A Satisfiability-Based Approach to Abstraction Refinement in Model Checking [1]

Bing Li [2] Chao Wang [3] Fabio Somenzi [4]

University of Colorado at Boulder

Abstract

We present an abstraction refinement algorithm for model checking of safety properties that relies exclusively on a SAT solver for checking the abstract model, testing abstract counterexamples on the concrete model, and refinement. Model checking of the abstractions is based on bounded model checking extended with checks for the existence of simple paths that help in deciding passing properties. All minimum-length spurious counterexamples are eliminated in one refinement step by a procedure that combines the analysis of the conflict dependency graph produced by the SAT solver while looking for concrete counterexamples with an effective abstraction minimization procedure.

1 Introduction

Model checking [CGP99] is an algorithmic approach to the verification of properties of reactive systems, which has been successfully applied to both hardware and software. Since model checking entails the exploration of a potentially very large state space, the alleviation of the so-called state explosion problem has been the object of much research. On the one hand, techniques have been developed that allow models with hundreds of state variables to be analyzed directly. On the other hand, abstraction has been used to allow the model checker to draw conclusions on the original, concrete model by examining a simpler, abstract one.

For systems with many state variables and many transitions, the symbolic approach has proved crucial. In symbolic model checking, sets of states and transition are described by their characteristic functions. Various forms of representation have been used for these functions, the most popular being Binary Decision Diagrams (BDDs) [Bry86], and Conjunctive Normal Form (CNF).

[1] This work was supported in part by SRC contract 2001-TJ-920 and NSF grant CCR-99-71195.
[2] Email: `Bing.Li@Colorado.EDU`
[3] Email: `wangc@Colorado.EDU`
[4] Email: `Fabio.Li@Colorado.EDU`

Classical BDD-based model checking [McM94] is based on the computation of fixpoints. For instance, the reachable states of a model are computed as the least fixpoint of the function $\lambda Z \,.\, I \vee \mathrm{Succ}(Z)$, which adds the successors of the states in Z to the initial states. Both the set of states and the successor relation are stored as BDDs. The fixpoint computation converges in a number of iterations that equals the maximum distance of a reachable state from the initial states. Checking for convergence is made easy by the strong canonicity of BDDs (identical sets share the same representation). BDD-based model checking can therefore prove properties almost as easily as it can disprove them.

Bounded Model Checking (BMC) [BCCZ99], on the other hand, formulates the reachability test as a series of satisfiability (SAT) checks for paths of bounded length. (To see if a path of length k to a set of states exists, the transition relation is unrolled k times.) For finite systems the process must eventually terminate: the length of the shortest path between two states cannot exceed the number of states. Hence, if no path is found with length up to the number of states, the target states are known to be unreachable. This observation, however, does not help for the kind of models that one encounters in practice. The diameter of the state graph would give a much better bound on k, but, unfortunately, it is hard to compute [BCCZ99]. For this reason, BMC has come to be regarded as an excellent *debugging* (as opposed to *verification*) technique. That is, classical BMC is particularly adept at finding counterexamples, but ill-suited to prove their absence.

The ability demonstrated by BMC to deal with models beyond the reach of BDD-based methods has sparked interest in the use of CNF and SAT for proof as well as refutation. Two main approaches have been pursued: The replacement of BDDs with CNF formulae in the fixpoint computation [ABE00,WBCG00,McM02], and the development of more effective termination criteria for BMC.

The opportunity of replacing BDDs with CNF formulae can be argued on the grounds that canonicity of representation makes BDDs somewhat inflexible. Hence, some functions that admit compact representations in CNF have exceedingly large BDDs. However, the inflexibility argument can also be used against CNF, and memoization techniques are more effective for BDDs. In fact, to date, CNF-based fixpoint computation has not demonstrated a consistent advantage over the classical BDD-based one. One may argue that the main reason for the success of BMC in finding counterexamples lies in its avoidance of the needless computation and storage of reachable states that are not on the error trace.

Several proposals have been made to improve BMC's ability to prove the non-existence of a path. It is straightforward to check for inductive invariants, since it only entails checking for the existence of a transition from a state that satisfies the invariant to one that does not. An extension of the inductive approach has been presented in [SSS00], in which termination occurs as soon as the length of the path reaches the length of the longest simple path from an initial state, or to a target state. A recent paper [McM03] proposes the analysis of the unsatisfiable formulae to allow termination when the *reverse sequential depth* of the model is reached.

Early termination in BMC requires additional checks beyond the one for the

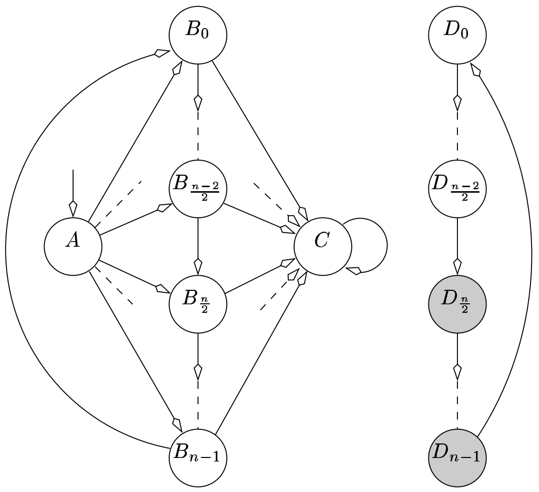

Fig. 1. Model with long simple path

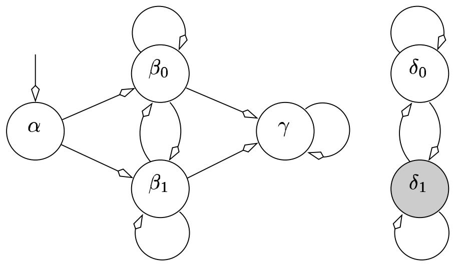

Fig. 2. Abstraction of the model of Fig. 1

existence of paths of certain lengths. These checks translate into more clauses in the CNF formulae whose satisfiability has to be established. For the approach of [SSS00], the number of extra clauses is quadratic in the length of the path. As a result, it is not surprising that finding counterexamples is slower than with pure BMC. The extra cost, however, appears to be worth paying, since it increases substantially the fraction of passing properties that can be decided. Unfortunately, there remain instances for which the additional termination tests are too expensive. Consider the model illustrated in Fig. 1. It has $2n + 2$ states, one of which is initial (A). The $n/2$ states $D_{n/2}, \ldots, D_{n-1}$ are the (unreachable) target states. The longest simple path from the initial state has length $n + 1$, while the longest simple path to a target state that does not visit any other target state has length $n/2$; the reverse sequential depth of the model is also $n/2$. Hence, the methods of [SSS00,McM03] will have to consider paths of length $n/2$ before they can declare the target states unreachable. By contrast, the forward sequential depth is 2.

Fig. 2 shows an abstraction of the model of Fig. 1. States A, B_i, C, and D_i are abstracted by α, $\beta_{\lfloor 2i/n \rfloor}$, γ, and $\delta_{\lfloor 2i/n \rfloor}$, respectively. The target state remains unreachable in this model, and the forward sequential depth is still 2; however, the longest simple path and the sequential depth are reduced. Though in general there is no guarantee that abstraction will shorten or even not lengthen the longest simple paths, or the shortest paths, this example illustrates how abstraction may help BMC, especially for passing properties.

Abstraction and BMC have been combined in more than one recent work, especially in the context of abstraction refinement. In abstraction refinement [Kur94], one starts with a coarse abstraction of the given, concrete model and keeps refining it until the property is decided. For universal properties like the reachability properties that are the focus of this paper, this often means that the abstract models simulate the concrete one [Mil71], and that either the property is shown to hold on an abstract model, or a counterexample is found in the concrete one. In [WHL$^+$01,CGKS02,CCK$^+$02,WLJ$^+$03] BMC is used to check whether counterexamples found in the abstract models can be *concretized*, that is, whether a counterexample can be found in the concrete model that is mapped by the abstraction onto the abstract counterexample. The first three of these methods also analyze the failed concretization test to guide the refinement. Therefore, they represent instances of counterexample-guided abstraction refinement. On the other hand, [WLJ$^+$03] analyzes the abstract model to decide how to refine it. Yet another approach is the one of [MA03], in which the abstract model is derived from a failing BMC run on the concrete model. This reversal of the customary order is attractive for those frequent cases in which paths of moderate length can be easily checked on the concrete model.

One common trait of the approaches to abstraction refinement mentioned so far is the application of a BDD-based model checker to the abstract models, and of SAT solvers to the concrete ones. By contrast, the objective of this paper is to explore what can be achieved with a SAT solver as the only decision procedure in the abstraction refinement framework. The rationale for combining BDDs and SAT is that each is well-suited to the task assigned to it: The SAT solver is good at checking the existence of a path of a given length in a large model, whereas the BDD-based model checker is better at proving the absence of certain paths, regardless of their lengths, in a model of moderate size. This observation is certainly well motivated when one regards the models for which abstraction refinement results have been reported in the literature; their sizes rarely exceed 1,000 binary state variables. As the models grow larger, however, we expect an approach purely based on SAT to become more competitive. Therefore, our goal is to eventually being able to switch between BDD-based model checking and SAT-based techniques for the analysis of the concrete model. In this paper we report on a significant step in that direction by presenting an algorithm for abstraction refinement that is purely based on SAT.

Our approach is similar to the ones discussed so far in the fact that abstractions are obtained by removing part of the state variables of the model; refinement then consists of reinstating some of the removed variables. The algorithm has three major components: the decision procedure for the abstract model is the one of [SSS00], which has already been mentioned. The second component—the choice of the refinement—combines elements of [WLJ$^+$03] and [CCK$^+$02]. Like the former, it addresses all the abstract counterexamples at once; like the latter, it analyzes the conflict dependency graph of the failed concretization test to derive a set of candidate state variables from which the ones that will be added to the abstract model are chosen. Finally, the third component is a heuristic procedure for abstraction minimization. This minimization is quite important in our approach, because the simultaneous elimination of all spurious counterexamples of a certain length tends to generate large sets of candidate variables. Our experimental evaluation of the SAT-based abstraction refinement algorithm compared it to both BMC (with and without early termination checks for passing properties) and to the best abstraction refinement algorithm available to us [WLJ$^+$03]. The results, discussed in Section 4, show that the new approach, though not uniformly superior, is more robust than the others, and is especially promising for the more challenging problems.

2 Preliminaries

Let $V = \{v_1, \ldots, v_n\}$ be a set. We designate by V' the set $\{v_1', \ldots, v_n'\}$ consisting of the primed version of the elements of V, and by V^i the set $\{v_1^i, \ldots, v_n^i\}$. We define an *open system* as a 4-tuple

$$\langle V, W, I, T \rangle \ ,$$

where V is the set of (current) state variables, W is the set of combinational variables, $I(V)$ is the initial state predicate, and $T(V, W, V')$ is the transition relation. The variables in V' are the next state variables. All sets are finite, and all variables range over finite domains.

We assume that the transition relation is given as the composition of elementary relations. If $W = \{w_1, \ldots, w_m\}$ with $m \geq n$, our assumption amounts to writing:

$$T(V, W, V') = \bigwedge_{1 \leq i \leq n} (v_i' \leftrightarrow w_i) \wedge \bigwedge_{1 \leq i \leq m} T_i(W, V) \ . \tag{1}$$

We consider the case of a sequential circuit, in which the variables in W are associated with the primary inputs and the outputs of the combinational logic gates of the circuit; the variables in V are associated with the memory elements. Each T_i is called a *gate relation* because it usually describes the behavior of a logic gate. For instance, if w_i is the output variable of a two-input AND gate with inputs w_j and v_k, then $T_i = w_i \leftrightarrow (w_j \wedge v_k)$. If, on the other hand, w_i is a primary input to the circuit, then $T_i = 1$. Each term of the form $v_i' \leftrightarrow w_i$ equates a next state variable to a combinational variable. (The output of the gate feeding the i-th memory element.)

In a sequential circuit, a state variable v_j is said to be in the *direct support* of variable v_i (w_i) if the memory element associated to v_j is connected to the memory element (logic gate) associated to v_i (w_i) by a path that goes through logic gates only. Variable v_i is in the *cone of influence* (COI) of v_i (w_i) if there is a path (of any kind) connecting v_j to v_i (w_i).

An open system Ω defines a labeled transition structure in the usual way, with states Q_Ω corresponding to the valuations of the variables in V, and transition labels corresponding to the valuations of the variables in W. Conversely, a set of states $S \subseteq Q_\Omega$ corresponds to a predicate $S(V)$ or $S(V')$. Predicate $S(V)$ ($S(V')$) is the characteristic function of S expressed in terms of the current (next) state variables. State $q \in Q_\Omega$ is an initial state if it satisfies $I(V)$. State set $S \subseteq Q_\Omega$ is *reachable* from state set S' in k steps if there is a path of length k in the labeled transition structure defined by Ω that connects some state in S' to some state in S; equivalently if

$$S'(V^0) \wedge \bigwedge_{1 \leq i \leq k} T(V^{i-1}, W^i, V^i) \wedge S(V^k) \tag{2}$$

is satisfiable. State set S is reachable from S' if there exists $k \in \mathbb{N}$ such that S is reachable in k steps from S'. A state set is reachable (in k steps) if it is reachable (in k steps) from I. When no confusion arises we shall identify a state $q \in Q_\Omega$ with the set $\{q\}$. A finite (infinite) sequence of states $\rho \in Q_\Omega^*$ ($\in Q_\Omega^\omega$) is a finite (infinite) *run* of Ω if the first state is initial, and every other state is reachable from its predecessor in one step. The set of all possible runs of Ω is the language of Ω, denoted by $L(\Omega)$.

A linear-time *safety property* P of Ω is a subset of Q_Ω^ω such that any infinite sequence over Q_Ω not in P has a finite prefix that cannot be extended to a sequence in P [AS85]. Open system Ω satisfies safety property P if $L(\Omega) \subseteq P$. Checking the satisfaction of an ω-regular safety property P by an open system Ω can be reduced to the reachability problem by composing Ω with an automaton $\mathcal{A}_P$ that accepts the inextensible prefixes of the sequences not in P. The property is satisfied by the open system if no state of the composition $\Omega \parallel \mathcal{A}_P$ that projects on an accepting state of $\mathcal{A}_P$ is reachable. In the sequel we restrict ourselves to ω-regular safety properties, and assume that the given open system already incorporates the property automaton. This assumption allows us to identify the property with a set of (accepting) states of the system, which we also denote by P. Hence, property P is satisfied by Ω if there is no $k \in \mathbb{N}$ such that

$$I(V^0) \wedge \bigwedge_{1 \leq i \leq k} T(V^{i-1}, W^i, V^i) \wedge \neg P(V^k) \tag{3}$$

is satisfiable. An invariant is a safety property that states that a certain predicate holds of all reachable states of Ω. In this case P is the set of states that satisfy that predicate.

The search for a k such that (3) is satisfiable can obviously be restricted to the range $\{0, \ldots, |Q_\Omega| - 1\}$. Hence, in theory, the process is guaranteed to terminate. In practice, the number of states is too large to be of any practical use, and tighter upper bounds for k are sought. In model checking approaches that are based on fixpoint computations [McM94,ABE00,WBCG00,McM02], the maximum value of k is provided by the number of iterations needed to reach convergence. On the other hand, for algorithms that directly check the satisfiability of (3), the diameter of the graph [BCCZ99] or bounds obtained from the structure of the hardware model have been proposed [BKA02]. Here we summarize a method proposed in [SSS00] that is of particular interest to us.

A *simple path* is one that visits a state at most once. If some state in $\neg P$ is reachable, there must exist a simple path from an initial state to it that does not go through any other states in I or $\neg P$. Hence, if no simple path of length k exists such that its first state is initial and no other state is initial, or such that its final state is in $\neg P$ and no other state is in $\neg P$, then, there is no path of length greater than or equal to k connecting a state in I to a state in $\neg P$. If in addition, there is no path of length less than k connecting I to $\neg P$, then $\Omega \models P$. Two sets of states S' and S are connected by a simple path of length k in Ω if

$$\Sigma_k(S', S) = S'(V^0) \wedge \bigwedge_{1 \le i \le k} T(V^{i-1}, W^i, V^i) \wedge S(V^k) \wedge \bigwedge_{0 \le j < i \le k} \; \bigvee_{1 \le l \le n} (v_l^i \neq v_l^j) \quad (4)$$

is satisfiable. Checking the two conditions above then amounts to checking that either of the following formulae is unsatisfiable.

$$\Sigma_k(I, Q) \wedge \bigwedge_{0 < i \le k} \neg I(V^i) \quad (5)$$

$$\Sigma_k(Q, \neg P) \wedge \bigwedge_{0 \le i < k} P(V^i) \ . \quad (6)$$

Note that the predicate corresponding to the set Q is true.

Abstract interpretation [CC77] provides a very flexible framework for the description of abstraction. In this paper, however, we consider the following restricted definition. Open system $\widehat{\Omega} = \langle \widehat{V}, \widehat{W}, \widehat{I}, \widehat{T} \rangle$ is an *abstraction* of Ω if

- $\widehat{V} \subseteq V$;
- $\widehat{W} \subseteq W$ such that $v_i \in \widehat{V}$ implies $w_i \in \widehat{W}$;
- $\widehat{I}(\widehat{V}) = \exists (V \setminus \widehat{V}) \,.\, I(V)$;
- $\widehat{T}(\widehat{V}, \widehat{W}, \widehat{V}') = \exists (V \setminus \widehat{V}) \,.\, \exists (W \setminus \widehat{W}) \,.\, \exists (V' \setminus \widehat{V}') \,.\, T(V, W, V')$.

(Note that w_i is the combinational variable associated to v_i'.) Property $\widehat{P}$ is the abstraction of property P with respect to $\widehat{\Omega}$ if $\widehat{P}(\widehat{V}) = \exists (V \setminus \widehat{V}) \,.\, P(V)$. If P is an ω-regular set and $\widehat{\Omega}$ satisfies (or models) $\widehat{P}$, then Ω satisfies P. That is,

$$\widehat{\Omega} \models \widehat{P} \rightarrow \Omega \models P \ . \quad (7)$$

This preservation result is the basis for the following abstraction refinement approach to the verification of P. One starts with a coarse abstraction $\widehat{\Omega}_0$ of the *concrete* open system Ω and checks whether $\widehat{\Omega}_0 \models \widehat{P}_0$. If that is the case, then $\Omega \models P$; otherwise, there exists a least $k' \in \mathbb{N}$ such that

$$\widehat{I}(\widehat{V}^0) \wedge \bigwedge_{1 \le i \le k'} T(\widehat{V}^{i-1}, \widehat{W}^i, \widehat{V}^i) \wedge \neg \widehat{P}(\widehat{V}^{k'}) \quad (8)$$

is satisfiable. The satisfying assignments to (8) are the shortest-length *abstract counterexamples* (ACEs). If $\widehat{\Omega}_0 \not\models \widehat{P}_0$ one or more ACEs are checked for *concretization*. That is, one checks whether (3) has solutions that agree with the ACE(s) being checked. Because of the additional constraints provided by the ACEs, a concretization test is often less expensive that the satisfiability check of (3). However, its failure only indicates that the abstract error traces are spurious. Therefore, if the concretization test fails, one chooses a refined abstraction $\widehat{\Omega}_1$ and repeats the process, until one of these cases occurs.

(i) $\widehat{\Omega}_i \models \widehat{P}_i$ for some i, in which case $\Omega \models P$ is inferred.

(ii) The concretization test passes for some i, in which case it is concluded that $\Omega \not\models P$ and the satisfying assignment found is returned as counterexample to P.

(iii) The refinement eventually produces $\widehat{\Omega}_i = \Omega$. In this final case, the satisfiability check of (8) answers the model checking question conclusively. This is an undesirable outcome because the purpose of abstraction is defeated.

When the refinement $\widehat{\Omega}_{i+1}$ of $\widehat{\Omega}_i$ is chosen with the help of the information provided by the failed concretization test, one talks of counterexample-guided abstraction refinement.

The cone of influence (direct support) of a property is the union of the cones of influence (direct supports) of all the variables mentioned in the property. Cone-of-influence reduction refers to the abstraction in which $\widehat{V}$ is the COI of the property. It is commonly applied before any model checking is attempted, because it satisfies

$$\widehat{\Omega} \models \widehat{P} \leftrightarrow \Omega \models P \ . \tag{9}$$

3 Algorithm

Our algorithm is shown in Fig. 3. Initially, an abstract model $\widehat{\Omega}$ is computed by collecting only the state variables (called *latches* henceforth) in the direct support of the property P. The algorithm then progressively increases L from its initial value 0 until either a counterexample of length L is found in the concrete system Ω, or it is concluded that no counterexample exists in the current abstract model. If at some point, the abstract model becomes the concrete model, the endgame is executed as described in Lines 14–19.

Lines 3–13 verify the abstract models. First, (5) and (6) are checked to see whether the simple path conditions are met. If either one is unsatisfiable, the property holds, and the algorithm terminates. Otherwise, the algorithm checks whether there is a counterexample of length L in the abstract model, by checking (3) on $\widehat{\Omega}$; if there is no length-L abstract counterexample, there is no counterexample of length up to L in the concrete model either. (This is because every abstract model simulates the concrete model; hence, if there is a real counterexample of length $L' \leq L$ in the concrete model, there must be a corresponding abstract counterexample of length $L'' \leq L'$. Since the counterexample length is increased in increments of one, we would have found this counterexample before.) Since there is no counterexample of length up to L (in either the abstract model or the concrete model), L is increased by one. On the other hand, if there is an abstract counterexamples of

```
boolean pureSAT($\Omega$,$P$) {
1   $L$ = 0;
2   $\widehat{\Omega}$ = CreateInitialAbstraction($\Omega$,$P$);
3   **while** ($\widehat{\Omega} \neq \Omega$) {
4       **if** ($\neg$ CheckSimplePath($\widehat{\Omega}$,$P$,$L$))
5           **return** TRUE;
6       **if** (ExistCex($\widehat{\Omega}$,$P$,$L$)) {
7           **if** (ExistCex($\Omega$,$P$,$L$))
8               **return** FALSE;
9           refinement = GetRefinementFromCA($\Omega$,$\widehat{\Omega}$,$P$,$L$);
10          $\widehat{\Omega}$ = AddRefinementToAbsModel($\widehat{\Omega}$, refinement);
11      }
12      $L = L + 1$;
13  }
14  **while** (CheckSimplePath($\Omega$,$P$,$L$)){
15      **if** (ExistCex($\Omega$,$P$,$L$))
16          **return** FALSE;
17      $L = L + 1$;
18  }
19  **return** TRUE;
}
```

Fig. 3. The PureSAT algorithm

```
set GetRefinementFromCA($\Omega$, $\widehat{\Omega}$,$P$,$L$) {
    nsVarSet = GetNextStateVarsFromCDG($\Omega$,$P$,$L$);
    sufficient = $\emptyset$ ;
    **while** (sufficient does not kill all length-$L$ counterexamples
                $\wedge$ nsVarSet is not empty) {
        someNsVars = PickVarsThreshold(nsVarSet, threshold);
        sufficient = sufficient $\cup$ someNsVars
        nsVarSet = nsVarSet $\setminus$ someNsVars
    }
    RCArray = ComputeRelativeCorrelationArray(sufficient,$\Omega$,$\widehat{\Omega}$);
    **return** RefinementMinimization($\widehat{\Omega}$,RCArray);
}
```

Fig. 4. The refinement algorithm

length L, (3) is checked on the concrete model to see if any concrete counterexample of the same length exists. If it does, the property fails; otherwise, the refinement step (Lines 9–10) is executed.

The goal of the refinement procedure is to find a minimal set of latches not in $\widehat{\Omega}$ which, after being added to the abstract model, can kill all the counterexamples of the length L. Our refinement algorithm is based on computing and analyzing the *unsatisfiable core* [GN03,ZM03] associated with the proof that there is no concrete counterexample of length L; hence, it is similar to the conflict analysis method proposed in [CCK+02]. However, our approach differs significantly from [CCK+02] in the following aspects:

(i) The authors of [CCK+02] first identify a single spurious abstract counterexample (by using BDD-based model checking), together with its failure index. (I.e., the time step from which the ACE is no longer concretizable in the concrete model.) A conflict dependency graph is built from the unsatisfiable BMC obtained by constraining the concrete model with the single spurious ACE up to the failure index time step. The refinement set is then computed by analyzing the conflict dependency graph. In our algorithm, however, we do not use a single abstract counterexample to constrain the BMC instance (and we do not compute the failure index). Rather, an unconstrained BMC instance (on the concrete model, for path length up to L) is used for the concretization test; such a BMC instance covers all the possible length-L spurious abstract counterexamples.

(ii) In [CCK+02], the *invisible* latches (those not currently in $\widehat{\Omega}$) are added to the refinement set if their corresponding literals at the failure index time step appear in the conflict dependency graph. In our algorithm, all the literals (which correspond to either latches or internal logic gates at different time steps) appearing in the unsatisfiable core are recorded in the SAT solver. However, only those invisible latches whose *next-state variable* literals (i.e., the literals corresponding to the input variable of a latch at a different time step) appear in the unsatisfiable core are added to the refinement set. This refinement set, when added to $\widehat{\Omega}$, is sufficient to kill all length-L spurious abstract counterexamples. Our algorithm for picking refinement variables is shown in Fig. 4. The original "sufficient set" (i.e., nsVarSet in the pseudo code) may or may not be minimal; hence, refinement minimization is used to get rid of the redundant latches in the refinement set before the function returns. In some cases, the number of redundant invisible latches in nsVarSet may be too large, causing REFINEMENTMINIMIZATION to spend too much time. The **while** loop, together with a threshold, is used to heuristically get a smaller "sufficient set" for the refinement minimization: Each time, only a certain number of invisible latches are picked from nsVarSet, after which (3) is checked to see if they are already sufficient.

(iii) Our refinement minimization algorithm is also somewhat different from [CCK+02]. Both methods remove redundant latches greedily. Each latch in turn is tentatively removed. If (3) remains unsatisfiable, the remaining latches are still sufficient, and the dropped latch is indeed redundant; otherwise, that latch is restored to the refinement set. In our method, the order in which invisible latches are removed in the minimization procedure is based on the *relative correlation* of each candidate latch to the current abstract model. The relative correlation of an invisible latch equals the ratio of the number of gates in the COI of this latch which are already in the abstract model divided by the total number of gates in the COI of this latch. Intuitively, the larger the relative correlation of a latch, the larger effect it will have when added to or subtracted from the current abstract model. The invisible latches of the current sufficient set are sorted by Function ComputeRelativeCorrelationArray: The one with the smaller relative correlation is considered of less importance, and thus will be tested for deletion earlier. In this way, we can concentrate on the important invisible latches and at the same time keep the refined abstract model small.

Our approach is also related to the one of [MA03]. Both approaches check all counterexamples of a certain lenght at once by a model checking run on the concrete model. The main differences are:

(i) We use SAT, instead of a BDD-based model checker, for the abstract model. This will give our method an advantage in proofs that require an abstract model of size comparable to that of the concrete one.

(ii) Our abstraction grows at each refinement, and we use refinement minimization to control its size, whereas the abstraction of [MA03] is computed from scratch each time. Refinement minimization requires repeated BMC runs; these, however, are runs on the abstract model. In the experiments reported in Section 4, refinement minimization was never the bottleneck, and it could be further sped up by using an incremental SAT solver.

4 Experimental Results

To evaluate the technique of Section 3, we compared four algorithms: an implementation of the BMC [BCCZ99] algorithm, BMC extended with the checks for simple paths [SSS00] (referred to as SSS), our PureSAT algorithm, and the Grab algorithm of [WLJ+03], which uses both BDDs and SAT. All the four algorithms are implemented in VIS-2.0 [B+96,VIS], and Chaff [MMZ+01] was use as the back-end SAT solver. The experiments were run under Linux on an IBM IntelliStation with a 1.7 GHz Intel Pentium 4 CPU and 2 GB of RAM.

The comparison was conducted on 26 models, either from industry or from VIS verification benchmarks [B+96,VIS] except for *lsp*. This model was created to illustrate the help BMC could get from abstraction. A simplified version of it appears in Fig. 1. Since in the concrete model, the longest simple path is long, SSS failed to complete, even though PureSAT finished within one second.

The results are shown in Table 1. The first column is the name of the model, the second column indicates whether each property passes or fails; if a property fails, the number in this column is the length of the counterexample. The third column gives the number of latches in the cone of influence of the property. The fourth column lists the time of BMC. A time in parentheses is the time elapsed when the process ran out of memory. In our experiments, the time limit was set to 8 hours.

Table 1
Experimental results. Boldface is used to highlight best CPU times

model	pass/	latches	BMC	SSS	PureSAT		Grab	
	cex length	in COI	time	time	time	final sz.	time	final sz.
lsp-p1	pass	12	>8h	>8h	**1**	3	**1**	3
D12-p1	16	48	**5**	25	37	23	14	23
D23-p1	5	85	**1**	**1**	3	25	20	21
D2-p1	14	94	**6**	25	20	48	180	48
D14-p1	14	96	**65**	83	1460	80	>8h	(75)
D1-p1	9	101	**1**	5	11	20	9	21
D1-p2	13	101	**2**	12	26	23	51	23
D1-p3	15	101	**3**	18	32	23	56	25
I12-p1	370	119	>8h	>8h	>8h	(12)	**2503**	16
B-p1	pass	124	>8h	>8h	2074	18	**173**	18
B-p2	17	124	150	675	247	7	**93**	7
B-p3	pass	124	>8h	>8h	>8h	(42)	**223**	43
B-p4	pass	124	>8h	(23708)	>8h	(43)	**393**	42
D22-p1	10	140	**2**	10	17	132	720	132
D24-p1	9	147	7	10	2	4	**1**	4
D24-p2	pass	147	>8h	16	6	8	**3**	8
D24-p3	pass	147	>8h	**1**	4	6	20	8
D24-p4	pass	147	>8h	**1**	4	6	43	8
D24-p5	pass	147	>8h	**1**	4	8	3	5
M0-p1	pass	221	>8h	(2537)	2156	13	**136**	16
D5-p1	31	319	58	592	155	13	**31**	18
D18-p1	23	506	**96**	795	4359	160	>8h	(99)
D16-p1	8	531	**10**	29	31	14	92	14
D20-p1	14	562	**26**	101	6228	232	>8h	(69)
rcu-p1	pass	2453	>8h	(3115)	**136**	11	195	10
IU-p2	pass	4493	(11331)	>8h	**1756**	14	>8h	(6)

The fifth column is the time of SSS; the sixth column shows the time for PURESAT; the seventh column is the number of latches in the final abstract model. If the time is greater than 8 hours, the number in parentheses in the next column is the number of latches in the abstract model when time ran out. The next two columns are the data for GRAB. All CPU times are in seconds except when noted.

The algorithm labeled BMC can check inductive invariants. However, no such properties are included in our set of experiments. From the table we can see that, in general, for passing properties, PURESAT is better than both BMC and SSS. For failing properties, with a few exceptions, BMC is best, while PURESAT is better than GRAB. For the largest model, like IU, whose COI contains 4493 latches, PURESAT is the only one being able to verify the property. Interestingly, GRAB and PURESAT fail to finish similar numbers of experiments (4 for GRAB and 3 for PURESAT). However, the two sets of failures are disjoint. This is an encouraging sign for the development of a hybrid algorithm that may switch between BDDs and SAT for the analysis of the abstract models.

Though PURESAT appears to be reasonably robust, there are only three cases in Table 1 in which it manages to be fastest. This is in part due to the fact that the implementation is still preliminary.

5 Conclusions

We have presented an abstraction refinement algorithm for model checking safety properties that uses a SAT solver as sole decision procedure. We have compared this algorithm to both BMC and to an abstraction refinement algorithm that uses both BDDs and CNF SAT. The new algorithm is competitive and was the only one to complete the largest test case. Our implementation is still preliminary. We plan to investigate the use of an incremental SAT solver like SATIRE [WKS01] in the abstraction minimization phase, which is currently the most time consuming part of the algorithm. We are also interested in the extension of the techniques of [WLJ$^+$03] to the SAT environment. This is not an entirely trivial task, since they are based on the knowledge of the sets of states at various distances along the paths connecting initial states to error states.

By its very nature, the PURESAT algorithm suffers, albeit in attenuated form, from the same problems that afflict the basic procedure used in analyzing the abstract models. Improvements like those proposed in [McM03] may boost PURESAT's performance. More generally, the integration with a BDD-based approach to the analysis of the abstract model should lead to a more robust and powerful approach to abstraction refinement.

References

[ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Tools and Algorithms for the Construction of Systems (TACAS)*, pages 411–425, 2000. LNCS 1785.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[B$^+$96] 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.

[BCCZ99] 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, March 1999. LNCS 1579.

[BKA02] 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.

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

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by constructions or approximation of fixpoints. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 238–250, 1977.

[CCK$^+$02] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 33–51. Springer-Verlag, November 2002. LNCS 2517.

[CGKS02] 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.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[GN03] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE'03)*, pages 886–891, Munich, Germany, March 2003.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[MA03] 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, April 2003. LNCS 2619.

[McM94] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[McM02] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In *Fifteenth Conference on Computer Aided Verification (CAV'03)*. Springer-Verlag, Berlin, July 2003. LNCS 2725. To appear.

[Mil71] R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[MMZ$^+$01] 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.

[SSS00] 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, November 2000. LNCS 1954.

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

[WBCG00] P. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 124–138. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[WHL$^+$01] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the Design Automation Conference*, pages 35–40, Las Vegas, NV, June 2001.

[WKS01] 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.

[WLJ$^+$03] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. Submitted for publication, April 2003.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*, pages 880–885, Munich, Germany, March 2003.