Int J Softw Tools Technol Transfer (2005) 7: 143–155 / Digital Object Identifier (DOI) 10.1007/s10009-004-0169-2

Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure

Bing Li, Chao Wang, Fabio Somenzi

University of Colorado at Boulder, USA
e-mail: {bli,wangc,Fabio}@Colorado.edu

Published online: 24 February 2005 – © Springer-Verlag 2005

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 an incremental procedure that combines the analysis of the conflict dependency graph produced by the SAT solver while looking for concrete counterexamples with an effective refinement minimization procedure.

Keywords: Bounded model checking – Abstraction refinement – Satisfiability problem – Unsatisfiability proof

1 Introduction

Model checking [11] is an algorithmic approach to the verification of properties of reactive systems that 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) [7] and Conjunctive Normal Form (CNF).

Classical BDD-based model checking [17] 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) [4], 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 [4]. 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

This work was supported in part by SRC contract 2003-TJ-920.

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 [1, 18, 23] 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 nonexistence 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 [22], 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 [19] 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 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 [22], 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.[1] 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 [19, 22] 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.

Figure 2 shows an abstraction of the model of Fig. 1. States A, B_i, C, and D_i are abstracted by α, $\beta_{\lfloor 2i/n \rfloor}$,

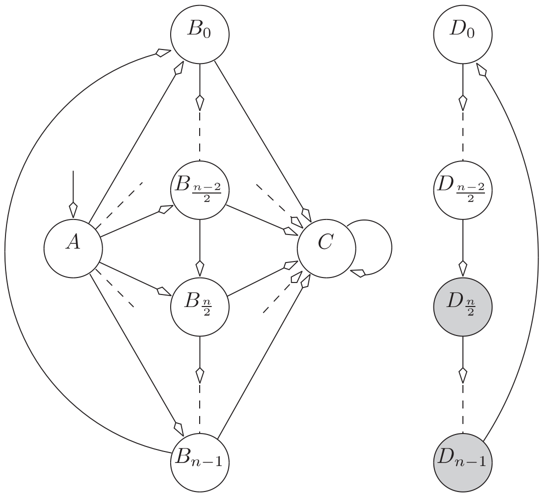


Fig. 1. Model with long simple path

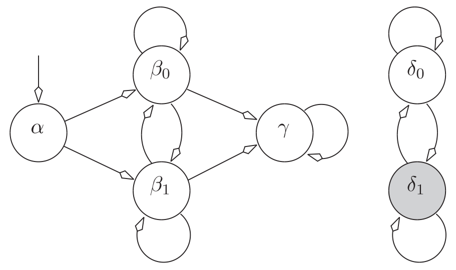


Fig. 2. Abstraction of the model of Fig. 1

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 [15], 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 article, this often means that the abstract models simulate the concrete one [20], and that either the property is shown to hold on an abstract model, or a counterexample is found in the concrete one. In [9, 10, 24, 26] 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, [26] analyzes the abstract model to decide

[1] In [14] it was shown how to reduce the number of clauses to $O(k \log^2 k)$. However, the extra clauses to enforce simple paths still represent a significant hurdle for the SAT solvers.

how to refine it. Yet another approach is the one of [16], 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 article 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 1000 binary state variables. As the models grow larger, however, we expect an approach based purely on SAT to become more competitive. Therefore, our goal is to eventually be able to switch between BDD-based model checking and SAT-based techniques for the analysis of the concrete model. In this article we report on a significant step in that direction by presenting an algorithm for abstraction refinement that is based purely on SAT.

Our approach is similar to those 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 four major components: the decision procedure for the abstract model is that of [22], which has already been mentioned. The second component is the concretization test: as in [5], a gradual refinement approach tries to prove counterexamples spurious without resort to the concrete model. The third component – the choice of the refinement – makes use of elements of [9, 16, 26]. Like the first two, it addresses all the abstract counterexamples of a certain length at once; like the last two, it analyzes the proof of nonexistence of counterexamples of a certain length to derive a set of candidate state variables from which the ones that will be added to the abstract model are chosen. Finally, the fourth 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 termination checks for passing properties) and to the best abstraction refinement algorithm available to us [26]. The results, discussed in Sect. 5, 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

2.1 Open systems

Let $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$ be sets of Boolean variables. 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 $\{v^i_1, \ldots, v^i_n\}$. Likewise, $W^i = \{w^i_1, \ldots, w^i_m\}$. An *open system* is 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 $T(V, W, V')$ is given by a *circuit graph*, that is, by a labeled graph $\mathcal{C} = (V \cup W, E)$ such that $m \geq n$, node $v_i \in V$ is labeled by $w_i \in W$, node $w_i \in W$ is labeled by a Boolean formula $T_i = w_i \leftrightarrow \delta_i(V, W)$, $(w_i, v_i) \in E$ for $i \in \{1, \ldots, n\}$, and, for $x \in V \cup W$, $w_i \in W$, $(x, w_i) \in E$ iff x appears in δ_i. The transition relation is then defined by:

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

In a sequential circuit, 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 with a combinational variable. (The output of the gate feeding the ith memory element.)

A state variable v_j is said to be in the *direct support* of variable v_i (w_i) if v_j is connected to v_i (w_i) by a path in $\mathcal{C}$ that goes through nodes in W (logic gates) only. Variable v_j 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) in $\mathcal{C}$.

Example 1. Figure 3 shows a simple sequential circuit with two state variables. The transition relation (1) is given by

$$T(V, W, V') = (v'_1 \leftrightarrow w_1) \wedge (v'_2 \leftrightarrow w_2) \wedge (w_2 \leftrightarrow (w_3 \wedge v_1)) .$$

Note that $T_1 = T_3 =$ true. If the only initial state of the circuit is $v_1 = 0, v_2 = 0$, then $I(V) = \neg v_1 \wedge \neg v_2$. Variable v_1 is in the direct support of w_2 and in the COI of v_2. □

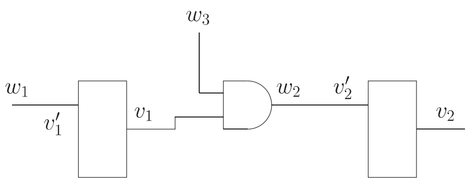


Fig. 3. A sequential circuit defining an open system with $n = 2$ and $m = 3$. The rectangles represent binary state elements

2.2 Proving safety properties

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

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

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 $\mathcal{L}(\Omega)$.

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 [2]. Open system Ω satisfies safety property P if $\mathcal{L}(\Omega) \subseteq P$. 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 what follows 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 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

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

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.

Example 2. Continuing Example 1, suppose that $P(V) = \neg v_2$. Then a counterexample of length 2 to P can be found by solving

$$\neg v_1^0 \wedge \neg v_2^0 \wedge (v_1^1 \leftrightarrow w_1^1) \wedge (v_2^1 \leftrightarrow w_2^1) \wedge (w_2^1 \leftrightarrow (w_3^1 \wedge v_1^0)) \wedge (v_1^2 \leftrightarrow w_1^2) \wedge (v_2^2 \leftrightarrow w_2^2) \wedge (w_2^2 \leftrightarrow (w_3^2 \wedge v_1^1)) \wedge v_2^2 \,.$$

A satisfying assignment for this formula is

$$\neg v_1^0 \wedge \neg v_2^0 \wedge w_1^1 \wedge \neg w_2^1 \wedge v_1^1 \wedge \neg v_2^1 \wedge w_1^2 \wedge w_2^2 \wedge w_3^2 \wedge v_1^2 \wedge v_2^2 \,.$$

The value of w_3^1 is irrelevant. □

The search for a k such that (3) is satisfiable can obviously be restricted to the range $\{0, \dots, |Q_\Omega| - 1\}$. 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 based on fixpoint computations [1, 17, 18, 23], 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 [4] or bounds obtained from the structure of the hardware model have been proposed [6]. Here we summarize a method proposed in [22] 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

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

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

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

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

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

2.3 Abstraction refinement

Abstract interpretation [8] provides a very flexible framework for the description of abstraction. In this article, 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

- $\widehat{V} \subseteq V$;
- $\widehat{W} = \widehat{W}_1 \cup (V \setminus \widehat{V})$;
- $\widehat{W}_1 \subseteq W$ 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'.) This definition entails that $\widehat{\Omega}$ simulates [20] Ω. Hence, every run of Ω has a matching run in $\widehat{\Omega}$. Property $\widehat{P}$ is the abstraction of property P with respect to $\widehat{\Omega}$ if $\widehat{P}(\widehat{V}) = \forall (V \setminus \widehat{V}) \,.\, P(V)$. If P is an ω-regular property and $\widehat{\Omega}$ satisfies (or models) $\widehat{P}$, then Ω satisfies P. That is,

$$\widehat{\Omega} \models \widehat{P} \rightarrow \Omega \models P \,. \tag{7}$$

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

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

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 ACEs being checked. Because of the additional constraints provided by the ACEs, a concretization test is often less expensive than 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.

1. $\widehat{\Omega}_i \models \widehat{P}_i$ for some i, in which case $\Omega \models P$ is inferred.
2. 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.
3. 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 predicate that defines 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

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

Example 3. For the circuit of Fig. 3, consider the following abstraction $\widehat{\Omega}$:

- $\widehat{V} = \{v_2\}$;
- $\widehat{W} = \{w_2, w_3, v_1\}$;
- $\widehat{I}(\widehat{V}) = \neg v_2$;
- $\widehat{T}(\widehat{V}, \widehat{W}, \widehat{V}') = (v_2' \leftrightarrow w_2) \wedge (w_2 \leftrightarrow (w_3 \wedge v_1)$.

Suppose $P(V) = \widehat{P}(\widehat{V}) = \neg v_2$. Then from (8) one finds that a counterexample of length 1 exists in $\widehat{\Omega}$ by solving

$$\neg v_2^0 \wedge (v_2^1 \leftrightarrow w_2^1) \wedge (w_2^1 \leftrightarrow (w_3^1 \wedge v_1^1)) \wedge v_2^1 \,.$$

The only satisfying assignment is $\neg v_2^0 \wedge w_2^1 \wedge w_3^1 \wedge v_1^1 \wedge v_2^1$. Note that this ACE cannot be concretized because v_1^0 cannot be set to 1 in the concrete model. After refinement adds v_1 to $\widehat{V}$, $\widehat{\Omega} = \Omega$. The counterexample found in Example 2 therefore shows that $\Omega \not\models P$. □

2.4 Satisfiability solvers

Many modern SAT solvers are based on clause recording. Whenever they detect a conflicting assignment to a formula f (one that causes f to evaluate to false), they conjoin a *conflict clause* to f. The new clause prevents the solver from attempting the same assignment again. It may also exclude from future consideration other parts of the search space that can be inferred to contain no satisfying assignments. Such inference is based on the analysis of the so-called *implication graph*, which shows which decisions and clauses are responsible for the conflict. A SAT solver maintains the implication graph by recording, for each assignment, the *level* at which it was made,[2] whether it was a decision, and, if not, the clause that implied it. When a conflict is detected, a cut separating the sink of the implication graph from the sources identifies a set of assignments sufficient to cause the conflict. The disjunction of the negation of those assignments is a conflict clause.

The clauses that make up the edges of an implication graph between the cut and the sink can be used to explain the conflict clause deduced from it. When using a SAT solver in model checking based on abstraction refinement, there are two important reasons to keep track

[2] The level starts from 0; it is incremented at every new decision and decreases when the algorithm backtracks.

of the explanations of conflict clauses. The first reason is the ability to identify an *unsatisfiable core* [13, 27] of the given formula when it is indeed unsatisfiable. This can be done by recursively replacing conflict clauses with those appearing between the chosen cut and the sink in the implication graphs that produced them. The process starts from the final conflict clause and terminates when only clauses of the original formula are left. When the check for existence of counterexamples of a certain length fails, the unsatisfiable core produced by the SAT solver can be used to guide the refinement of the abstract model.

The second reason to preserve the mapping between conflict clauses and the edges of their implication graphs is to help in the incremental solution of sequences of SAT instances [12, 25]. If f and f' are two CNF formulae, and γ is a conflict clause of f, then γ is also a conflict clause of f' if f' contains all the clauses of f in the implication graph for γ up to the cut that identifies γ. Therefore, if f' is obtained by slight modification of f, many conflict clauses derived in checking the satisfiability of f may be added to f' inexpensively if their "explanations" in terms of clauses of f are saved. As a special case, if f' contains all the clauses of f, then all conflict clauses deduced while checking the satisfiability of f are also valid for f'. Sequences of closely related SAT instances are found in our algorithm for abstraction refinement. They are discussed in Sect. 3.4 together with the exploitation of inherited conflict clauses to speed up SAT checking.

3 Algorithm

In this section we present an algorithm for abstraction refinement that uses SAT as decision procedure on both abstract and concrete models. The input to the algorithm is an open system $\Omega = \langle V, W, I, T \rangle$ whose transition relation T is specified by a circuit graph $\mathcal{C} = (V \cup W, E)$ and a predicate $P(V)$ describing a set of accepting states. We assume that cone-of-influence reduction is applied before invoking the abstraction refinement procedure so that all vertices of $\mathcal{C}$ are in the COI of the property.

3.1 The PureSAT algorithm

Our algorithm is shown in Fig. 4. Initially, an abstract model $\widehat{\Omega}$ is computed by collecting in $\widehat{V}$ only the state variables that appear in $P(V)$; hence, $\widehat{P} = P$ throughout. In lines 3–18 the algorithm progressively increases L from its initial value of 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 $\widehat{\Omega}$. If at some point the abstract model becomes the concrete model, the endgame described in lines 19–24 is executed.

For each length L, (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.

```
boolean PURESAT($\Omega$, $P$, $\mathcal{C}$) {
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       $\widetilde{\Omega} = \widehat{\Omega}$;
7       while (EXISTCEX($\widetilde{\Omega}$,$P$,$L$)) {
8           if ($\widetilde{\Omega} == \Omega$)
9               return FALSE;
10          else
11              $\widetilde{\Omega}$ = ADDLAYER($\widetilde{\Omega}$, $\Omega$, $\mathcal{C}$);
12      }
13      if ($\widetilde{\Omega} \neq \widehat{\Omega}$) {
14          refinement = GETREFFROMCA($\widehat{\Omega}$,$\widetilde{\Omega}$,$P$,$L$);
15          $\widehat{\Omega}$ = ADDREFTOABSMODEL($\widehat{\Omega}$, refinement);
16      }
17      $L = L + 1$;
18  }
19  while (CHECKSIMPLEPATH($\Omega$,$P$,$L$)) {
20      if (EXISTCEX($\Omega$,$P$,$L$))
21          return FALSE;
22      $L = L + 1$;
23  }
24  return TRUE;
}
```

Fig. 4. The PureSAT algorithm

Otherwise, the algorithm checks whether there exists a counterexample of length L (lines 7–12). The check is incremental: since every abstract model simulates the concrete one, a sequence of increasingly refined abstract models is used to establish the existence of a counterexample. If any model $\widetilde{\Omega}$ in the sequence has no counterexample of length L, neither does the concrete model Ω. In the first iteration of the inner **while** loop, (3) is checked on the current abstract model $\widehat{\Omega}$. If abstract counterexamples exist, function ADDLAYER produces a coarse refinement as the next element in the sequence.

The loop terminates as soon as it is known whether a counterexample of length L exists in the concrete model Ω. When no such counterexample exists, if $\widetilde{\Omega} \neq \widehat{\Omega}$ at line 13, then the abstract model admits spurious ACEs. Therefore, it is refined by adding to $\widehat{V}$ a minimal set of variables from $\widetilde{V}$ sufficient to rule out all counterexamples of length L. (lines 14–15). The resulting model has no counterexamples of length up to L.

Lemma 1. *The abstract model computed at line 15 of algorithm* PURESAT *has no counterexample to P of length less than or equal to L.*

Proof. Let $\widehat{\Omega}_i$ be the abstract model computed at line 15 when $L = i$. By definition of refinement, $\widehat{\Omega}_i$ has no counterexamples of length i. We can then prove by induction that $\widehat{\Omega}_L$ has no counterexamples of length i for $i < L$.

The base for $L = 0$ is trivially established. For $L > 0$, we can assume that $\widehat{\Omega}_{L-i}$ has no counterexample of length less than L. Since $\widehat{\Omega}_{L-i}$ is an abstraction of $\widehat{\Omega}_L$, a counterexample of length less than L in the latter would have a matching run in the former, which would contradict the inductive hypothesis. □

3.2 Concretization test

For each value of L, algorithm pureSAT checks whether the concrete model Ω has a counterexample to P of that length. This test proceeds incrementally, starting from the current abstract model $\widehat{\Omega}$, and possibly reaching Ω. At each iteration of the **while** loop at lines 7–12 of Fig. 4, a model $\widetilde{\Omega}$ is checked. Initially, $\widetilde{\Omega} = \widehat{\Omega}$; in the worst case, eventually $\widetilde{\Omega} = \Omega$; however, the loop terminates before this condition is met often enough to justify the incremental approach. If a given $\widetilde{\Omega} \neq \Omega$ has counterexamples of length L, AddLayer is invoked to produce the model to be checked at the next iteration.

Procedure AddLayer adds variables from $V \setminus \widetilde{V}$ to $\widetilde{\Omega}$ with two objectives: to allow the concretization check to complete quickly and to help produce a better refinement if one is necessary.

When no concrete counterexamples are found, the two goals are met if AddLayer arrives in few iterations at a small $\widetilde{\Omega}$ such that ExistCex($\widetilde{\Omega}$,P,L) returns false. By producing a short sequence of small models, the time spent in checking for counterexamples in them is kept short. In addition, the sizes of the unsatisfiable cores produced when checking (3) grow with the sizes of the models. A smaller $\widetilde{\Omega}$ therefore tends to save time also during refinement minimization. When there are concrete counterexamples, eventually the concrete model Ω must be checked. Hence, only a few iterations of the **while** loop should be taken.

These requirements are addressed by the following procedure. The variables in $V \setminus \widetilde{V}$ are ranked relative to $\widetilde{\Omega}$ according to the same criteria used in the computation of refinement. If the variables at the lowest level make up less than a set fraction of the variables in $V \setminus \widetilde{V}$, they are all added to the model to obtain the new $\widetilde{\Omega}$. Otherwise, only that fraction is added. Since a variable's level is the first sorting criterion, all new variables come from the lowest level; hence, they are guaranteed to have some influence on the behavior of the model.

If there are too many iterations of the **while** loop, the benefit of finding a more compact conflict core does not outweigh the cost of checking the many abstract models for counterexamples. So, if there are still counterexamples after the first few rounds, the algorithm skips to the concrete model.

3.3 Refinement

The goal of the refinement procedure is to find a minimal set of state variables not in $\widehat{\Omega}$ that, when added to

```
set GetRefFromCA($\widehat{\Omega}$, $\widetilde{\Omega}$,$P$,$L$) {
1  nsVarSet = GetNextStateVarsFromCDG($\widetilde{\Omega}$,$P$,$L$);
2  RCArray = SortCandidateArray(nsVarSet,$\widetilde{\Omega}$,$\widehat{\Omega}$);
3  sufficient = $\emptyset$ ;
4  while (sufficient does not kill all length-$L$ counterexamples
5           $\wedge$ RCArray is not empty) {
6      someNsVars = PickVars(RCArray, threshold);
7      sufficient = sufficient $\cup$ someNsVars;
8      RCArray = nsVarSet $\setminus$ someNsVars;
9  }
10 return RefinementMinimization($\widehat{\Omega}$,sufficient);
}
```

Fig. 5. The refinement algorithm

the abstract model, can suppress all counterexamples of length L. Care is put into keeping the abstract model small because the success of the termination criterion based on (5) and (6) is affected by the complexity of $\widehat{\Omega}$.

Our algorithm for picking refinement variables is shown in Fig. 5. When GetRefFromCA is called, $\widehat{\Omega}$ has counterexamples to P of length L, while $\widetilde{\Omega}$ does not because ExistCex just returned false. The addition to $\widehat{\Omega}$ of the state variables in $\widetilde{V} \setminus \widehat{V}$ found in the unsatisfiable core of (3) as next state variables is sufficient to suppress all spurious counterexamples of length L. Variable v_i is used as next state variables in the unsatisfiable core of (3) if w_i^j appears in some clause in that core for some j between 0 and L.

Sufficiency of the refinement can be argued by observing that the candidate variables are all the variables in a set of clauses from which a contradiction can be derived. Refinement with respect to all those variables will add to the SAT problem for the abstract model all the clauses that are necessary to obtain the same contradiction. If state variable v_i does not appear as next state variable in the unsatisfiable core, then the corresponding term T_i in (1) is irrelevant to the proof of unsatisfiability.

The original "sufficient set" (nsVarSet in Fig. 5) extracted from the unsatisfiable core may be nonminimal; hence, RefinementMinimization removes redundant state variables from the refinement set. This procedure tentatively removes one variable at a time from the refinement and checks whether ExistCex still returns false. Since the order in which candidate variables are considered by the minimization procedure affects the result, SortCandidateArray sorts the variables so that those considered less valuable are removed first.

The number of redundant state variables in nsVarSet may be quite large, causing too many calls to ExistCex. The **while** loop of lines 4–9 is used to heuristically get a smaller "sufficient set" for the refinement minimization: each time, only a certain number of state variables are picked from RCArray, after which (3) is checked to see if they are already sufficient.

In our method, the order in which state variables are removed in the minimization procedure is based on two criteria: the *level* of each candidate state variable and, as a tie-breaker, its *relative correlation* to the current abstract model. The level of a state variable $v \in \widetilde{V} \setminus \widehat{V}$ w.r.t. $\widehat{\Omega}$ is the least number of state variables on a path in $\mathcal{C}$ connecting v to any state variable in $\widehat{V}$. The smaller its level, the more important v is deemed. The relative correlation of $v \in \widetilde{V} \setminus \widehat{V}$ equals the ratio of the number of direct predecessors of v in $\mathcal{C}$ that are in $\widehat{\Omega}$ to the total number of nodes in the COI of v. Intuitively, the larger the relative correlation of a state variable, the greater effect it will have when added to or subtracted from the current abstract model. The state variables with the larger levels and smaller relative correlations are considered of less importance and thus will be tested for deletion earlier. In this way, we can concentrate on the important candidates and keep the refined abstract model small.

3.4 Incremental SAT solver

An incremental SAT solver exploits the similarities among SAT instances that form a sequence by using the conflict clauses generated from the previous instance to guide the search for a solution to the current instance. We assume that a sequence of SAT instances $(\Delta_0, \ldots, \Delta_l)$ is specified by a sequence of pairs of sets of clauses $((A_0, B_0), \ldots, (A_l, B_l))$ such that $\Delta_0 = A_0$, $B_0 = \emptyset$, and, for $1 \leq i \leq l$, $\Delta_i = (\Delta_{i-1} \setminus B_i) \cup A_i$. That is, the second and successive instances are obtained by removing some clauses from the instances immediately preceding them and then adding some other clauses.

If Γ_i is the set of conflict clauses produced while solving Δ_i, $\gamma \in \Gamma_i$ is implied by the conjunction of the clauses in Δ_i. If $B_{i+1} = \emptyset$, it follows that γ is implied also by the conjunction of the clauses in Δ_{i+1}. Adding Γ_i to Δ_{i+1} does not change the answer to the SAT problem, but it may help the solver by precluding examination of assignments that are known not to be satisfying. If, on the other hand, $B_{i+1} \neq \emptyset$, $\gamma \in \Gamma_i$ may or may not be implied by Δ_{i+1}. However, if none of the clauses that were used in deriving γ is in B_{i+1}, then it is still correct to add γ to Δ_{i+1}. This test is conservative, but inexpensive.

Keeping all valid conflict clauses from previous instances may result in too many clauses with very many literals, which may have a negative impact on search time. In our implementation, we address this problem by modifying the conflict clause deletion strategy: we decrease the interval between successive scans of the database for clauses to be deleted. To avoid wasting time when there are few large conflict clauses generated, we use a threshold. At the end of each interval, we search for clauses to be deleted only if the number of new large conflict clauses exceeds the threshold.

The PURESAT algorithm offers several opportunities to apply an incremental SAT solver. In our implementation, we deal incrementally with two sequences of instances: one is the sequence of checks of (3) (existence of counterexamples), the other is the sequence that interleaves the checks of (5) and (6) (existence of simple paths). In the former, the instances for the same value of L form monotonically increasing subsequences, in which all conflict clauses can be reused. However, when L is incremented, the clauses accumulated for models other than $\widehat{\Omega}$ are discarded.

Two more opportunities of applying incremental SAT solving exist in the PURESAT algorithm. – in the loop of lines 5–9 in Fig. 5 and in refinement minimization. However, the average number of iterations of the loop in Fig. 5 is too low to warrant incremental solving, while the SAT instances occurring in refinement minimization are monotonically decreasing; hence, they are not well suited to incremental solving.

4 Related work

Our refinement algorithm is based on computing and analyzing the unsatisfiable core associated with the proof that there is no concrete counterexample of length L; hence, it is similar to the conflict analysis method proposed in [9]. However, our approach differs significantly from that of [9] in the following respects:

1. The authors of [9] 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 Ω.) A conflict dependency graph is built from the unsatisfiable formula 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 consequently we do not compute the failure index). Rather, unconstrained BMC instances (on a sequence of increasingly concrete models) is used for the concretization test; such BMC instances cover all the possible length-L spurious abstract counterexamples.
2. In [9], the *invisible* state variables (those in $V \setminus \widehat{V}$) 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 state variables or internal logic gates at different time steps) appearing in the unsatisfiable core are recorded in the SAT solver. However, only those invisible state variables whose *next-state-variable* literals appear in the unsatisfiable core are added to the refinement set.
3. Our refinement minimization algorithm is also somewhat different from [9]. Both methods remove redundant state variables greedily, but they differ in the order in which variables are considered.

Our algorithm is also related to that of [16]. The `pureSAT` algorithm starts from the abstract model and checks concretization for increasing counterexample lengths. By contrast, the method of [16] analyzes the concrete model; for some lengths of counterexamples it builds an abstraction from the failed proof of their existence. Both approaches check all counterexamples of a certain length at once. The main differences are:

1. We use SAT, instead of a BDD-based model checker, for the abstract model. Some abstract models are more suitable for BDD-based model checking; for instance, those that have very long simple paths. Other models have unwieldy BDDs and are more suitable for SAT-based induction proofs. Hence, neither approach dominates the other. As pointed out in Sect. 1, we expect a combination of the two approaches to provide significant advantages.
2. The `pureSAT` algorithm checks concretization incrementally. In many cases, a property can be proved without ever checking the concrete model. An additional advantage of this approach is that unsatisfiable cores come from smaller models. Finally, the abstract model provides guidance in checking the concrete one in the form of conflict clauses of the more abstract models that the incremental SAT solver may use to guide the search of counterexamples in Ω.
3. Our abstraction grows at each refinement, and we use refinement minimization to control its size, whereas the abstraction of [16] is computed from scratch each time. Refinement minimization requires repeated BMC runs; these, however, are not on the full concrete model. Recomputing the abstraction from scratch may lead to smaller abstract models. Building it incrementally allows PURESAT to check only for paths of length L instead of length up to L.

In [9, 10, 26], once counterexamples of length L are found on an abstract model, the concrete model is checked immediately to see if a real counterexample of the same length can be found. The method of [16] also checks the concrete model for increasing values of L. If the concrete model is very large, this approach may suffer from the state explosion problem. In [5], the authors propose a *layered reconstruction algorithm* in which, instead of checking directly the concrete model when abstract counterexamples are found, the state variables in $V \setminus \widehat{V}$ are divided into layers, and as each layer is added in turn, the resulting model is checked for surviving counterexamples of length L. In the PURESAT algorithm, we have adopted a similar layered approach in our incremental concretization test, with the following main differences:

1. While the approach of [5] is completely based on BDDs, our approach is entirely based on SAT. The main consequence of this choice is that we are less concerned about incremental concretization reaching Ω. We do not employ backtracking, and we limit the number of refinements during each concretization test to save time. We can also easily benefit from incremental SAT solving.
2. The computations of the layers differ in the two methods. In particular, in our algorithm we use two criteria: the level of a state variable w.r.t. the current abstract model as the primary criterion and the relative correlation as the second one.

The application of incremental SAT solving to bounded model checking was proposed in [25]. In [12], a new encoding scheme for BMC was presented that allows efficient use of incrementality. The main observation behind that encoding is that removal of unit clauses can be implemented very efficiently. Our incremental solver does not make use of this observation and is therefore closer to that of [25]. Unlike either [25] or [12], we use incremental SAT solving for abstraction refinement rather than plain BMC.

5 Experimental results

To evaluate the technique of Sect. 3, we compared four algorithms: our implementation of the BMC algorithm of [4], BMC extended with the checks for simple paths [22] (referred to as SSS), our PURESAT algorithm, and the GRAB algorithm of [26], which uses both BDDs and SAT. All four algorithms are implemented in VIS-2.0 [3] (see also `http://vlsi.colorado.edu/~vis`). We used Chaff [21] as the SAT solver. We added to it the ability to extract unsatisfiable cores and to perform incremental checks. The experiments were run under Linux on an IBM IntelliStation with a 1.7-GHz Intel Pentium IV CPU and 2 GB of RAM.

The comparison was conducted on 26 models, either from industry or from VIS verification benchmarks (see also `http://vlsi.colorado.edu/~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 quite long, SSS failed to complete, even though PURESAT finished within 1 s.

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

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

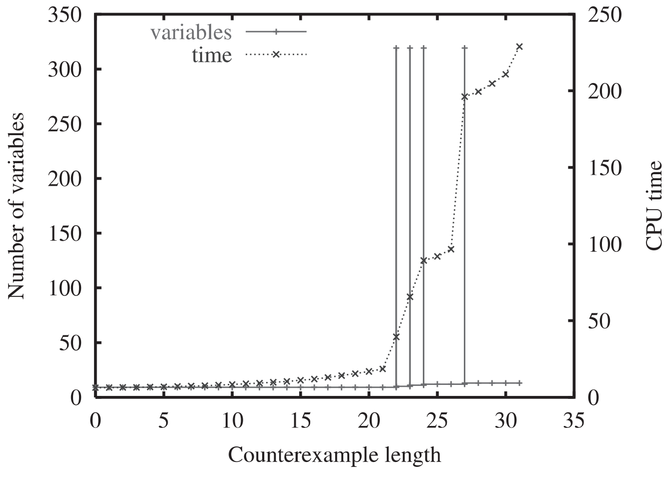
Model	Pass/ cex length	Latches in COI	BMC time	SSS time	Grab time	Grab final sz.	PureSAT time	PureSAT final sz.
lsp-p1	Pass	12	NA	>8h	**1**	3	**1**	3
D12-p1	16	48	**5**	25	14	23	22	23
D23-p1	5	85	**1**	**1**	20	21	**1**	25
D2-p1	14	94	**6**	25	180	48	10	48
D14-p1	14	96	**65**	83	>8h	(75)	1294	79
D1-p1	9	101	**1**	5	9	21	6	20
D1-p2	13	101	**2**	12	51	23	13	23
D1-p3	15	101	**3**	18	56	25	17	23
I12-p1	370	119	>8h	>8h	**2503**	16	>8h	(12)
B-p1	Pass	124	NA	>8h	**173**	18	825	18
B-p2	17	124	150	675	**93**	7	113	7
B-p3	Pass	124	NA	>8h	**223**	43	>8h	(42)
B-p4	Pass	124	NA	MO(23708)	**393**	42	>8h	(42)
D22-p1	10	140	**2**	10	720	132	10	132
D24-p1	9	147	7	10	**1**	4	**1**	4
D24-p2	Pass	147	NA	16	**3**	8	8	8
D24-p3	Pass	147	NA	**1**	20	8	3	6
D24-p4	Pass	147	NA	**1**	43	8	3	6
D24-p5	Pass	147	NA	**1**	3	5	3	8
M0-p1	Pass	221	NA	MO(2537)	**136**	16	1188	13
D5-p1	31	319	58	592	**31**	18	53	13
D18-p1	23	506	**96**	795	>8h	(99)	2721	166
D16-p1	8	531	**10**	29	92	14	20	14
D20-p1	14	562	**26**	101	>8h	(69)	7445	229
rcu-p1	Pass	2453	NA	MO(3115)	195	10	**87**	10
IU-p2	Pass	4493	MO(11331)	>8h	>8h	(6)	**1060**	14

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 most failing properties BMC is best, while PureSAT is better than Grab. For the largest models, like IU, whose COI contains 4493 state variables, PureSAT is the only one able to verify the property. Interestingly, Grab and PureSAT fail to finish similar numbers of experiments (four for Grab and three 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.

The effects on performance of incremental concretization and incremental SAT solving are shown in Table 2. In this table, IC means incremental concretization, NIC means nonincremental concretization, IS means incremental SAT, and NIS means nonincremental SAT. The sizes of the final abstract models are the same for PureSAT(NIC, NIS), PureSAT(IC, NIS), and PureSAT(IC, IS), except for D24-p3 (6, 5, 6), D24-p4 (6, 5, 6), and D24-p5 (6, 8, 8). These small differences are due to the differences in the search order followed by the SAT solver.

Table 2 makes it clear that incremental SAT solving almost always decreases CPU time. Incremental concretization helps most of the time, though for some models (e.g., D20-p1) it is not effective because most of the concretization checks end up dealing with Ω. Figure 6 and Table 3 take a closer look at the effects of incremental concretization on model D5-p1. The figure compares two runs, with and without incremental concretization. Each graph shows the numbers of state variables in the models and the CPU times as a function of the counterexample length L. When refinement occurs, both the size of the abstract model and the size of the largest $\widetilde{\Omega}$ examined during the concretization check are shown as the endpoints of a vertical line. For example, the first refinement of the initial abstraction happens for $L = 22$. Without incremental concretization (left graph) all 319 state variables are included in $\widetilde{\Omega}$, whereas the incremental check requires only the analysis of a model with 23 state variables. In both cases, $\widehat{\Omega}$ contains 9 state variables.

Checking smaller models reduces the total CPU time by over 50%, even without incremental SAT. The graphs of Fig. 6 clearly show that the time penalty is incurred when refinement takes place. Table 3 examines the nature of this time penalty; it shows that the major effect of incremental concretization is to speed up refinement minimization. This happens because smaller unsatisfiable cores are usually extracted from smaller models. (Practical methods that produce proofs of unsatisfiability do not



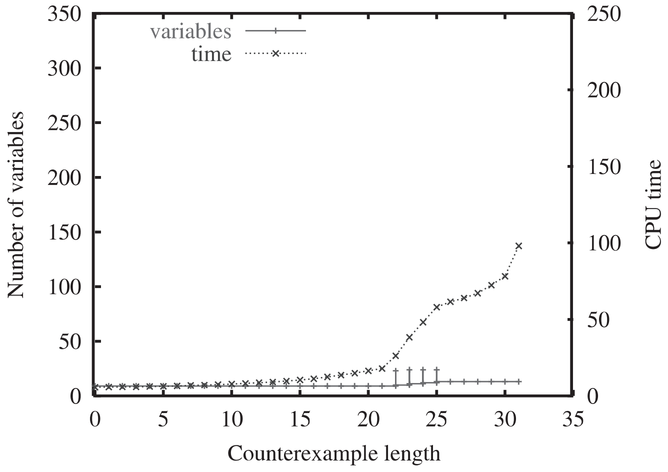


Fig. 6. Model sizes and CPU time for D5-p1 without and with incremental concretization (no incremental SAT in either case)

Table 2. Comparison between PureSAT with and without incremental concretization (IC) and with and without incremental SAT (IS). Boldface is used to highlight best CPU times

Model	NIC,NIS time	IC,NIS time	IC,IS time
lsp-p1	**1**	**1**	**1**
D12-p1	33	26	**22**
D23-p1	2	2	**1**
D2-p1	20	22	**10**
D14-p1	1393	1345	**1294**
D1-p1	9	8	**6**
D1-p2	19	23	**13**
D1-p3	28	17	**13**
I12-p1	>8h	>8h	>8h
B-p1	846	857	**825**
B-p2	316	198	**113**
B-p3	>8h	>8h	>8h
B-p4	>8h	>8h	>8h
D22-p1	15	18	**10**
D24-p1	2	2	**1**
D24-p2	**8**	9	**8**
D24-p3	**2**	5	3
D24-p4	4	**2**	3
D24-p5	**3**	**3**	**3**
M0-p1	2826	2134	**1188**
D5-p1	222	98	**53**
D18-p1	3875	2844	**2721**
D16-p1	29	24	**20**
D20-p1	**7292**	7717	7445
rcu-p1	123	**87**	**87**
IU-p2	1683	1568	**1060**

guarantee proofs that are minimal in terms of either number of clauses or variables appearing in the unsatisfiable core.) Minimization produces refinements of the same sizes for D5-p1 (one variable in all cases) with and without incremental concretization check. The time advantage comes from examining fewer candidates on smaller models.

Table 3. Comparison of refinement procedures with and without incremental concretization for D5-p1

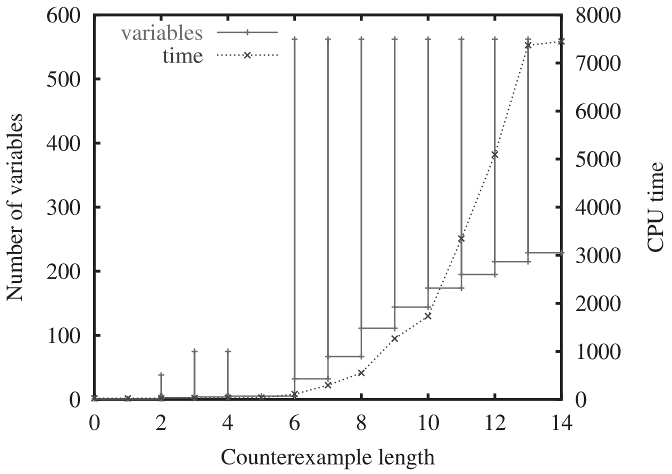
Ref. round	Variables in core NIC	IC	Ref. min. time NIC	IC	Concr. time NIC	IC
1	17	8	11.24	5.62	3.29	0.51
2	16	8	14.61	7.34	4.64	0.59
3	22	7	19.16	6.51	4.42	0.53
4	50	6	78.69	5.72	7.41	0.61

Despite the advantages of dealing with smaller unsatisfiable cores, our experiments indicate that the number of rounds in the concretization check should be kept small. We obtained the best results when the number of rounds was limited to two.

Though PureSAT appears to be reasonably robust, there are a few cases in which it performs poorly compared to competing techniques. The profiles of two such runs are shown in Fig. 7.

Model D20-p1 is representative of a group, which also includes D14-p1 and D18-p1, for which pureSAT does substantially worse than BMC and SSS on failing properties. The left graph in Fig. 7 shows that incremental concretization is not effective for D20-p1. As a consequence, refinement minimization is expensive and accounts for the majority of the CPU time. In this case, the effort put into keeping the abstract model $\widehat{\Omega}$ small is wasted: the property is eventually found to fail, and for the values of L up to 31 (the length of the shortest counterexamples), the simple path checks of (5) and (6) are not very expensive even on the concrete model – as witnessed by the CPU time of SSS.

For model b-p3, pureSAT runs out of time, while Grab proves that the property passes in less than 4 min. The right graph in Fig. 7 shows that pureSAT quickly reaches the stage where no counterexamples are found in the abstract model, but then it is unable to reach a value of L sufficient to prove the property true. In this case,



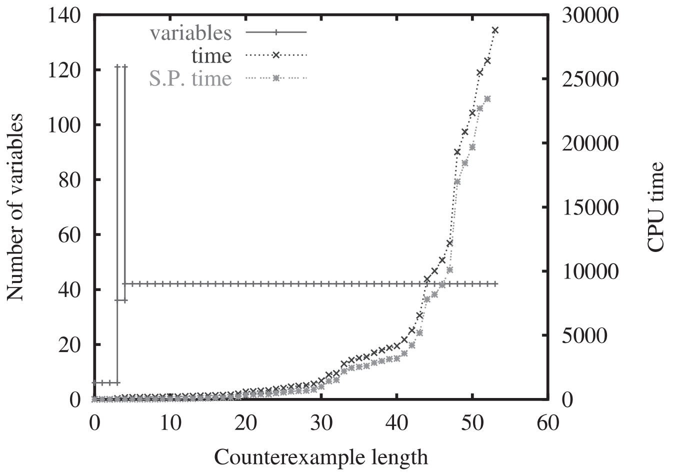


Fig. 7. Two hard problems for PURESAT: D20-p1 (*left*) and b-p3 (*right*)

the graph shows also the time (S.P. time) spent in checking (5) and (6), which clearly dominates the total CPU time. It is consistent with this observation that SSS also runs out of time on b-p3. Even though the abstract model produced by PURESAT has one fewer state variable than that of GRAB, abstraction fails to sufficiently reduce the length of the longest simple paths; hence, a BDD-based approach like that of GRAB (or [16]) is more effective.

Since checking for simple paths is sometimes expensive, one may choose to use only either (5) or (6). However, in our experiments, both tests helped establish termination for passing properties. When the test for (5), which is the most effective of the two, was disabled, the number of experiments successfully completed decreased, and no significant speedups were observed.

6 Conclusions

We have presented an abstraction refinement algorithm for model checking safety properties that uses a SAT solver as the 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 are interested in the extension of the techniques of [26] to the SAT environment. This is not an entirely trivial task since those techniques 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 [19] may boost PURESAT's performance. Improved extraction of unsatisfiable cores may speed up the abstraction minimization phase, which is currently the most time-consuming part of the algorithm. 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

1. Abdulla PA, Bjesse P, Eén N (2000) Symbolic reachability analysis based on SAT-solvers. In: 6th international conference on tools and algorithms for the construction of systems (TACAS). Lecture notes in computer science, vol 1785. Springer, Berlin Heidelberg New York, pp 411–425
2. Alpern B, Schneider FB (1985) Defining liveness. Inf Process Lett 21:181–185
3. Brayton RK et al. (1996) VIS: A system for verification and synthesis. In: Henzinger T, Alur R (eds) 8th conference on computer-aided verification (CAV'96). Lecture notes in computer science, vol 1102. Springer, Berlin Heidelberg New York, pp 428–432
4. Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: 5th international conference on tools and algorithms for construction and analysis of systems (TACAS'99), Amsterdam, The Netherlands, March 1999. Lecture notes in computer science, vol 1579. Springer, Berlin Heidelberg New York, pp 193–207
5. Barner S, Geist D, Gringauze A (2002) Symbolic localization reduction with reconstruction layering and backtracking. In: Brinksma E, Larsen KG (eds) 14th conference on computer-aided verification (CAV 2002), July 2002. Lecture notes in computer science, vol 2404. Springer, Berlin Heidelberg New York, pp 65–77
6. Baumgartner J, Kuehlmann A, Abraham J (2002) Property checking via structural analysis. In: Brinksma E, Larsen KG (eds) 14th conference on computer-aided verification (CAV'02), July 2002. Lecture notes in computer science, vol 2404. Springer, Berlin Heidelberg New York, pp 151–165
7. Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput C-35(8):677–691
8. Cousot P, Cousot R (1977) 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, pp 238–250
9. Chauhan P, Clarke E, Kukula J, Sapra S, Veith H, Wang D (2002) Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In: Aagaard MD, O'Leary JW (eds) Formal methods in computer aided design, November 2002. Lecture notes in computer science, vol 2517. Springer, Berlin Heidelberg New York, pp 33–51

10. Clarke E, Gupta A, Kukula J, Strichman O (2002) SAT based abstraction-refinement using ILP and machine learning. In: Brinksma E, Larsen KG (eds) 14th conference on computer-aided verification (CAV 2002), July 2002. Lecture notes in computer science, vol 2404. Springer, Berlin Heidelberg New York, pp 265–279
11. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge, MA
12. Eén N, Sörensson N (2003) Temporal induction by incremental SAT solving. In: 1st international workshop on bounded model checking. Electronic notes in theoretical computer science, 89(4). `http://www.elsevier.nl/locate/entcs/`
13. Goldberg E, Novikov Y (2003) Verification of proofs of unsatisfiability for CNF formulas. In: Design, automation and test in Europe (DATE'03), Munich, Germany, March 2003, pp 886–891
14. Kröning D, Strichman O (2003) Efficient computation of recurrence diameters. In: 4th international conference on verification, model checking, and abstract interpretation, January 2003. Lecture notes in computer science, vol 2575. Springer, Berlin Heidelberg New York, pp 298–309
15. Kurshan RP (1994) Computer-aided verification of coordinating processes. Princeton University Press, Princeton, NJ
16. McMillan KL, Amla N (2003) Automatic abstraction without counterexamples. In: International conference on tools and algorithms for construction and analysis of systems (TACAS'03), Warsaw, Poland, April 2003. Lecture notes in computer science, vol 2619. Springer, Berlin Heidelberg New York, pp 2–17
17. McMillan KL (1994) Symbolic model checking. Kluwer, Boston
18. McMillan KL (2002) Applying SAT methods in unbounded symbolic model checking. In: Brinksma E, Larsen KG (eds) 14th conference on computer-aided verification (CAV'02), July 2002. Lecture notes in computer science, vol 2404. Springer, Berlin Heidelberg New York, pp 250–264
19. McMillan KL (2003) Interpolation and SAT-based model checking. In: Hunt WA Jr, Somenzi F (eds) 15th conference on computer-aided verification (CAV'03), July 2003. Lecture notes in computer science, vol 2725. Springer, Berlin Heidelberg New York, pp 1–13
20. Milner R (1971) An algebraic definition of simulation between programs. In: Proceedings of the 2nd international joint conference on artificial intelligence, pp 481–489
21. Moskewicz M, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Proceedings of the design automation conference, Las Vegas, June 2001, pp 530–535
22. Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Hunt WA Jr, Johnson SD (eds) Formal methods in computer-aided design, November 2000. Lecture notes in computer science, vol 1954. Springer, Berlin Heidelberg New York, pp 108–125
23. Williams P, Biere A, Clarke EM, Gupta A (2000) Combining decision diagrams and SAT procedures for efficient symbolic model checking. In: Emerson EA, Sistla AP (eds) 12th conference on computer-aided verification (CAV'00), July 2000. Lecture notes in computer science, vol 1855. Springer, Berlin Heidelberg New York, pp 124–138
24. Wang D, Ho P-H, Long J, Kukula J, Zhu Y, Ma T, Damiano R (2001) Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In: Proceedings of the design automation conference, Las Vegas, June 2001, pp 35–40
25. Whittemore J, Kim J, Sakallah K (2001) SATIRE: A new incremental satisfiability engine. In: Proceedings of the design automation conference, Las Vegas, June 2001, pp 542–545
26. Wang C, Li B, Jin H, Hachtel GD, Somenzi F (2003) Improving Ariadne's bundle by following multiple threads in abstraction refinement. In: Proceedings of the international conference on computer-aided design, November 2003, pp 408–415
27. Zhang L, Malik S (2003) Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications. In: Design, automation and test in Europe (DATE'03), Munich, Germany, March 2003, pp 880–885