Symbolic Reachability Analysis based on SAT Solvers

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

[1] Uppsala University and Prover Technology, Sweden,
`parosh@docs.uu.se`
[2] Chalmers University of Technology and Prover 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 which 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 boolean expressions, 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 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 the canonicity can in some cases lead to a space blowup, making memory a bottleneck in the application of BDD-based algorithms. There are well-known 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. Typically BDD-based methods can handle systems with hundreds of boolean variables.

A related approach is that of using satisfiability solvers, such as Stålmarck's method [Stå] and the Davis-Putnam procedure [Zha97]. These methods have already been used successfully for verifying industrial systems [Bor97b,SS90] [Bor97a,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 are occasionally able to handle propositional formulas with thousands of variables. Furthermore, typical 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 the 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 – forwards and backwards 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 formulas is quantification over propositional variables. Therefore, we provide a simple procedure for the removal of quantifiers which gives an 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 very simple techniques, our method can perform very 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. The BMC procedure searches for counterexamples by "unrolling" the transition relation k steps, for increasing values of k. At each step k, the unrolling characterizes the set of paths of length k through the transition relation, and is described as a formula (without quantifiers). If no counterexample is found, the search is terminated when the value of k is equal to the *diameter* of the system. However, the value of the diameter is usually hard to compute, making BMC incomplete in practice. Furthermore, for "deep" transition systems, formulas characterizing the set of reachable states may be much smaller than those characterizing the set of 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 too large formulas.

Our representation of formulas is closely related to *Binary Expression Diagrams (BEDs)* [AH97,HWA97]. In fact there are straightforward linear space translations from each of the representations to the other. Consequently, RBCs share the nice properties of BEDs, such as being exponentially more succinct than BDDs [AH97]. The main difference between our approach and that 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 operator vertices, or by a special transformation that rewrites the representation into a form where naive expansion can be applied. We use a similar algorithm which 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 application 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 model our systems as *(synchronous) circuits* which operate over the boolean domain. The operation of a circuit is controlled by a global clock. A circuit contains the following components:

- *Circuit inputs*: During each clock-cycle, a circuit input receives a random value which is either *true* or *false*.
- *Gates*: The output of a gate is a boolean combination of its inputs.
- *Latches*: A latch represents a memory unit with one input and one output. The value of the output during a clock-cycle is equal to the value of the input in the previous clock-cycle.

We use *Bool* to denote the domain of booleans. For a vector $s \in Bool^n$, we use s_k to denote the k^{th} element of s, for $k : 1 \leq k \leq n$. In the sequel, we assume a circuit with m circuit inputs indexed by $1, \ldots, m$, and n latches indexed by $1, \ldots, n$. The inputs of gates and latches, are connected to the inputs and to the outputs of other gates and latches. This implies that the output of a latch, in a given clock-cycle, is a boolean combination of circuit inputs and outputs of latches in the previous clock-cycle. In other words, for latch k, there is a function $f_k : \mathit{Bool}^m \times \mathit{Bool}^n \longrightarrow \mathit{Bool}$, where $f_k(i, s)$ describes the output of latch k during the current clock-cycle, in terms of i and s which are the values of circuit inputs and latches, respectively, in the previous clock-cycle.

We represent the behaviour of a circuit as a transition system (S, T), where $S = \mathit{Bool}^n$ is a finite set of states and $T \subseteq \mathit{Bool}^n \times \mathit{Bool}^n$ is a finite set of *transitions*. Intuitively, a state s maps each latch k to a boolean value s_k. The transition relation describes the computations performed during each clock-cycle, and is defined by $T(s, s') = \bigwedge_{k=1}^{n} (s'_k \leftrightarrow \exists i.\ f_k(i, s))$. The argument s' represents the "new state" of the circuit (after performing the transition), while i and s describe the values of circuit inputs, and the state of the circuit before performing the transition. The values of circuit inputs are existentially quantified, corresponding to the fact that they can have a random value which is either *true* or *false* during each clock-cycle.

Once the state variables under consideration have been fixed, any set of states can be represented by a propositional formula ϕ that is satisfied precisely by the states in the set.

3 Reachability Analysis

We work with two formulas $I(s)$ and $B(s)$, characterizing a set of *initial* and *bad* states, respectively. We define the reachability problem as that of checking whether the bad states are reachable from the initial states, or in other words to determine whether there is a sequence $s_0, s_1, \ldots, s_n$ of states, such that $s_0 \in I$, $s_n \in B$, and $(s_k, s_{k+1}) \in T$, for each $k : 1 \leq k < n$. There are two basic techniques for performing reachability analysis. Both perform fixpoint iterations generating a sequence $\Phi_0(s), \Phi_1(s), \Phi_2(s), \ldots, \Phi_n(s)$ of formulas.

In *forward* reachability analysis, we define $\Phi_0(s) = I(s)$, and $\Phi_{j+1}(s') = \exists s.\ T(s, s') \wedge \Phi_j(s)$. The formula $\Phi_j(s)$ characterizes the set of states which are reachable from an initial state by a path of length j. Notice that ϕ_j is a *Quantified Boolean Formula (QBF)*. We terminate if we reach a point n such that either (i) $\Phi_n(s) \wedge B(s)$ is satisfiable: this means that the set of reachable states contains a bad state; hence we answer the reachability problem positively; or (ii) if $\Phi_n(s) \implies \bigvee_{k=0}^{n-1} \Phi_k(s)$ holds: this implies that we have reached the fixpoint without encountering a bad state; consequently the answer to the reachability question is negative.

In *backward* reachability analysis, we define $\Phi_0(s) = B(s)$, and $\Phi_{j+1}(s) = \exists s'.\ T(s, s') \wedge \Phi_j(s')$. In a similar manner to forward reachability analysis, we terminate either if $\Phi_n(s) \wedge I(s)$ is satisfiable, or if $\Phi_n(s) \implies \bigvee_{k=0}^{n-1} \Phi_k(s)$.

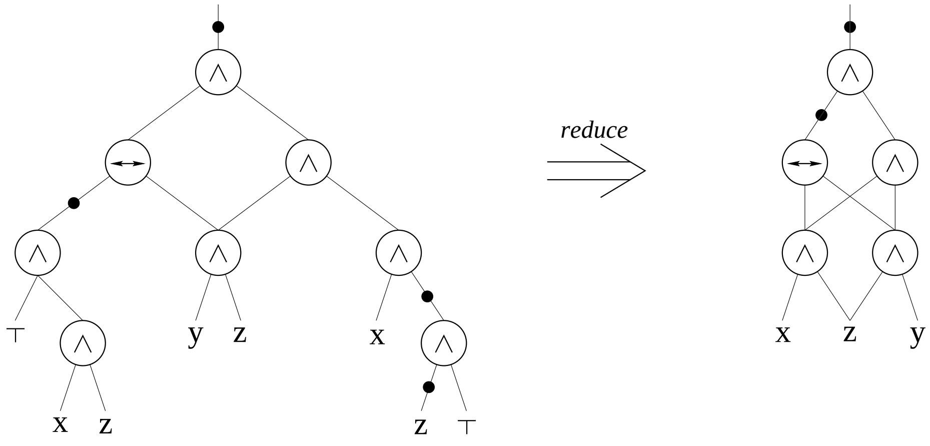

Fig. 1. A non-reduced *Boolean Circuit* and its reduced form.

4 Representation of Formulas

Let **Vars** denote the set of 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 $op(v) \in$ **Op**, and two edges $left(v), right(v) \in \mathbf{E}$.
- Each leaf $v \in \mathbf{V_L}$ has one attribute: $var(v) \in$ **Vars**.
- Each edge $e \in \mathbf{E}$ has two attributes: $sign(e) \in$ **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 1* shows an example of a BC.

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

1. All common subformulas are shared so that there are no two vertices with 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 distinct, $left(v) \neq right(v)$.

```
reduce(AND, left ∈ RBC, right ∈ RBC)          reduce(EQUIV, left ∈ RBC, right ∈ RBC)
    if   (left = right)   return left              if   (left = right)   return ⊤
    elif (left = ¬right)  return ⊥                 elif (left = ¬right)  return ⊥
    elif (left = ⊤)       return right             elif (left = ⊤)       return right
    elif (right = ⊤)      return left              elif (left = ⊥)       return ¬right
    elif (left = ⊥)       return ⊥                 elif (right = ⊤)      return left
    elif (right = ⊥)      return ⊥                 elif (right = ⊥)      return ¬left
    else                  return NIL               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))     – We store vertices
    if (result = NIL)
        result := insert(RBC_env, (op, left, right))
    return id(result, sign)
```

Fig. 2. Pseudo-code for creating a composite RBC from two existing RBCs.

4. If $op(v) = \leftrightarrow$ 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 $(\phi \wedge \psi) \iff (\psi \wedge \phi)$ (in 5 above) and $\neg(\phi \leftrightarrow \psi) \iff (\neg\phi \leftrightarrow \psi)$ (in 4 above).

The RBCs are created in an implicit environment, where all existing subformulas are tabulated. We use the environment to assure property (1). *Figure 2* 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 occuring in the formula ϕ. It should be noted that the above properties only takes a constant time to maintain in *mk_Comp*.

5 Quantification

In the reachability algorithm we use the notion of quantified boolean formulas (QBF). The quantifiers are introduced for expressing the next set of reachable states Φ_{j+1}, in terms of the current set of reachable states Φ_j. The quantified formulas are never stored, but immediately resolved into 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 3 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 whenever possible:

Inlining:

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

Scope Reduction:

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

When applicable, inlining is an effective method of resolving quantifiers as it immediately removes all occurrences of x. The applicability of the transformation relies on the fact that the formulas quantified over for reachability often have this particular structure. This is especially true for backward reachability as the transition relation is a conjunction of next state variables defined in terms of current state variables $\bigwedge_i s'_i \leftrightarrow \phi(s_0, \ldots, s_n)$ which yields formulas that fit the rule.

The first step of the inlining algorithm collects the toplevel conjunction of the RBC. This is necessary as a conjunction on the form $\bigwedge\{\phi_0, \phi_1 \ldots \phi_n\}$ must be encoded using binary AND-operators. If one of the collected conjuncts is a definition $(x \leftrightarrow \psi)$, we remove it from the conjunction and substitute x for ψ in the remaining conjuncts, then re-encode the resulting conjunction as an RBC.

If inlining is not applicable to the formula (and variable) at hand, the algorithm 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 even 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 subformulas processed.

```
– 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)   – Merges all binary ANDs at the top of φ into the
                                 "big" conceptual conjunction (returned as a set).
    ψ := findDef(C, x)         – Returns 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. 3. Pseudo-code for performing existential quantification over one variable. By ϕ_{left} we denote *left*(*target*(ϕ)) etc. We use $\wedge$, $\vee$ as abbreviations for calls to *mk_Comp*.

6 Satisfiability

Given an RBC we want to investigate whether there exists a satisfying assignment for the corresponding formula. We solve the problem by mapping the RBC back to a formula that is fed to an external SAT-solver. The naive translation, where the graph is unfolded to a tree which is linearised to 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 which is used in place of the corresponding subformula. The generated formula is the conjunction of the definitions of the internal nodes conjoined with the literal defining the top node.

Example. The right-hand RBC in *Figure 1* is mapped to the following formula in which the i_x 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 is obtained by discarding the values of the internal variables.

Remark. The use of SAT-solvers can also be applied to BEDs, although this seems not to have been explored so far. Instead the BEDs are translated into BDDs [AH97]. This is feasible only if the resulting BDD is of a manageable size. It would probably be useful to try both building a BDD and applying several SAT-solvers concurrently in order to decide satisfiability.

7 Experimental Results

Based on our method, we have implemented a tool FixIt [Eén99] for performing symbolic reachability analysis. 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 package. We have carried out preliminary experiments on three benchmarks: a *multiplier*, a *barrel shifter* (from the BMC package), and a *swapper* (defined by the authors). All the examples are parametrized by the size of the system. The first two benchmarks are known to be hard for BDD-based methods.

We present only 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 1-2 MB in our experiments. We also know that the memory requirements of Prover are relatively low (on the order of tens of megabytes for difficult formulas). In all the experiments, Prover outperforms Sato, so we only present the measurements for Prover. The test results for FixIt are compared against results obtained from VIS release 1.3, BMC version 1.0f and SMV version 2.5.3.1d.

Bit	FixIt Fwd sec	FixIt Bwd sec	FixIt Unroll sec	BMC sec	VIS sec
0	1.6	2.9	1.5	1.7	9.3
1	1.8	3.1	1.6	1.9	9.4
2	1.9	3.7	1.6	2.5	9.4
3	2.6	4.8	1.8	3.8	9.6
4	3.7	6.6	1.9	6.5	10.8
5	7.0	10.5	2.4	14.5	17.9
6	17.0	20.1	3.5	48.6	55.4
7	62.1	47.4	8.2	245.6	[>384 Mb]
8	277.9	150.3	27.1	1438.4	–
9	1423.6	544.31	111.3	8721.1	–
10	7878.9	2078.0	487.0	24926.3	–
11	18871.6	8134.1	2132.5	–	–
12	–	30330.0	10156.9	–	–

Table 1. Experimental results for the multiplier.

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 combinatorial multiplier of the ISCAS'85 benchmarks by 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 combinatorial circuit.

Table 1 presents the results for the multiplier. The SAT-based methods outperform VIS on all system sizes. 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 generated diameter formula, which means that unroll verification (and BMC) should be considered a partial result.

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 are $\log|R|$ 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 preimage function, and hence FixIt terminates after only one iteration. SMV is unable to build the BDDs characterising the circuits for larger problem instances.

Size	${}^{\text{FixIt}}$Fwd sec	${}^{\text{FixIt}}$Bwd sec	${}^{\text{FixIt}}$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	0.0
4	3.0	0.1	0.2	0.0	0.0	0.1
5	42.4	0.2	0.3	0.1	0.1	75.59
6	848.9	0.2	0.4	0.5	0.1	3348.4
7	[>3 h]	0.4	0.6	0.5	0.2	[>3 h]
8	–	0.5	0.8	2.2	0.3	–
9	–	0.8	1.3	4.3	0.6	–
10	–	1.1	2.1	16.8	0.8	–
11	–	1.5	2.1	5.4	1.1	–
12	–	2.3	3.9	52.4	1.5	–
13	–	2.6	3.6	11.7	2.0	–
14	–	3.2	7.6	168.2	2.6	–
15	–	3.7	8.3	154.4	3.5	–
16	–	4.3	12.6	310.0	4.4	–
17	–	6.7	10.4	58.5	7.9	–
18	–	8.7	14.5	?	?	–
19	–	9.2	21.0	?	?	–
20	–	13.5	50.5	?	?	–
...						
30	–	51.4	528.0	?	?	–
...						
40	–	230.5	2458.0	?	?	–
...						
50	–	501.5	8797.0	?	?	–

Table 2. Experimental results for the barrel shifter.

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 not generate formulas for instances larger than 17. The memory requirements for a size 17 formula is 2.2 MB, and for larger sizes we received segmentation faults.

The Swapper. A set of N nodes, capable of storing a single bit are connected linearly:

$$\boxed{1}-\boxed{2}-\boxed{3}-\boxed{4}- \cdots -\boxed{N}$$

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 where exactly the first $|n/2|$ nodes are set to 1 is reachable from the single initial state where exactly the last $|n/2|$ 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 15, but does not scale up as well as VIS and SMV, as the representations get too large. This stresses the importance of maintaining a compact representation during deep reachability problems; something that is currently not done by FixIt.

Size	FixIt Fwd sec	FixIt Bwd sec	FixIt Unroll sec	BMC sec	VIS sec	SMV sec
3	0.2	0.2	0.18	0.01	0.3	0.03
4	0.3	0.3	0.17	0.01	0.3	0.05
5	0.6	0.5	0.27	0.08	0.3	0.04
6	0.9	1.5	1.76	7.24	0.4	0.06
7	1.7	3.7	131.19	989.51	0.4	0.06
8	3.8	10.4	[>2 h]	[>2 h]	0.4	0.08
9	9.7	58.9	–	–	0.4	0.11
10	27.7	187.1	–	–	0.4	0.11
11	74.1	779.2	–	–	0.5	0.18
12	238.8	4643.2	–	–	0.6	0.23
13	726.8	–	–	–	0.7	0.30
14	2685.7	–	–	–	0.7	0.44
15	–	–	–	–	0.7	0.61
...						
20	–	–	–	–	1.6	7.88
...						
25	–	–	–	–	3.3	52.97
...						
30	–	–	–	–	15.1	263.08
...						
35	–	–	–	–	39.1	–
...						
40	–	–	–	–	89.9	–

Table 3. Experimental results for the swapper.

However, BMC does even worse, even though the problem is a strict search for counterexamples, something that BMC is generally good at. This suggests that for deep systems, fixpoint methods are superior to the bounded model checking approach.

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 in the long term 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. Although this article deals only with reachability analysis, a similar approach can be applied to convert any

BDD-based algorithm into a corresponding SAT-based algorithm. For example, we have already implemented a prototype of a model checker for general (fair) CTL formulas.

We are currently surveying simplification methods to maintain compact representations. One promising approach [AH97] is to improve the local reduction rules to span over multiple levels of the RBC graphs. Other methods include utilizing the structure of big conjunctions or disjunctions and simplifying formulas using algorithms based on Stålmarck's notion of formula saturation [Bje99].

Extensions to the representation by the trinary connective *if-then-else*, and by substitution nodes [HWA97] are also under consideration. We are experimenting with heuristics for choosing good quantification orderings.

It is also interesting to investigate whether standard BDD-based model checking techniques, such as front simplification and approximate analysis, can be carried over into our domain.

Acknowledgements

The implementation of FixIt [Eén99] was done as a Master's thesis at Prover Technology, Stockholm. Thanks to Purushothaman Iyer, Bengt Jonsson, Mary Sheeran and Gunnar Stålmarck for giving valuable feedback on earlier drafts.

References

[AH97] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams. In *Proc.* 12^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 88–98, 1997.

[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.

[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.

[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.

[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.

[Bje99] Per 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.

[Bor97a] Arne Borälv. A fully automated approach for proving safety properties in interlocking software using automatic theorem-proving. In *Proceedings of the Second International ERCIM Workshop on Formal Methods for Industrial Critical Systems*, pages 39–62, 1997.

[Bor97b] Arne 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.

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

[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.

[Eén99] Niklas Eén. Symbolic reachability analysis based on SAT-solvers. Master's thesis, Dept. of Computer Systems, Uppsala university, 1999.

[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.

[HWA97] Henrik Hulgaard, Poul Frederick Williams, and Henrik Reif Andersen. Combinational logic-level verification using boolean expression diagrams. In *3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1997.

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

[Pap94] Christos Papadimitriou. *Computatitional complexity.* Addison-Wesley, 1994.

[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.

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

[Stå] G. Stålmark. 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).

[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.