. The remainder of the paper explains our solutions, and evaluates the resulting reachability checker.

## 4 Representation of Formulas

Let **Bool** denote the set of booleans; **Vars** denote the set of propositional variables, including a special variable $\top$ for the constant *true*; and **Op** denote the set $\{\leftrightarrow, \wedge\}$.

We introduce the representation *Boolean Circuit* (BC) for propositional formulas. A BC is a directed acyclic graph, $(\mathbf{V}, \mathbf{E})$. The vertices $\mathbf{V}$ are partitioned into internal nodes, $\mathbf{V_I}$, and leaves, $\mathbf{V_L}$. The vertices and edges are given attributes as follows:

- Each internal vertex $v \in \mathbf{V_I}$ has three attributes: A binary operator $op(v) \in \mathbf{Op}$, and two edges $left(v), right(v) \in \mathbf{E}$.
- Each leaf $v \in \mathbf{V_L}$ has one attribute: $var(v) \in \mathbf{Vars}$.
- Each edge $e \in \mathbf{E}$ has two attributes: $sign(e) \in \mathbf{Bool}$ and $target(e) \in \mathbf{V}$.

We observe that negation is coded into the edges of the graph, by the *sign* attribute. Furthermore, we identify *edges* with *subformulas*. In particular, the whole formula is identified with a special top-edge having no source vertex. The interpretation of an edge as a formula is given by the standard semantics of $\wedge, \leftrightarrow$ and $\neg$ by viewing the graph as a parse tree (with some common sub-expressions shared). Although $\wedge$ and $\neg$ are functionally complete, we choose to include $\leftrightarrow$ in the representation as it would otherwise require three binary connectives to express. Figure 3 shows an example of a BC.

A *Reduced Boolean Circuit* (RBC) is a BC satisfying the following properties:

```
reduce(AND, left ∈ RBC, right ∈ RBC)
    if    (left = right)   return left
    elif  (left = ¬right)  return ⊥
    elif  (left = ⊤)       return right
    elif  (right = ⊤)      return left
    elif  (left = ⊥)       return ⊥
    elif  (right = ⊥)      return ⊥
    else                   return NIL

reduce(EQUIV, left ∈ RBC, right ∈ RBC)
    if    (left = right)   return ⊤
    elif  (left = ¬right)  return ⊥
    elif  (left = ⊤)       return right
    elif  (left = ⊥)       return ¬right
    elif  (right = ⊤)      return left
    elif  (right = ⊥)      return ¬left
    else                   return NIL

mk_Comp(op ∈ Op, left ∈ RBC, right ∈ RBC, sign ∈ Bool)
    result := reduce(op, left, right)
    if (result ≠ NIL)
        return id(result, sign)     – id returns result or ¬result depending on sign

    if (right < left)
        (left, right) := (right, left)     – Swap the values of left and right

    if (op = EQUIV)
        sign  := sign xor sign(left) xor sign(right)
        left  := unsigned(left)
        right := unsigned(right)

    result := lookup(RBC_env, (op, left, right))     – Look for vertex in environment
    if (result = NIL)
        result := insert(RBC_env, (op, left, right))
    return id(result, sign)
```

**Fig. 4.** Pseudo-code for creating a composite RBC from two existing RBCs

1. All common subformulas are shared so that no two vertices have identical attributes.
2. The constant $\top$ never occurs in an RBC, except for the single-vertex RBCs representing *true* or *false*.
3. The children of an internal vertex are syntactically distinct, $\mathit{left}(v) \neq \mathit{right}(v)$.
4. If $\mathit{op}(v) = \ \leftrightarrow$ then the edges to the children of $v$ are unsigned.
5. For all vertices $v$, $\mathit{left}(v) < \mathit{right}(v)$, for some total order $<$ on BCs.

The purpose of these constraints is to identify as many equivalent formulas as possible, and thereby increase the amount of subformula sharing. For this reason we allow only one representation of $\neg(\phi \leftrightarrow \psi) \iff (\neg\phi \leftrightarrow \psi)$ (in 4 above), and $(\phi \wedge \psi) \iff (\psi \wedge \phi)$ (in 5 above).

The RBCs are created in an implicit environment, where all existing subformulas are tabulated. We use the environment to assure property (1). Figure 4 shows the only non-trivial constructor for RBCs, *mk_Comp*, which creates a composite RBC from two existing RBCs (we use $x \in \mathrm{Vars}(\phi)$ to denote that $x$ is a variable occurring in the formula $\phi$). It should be noted that the above properties only takes constant time to maintain in *mk_Comp*.

## 5 Quantification

In the reachability algorithms we make use of the operation *toProp* to translate QBF formulas into equivalent propositional formulas. We reduce the translation of a set of existential quantifiers to the iterated removal of a single quantifier after we have chosen a quantification order. In the current implementation an arbitrary order is used, but we are evaluating more refined approaches.

Figure 5 presents the quantification algorithm of our implementation. By definition we have:

$$\exists x \,.\, \phi(x) \iff \phi(\bot) \vee \phi(\top) \qquad (*)$$

The definition can be used to naively resolve the quantifiers, but this may yield an exponential blowup in representation size. To try to avoid this, we use the following well-known identities (applied from left to right) whenever possible:

*Inlining:*

$$\exists x \,.\, (x \leftrightarrow \psi) \wedge \phi(x) \iff \phi(\psi) \qquad (\text{where } x \notin \text{Vars}(\psi))$$

*Scope Reduction:*

$$\exists x \,.\, \phi(x) \wedge \psi \iff (\exists x.\phi(x)) \wedge \psi \qquad (\text{where } x \notin \text{Vars}(\psi))$$
$$\exists x \,.\, \phi(x) \vee \psi(x) \iff (\exists x.\phi(x)) \vee (\exists x.\psi(x))$$

When applicable, *inlining* is an effective method of resolving quantifiers as it immediately removes all occurrences of the quantified variable $x$. The applicability of the transformation relies on the fact that the formulas occurring in reachability often have a structure that matches the rule. This is particularly true for backward reachability as the transition relation is a conjunction of next state variables defined in terms of current state variables $\bigwedge_i v_i' \leftrightarrow \psi_i(s)$.

The first step of the inlining algorithm temporarily changes the representation of the top-level conjunction. From the binary encoding of the RBC, we extract an equivalent set representation $\bigwedge\{\phi_0, \phi_1, \ldots, \phi_n\}$. If the set contains one or more elements of the form $x \leftrightarrow \psi$, the smallest such element is removed from the set and its right-hand side $\psi$ is substituted for $x$ in the remaining elements. The set is then re-encoded as an RBC.

If inlining is not applicable to the formula (and variable) at hand, the translator tries to apply the *scope reduction* rules as far as possible. This may result in a quantifier being pushed through an Or (represented as negated And), in which case inlining may again be possible.

For subformulas where the scope can no longer be reduced, and where inlining is not applicable, we resort to *naive quantification* (*). Reducing the scope as much as possible before doing this will help prevent blowups. Sometimes the quantifiers can be pushed all the way to the leaves of the RBC, where they can be eliminated.

Throughout the quantification procedure, we may encounter the same subproblem more than once due to shared subformulas. For this reason we keep a table of the results obtained from all previously processed subformulas.

```
– Global variable processed tabulates the results of the performed quantifications.

quant_naive(φ ∈ RBC, x ∈ Vars)
    result = subst(φ, x, ⊥) ∨ subst(φ, x, ⊤)
    insert(processed, φ, x, result)
    return result

quant_reduceScope(φ ∈ RBC, x ∈ Vars)
    if (x ∉ Vars(φ)) return φ
    if (φ = x)       return ⊤

    result := lookup(processed, φ, x)
    if (result ≠ NIL)
        return result

    – In the following φ must be composite and contain x:
    if (φ_op = EQUIV)
        result := quant_naive(φ, x)
    elif (not φ_sign)    – Operator AND, unsigned
        if    (x ∉ Vars(φ_left))  result := φ_left  ∧  quant_reduceScope(φ_right, x)
        elif  (x ∉ Vars(φ_right)) result := quant_reduceScope(φ_left, x)  ∧  φ_right
        else                      result := quant_naive(φ, x)
    else    – Operator AND, signed ("OR")
        result := quant_inline(¬φ_left, x)  ∨  quant_inline(¬φ_right, x)

    insert(processed, φ, x, result)
    return result

quant_inline(φ ∈ RBC, x ∈ Vars)    – "Main"
    C := collectConjuncts(φ)   – Merge all binary ANDs at the top of φ into a
                                 "big" conceptual conjunction (returned as a set).
    ψ  := findDef(C, x)        – Return the smallest formula ψ such that (x ↔ ψ)
                                 is a member of C.

    if (ψ ≠ NIL)
        C' := C \ (x ↔ ψ)                      – Remove definition from C.
        return subst(makeConj(C'), x, ψ)       – makeConj builds an RBC.
    else
        return quant_reduceScope(φ, x)
```

**Fig. 5.** Pseudo-code for performing existential quantification over one variable. By $\phi_{left}$ we denote $left(target(\phi))$ etc. We use $\wedge$, $\vee$ as abbreviations for calls to *mk_Comp*

## 6 Satisfiability

Given an RBC, we want to decide whether there exists a satisfying assignment for the corresponding formula by applying an external SAT-solver. The naive translation—unfold the graph to a tree and encode the tree as a formula—

has the drawback of removing sharing. We therefore use a mapping where each internal node in the representation is allocated a fresh variable. This variable is used in place of the subformula that corresponds to the internal node. The generated formula is the conjunction of all the definitions of internal nodes and the literal that defines the top edge.

*Example 2.* The right-hand RBC in Figure 3 is mapped to the following formula in which the $i_k$ variables define internal RBC nodes:

$$\begin{aligned} & (i_0 \leftrightarrow \neg i_1 \wedge \ i_2) \\ \wedge \ & (i_1 \leftrightarrow i_3 \leftrightarrow i_4) \\ \wedge \ & (i_2 \leftrightarrow i_3 \wedge i_4) \\ \wedge \ & (i_3 \leftrightarrow x \wedge z) \\ \wedge \ & (i_4 \leftrightarrow z \wedge y) \\ \wedge \ & \neg i_0 \end{aligned}$$

A formula resulting from the outlined translation is *not* equivalent to the original formula without sharing, but it will be satisfiable if and only if the original formula is satisfiable. Models for the original formula are obtained by discarding the values of internal variables.

## 7 Experimental Results

We have implemented a tool FixIt [Eén99] for performing symbolic reachability analysis based on the ideas presented in this paper. The tool has a *fixpoint mode* in which it can perform both forward and backward reachability analysis, and an *unroll mode* where it searches for counterexamples in a similar manner to the BMC procedure. We have carried out preliminary experiments on three benchmarks: a *multiplier* and a *barrel shifter* (both from the BMC distribution), and a *swapper* (defined by the authors). The first two benchmarks are known to be hard for BDD-based methods.

In all the experiments, Prover outperforms Sato, so we only present measurements made using Prover. Furthermore, we only present time consumption. Memory consumption is much smaller than for BDD-based systems. Garbage collection has not yet been implemented in FixIt, but the amount of simultaneously referenced memory peaks at about 5-6 MB in our experiments. We also know that the memory requirements of Prover are relatively low (worst case quadratic in the formula size). The test results for FixIt are compared with results obtained from VIS release 1.3, BMC version 1.0f and Cadence SMV release 09-01-99.

**The Multiplier.** The example models a standard 16×16 bit shift-and-add multiplier, with an output result of 32 bits. Each output bit is individually verified against the C6288 combinational multiplier of the ISCAS'85 benchmarks by

**Table 1.** Experimental results for the multiplier

| Bit | FixIt Fwd sec | FixIt Bwd sec | FixIt Unroll sec | BMC sec | VIS sec | SMV sec |
|---|---|---|---|---|---|---|
| 0 | 0.8 | 2.0 | 0.7 | 1.0 | 5.3 | 41.4 |
| 1 | 0.9 | 2.3 | 0.7 | 1.4 | 5.4 | 41.3 |
| 2 | 1.1 | 3.0 | 0.8 | 2.0 | 5.3 | 42.5 |
| 3 | 1.8 | 3.9 | 0.9 | 4.0 | 5.5 | 42.6 |
| 4 | 3.0 | 6.1 | 1.2 | 8.2 | 6.2 | [>450 MB] |
| 5 | 7.2 | 9.9 | 1.8 | 19.9 | 10.2 | – |
| 6 | 24.3 | 21.5 | 3.8 | 66.7 | 32.9 | – |
| 7 | 100.0 | 61.9 | 11.8 | 304.6 | 153.5 | – |
| 8 | 492.8 | 224.7 | 45.2 | 1733.7 | [>450 MB] | – |
| 9 | 2350.6 | 862.6 | 197.8 | 9970.8 | – | – |
| 10 | 11927.5 | 3271.0 | 862.8 | 54096.8 | – | – |
| 11 | 60824.6 | 13494.3 | 3838.0 | – | – | – |
| 12 | – | 50000.0 | 16425.8 | – | – | – |

checking that we cannot reach a state where the computation of the shift-and-add multiplier is completed, but where the selected result bit is not consistent with the corresponding output bit of the combinational circuit.

Table 1 presents the results for the multiplier. The SAT-based methods outperform both VIS and SMV. The unroll mode is a constant factor more efficient than the fixpoint mode. However, we were unable to prove the diameter of the system by the diameter formula generated by BMC, which means that the verification performed by the unroll method (and BMC) should be considered partial.

**The Barrel Shifter.** The barrel shifter rotates the contents of a register file $R$ with one position in each step. The system also contains a fixed register file $R_0$, related to $R$ in the following way: if two registers from $R$ and $R_0$ have the same contents, then their neighbours also have the same contents. We constrain the initial states to have this property, and the objective is to prove that it holds throughout the reachable part of the state space. The width of the registers is $\log |R|$ bits, and we let the BMC tool prove that the diameter of the circuit is $|R|$.

Table 2 presents the results for the barrel shifter. No results are presented for VIS due to difficulties in describing the extra constraint on the initial state in the VIS input format.

The backward reachability mode of FixIt outperforms SMV and BMC on this example. The reason for this is that the set of bad states is closed under the pre-image function, and hence FixIt terminates after only one iteration. SMV is unable to build the BDDs characterising the circuits for larger problem instances. The BMC tool has to unfold the system all the way up to the diameter, producing very large formulas; in fact, the version of BMC that we used could

**Table 2.** Experimental results for the barrel shifter

| $\|R\|$ | FixIt Fwd sec | FixIt Bwd sec | FixIt Unroll sec | BMC sec | Diam sec | SMV sec |
|---|---|---|---|---|---|---|
| 2 | 1.7 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 |
| 3 | 2.3 | 0.1 | 0.1 | 0.0 | 0.0 | 0.1 |
| 4 | 3.0 | 0.1 | 0.2 | 0.0 | 0.0 | 0.1 |
| 5 | 42.4 | 0.2 | 0.3 | 0.1 | 0.1 | 44.2 |
| 6 | 848.9 | 0.2 | 0.5 | 0.3 | 0.1 | [>450 MB] |
| 7 | 5506.6 | 0.4 | 0.5 | 0.4 | 0.2 | – |
| 8 | [>3 h] | 0.5 | 1.0 | 1.2 | 0.3 | – |
| 9 | – | 0.8 | 1.6 | 2.4 | 0.6 | – |
| 10 | – | 1.1 | 2.3 | 8.6 | 0.8 | – |
| 11 | – | 1.5 | 2.3 | 3.3 | 1.1 | – |
| 12 | – | 2.3 | 4.1 | 25.6 | 1.5 | – |
| 13 | – | 2.6 | 3.9 | 7.1 | 2.0 | – |
| 14 | – | 3.2 | 7.8 | 80.1 | 2.6 | – |
| 15 | – | 3.7 | 8.6 | 75.1 | 3.5 | – |
| 16 | – | 4.3 | 12.1 | 150.0 | 4.4 | – |
| 17 | – | 6.7 | 11.0 | 34.6 | 7.9 | – |
| 18 | – | 8.7 | 30.5 | ? | ? | – |
| 19 | – | 9.2 | 15.6 | ? | ? | – |
| 20 | – | 13.5 | 49.1 | ? | ? | – |
| ... | | | | | | |
| 30 | – | 51.4 | 452.1 | ? | ? | – |
| ... | | | | | | |
| 40 | – | 230.5 | 2294.7 | ? | ? | – |
| ... | | | | | | |
| 50 | – | 501.5 | 8763.3 | ? | ? | – |

not generate formulas for larger instances than size 17 (a size 17 formula is 2.2 MB large). The oscillating timing data for the SAT-based tools reflects the heuristic nature of the underlying SAT-solver.

**The Swapper.** $N$ nodes, each capable of storing a single bit, are connected linearly:


At each clock-cycle (at most) one pair of adjacent nodes may swap their values. From this setting we ask whether the single final state in which exactly the first $\lfloor N/2 \rfloor$ nodes are set to 1 is reachable from the single initial state in which exactly the last $\lfloor N/2 \rfloor$ nodes are set to 1. Table 3 shows the result of verifying this property.

Both VIS and SMV handle the example easily. FixIt can handle sizes up to 14, but does not scale up as well as VIS and SMV, as the representations get too large. This illustrates the importance of maintaining a compact representation during deep reachability problems; something that is currently not done

**Table 3.** Experimental results for the swapper

| N | FixIt Fwd sec | FixIt Bwd sec | FixIt Unroll sec | BMC sec | VIS sec | SMV sec |
|---|---|---|---|---|---|---|
| 3 | 0.2 | 0.2 | 0.2 | 0.0 | 0.3 | 0.0 |
| 4 | 0.3 | 0.3 | 0.2 | 0.0 | 0.3 | 0.0 |
| 5 | 0.6 | 0.5 | 0.3 | 0.1 | 0.3 | 0.0 |
| 6 | 0.9 | 1.5 | 1.8 | 7.2 | 0.4 | 0.1 |
| 7 | 1.7 | 3.7 | 131.2 | 989.5 | 0.4 | 0.1 |
| 8 | 3.8 | 10.4 | [>2 h] | [>2 h] | 0.4 | 0.1 |
| 9 | 9.7 | 58.9 | – | – | 0.4 | 0.1 |
| 10 | 27.7 | 187.1 | – | – | 0.4 | 0.1 |
| 11 | 74.1 | 779.2 | – | – | 0.5 | 0.2 |
| 12 | 238.8 | 4643.2 | – | – | 0.6 | 0.2 |
| 13 | 726.8 | [>2 h] | – | – | 0.7 | 0.3 |
| 14 | 2685.7 | – | – | – | 0.7 | 0.4 |
| 15 | [>2 h] | – | – | – | 0.7 | 0.6 |
| ... | | | | | | |
| 20 | – | – | – | – | 1.6 | 7.9 |
| ... | | | | | | |
| 25 | – | – | – | – | 3.3 | 53.0 |
| ... | | | | | | |
| 30 | – | – | – | – | 15.1 | 263.0 |
| ... | | | | | | |
| 35 | – | – | – | – | 39.1 | 929.6 |
| ... | | | | | | |
| 40 | – | – | – | – | 89.9 | 2944.3 |

by FixIt. However, BMC does even worse, even though the problem is a strict search for an existing counterexample—something BMC is generally good at. This shows that fixpoint methods can be superior both for proving unreachability and detecting counterexamples for certain classes of systems.

## 8 Conclusions and Future Work

We have described an alternative approach to standard BDD-based symbolic model checking which we think can serve as a useful complement to existing techniques. We view our main contribution as showing that with relatively simple means it is possible to modify traditional algorithms for symbolic reachability analysis so that they work with SAT-procedures instead of BDDs. The resulting method gives surprisingly good results on some known hard problems.

SAT-solvers have several properties which make us believe that SAT-based model checking will become an interesting complement to BDD-based techniques. For example, in a proof system like Stålmarck's method, formula size does not play a decisive role in the hardness of satisfiability checking. This is particularly interesting since industrial applications often give rise to formulas which are extremely large in size, but not necessarily hard to prove.

There are several directions for future work. We are currently surveying simplification methods that can be used to maintain compact representations. One promising approach [AH97] is to improve the local reduction rules to span over multiple levels of the RBC graphs. We are also interested in exploiting the structure of big conjunctions and disjunctions, and in simplifying formulas using algorithms based on Stålmarck's notion of formula saturation [Bje99]. As for the representation itself, we are considering adding *if-then-else* and substitution nodes [HWA97]. Other ongoing work includes experiments with heuristics for choosing good quantification orderings.

In the longer term, we will continue to work on conversions of BDD-based algorithms. For example, we have already implemented a prototype model checker for general (fair) CTL formulas. Also, employing traditional BDD-based model checking techniques such as front simplification and approximate analysis are very likely to improve the efficiency of SAT-based model checking significantly.

Many important questions related to SAT-based model checking remain to be answered. For example, how should the user choose between bounded and fixpoint-based model checking? How can SAT-based approaches be combined with standard approaches to model checking?

## Acknowledgements

The implementation of FixIt was done as a Master's thesis at Prover Technology, Stockholm. Thanks to Purushothaman Iyer, Bengt Jonsson, Gordon Pace, Mary Sheeran and Gunnar Stålmarck for giving valuable feedback on earlier drafts.

This research was partially supported by TFR, the ASTEC competence center for advanced software technology, and the ARTES network for real-time research and graduate education in Sweden.

## References

AH97. H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *Proc.* $12^{th}$ *IEEE Int. Symp. on Logic in Computer Science*, pages 88–98, 1997. 413, 424

BCC+99. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, 1999. 412

BCCZ99. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS '98,* $8^{th}$ *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 1999. 412

BCMD92. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98:142–170, 1992. 411

BCRZ99. A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs. In *Proc.* $11^{th}$ *Int. Conf. on Computer Aided Verification*, 1999. 412

Bje99. P. Bjesse. Symbolic model checking with sets of states represented as formulas. Technical Report CS-1999-100, Department of Computer Science, Chalmers technical university, March 1999. 412, 424

Bor97. A. Borälv. The industrial success of verification tools based on Stålmarck's method. In *Proc.* $9^{th}$ *Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10, 1997. 412

Bor98. A. Borälv. Case study: Formal verification of a computerized railway interlocking. *Formal Aspects of Computing*, 10(4):338–360, 1998. 412

Bry86. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986. 411

CES86. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986. 411

Eén99. N. Eén. Symbolic reachability analysis based on SAT-solvers. Master's thesis, Dept. of Computer Systems, Uppsala university, 1999. 412, 420

GvVK95. J.F. Groote, S.F.M. van Vlijmen, and J.W.C. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *COMPASS'95*, 1995. 412

HWA97. H. Hulgaard, P.F. Williams, and H.R. Andersen. Combinational logic-level verification using boolean expression diagrams. In *3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1997. 413, 424

McM93. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. 411

Pap94. C. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994. 412

QS82. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *5th International Symposium on Programming, Turin*, volume 137 of *Lecture Notes in Computer Science*, pages 337–352. Springer Verlag, 1982. 411

SS90. G. Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In *SAFECOMP'90*, pages 31–36. Pergamon Press, 1990. 412

SS00. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's method of propositional proof. *Formal Methods In System Design*, 16(1), 2000. 412

Stå. G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467 076 (approved 1992), US patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995). 412

Zha97. H. Zhang. SATO: an efficient propositional prover. In *Proc. Int. Conference om Automated Deduction (CADE'97)*, volume 1249 of *LNAI*, pages 272–275. Springer Verlag, 1997. 412