Automatic Invariant Strengthening to Prove Properties in Bounded Model Checking*

Mohammad Awedh Fabio Somenzi
University of Colorado at Boulder

Abstract

In this paper, we present a method that helps improve the performance of Bounded Model Checking by automatically strengthening invariants so that the termination proof may be obtained by analyzing shorter paths. The strengthening technique identifies sets of states as byproducts of the termination checks. It then uses SAT-based preimage computations to extend those sets. Our approach may substantially speed up the verification of both failing and passing properties. We present experimental results showing that our new method improves the performance of BMC significantly.

Categories and Subject Descriptors
B.6.3 [**Logic design**]: Design aids—*Verification*
General Terms: Verification, Algorithms
Keywords: Bounded Model Checking, SAT

1. Introduction

Bounded Model Checking (BMC, [5]) is a model checking approach, based on Boolean satisfiability (SAT), for linear time properties typically expressed in Linear Time Logic (LTL). In BMC, a propositional formula is constructed such that a counterexample of bounded length for the LTL formula exists if and only if the propositional formula is satisfiable.

The original Bounded Model Checking algorithm [5], although complete in theory, is limited in practice to falsification of LTL properties. BMC can prove that an LTL property ψ passes on a model $\mathcal{M}$ only if a bound, κ, is known such that, if no counterexample of length up to κ is found ($\mathcal{M} \not\models_\kappa \mathsf{E}\neg\psi$,) then $\mathcal{M} \models \mathsf{A}\psi$. Several methods exist to compute a suitable κ, all of them depending on $\mathcal{M}$, ψ, and the BMC encoding scheme. Some methods are straightforward, but usually poor. Finding the optimum value of κ, however, is at least as hard as checking whether $M \models \mathsf{A}\psi$ [8].

Practical methods to proving invariants with BMC therefore resort to conservative estimates of κ and can be interpreted as variants of the inductive proof method. An invariant is *inductive* if it holds in all initial states of the model and in all successors of states that satisfy the invariant. Invariants may be proved inductive by both fixpoint-based and BMC-like techniques. When computing fixpoints (typically with BDDs), an invariant is inductive if the backward reachability computation started from the states violating the invariant (the *bad states*) converges in one iteration without including any initial state.

*This work was supported by SRC contract 2005-TJ-920.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

With SAT solvers, the classical approach is followed: One first checks the satisfiability of the conjunction of the initial states and the bad states. If that conjunction is unsatisfiable, one then checks for satisfiability the transition relation of the model when the present state is constrained to satisfy the invariant and the next state is constrained to violate it. A negative result proves the inductive step.

Inductive invariants are among the easiest to prove, but, in many cases one has to deal with non inductive invariants. In that case one may use an *auxiliary invariant*, chosen in such a way that its conjunction with the given invariant makes it inductive. If one chooses the auxiliary invariant so as to leave out the unreachable good states with bad successors, one obtains a conjunction on which the inductive step will succeed.

One can leave it to the user of the model checker to supply the auxiliary invariant. Since it leverages human ingenuity, this approach is powerful, but also time-consuming. Other approaches are more automated and in fact, the termination criteria commonly employed by bounded model checkers can be seen as coming up with auxiliary invariants. The reachable states of the model provide the ultimate auxiliary invariant—one that is sufficient to prove any true invariant—but is also often prohibitively expensive.

The approach of [14] relies on the fact that if a counterexample to an invariant exists, then there is a simple path from an initial state to a failure state that goes through no other initial or failure state. An invariant holds if all states of all paths of length k starting from the initial states satisfy the invariant, and there is no simple path of length $k+1$ starting at an initial state or leading to a failure state, and not going through any other initial or failure states. Therefore, the invariant is strengthened by eliminating those states that have some bad states among their successors at distance k or less.

In this approach, the estimate of κ is given by the forward and backward recurrence radii of the model [5]; these radii are found by solving a sequence of propositional SAT instances rather than QBF SAT instances. Hence, they are easier to compute. However, the resulting bound is not as tight as the one based on the radii.

In [2], we use BDD-based analysis to find lower bounds on the reachable states, and use these bounds to strengthen the invariant.

The approach of [13] does not explicitly compute the backward radius, but it only examines counterexamples of length up to it to prove termination in invariant checking. For a given path length, the algorithm iterates over approximations of the states that are reachable from the initial states, R. The computed R is an auxiliary invariant, which, being a fixpoint of the reachability relation, guarantees that the inductive step will go through.

The technique of [1] combines the approach of [14] with the classic manual approach to invariant strengthening with the aim of limiting the length of paths that must be examined by the SAT solver. It therefore trades automation for scalability.

The objective of invariant strengthening is to remove unreachable states that impede the inductive step. This paper proposes a method that extends the one of [14] by obtaining sets of unreach-

able states as byproducts of failed attempts to prove termination and by extending those byproducts with SAT-based unbounded model checking techniques [12, 10, 11]. Knowing that a state is not reachable allows the bounded model checker to exclude it from any path it searches: Such a state cannot be part of a counterexample and should be avoided also in the simple paths that are built during the checks for termination. All unreachable states identified by our algorithm are known to be on a path to some bad state. The states that the algorithm tries to prove unreachable, but turn out to be reachable, are therefore also quite useful in practice, because they signal that the invariant fails. This early warning may come much sooner than the actual detection of the counterexample, and may save considerable computation resources.

2. Preliminaries

We define the model $\mathcal{M}$ under verification as a *Kripke structure*. A Kripke structure $\mathcal{M} = \langle S, \delta, I, L\rangle$ consists of a finite set of states S whose connections are described by the transition relation $\delta \subseteq S \times S$. If $(s,t) \in \delta$, then there is a transition from state s to state t in $\mathcal{M}$. The transition relation δ is total, i.e., for every state $s \in S$ there is a state $t \in S$ such that $(s,t) \in \delta$. $I \subseteq S$ is the set of initial states of the model. The labeling function $L : S \to 2^{AP}$ indicates what atomic propositions hold at each state. We write $\delta(s,t)$ for $(s,t) \in \delta$. Likewise, we write $I(s)$ to indicate that s is an initial state, and, for $p \in AP$, $p(s)$ to indicate that $p \in L(s)$.

A path π in $\mathcal{M}$ is a non-empty sequence $(s_0, s_1, \ldots)$ of states in $\mathcal{M}$ such that $\delta(s_i, s_{i+1})$ for all $0 \leq i < |\pi|$. We let $\pi[i] = s_i$ be the i-th state of π, and $\pi[i, \ldots, j] = [s_i, \ldots, s_j]$ be the segment of π from position i to position j, included. A *simple* path is a cycle free path. The *recurrence diameter* rd of $\mathcal{M}$ is the longest simple path in $\mathcal{M}$. The *recurrence radius* rr of $\mathcal{M}$ is the longest simple path in $\mathcal{M}$ that starts form a state in I.

Image computation finds 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 $P(t) = \exists s \,.\, Q(s) \wedge \delta(s,t)$. Pre-image computation finds the predecessors $P(s)$ of a set of states $Q(t)$ in the graph described by $\delta(s,t)$, and is defined by $P(s) = \exists t \,.\, Q(t) \wedge \delta(s,t)$.

A *safety* property is such that every counterexample to it has a finite prefix that, however extended to an infinite path, yields a counterexample. Though in principle a counterexample to a linear-time property is an infinite sequence of states, it is sufficient to present an initialized simple path that leads to a *bad state*—one from which all extensions to infinite paths result in counterexamples.

Invariants are safety properties of the form $\mathsf{G}\, p$, where p is a propositional formula. Reachability analysis can be used to check the validity of invariants. It may compute all the states reachable from the initial states and prove that p holds at all of them (forward reachability); or, it may compute all the states from which a state that violates p 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) [7] for representing Boolean functions.

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

Conventional SAT solvers can be augmented to get all satisfying assignments to a propositional formula. In principle, to get more than one satisfying assignment, it is enough to force the SAT solver to continue the search after getting each satisfying assignment [12]. Once the SAT solver finds a satisfying assignment a *blocking clause* is generated by taking its complement. The resulting clause prevents the SAT solver from returning the same assignment again. Various optimizations are possible to reduce the number of blocking clauses [12, 10, 11].

SAT-based pre-image computation [12, 11] enumerates the satisfying assignments to an *objective*. The objective represents the transition relation δ of $\mathcal{M}$. The returned blocking clauses represent predecessors of a given set of states. Iteration of this computation results in an algorithm for backward reachability analysis whose results are in clausal form. Therefore it is convenient when the sets of states are to be used as constraints for SAT solvers.

3. Invariant Strengthening Algorithm

We verify an invariant $\mathsf{G}\, p$ (i.e., always p), where p is a propositional formula, by attempting to prove or disprove the reachability of *bad states*, states that satisfy $\neg p$, from the initial states I. We use BMC [5] augmented with induction proof [14] to prove $\mathsf{G}\, p$. If there is no counterexample of length k to $\neg p$, we search for a simple path of length $k+1$ to $\neg p$ that is not going through any other bad states. If no such path exists, the invariant holds.

The basic idea of our algorithm is to learn about the reachability of some bad states while searching for simple paths, and use this information to expedite the next search for counterexamples or simple paths. The next two lemmas justify our approach.

Lemma 1 *Let $\mathcal{M} = \langle S, \delta, I, L\rangle$ be a model and let $s \in S$ be a state satisfying $\neg p$. If for some $k \geq 0$ (1) there is no counterexample to $\mathsf{G}\, p$ of length up to k, and (2) there is no simple path into state s of length k such that the only state along the path satisfying $\neg p$ is s, then no counterexample to $\mathsf{G}\, p$ has s as the first state violating the invariant.*

Lemma 2 *Let s be a state satisfying the conditions of Lemma 1. Then, if a state t has a path reaching s without any other state satisfying $\neg p$ besides s, then if t is on any shortest counterexample to $\mathsf{G}\, p$, it is preceded by one state violating the invariant.*

Fig. 1 illustrates our algorithm. If we fail to find a counterexample, ce_k, to a bad state, we search for a simple path, sp_{k+1}, to any bad state that is not going through any other bad states. If we find sp_{k+1}, we set q to this bad state, $sp_{k+1}[k+1]$, and we set C to all states in sp_{k+1}. We use the next search for simple path to prove that q is a *useless* state: either unreachable from I or can be reached from any other bad state. Before proving q is a useless state, we add to C all the states in sp_{k+1} excluding the last state $sp_{k+1}[k+1]$. If we prove that q is a useless state, we add to R all the states, including C, in the paths that reach C and do not go through any other bad states. R represents useless states in term of *Blocking* clause [12]. R is used in the future search for counterexample or simple path to prevent the SAT solver from trying the useless states.

We use the function $searchForCounterExample(\neg p \cup C, R, k)$ to search for a counterexample of length k. It searches for an initialized path of length k to a state in $(\neg p \cup C)$ that does not go through any state in R. It actually does two things to improve the search: First, it instructs the SAT solver not to go through any state in R—this may speed up the search because it prevents the SAT solver from trying to find a path through states that are known not to be on any shortest counterexample. Second, if a path to a state in C is found, the property is known to be violated. This corresponds to strengthening the invariant, while maintaining equisatisfiability. Detecting the existence of a counterexample while checking for a

```
checkInvariant(p){
    q = ∅; C = ∅; R = ∅; k = 0; flag = 0;
    while(true)
        ce_k = searchForCounterExample(¬p ∪ C, R, k);
        if (ce_k)
            if (ce_k[k] ∈ ¬p) return (fail, ce_k)
            else flag = 1;
        if (flag == 1) continue;
        if (q == ∅) t = ¬p;
        else t = q;
        sp_{k+1} = searchForSimplePath(p, t, R, k);
        if(sp_{k+1})
            if (q == ∅)
                q = sp_{k+1}[k + 1];
                C = {sp_{k+1}[0, ..., k + 1]};
            else C = C ∪ {sp_{k+1}[0, ..., k]};
        else
            if (q == ∅) return (pass)
            else
                R = R ∪ { E(¬(R ∪ p) U C)};
                q == ∅; C = ∅;
        k = k + 1;
}
```

Figure 1: BMC algorithm with invariant strengthening.

path that is up to a factor of two shorter may have significant benefits on the runtime because the cost of the successive checks often grows very fast with the length of the paths.

If a path ce_k is found, the existence of a counterexample to $\neg p$ is guaranteed. If the last state in ce_k is a bad state, then ce_k is already the shortest counterexample. Suppose it is not. In some cases the fact that the property fails may be all that is needed to know. However, if a counterexample is actually needed, there are two options. One is to concatenate a path from an initial state to the state in C that has been proved reachable with a path from that state in C to a bad state (which is guaranteed to exist). The other option is to continue the search for a shortest counterexample, but to disable the check for termination, which is usually rather time-consuming. This last option is the one implemented in the algorithm of Figure 1.

The function $searchForSimplePath(p, t, R, k)$ is used to search for a simple path of length k to t, sp_{k+1}, which goes through states in p and does not go through any states in R. The addition of the constraint R to the search strengthens the invariant and prevents the SAT solver from going through the states in R. If q is $\emptyset$, t is set to $\neg p$, else it is set to q. If no such simple path exists, sp_{k+1}, then the invariant is true if t is $\neg p$.

The evaluation of the CTL formula $\mathsf{E}(\neg(R \cup p)\,\mathsf{U}\,C)$ in Fig. 1 computes the states on a path leading to a state in C and entirely contained in $\neg(R \cup p)$ (except, possibly, for the last state). Because of Lemma 2, these states cannot be on any shortest counterexample.

The computation is carried out using SAT-based backward reachability [10, 11]. The result is a set of *blocking* clauses that are used to constrain further SAT searches. The computation may blow up in some cases, so we limit the number of computed clauses to a certain threshold. This limitation has no effect on correctness or on the completeness of the algorithm.

Theorem 1 *The algorithm of Figure 1 is sound and complete.*

In practice, in the implementation of the algorithm of Figure 1 we may impose limits on the length of the paths that are analyzed or on the computational resources. The resulting algorithm is no longer complete, but guarantees termination within certain bounds.

4. Experiments

We have implemented our method in VIS [6, 15]. We use CirCUs [9] as the SAT solver. The results presented in Table 1 are for models that are from industry, and from the Texas-97 and VIS Verification Benchmark sets [15]. The experiments were run 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 first column in Table 1 is the name of the model. The column labeled St indicates whether each property passes (*P*), fails (*F*), or remains undecided (*U*). If a property fails, then, the column labeled k indicates the length of the counterexample; k indicates the length of the induction depth if the property passes. If the property undecided, then k indicates the depth at which a model checking method reaches before it timed out.

The column labeled T in Table 1 gives the run times in seconds; boldface is used to highlight best run times; a TO in this column indicates a time greater than 1800 s. CPU times for all experiments are for both counterexample detection and termination checks.

Table 1 compares the CPU time of our new method *ais* to the standard BMC command in VIS bmc augmented with the induction proof. It also shows the CPU time of BDD-based model checking commands ci and mc in VIS. The ci command performs invariant checking based on forward reachability, while the mc performs invariant checking based on backward reachability.

Each entry in Table 1 consists of two groups. Group labeled bmc/ais shows the results of applying bmc in the first row and the results of applying ais in the second row. Group labeled ci/mc shows the results of applying ci in the first row and the results of applying mc in the second row.

In Table 1, *ais* is faster than *bmc* in proving 8 out of 9 passing properties (proving by both *ais* and *bmc*), and in 11 out of 12 failing properties. Both methods fail to decide 5 properties.

Our method helps to reduce the induction depth. In model *Vending2*, our method reduces the induction depth from 17 to 15. For the model *MPEG(product)*, our method proves the property passes in 27 steps, while *bmc* times out before reaching any conclusion. In addition, *bmc* times out before proving the property of the model *PPC60X_bus2*, while *ais* proves the property passes in 5 steps.

Our method increases our confidence of the absence of a counterexamle by going much deeper than *bmc* within the same time. In model *IBM(11_batch_3)*, *ais* continues the search to $k = 31$ while *bmc* times out at $k = 22$. In addition, *ais* reaches $k = 37$ for the model *Peterson* before it times out; *bmc* times out at $k = 32$.

Applying our method helps *bmc* to compete with the BDD-based methods. For the model *CacheCo(3P5) bmc* is slower than *ci*, but *ais* is faster than *ci*.

For failing properties, our method uses the *useless* states to expedite the search for a counterexample. For the model *CacheCo(3P5)*, the model check time is reduced from 449.96 to 73.95 seconds. We prove the existence of a counterexample only one step before finding the shortest one.

5. Conclusions

We have presented a new method that improves the performance of BMC. This method attempts to learn the reachability of states in the model while applying BMC for searching for counterexamples and simple paths. The reachability information is used to strengthen the invariant so as to allow inductive proof of termination with the analysis of shorter paths than in the method of [14].

Table 1: Comparison of AIS to other methods.

Model	*bmc/ais*			*ci/mc*	
	St	k	$T(s)$	St	$T(s)$
Blackjack(2)	F	13	219.41	F	496.7
	F	13	**212.34**	U	TO
CacheCo(2P1)	F	15	23.51	F	**0.7**
	F	15	5.86	F	166.6
CacheCo(3P1)	F	15	291.29	F	**1.5**
	F	15	64.89	U	TO
CacheCo(3P3)	F	18	593.84	F	**2.3**
	F	18	569.05	U	TO
CacheCo(3P4)	F	20	885.2	F	**8.8**
	F	20	152.35	U	TO
CacheCo(3P5)	F	15	449.96	F	80.34
	F	15	**73.95**	U	TO
D4(1)	F	24	891.26	U	TO
	F	24	**388.46**	U	TO
D4(2)	P	10	**13.49**	U	TO
	P	9	19.76	U	TO
Dcnew	F	5	17.95	F	**0.03**
	F	5	7.07	F	0.01
Heap	U	24	TO	P	1.9
	U	24	TO	P	**0.61**
IBM(4_batch)	F	24	981.87	U	TO
	F	24	**685.04**	U	TO
IBM(6_batch)	F	31	779.97	U	TO
	F	31	**604.23**	U	TO
IBM(11_batch_3)	U	22	TO	U	TO
	U	31	TO	U	TO
IBM(17_2_batch_2)	U	42	TO	U	TO
	U	49	TO	U	TO
MPEG(ppp1)	P	5	10.95	P	**4.0**
	P	4	6.82	P	13.7
MPEG(ppp2)	P	8	14.94	P	**6.2**
	P	6	14.12	P	18.3
MPEG(product)	U	35	TO	P	294.4
	P	27	734.45	P	**118.23**
Peterson	U	32	TO	P	0.05
	U	37	TO	P	**0.02**
PPC60X_bus1	U	24	TO	P	**1.4**
	P	21	187.99	P	4.64
PPC60X_bus2	U	18	TO	P	**1.21**
	P	5	10.38	P	4.90
ProdCell(19)	P	21	401.94	P	**1.2**
	P	21	285.43	p	10.2
ProdCell(23)	F	14	84.44	F	**0.8**
	F	14	37.44	F	1.21
Solitaire	F	12	216.79	F	**0.1**
	F	12	186.23	F	0.4
Swap4	U	10	TO	P	**0.01**
	U	13	TO	P	**0.01**
Vending1	P	12	65.23	P	0.9
	P	12	41.03	P	**0.1**
Vending2	P	17	98.18	P	0.8
	P	15	40.24	P	**0.08**

Our method can be seen as identifying *useless states* that can be used to constrain the search for counterexamples and simple paths. In addition, it may prove the existence of a counterexample very early so that one can stop the check for termination. Our experimental results confirm the advantage of the technique, which, although in the worst case has the same termination depth as the standard algorithm based on simple paths, often performs substantially better, for both passing and failing properties.

In this paper we have presented the application of the invariant strengthening technique to the verification of invariants. However, the principles may be applied to checking general linear-time temporal properties according to the approach of [3, 4]. We are currently investigating that extension.

References

[1] R. Armoni, L. Fix, R. Fraer, S. Huddleston, N. Piterman, and M. Vardi. SAT-based induction for temporal safety properties. In *BMC*, Boston, MA, July 2004.

[2] M. Awedh and F. Somenzi. Increasing the robustness of bounded model checking by computing lower bounds on the reachable states. In *FMCAD*, pages 230–244, Austin, TX, Nov. 2004. Springer. LNCS 3312.

[3] M. Awedh and F. Somenzi. Proving more properties with bounded model checking. In R. Alur and D. Peled, editors, *CAV*, pages 96–108. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[4] M. Awedh and F. Somenzi. Termination criteria for bounded model checking: Extensions and comparison. In *BMC*, Edimburgh, UK, July 2005. To appear in Electronic Notes in Theoretical Computer Science.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[6] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *CAV'96*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

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

[8] E. Clarke, D. Kröning, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *VMCAI*, pages 85–96, Venice, Italy, Jan. 2004. Springer. LNCS 2937.

[9] H. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In R. Alur and D. Peled, editors, *CAV*, pages 519–522. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[10] H. Jin, H. Han, and F. Somenzi. Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit. In *TACAS*, pages 287–300, Apr. 2005. LNCS 3440.

[11] H. Jin and F. Somenzi. Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In *DAC*, pages 750–753, Anaheim, CA, June 2005.

[12] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[13] K. L. McMillan. Interpolation and SAT-based model checking. In W. A. Hunt, Jr. and F. Somenzi, editors, *CAV*, pages 1–13. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[14] 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, *FMCAD*, pages 108–125. Springer-Verlag, Nov. 2000. LNCS 1954.

[15] URL: http://vlsi.colorado.edu/∼vis.