Increasing the Robustness of Bounded Model Checking by Computing Lower Bounds on the Reachable States

Mohammad Awedh and Fabio Somenzi

University of Colorado at Boulder
{awedh,fabio}@colorado.edu

Abstract. Most symbolic model checkers are based on either Binary Decision Diagrams (BDDs), which may grow exponentially large, or Satisfiability (SAT) solvers, whose time requirements rapidly increase with the sequential depth of the circuit. We investigate the integration of BDD-based methods with SAT to speed up the verification of safety properties of the form $\mathsf{G}\, f$, where f is either propositional or contains only the next-time temporal operator X. We use BDD-based reachability analysis to find lower bounds on the reachable states and the states that reach the bad states. Then, we use these lower bounds to shorten the counterexample or reduce the depth of the induction step (termination depth). We present experimental results that compare our method to a pure BDD-based method and a pure SAT-based method. Our method can prove properties that are hard for both the BDD-based and the SAT-based methods.

1 Introduction

BDD-based symbolic model checking [11] is widely used in formal verification of hardware and software systems. Recently, however, this technique has been challenged by the use of propositional satisfiability (SAT) with the introduction of Bounded Model Checking (BMC) [2]. BMC has been able to refute LTL formulae for models that have proved too hard for BDD-based methods. The disadvantages of this technique are that it is not complete in practice because a tight bound of the maximum length of a counterexample is often not available; and that its time requirements rapidly increase as the depth of the search increases.

The issue of completeness is addressed in [16] and [8], which add an induction proof to BMC so that both verification and falsification of safety properties becomes possible. In [12], the same problem is solved by the use of interpolants. The induction proof of [16] concludes that an invariant holds if all states of all paths of length k starting from the initial states satisfy the invariant, and, moreover, there is no simple path of length $k + 1$ starting at an initial state or leading to a state that violates the property. The disadvantage of this method is that the number and sizes of the SAT instances increase. In addition, the induction proof depends on the longest simple paths between two states, which may be much longer than the shortest paths between them.

In a simple path each state differs form all the others. This condition can be easily expressed with a number of clauses that is quadratic in the length of the path k. Recent work [10] reduces the number of additional clauses to $O(k \log^2 k)$: A bitonic sorting network is used to obtain an ordered permutation of the states in the path. A path

contains two equal states if and only if its corresponding ordered (sorted) permutation contains two equal adjacent states.

The interpolation method [12] uses the refutation proofs generated by the SAT solver to compute an *interpolant*. An interpolant can be used to compute an over-approximation of the reachable states.

Reachability analysis computes a set of states that are reachable from a given set of states; it may prove that an invariant holds at all states that are reachable from the initial states (forward reachability); or it may prove that no initial state has a path to a state that violates the invariant (backward reachability). The success of using reachability analysis to check invariants owes much to the use of the canonical Binary Decision Diagrams (BDDs) [4] to represent Boolean functions. However, for large systems, BDDs may grow too large. Several techniques have been proposed to overcome this limitation. Among them, High Density reachability analysis [15] attempts to reduce the size of the BDD representation of the reached states, while reaching as many states as possible. At each iteration, a subset of the newly reached states is chosen such that its BDD represents many states with few nodes. Hence, the high-density approach computes an under-approximation of the reachable states.

A combination of over-approximated forward reachability and exact backward reachability is described in [5]; however, this technique is limited by the exact backward traversal step which may become very expensive.

Target enlargement is used in [1] to prove reachability. The authors use SAT-based bounded model checking to check for the reachability of the current enlarged target. If reachability can not be proved, then they perform composition-based pre-image computations until the size of the BDD that represents the enlarged target exceeds a given limit. If the property is not proved, a simpler enlarged target is constructed and added to the netlist of the model for a subsequent verification flow. The enlargement of the target states is bounded by an over-approximation of the diameter of the circuit that can be computed by analyzing the netlist representing the model. This requires a model to have special structure; hence, this bound is not always useful.

In this paper, we present a new technique for symbolic model checking that integrates BDD-based reachability analysis and BMC augmented with induction. We use SAT to check for the existence of a counterexample of length k or to find the termination depth, which is bounded by the recurrence diameter [2]. The path of length k does not necessarily connect initial states to target states as in traditional BMC; instead, BDD-based reachability analysis reduces the gap between the initial states and the target states. Since for large systems, reachability analysis is very expensive, we aggressively under-approximate.

We compute a subset of the states that are reachable from the initial states; this subset includes all the initial states. Then, we compute all the successors of the states in this subset. Similarly, we compute a subset of the states from which the target states can be reached; this subset includes all the target states. Then, we compute all the predecessors of the states in this subset. Finally, we look for a path connecting these two subsets of states or try to prove that no such path exists.

Our method differs form the method of [1] in that we control the size of the intermediate BDDs by under-approximation rather than applying early quantification of

primary input variables. Our approach has the advantage that it goes deeper in the reachable states, and hence has more chances to reduce the counterexample and the termination depth. Another important difference is that we use the SAT-based method after we compute the under-approximation of the enlarged target and initial states, whereas the method of [1] alternates between using SAT to check for the reachability of the enlarged target states and BDDs to compute the enlarged target states.

BDD-based method and SAT-based method have been combined in the context of abstraction refinement [7, 13]. In [7, 13], the SAT-based method is used to check whether a counterexample found in the abstract model is a true counterexample in the concrete model. If the SAT instance is unsatisfiable, [13] uses the proofs of unsatisfiability derived from the SAT-based method as a guide to refine the abstract model.

Integrating BDD-based methods with SAT-based methods helps one to check properties that are hard for both approaches. Specifically, our method simplifies the SAT instances, while using BDDs only for manageable subsets of the reachable states.

We present an experimental comparison of our new method with the methods implemented in VIS [3]. For invariants, we compare our method to the BDD-based invariant checking in VIS, and to VIS's BMC. For safety properties that contain only the next-time temporal operator X, we compare our method to the BDD-based LTL model checking in VIS and to VIS's BMC.

In Section 2, we review background material. In Section 3, we present our approach, while in Section 4 we explain how to generate a counterexample for failing properties. We present experimental results In Section 5 and conclude in Section 6.

2 Background

We model the systems to be verified as *Kripke structures* and we specify properties in Linear-time Temporal Logic (LTL).

Definition 1. *A* Kripke structure M *over a set of atomic propositions* AP *is a 4-tuple* $M = \langle S, I, \Delta, L\rangle$ *where:*

- S *is a finite set of states.*
- $I \subseteq S$ *is a set of initial states.*
- $\Delta \subseteq S \times S$ *is a transition relation that is total: For every state* $s \in S$ *there is a state* $t \in S$ *such that* $\Delta(s, t)$.
- $L : S \to 2^{AP}$ *is a function that labels each state with a set of atomic propositions that are true in that state. For* $p \in AP$*, we write the predicate* $p(s)$ *to indicate* $p \in L(s)$.

Each state is a valuation of the *state variables*. Current state $s \in S$ is defined over the state variables $V_s = \{s_1, \ldots, s_n\}$. Next state $t \in S$ is defined over the state variables $V_t = \{t_1, \ldots, t_n\}$. The state at time i is written s^i.

Definition 2. *A path* π *in Kripke structure* M *is an infinite sequence of states* $s^0, s^1, \ldots$ *such that for any two consecutive states* s^i *and* s^{i+1} *in* π $\Delta(s^i, s^{i+1})$ *holds. We define* $\pi^i = (s^i, s^{i+1}, \ldots)$.

Definition 3. *An* LTL formula *is defined as follows*

- *An atomic proposition* $p \in AP$*,* true*, and* false *are LTL formulae.*
- *If* f *and* g *are LTL formulae, then so are* $\neg f$*,* $f \wedge g$*,* $f \vee g$*,* $\mathsf{X}\, f$*, and* $f \,\mathsf{U}\, g$*.*

A propositional formula *is an LTL formula that does not contain the temporal operators (*X *and* U*). We define* $f \,\mathsf{R}\, g = \neg(\neg f \,\mathsf{U}\, \neg g)$*,* $\mathsf{F}\, f = \mathsf{true} \,\mathsf{U}\, f$*, and* $\mathsf{G}\, f = \mathsf{false} \,\mathsf{R}\, g$*.*

The semantic of LTL formulae are defined over infinite paths. An atomic proposition p holds along a path π if $p(s^0)$ holds. Satisfaction for true, false, and the Boolean connectives is defined in the obvious way; $\pi \models \mathsf{X}\, f$ iff $\pi^1 \models f$; and $\pi \models f \,\mathsf{U}\, g$ iff there exists $i \geq 0$ such that $\pi^i \models g$, and for $j < i$, $\pi^j \models f$.

A *safety* property states that something bad will never happen. Sistla [17] provides a syntactic characterization of LTL safety formulae: Every propositional formula is a safety formula, and if f and g are safety formulae, then so are $f \vee g$, $f \wedge g$, $\mathsf{X}\, f$, $\mathsf{G}\, f$, and $f \,\mathsf{R}\, g$. Not all safety properties are captured by this definition. The violation of safety property $\mathsf{G}\, f$ is witnessed by a path starting from an initial state and leading to a state that violates f. We call the set of states that violate f *target states*, and we denote them by T.

Our method works on LTL safety properties of the form $\mathsf{G}\, f$, where f is either a propositional formula, or a path formula that contains only the temporal operator X (e.g., $\mathsf{G}(p \rightarrow \mathsf{X}\, q)$). For these properties, we use BDD operations to find the states satisfying $\neg f$. Since no fixpoint computations are required, these BDD operations seldom result in unwieldy BDDs[1].

Image computation is the process of computing the successors $P(t)$ of a set of states $Q(s)$ in the state transition graph described by $\Delta(s,t)$, and is defined by:

$$P(t) = \exists s \,.\, Q(s) \wedge \Delta(s,t) \;\; . \tag{1}$$

Pre-image computation is the process of computing the predecessors $P(s)$ of a set of states $Q(t)$ in the state transition graph described by $\Delta(s,t)$, and is defined by:

$$P(s) = \exists t \,.\, Q(t) \wedge \Delta(s,t) \;\; . \tag{2}$$

A set of states $\{s^0, s^1, \ldots, s^n\}$ forms a *path* of length n if it satisfies:

$$path_n = \bigwedge_{0 \leq i < n} \Delta(s^i, s^{i+1}) \;\; .$$

A *simple path* is a *path* along which all states are unique. It satisfies:

$$simplePath_n = path_n \wedge \bigwedge_{0 \leq i < j \leq n} (s^i \neq s^j) \;\; .$$

Invariants are safety properties of the form $\mathsf{G}\, f$, where f is a propositional formula. If f holds in all reachable states, the invariant is called an inductive invariant. Reachability analysis can be used to check the validity of invariants. It may compute all the

[1] We explain in Sect. 6 how to generalize our method to verify all safety properties.

states reachable from the initial states and prove that f holds at all of them (forward reachability); or, it may compute all the states from which a state that violates f may be reached, and prove that no initial state is included in them (backward reachability). The success of this technique often depends on the use of the canonical Binary Decision Diagrams (BDD) [4] for representing Boolean functions. However, in some cases, BDDs grow too large and make reachability analysis too expensive. High Density reachability analysis [15] was introduced to contain the size of BDDs.

High Density reachability analysis mixes breadth-first and depth-first searches in computing the reachable states. A BDD is used to represent the reachable states, but its size is controlled by dropping states that decrease its density. The density of a BDD is the ratio of the number of states it represents to the number of its nodes. By controlling the density of the BDD that represents the reachable state, one obtains a BDD that represents an under-approximation of the reachable states.

Boolean Satisfiability (SAT) is a well-known NP-complete problem. It consists of determining a satisfying variable assignment of a given propositional formula or determining that no such assignment exists. Many SAT solvers assume that the propositional formula is represented in conjunctive normal form (CNF). A CNF formula is a conjunction of clauses. Each clause is a disjunction of literals. A literal is a variable or its complement.

Bounded Model Checking (BMC) [2] is a SAT-based model checking approach for linear time properties. BMC reduces the search for a counterexample to propositional satisfiability. Given a model M, an LTL formula φ, and a bound k, BMC tries to falsify $M \models \varphi$ by proving the existence of a witness of length k for the negation of the LTL formula.

BMC generates a propositional formula $[\![M, \neg\varphi]\!]_k$ that is satisfiable if and only if a counterexample to φ of length k exists; $[\![M, \neg\varphi]\!]_k$ is defined as follows:

$$[\![M, \neg\varphi]\!]_k = [\![M]\!]_k \wedge [\![\neg\varphi]\!]_k \;\; , \tag{3}$$

where $[\![M]\!]_k$ is a propositional formula describing a path of length k that starts at the initial states:

$$[\![M]\!]_k = I \wedge path_k \;\; , \tag{4}$$

and $[\![\neg\varphi]\!]_k$ expresses the satisfaction of $\neg\varphi$ along that path. For a safety property $\varphi = \mathsf{G}\, f$, we are looking for a state that violates f. Hence, $[\![\neg\varphi]\!]_k$ is defined as:

$$[\![\neg\varphi]\!]_k = \bigvee_{0 \leq i \leq k} \neg f_i \;\; . \tag{5}$$

In [16], the authors use induction to verify invariants. An invariant $\mathsf{G}\, f$ holds if all states of every path of length k starting from an initial state satisfy f; and if f holds in all the states of some simple path of length k (not necessarily starting from the initial states), then it also holds in all the successors of the last state of the path. We refer to k as the *termination depth*.

SAT can be used to prove the induction in two steps. First, one calls the SAT solver on a propositional formula that describes an initialized path of length k to a target state. If the SAT solver returns *unsatisfiable*, then one calls the SAT solver on a propositional

formula that describes a simple path of length k to a target state, such that no other state along the path is a target state. If the SAT solver returns *unsatisfiable*, then the formula passes. Likewise, one can check for the existence of a simple path such that the first state, and no other state, is initial.

3 The Algorithm

We verify a safety property $\mathsf{G}\, f$, where f is either propositional or contains only the next-time temporal operator X, by attempting to prove or disprove the reachability of $\neg f$ from the initial states I. We integrate the BDD-based under-approximation reachability analysis with the SAT-based method in procedure bdd_sat. As illustrated in Fig. 1, first we compute the *outer boundary* of a subset of the states reachable from the initial states I by calling the function *ComputeReach(I, T, Fwd)*. If we cannot conclude, that is, if no target state has been reached, we find the outer boundary of a subset of the states that reach the target states T by calling the function *ComputeReach(T, I, Bwd)*. If we could not reach any conclusion because the results of the two reachability analyses do not intersect, then we invoke the SAT solver to find a path between the new sets of states, or to prove that such a path does not exist and the property holds. We call the newly computed subsets the *boundary states*. Figure 2 depicts our new technique.

```
bdd_sat(I, T) {
    (status, newI) = ComputeReach(I, T, Fwd);
    if (status == undecided ) {
        (status, newT) = ComputeReach(T, newI, Bwd);
        if (status == undecided ) {
            status = Check(newI, newT);
        }
    }
    return status;
}
```

Fig. 1. The bdd_sat algorithm.

Figure 4 shows the algorithm *ComputeReach(S, E, dir)* that returns either a forward or a backward under-approximation of the reachable states. In this algorithm, R at the i-th iteration of the *while* loop denotes the set of states proved reachable in i or fewer iterations. New_i denotes the set of new states at iteration i. We store New_i for later use in the generation of a counterexample in case the property fails. The frontier F denotes the set of states in the interval between the newly reached states New_i and the reached states R. We control the size of the BDD of the approximate reachable states by computing partial images of the frontier set, and we stop computing the reachable states when no new states are reached, when the size of the BDD of the approximate reachable

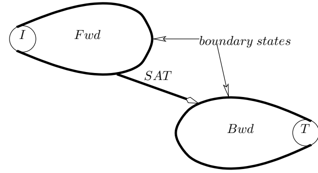

Fig. 2. bdd_sat technique

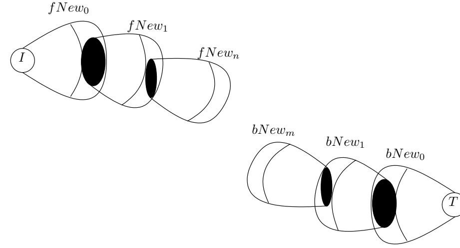

Fig. 3. Forward and Backward under-approximation reachability analysis

states exceeds a certain threshold, or when a counterexample is found, whichever comes first.

To compute approximate forward reachability, we call the function *ComputeReach(I, T, Fwd)*. Initially, the frontier F is equal to the initial states I. At each iteration, we compute a set of states that can be reached in one time frame by calling *computeImage(F, Fwd)*. The iteration is terminated if the set of newly reached states, New_i, is empty. Then, we check if New_i intersects the target states. If it does, the property fails otherwise, we compute the new frontier F by calling the BDD operation *BddBetween(*New_i*, R)*, where *BddBetween* picks a set of states between New_i and R such that its BDD size is not larger than the BDD size of either. The set of reachable states R is the collection of all new reachable states and the initial states I.

In order to deal with the possible explosion of BDD sizes, the new frontier F is calculated by computing an under-approximation of the BDD that represents the states in *BddBetween(*New_i*, R)*. If the size of the resulting BDD exceeds a certain limit (500 BDD nodes in our experiments), we compute a subset of the frontier by extracting a cube that is closest to the target states by calling *ComputeCloseCube*. This function extracts an implicant of F that is at minimum Hamming distance from T, thus selecting states expected to be closer to the target. Figure 3 illustrates the above algorithm graphically.

Finally, we compute the outer boundary of the under-approximation of the reachable states R (boundary states):

$$S_i = \textit{image}(R) - R \ .$$

S_i is the set of states that are reachable in one step from R but are not in R. Hence, we reduce the size of the BDD that represents the *boundary states*. In consequence we reduce the number of generated clauses when using the SAT-based method. Similarly, we use approximate backward reachability, *ComputeReach(T, I, Bwd)*, to compute the outer boundary S_t of the subset of the states that reach the target states.

If the set of boundary states S_i is empty, then R contains the complete reachable states. So, if R intersects T, then the property fails, otherwise the property passes. Similarly, if S_t is the empty set, a null intersection of R with $newI$ indicates that the property passes; otherwise the property fails. Finally, if S_i and S_t do not intersect, then we use

BMC to find a path between S_i and S_t, or use the induction proof to prove that there is no path between S_i and S_t and hence the property holds.

Both S_i and S_t are represented by BDDs. So, we need to generate a propositional formula in CNF from a BDD when using the SAT-based method. We follow the method that is presented in [6], which decomposes the BDD into sub-BDDs and introduces an auxiliary variable for each sub-BDD. The CNF is generated for each sub-BDD F by expressing $\neg F$ in disjunctive normal form (DNF), and generating a clause for each disjunct in the DNF.

```
ComputeReach(S, E, dir) {
    status = undecided;
    if(S ∩ E ≠ ∅) return(fail, ∅);
    R = New_0 = F = S;
    while (size(R) ≤ threshold_1 ) {
        To = computeImage(F, dir);
        New_i = To − R;
        if (New_i == ∅) break;
        if(New_i ∩ E ≠ ∅) return(fail, ∅);
        R = R ∪ New_i;
        F = BddBetween(New_i, R);
        F = BddUnderApprox(F);
        if(size(F) > threshold_2)
            F = ComputeCloseCube(F, T);
    }
    B = computeImage(R, dir) − R
    if(B == ∅) {
        B = R;
        if(R ∩ E ≠ ∅) status = fail;
        else status = pass;
    }
    return(status, B)
}
```

Fig. 4. Compute under-approximate reachable states R starting from set of states S and heading toward set of states E. Based on dir, $computeImage(R, dir)$ performs either forward image computation, or backward image computation

We use Fig. 5, which shows a fragment of a Kripke structure, to illustrate how our method reduces the termination depth. States $0, \ldots, 9$ are unreachable from the initial states (which are not shown) and state 9 is a bad state—one that violates f. The longest simple path reaching this state is of length 9. When we apply backward reachability for two steps, states 8, 3, and 7 are added to the target states. As a result, the length of the longest simple paths to a target state that does not go through another target state decreases to 3.

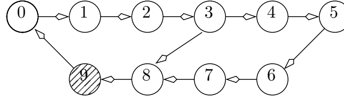

Fig. 5. Induction proofs can be sped up by reachability lower bounds

4 Counterexample Generation

If a property fails, we generate a counterexample to help debugging. This counterexample may not be a shortest counterexample due to the approximation of forward and backward reachability. In our method, we prove that a property fails in three ways: Using only forward approximate reachability analysis; using both forward and backward approximate reachability analyses; and using both reachability analyses and SAT.

If we initially set $S_t = T$, then we combine the above three cases into two situations. If the boundary states S_i and S_t do not overlap, we use the variable assignment returned by the SAT solver to construct a path which starts from state $s^n \in S_i$ and ends at state $s^m \in S_t$. If $s^n \notin I$, we generate a path from $s^i \in I$ to s^n. Similarly, we generate a path form s^m to $s^t \in T$ when $s^m \notin T$. If the boundary states S_i and S_t overlap, then $s^n \in S_i \cap S_t$ and $s^m = s^n$.

In summary, if there exists a counterexample in M between I and T, then this counterexample can be constructed from the following three paths, each of which may be empty:

- A path from $s^i \in I$ to $s^n \in S_i$ when $s^n \notin I$.
- A path from $s^m \in S_t$ to $s^t \in T$ when $s^m \notin T$.
- A path from s^n to s^m when $S_i \cap S_t = \emptyset$.

5 Experimental Results

We have implemented our new method in VIS [3, 18]. We implemented both the basic BMC algorithm of [2] and the induction proof as in [16]. We use zChaff [14] to check for the satisfiability of propositional formulae. The experiments were performed on an IBM IntelliStation with a 1.7 GHz Pentium IV CPU and 2 GB of RAM running Linux. The datasize limit was set to 1.5 GB.

The results presented in the following tables are for models that are either from industry or from the VIS Verification Benchmark set [19]. For each model, we count each safety property $\mathsf{G}\, f$, where f is either propositional or contains only the next-time temporal operator X, as a separate experiment. We exclude experiments such that all the methods that we compare finish in less than 60 seconds. For all experiments, we increase the value of the bound k by 1 every time, and we check for termination at each step.

In our experiments, the threshold value that we use to control the size of the BDD of the approximate reachable states is 10000. We have chosen this value so that all models of different sizes use our method efficiently. If the size of this threshold is too small,

Table 1. Comparison of bdd_sat, bmc, and ci on invariants

	state		bdd_sat					bmc			ci	
Model	vars	#	img	pre	st	k	t (s)	st	k	t (s)	st	t (s)
am2910	99	1	0	0	P	$I(0)$	**0.1**	P	$I(0)$ [1]	**0.1**		TO [2]
b12	115	1	21	0	F	2	**4.29**	F	14	137.49	F	5.7
b12abs	49	1	9	0	F	4	**4.04**	F	13	62.83	F	147.8
ball	85	1	0	0	P	$I(0)$	**0.07**	P	$I(0)$	0.1		TO
		2	8	5	U	20	89.93	U	20	**18.76**		TO
black jack	102	1	7	18			TO			TO		TO
		2	7	16	F	8	247.45	F	13	**19.13**		TO
bpb	36	1	25	0	F	-	**15.32**	F	9	193.45	F	40.00
D4	230	1	19	3	F	14	971.46	F	24	**23.12**	F	77.3
		2	19	3			TO	U	20	**90.7**	P	238.6
D14	96	1	9	2	F	9	208.49	F	14	**65.88**		TO
		2	0	0	P	$I(0)$	0.12	P	$I(0)$	**0.1**		TO
D16	531	1	5	3	F	-	**2.27**	F	8	8.38		TO
		2	0	0	P	$I(0)$	0.5	P	$I(0)$	**0.4**		TO
D18	506	1	12	5	F	10	247.11	F	23	**83.78**	F	129.34
D24	238	1	8	0	F	-	**0.65**	F	9	2.29		TO
		2	15	10	P	$I(6)$	58.19	P	$I(10)$	**9.39**		TO
		3	11	2	P	-	**10.0**	U	20	75.55		TO
daio_receiver_b	53	1	11	0	U	20	373.67	U	20	9.08	F	**1.7**
dekker	6	1	13	0	P	-	0.01	P	18	303.03	P	**0.0**
IBM01	141	1	15	3	F	-	**0.73**	F	14	70.96	F	4.4
IBM02_1	157	1	37	4	U	20	1310.51	U	20	81.12	P	**4.9**
nd3	100	1	0	0	P	$I(0)$	**0.3**	P	$I(0)$	**0.3**		TO
palu	37	1	6	4	F	1	**575.22**			TO		TO
peterson	6	1	10	0	P	-	0.01	U	20	126.94	P	**0.0**
ppprod	140	1	7	0	P	-	**0.2**	P	$I(6)$	66.6	P	9.3
s1269b	37	1	0	0	P	$I(0)$	0.1	P	$I(0)$	**0.0**		TO
		2	14	1	P	$I(0)$	19.52	P	$I(1)$	**0.79**		TO
s1423	74	1	3	0	P	-	**0.0**	P	$I(4)$	0.06		TO
		2	13	4	P	-	**0.68**	U	20	19.36		TO
		3	10	9	P	$I(48)$	**1720.22**			TO		TO
		4	9	324	F	-	**5.98**	F	61	161.98		TO
		5	10	22	F	12	97.18	F	35	**9.52**		TO
soap	140	1	0	0	P	$I(0)$	**0.1**	P	$I(0)$	**0.1**	P	60.21
Tom_P3	254	1	8	3	F	-	**4.41**	F	16	397.45	F	10.3
Tom_P4	254	1	8	2	F	-	**0.83**	F	11	75.30	F	3.3
UsbPhy	87	1	0	0	P	AT[3]	**0.0**	P	AT	**0.0**	P	88.7
		2	6	6	F	28	777.46	F	36	88.57	F	**8.5**
viper	215	1	4	1	F	-	260.77	F	4	**20.66**	F	623.4
		2	4	2	F	0	312.84	F	5	**36.17**	F	1620.3
vsaR	66	1	86	2	P	-	4.43	P	I(2)	**1.09**		TO
		2	165	8	F	5	35.8	F	23	**33.13**		TO
		3	15	2	P	-	**2.52**	U	30	120.02		TO

[1] Inductive proof (termination depth)
[2] Time Out
[3] Always True

Table 2. Comparison of bdd_sat, bmc, and ltl on safety properties containing X

	state		bdd_sat					bmc			ltl	
Model	vars	#	img	pre	st	k	t (s)	st	k	t (s)	st	t (s)
am2910	99	1	0	1	P	AT	**6.8**	U	50	534.33		TO
		2	3	2	P	-	1535.82	U	50	**510.34**		TO
bpb	36	1	5	1	P	-	**0.0**	U	30	TO	P	4.1
		2	9	1	P	-	**0.05**	U	32	TO	P	93.2
fabric	85	1	26	2	P	$I(2)$	37.71	U	50	90.43	P	**1.5**
heap	22	1	1	2	P	-	**0.1**	U	50	131.29	P	0.3
		2	0	1	P	AT	**0.1**	U	50	133.09	P	0.3
3proc	48	1	0	1	P	AT	**0.0**	U	50	213.36	P	8.3
		2	50	2	F	-	**0.16**	F	14	7.37	F	1124.2
PPC60X_bus	47	1	1	1	F	-	**0.51**	F	4	1.11	F	1.0
		2	1	1	F	-	**0.4**	F	3	0.63	F	7.0
		3	0	1	P	AT	6.3	U	50	215.54	P	**1.1**
		4	2	1	F	-	**0.0**	F	5	1.6	F	2.1
viper	215	1	2	2	P	-	**25.31**	U	42	TO		TO

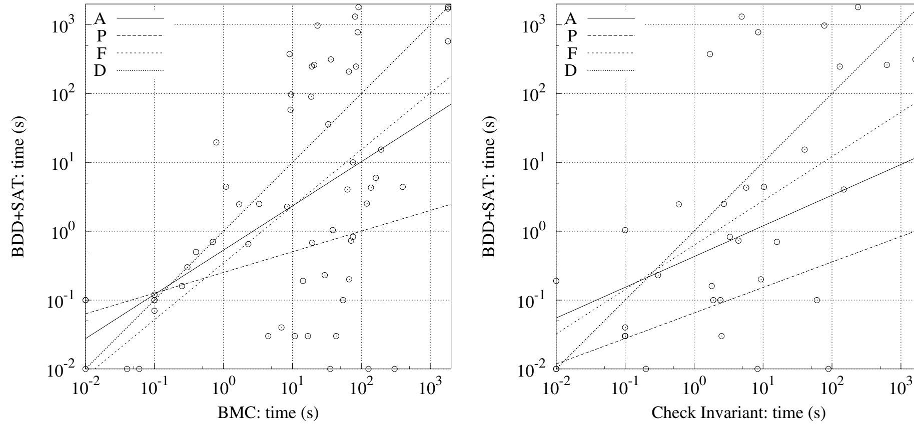

Fig. 6. Run times of bdd_sat against other methods

larger models will not benefit from using approximate reachability. However, a larger value of this threshold causes a large number of clauses to be generated when applying the SAT-based method, and hence a slower model checking.

Table 1 shows the results of using our method for the invariants $\mathsf{G}\, f$, where f is a propositional formula. We compare the performance of our method *bdd_sat* to the pure SAT-based method *bmc* and the pure BDD-based method *check invariant*, both in VIS. The first column in this table is the name of the model, the second is the number of state variables, and the third, labeled $\#$, is the property number. The remaining columns are divided into three groups: *bdd_sat*, *bmc*, and *ci*, respectively. The column labeled st in each group indicates whether each property passes (*P*), fails (*F*), or remains undecided (*U*); if a property fails, the number in the column labeled k is the length of the counterexample. The columns labeled t give the times in second for each method; a *TO* in this column indicates a time greater than 1800 s. The columns labeled *img* and *pre* give the number of image and pre-image computations respectively.

The column labeled k in each group may provide additional information. If the entry in this column is a dash, it indicates that the *bdd_sat* method proves the property without using the SAT-based part. A $I(n)$ in this column, indicates that the method proves that the property passes using the induction proof with termination depth of n. If n equals 0 the property is an *inductive invariant*.

Table 2 shows the results of using our method for safety properties $\mathsf{G}\, f$, where f is not propositional, but contains only the temporal operator X. It is organized similarly to Table 1, except that we use the pure BDD-based method *ltl*, which checks for language emptiness, instead of *check invariant*. Passing properties of this form cannot be proved by *bmc* because the induction proof is restricted to invariants in our implementation.

From both tables one sees that our new method could prove properties that are hard for both the BDD-based and the SAT-based methods. In Table 1, our method fails to decide 5 out of 43 properties, whereas *bmc* fails to decide 11 and *ci* fails to decide 24. In Table 2, *bdd_sat* decides all the 14 properties, whereas *bmc* fails to decide 10 of them and *ltl* fails to decide 3 of them. For example, in Table 1 *bdd_sat* refutes the property for the model *palu* in less than ten minutes, while *bmc* and *ci* time out. In addition, for the model *viper* in Table 2, *bdd_sat* proves the property true in 25.31 seconds, while *ltl* and *bmc* do not reach a decision in 1800s.

Our new method decreases the length of the paths to be found by SAT and reduces the termination depth. For model *b12* in Table 1, *bdd_sat* finds a counterexample in 2 steps, while *bmc* finds the counterexample in 14 steps. In model *D24* in Table 1, second property, *bdd_sat* proves the property with termination depth 6; *bmc* proves the same property with termination depth 10. For model *fabric* in Table 2, *bdd_sat* proves the property with termination depth 2; *bmc* can not reach a decision in 1800s.

For a given value of k, our method explores longer paths than *BMC*. So, our method has more chances to conclude for the same value of k. For the second property of the *ball* model in Table 1, *bdd_sat* computes 8 images and 5 preimages; hence, *bdd_sat* actually explores paths of length up to 33.

Our method can go much deeper than *BMC* within the same amount of memory. For the *IBM02_1* model, *bdd_sat* computes 37 images and 4 pre-images before calling the SAT solver. For the same example, if we run *bmc* for $k = 36$, zChaff runs out

of memory. The search of a larger fraction of the state space explains the longer time reported in Table 1 for *bdd_sat* in this experiment.

For inductive invariants, whose termination depth is 0, both *bdd_sat* and *bmc* perform better than *ci*. For instance, for model *nd3* in Table 1, *bdd_sat* and *bmc* prove the property in 0.3 seconds, while *ci* times out.

By only using the BDD-based part of our method, we have successfully proved properties that are hard for the pure BDD-based method. For the model *vsaR* in Table 1, we have proved the properties in a short amount of time while *ci* times out. Similarly, *ltl* times out for the model *viper* in Table 2.

Our method has a major advantage over *ci* because it combines forward and backward searches. For examples like the inductive invariants, backward reachability may be much more efficient than forward reachability. For model *s1269b* in Table 1, *bdd_sat* proves the second property with termination depth of 0, while *ci* times out. Backward reachability proves that all bad states are unreachable in one step, while approximate forward reachability does not take as much time as *ci*.

In addition, our method can decide that a formula passes or fails by checking if the formula is equivalent to either true or false before using either the BDD-based or the SAT-based method. The first property of model *UsbPhy* in Table 1 and *am2910* in Table 2 are both proved to pass by showing that the BDD of both properties is the constant one.

Varying the degree of approximation in *bdd_sat* obviously produces different results, and no specific setting works uniformly well for all examples. One should spend less time in reachability analysis, or even skip it altogether, for those models for which pure BMC is fast. Our implementation uses a fixed approach, which leads to mixed results. This is illustrated in Table 1 by the second property of model *blackjack* and by the only property of model *bpb*. Comparing to the pure SAT-based method, the BDD-based under-approximation method of *bdd_sat* significantly slows down verification for *blackjack*. In contrast, it significantly reduces the verification time of *bpb*. In both cases, *bdd_sat* is faster than *ci*. More importantly, aggressive approximation causes the fraction of time spent in reachability analysis to decrease as the size of the model increases. Hence, our mixed approach is more robust than one based exclusively on SAT.

Occasionally, *bdd_sat* fails to decide properties that *ci* successfully verifies. This is because SAT-based methods are not uniformly superior to BDD-based methods. Indeed, for all the experiments in which *bdd_sat* fails to decide the properties, *bmc* also fails.

In Fig. 6, we plot the performance of *bdd_sat* against the performance of *bmc* (left graph) and *check invariant* (right graph) methods. The vertical axes give the times in seconds taken by the *bdd_sat* method. The horizontal axes give the times in seconds taken by the other methods. Each point represents one model checking run. Points below the diagonal,labeled D, indicate a faster run time for the *bdd_sat* method. In the direct comparison over 59 experiments, our method performs better than *bmc* in 30 of them, and ties with *bmc* in 9 of them. Our method outperforms *ci* in 45 experiments, and ties with *ci* in 5 experiments, *ci* times out in 23 experiments.

We also plot $y = a \cdot x^b$ in Fig. 6, where a and b are obtained by least-square fitting of the experiment data. The line labeled A is for all properties, the line labeled P is for the passing properties, and the one labeled F is for the failing properties.

Using Student's t test, we can infer that the improvement by our method is statistically significant. However, the improvement for the passing properties is statistically more significant than the improvement of the failing properties.

6 Conclusions

We have presented a new symbolic model checking method that integrates SAT-based BMC with BDD-based approaches. Our method reduces the lengths of the paths to be examined by the SAT solver to find a counterexample or to prove termination. As a result, our new method can prove properties that are hard for both the BDD-based and the SAT-based methods. Since forward and backward under-approximations contain all the initial states and all the bad states, respectively, then given enough resources, our method is correct and complete.

We have tested our method on safety property $\mathsf{G}\, f$ (f is either a propositional formula or a path formula that contains only the temporal operator X). However, our method could be used to verify other safety properties. For instance, a counterexample to the safety property $\mathsf{G}(p \to q \,\mathsf{R}\, p)$ is an initialized path that starts at a state that satisfies $p \wedge \neg q$, goes through states that satisfy $\neg q$, and ends at a state that satisfies $\neg p$. The BDD-based methods can be used to enlarge the initial and target states, and the SAT-based method is used to look for a path between the two enlarged sets of states along which $\neg q$ is true.

In our method, we use BDDs to improve the SAT-based method. We may do the opposite by letting SAT help the BDD-based methods. One way to do so is to use the SAT solver to get an over-approximation of the reachable states. In [12], when the SAT solver finds the formula unsatisfiable, an *interpolant* is derived from a proof of unsatisfiability. This interpolant is an over-approximation of the reachable states. This information could be used to reduce the size of BDDs in the BDD-based method. In addition, we may analyze the proof of unsatisfiability produced by the SAT solver, to extract hints for BDD-based guided search, or variable orders for the BDDs,

Because the search for a counterexample and the induction proof are growing incrementally, our technique should benefit from the use of an *incremental* SAT solver [20, 9].

References

[1] J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 151–165. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
[5] G. Cabodi, P. Camurati, and S. Quer. Symbolic exploration of large circuits with enhanced forward/backward traversals. In *Proceedings of the Conference on European Design Automation*, pages 22–27, Grenoble, France, Sept. 1994.
[6] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based bounded model checking by means of BDD-based approximate traversal. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 898–905, Munich, Germany, Mar. 2003.
[7] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV 2002)*, pages 265–279. Springer-Verlag, July 2002. LNCS 2404.
[8] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W. A. Hunt, Jr. and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 1–13. Springer-Verlag, Boulder, CO, July 2003. LNCS 2725.
[9] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. First International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/.
[10] D. Kröning and O. Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking, and Abstract Interpretation*, pages 298–309, New York, NY, Jan. 2003. Springer. LNCS 2575.
[11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.
[12] K. L. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt, Jr. and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 1–13. Springer-Verlag, Berlin, July 2003. LNCS 2725.
[13] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17, Warsaw, Poland, Apr. 2003. LNCS 2619.
[14] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.
[15] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–158, San Jose, CA, Nov. 1995.
[16] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 108–125. Springer-Verlag, Nov. 2000. LNCS 1954.
[17] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects in Computing*, 6:495–511, 1994.
[18] URL: http://vlsi.colorado.edu/∼vis.
[19] Vis verification benchmarks. http://vlsi.colorado.edu/∼vis.
[20] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference*, pages 542–545, Las Vegas, NV, June 2001.