Model Checking with Boolean Satisfiability

Joao Marques-Silva

School of Electronics and Computer Science
University of Southampton
jpms@ecs.soton.ac.uk

Abstract

The evolution of SAT algorithms over the last decade has motivated the application of SAT to model checking, initially through the use of SAT in bounded model checking and, more recently, in unbounded model checking. This paper provides an overview of modern SAT algorithms, SAT-based bounded model checking and some of the most promising approaches for unbounded model checking, namely induction and interpolation. Moreover, the paper details a number of techniques that have proven effective in using SAT solvers in model checking.

1 Introduction

Boolean Satisfiability has been the subject of remarkable improvements over the last decade [25,26,31,17,14]. From the original algorithm proposed by M. Davis and H. Putnam in the mid 60s [13,12], SAT algorithms have evolved to integrate extremely effective techniques, including clause learning and non-chronological backtracking [25,26], advanced data structures and adaptive branching heuristics [31], and search restart policies [18]. The improvements in SAT algorithms motivated a number of practical applications of SAT, one of the most successful being symbolic model checking of state transition systems [5,6,8,28,29,34,35]. The success of SAT-based symbolic model checking motivated its widespread adoption by industry, and a number of vendors have included SAT-based Model Checking in their tools.

The use of SAT in symbolic model checking was first proposed in the form of Bounded Model Checking (BMC) [5], where a counterexample is searched for increasing unfoldings of a finite state automaton. The original BMC work has been shown to be extremely useful for finding counterexamples but, unless the recurrence (or the reachability) diameter of the automaton is known [4], the BMC procedure is incomplete. Different solutions have been proposed for ensuring the completeness of BMC [34,8,21,19,29], with interpolants [29] and induction [34] being arguably the most promising.

This paper provides an overview of the advances in SAT-based model checking, including bounded and unbounded model checking, emphasizing the most effective techniques.

The paper is organized as follows. The next section provides a brief overview of model checking for finite state transition systems. Section 2 introduces standard SAT definitions and overviews modern SAT solvers. Sections 3 and 4 describe, respectively, unsatisfiability proofs and interpolants. Afterwards, Section 6 reviews SAT-based bounded model checking, and Section 7 reviews the most widely used approaches for unbounded model checking. The paper concludes in Section 8.

2 Propositional Satisfiability

This section introduces the notation used throughout, and reviews basic concepts in Boolean Satisfiability (SAT).

2.1 Definitions

Propositional formulas are defined over finite sets of Boolean variables $X = \{x_1, x_2, \ldots\}$, $X_1 = \{x_{11}, x_{12}, \ldots\}$, $X_2 = \{x_{21}, \ldots\}$, etc., where each variable can be assigned value 1 (True) or 0 (False). In what follows propositional formulas are represented by $\psi_1, \psi_2, \ldots$. When relevant other subscripts can be used, e.g. ψ_a, ψ_b, etc. The term *predicate* is used to denote the propositional formula representing the characteristic function of a set, function or relation. Hence, in the paper the terms *predicate* and *propositional formula* are used interchangeably. For specific cases, letters and names representing sets or relations are also used for denoting the associated predicates, examples include I, T, F, P, Q and Bmc.

Most SAT algorithms assume that formulas are represented in conjunctive normal form (CNF). A CNF formula φ consists of a conjunction of clauses ω, each of which consists of a disjunction of literals. A literal is either a variable x_i or its complement $\neg x_i$. A CNF formula can also be viewed as a set of clauses, and each clause can be viewed as a set of literals. Throughout this paper, the representation used will be clear from the context.

When referring to propositional formulas in CNF, we associate with each propositional formula $\psi_a(X_a)$ a CNF formula $\varphi_a(X_a, U_a)$, where U_a denotes a set of auxiliary Boolean variables. Formulas in CNF consist of a conjunction of clauses (each clause represented by ω_i), where each clause consists of a disjunction of literals (represented by l_j). When used in an expression, a propositional formula ψ is interpreted as a predicate, and so corresponds to $\psi = 1$. Similarly, when the propositional formula $\neg\psi$ is used in an expression, it corresponds to $\psi = 0$.

It will also be necessary to map propositional formulas from one set of variables to another set of variables. The notation $\psi(Y/Y_k)$ is used to denote that the propositional formula ψ, defined over the set of variables Y, is mapped into the set of variables Y_k. Moreover, variables associated with states are preferably represented as set Y, and Y_k when referring to the state variables in time step k.

Transition relations are often represented in propositional logic, for example in the form of Boolean circuits. Mapping propositional formulas to CNF is performed in space linear in the size of the original formula, by using additional variables. A number of alternatives exist, the most widely used are due to Tseitin [36] and Plaisted and Greenbaum [32]. In the algorithms described below, any encoding to CNF assumes a set of new auxiliary variables, required for mapping from a propositional representation to CNF. In terms of notation, Boolean circuit variables are preferably represented as sets X or W, respectively X_k and W_k for variables in time step k, and finally auxiliary variables used in the CNF representation are preferably represented as sets W or Z.

2.2 SAT Algorithms

Despite the vast number of alternative algorithms for SAT, the most effective for model checking are based on backtrack search with clause learning. These algorithms are referred to as *conflict-driven clause learning* (CDCL) SAT solvers, and are overviewed in this section. CDCL SAT solvers are derived from the well-know DPLL SAT algorithm, consisting of the algorithm described in [12], but including techniques first proposed in [13].

In the context of search algorithms for SAT, variables can be *assigned* a logic value, either 0 or 1. Alternatively, variables may also be *unassigned*. Assignments to the problem variables can be defined as a function $\nu : X \rightarrow \{0, u, 1\}$, where u denotes an *undefined* value used when a variable has not been assigned a value in $\{0, 1\}$. Given an assignment ν, if all variables are assigned a value in $\{0, 1\}$, then ν is referred to as a *complete assignment*. Otherwise it is a *partial assignment*.

Assignments serve for computing the values of literals, clauses and the complete CNF formula, respectively, l^ν, ω^ν and φ^ν. A total order is defined on the possible assignments, $0 < u < 1$. Moreover, $1 - u = u$. As a result, the following definitions apply:

$$l^\nu = \begin{cases} \nu(x_i) & \text{if } l = x_i \\ 1 - \nu(x_i) & \text{if } l = \neg x_i \end{cases} \tag{1}$$

$$\omega^\nu = \max \{ l^\nu \mid l \in \omega \} \tag{2}$$

$$\varphi^\nu = \min \{ \omega^\nu \mid \omega \in \varphi \} \tag{3}$$

The assignment function ν will also be viewed as a set of tuples (x_i, v_i), with each x_i distinct and $v_i \in \{0, 1\}$. Adding a tuple (x_i, v_i) to ν corresponds to assigning v_i to x_i, such that $\nu(x_i) = v_i$. Removing a tuple (x_i, v_i) from ν, with $\nu(x_i) \neq u$, corresponds to assigning u to x_i.

Clauses are characterized as *unsatisfied*, *satisfied*, *unit* or *unresolved*. A clause is unsatisfied if all its literals are assigned value 0. A clause is satisfied if at least one of its literals is assigned value 1. A clause is unit if all literals but one are assigned value 0, and the remaining literal is unassigned. Finally, a clause is unresolved if it is neither unsatisfied, nor satisfied, nor unit.

A key procedure in SAT solvers is the *unit clause rule* [13]: if a clause is unit, then its sole unassigned literal must be assigned value 1 for the clause to be satisfied. The iterated application of the unit clause rule is referred to as *unit propagation* or *Boolean constraint propagation* (BCP) [37]. In modern CDCL solvers, as in most implementations of DPLL, logical consequences are derived with unit propagation. Unit propagation is applied after each branching step (and also during preprocessing), and is used for identifying variables which must be assigned a specific Boolean value. If an unsatisfied clause is identified, a *conflict* condition is declared, and the algorithm backtracks.

In CDCL SAT solvers, each variable x_i is characterized by a number of properties, including the *value*, the *antecedent* (also referred to as the *reason*), and the *decision level*, denoted respectively by $\nu(v_i) \in \{0, u, 1\}$, $\alpha(x_i) \in \varphi \cup \{\text{NIL}\}$, and $\delta(x_i) \in \{-1, 0, 1, \ldots, |X|\}$. A variable x_i that is assigned a value as the result of applying the unit clause rule is said to be *implied*. The unit clause ω used for implying variable x_i is said to be the antecedent of x_i, $\alpha(x_i) = \omega$. For variables that are decision variables or are unassigned, the antecedent is NIL. Hence, antecedents are only defined for variables whose value is implied by other assignments. The decision level of a variable x_i denotes the depth of the decision tree at which the variable is assigned a value in $\{0, 1\}$. The decision level for an unassigned variable x_i is -1, $\delta(x_i) = -1$. The decision level associated with variables used for branching steps (i.e. *decision assignments*) is specified by the search process, and denotes the current depth of the *decision stack*. Hence, a variable x_i associated with a decision assignment is character-

$$\begin{tikzcd}
x_{31} = 0 \,@\, 3 \arrow[dr, "\omega_1"] & & & & \\
 & x_2 = 0 \,@\, 5 \arrow[dr, "\omega_3"] & & x_5 = 0 \,@\, 5 \arrow[dr, "\omega_6"] & \\
x_1 = 0 \,@\, 5 \arrow[ur, "\omega_1"] \arrow[dr, "\omega_2"'] & & x_4 = 1 \,@\, 5 \arrow[ur, "\omega_4"] \arrow[dr, "\omega_5"'] & & \kappa \\
 & x_3 = 0 \,@\, 5 \arrow[ur, "\omega_3"'] & & x_6 = 0 \,@\, 5 \arrow[ur, "\omega_6"'] & \\
 & & x_{21} = 0 \,@\, 2 \arrow[ur, "\omega_5"'] & &
\end{tikzcd}$$

Fig. 1. Implication graph for example 1

ized by having $\alpha(x_i) = \text{NIL}$ and $\delta(x_i) > 0$. Alternatively, the decision level of x_i with antecedent ω is given by:

$$\delta(x_i) = \max(\{0\} \cup \{\delta(x_j) \,|\, x_j \in \omega \wedge x_j \neq x_i\}) \tag{4}$$

The notation $x_i = v \,@\, d$ is used to denote that $\nu(x_i) = v$ and $\delta(x_i) = d$. Moreover, the decision level of a literal is defined as the decision level of its variable, $\delta(l) = \delta(x_i)$ if $l = x_i$ or $l = \neg x_i$.

During the execution of a DPLL-style SAT solver, assigned variables as well as their antecedents define a directed acyclic graph $I = (V_I, E_I)$, referred to as the *implication graph* [25].

The vertices in the implication graph are defined by all assigned variables and one special node κ, $V_I \subseteq X \cup \{\kappa\}$. The edges in the implication graph are obtained from the antecedent of each assigned variable: if $\omega = \alpha(x_i)$, then there is a directed edge from each variable in ω, other than x_i, to x_i. If unit propagation yields an unsatisfied clause ω_j, then a special vertex κ is used to represent the unsatisfied clause. In this case, the antecedent of κ is defined by $\alpha(\kappa) = \omega_j$.

Example 1 (Implication Graph) *Consider the CNF formula:*

$$\begin{aligned} \varphi_1 &= \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5 \wedge \omega_6 \\ &= (x_1 \vee x_{31} \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge \\ &\quad (\neg x_4 \vee \neg x_5) \wedge (x_{21} \vee \neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6) \end{aligned} \tag{5}$$

Assume decision assignments $x_{21} = 0@2$ and $x_{31} = 0@3$. Moreover, assume the current decision assignment $x_1 = 0@5$. The resulting implication graph is shown in figure 1, and yields a conflict because clause $(x_5 \vee x_6)$ becomes unsatisfied.

In the presence of conflicts, modern CDCL SAT solvers learn new clauses [25]. Learnt clauses are then used for implementing non-chronological backtracking [25].

Example 2 (Clause Learning) *For the CNF formula of Example 1, a new clause* $(x_1 \vee x_{31} \vee x_{21})$ *is learnt by analyzing the causes of the conflict [25]. The structure of the conflicts can be exploited by identifying* unique implication points *(UIPs) [25]. For this example,* $x_4 = 1@5$ *is a UIP, and so the learnt clause would be* $(\neg x_4 \vee x_{21})$*. The first CDCL SAT solver [25] used UIPs to learn more clauses, and so it would learn* $(x_1 \vee x_{31} \vee x_{21})$*,* $(\neg x_4 \vee x_{21})$ *and* $(x_1 \vee x_{31} \vee x_4)$*. Recent CDCL SAT solvers [31,14] stop clause learning at the first UIP, and so only learn clause* $(\neg x_4 \vee x_{21})$*.*

Algorithm 1 shows the standard organization of a CDCL SAT solver, which essentially follows the organization of DPLL. With respect to DPLL, the main differences are the call to function ConflictAnalysis each time a conflict is identified, and the call to Backtrack when backtracking takes place. Moreover, the Backtrack procedure allows for backtracking non-chronologically.

In addition to the main CDCL function, the following auxiliary functions are used:

- UnitPropagation consists of the iterated application of the unit clause rule. If an unsatisfied clause is identified, then a conflict indication is returned.
- PickBranchingVariable consists of selecting a variable to assign and the respective value.
- ConflictAnalysis consists of analyzing the most recent conflict and learning a new clause from the conflict. The organization of this procedure is described elsewhere [25].
- Backtrack backtracks to the decision level computed by ConflictAnalysis.
- AllVariablesAssigned tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable. An alternative criterion to stop execution of the algorithm is to check whether all clauses are satisfied. However, in modern SAT solvers that use lazy data structures, clause state cannot be maintained accurately, and so the termination criterion must be whether all variables are assigned.

Arguments to the auxiliary functions are assumed to be passed by reference. Hence, φ and ν are supposed to be modified during execution of the auxiliary functions.

The typical CDCL algorithm shown does not account for a few often used techniques as well as key implementation details. A state of the art SAT solver implements the typical CDCL algorithm shown above, and also uses

Algorithm 1 Typical CDCL algorithm

```
CDCL($\varphi, \nu$)
1  if (UnitPropagation($\varphi, \nu$) == CONFLICT)
2     then return UNSAT
3  $dl \leftarrow 0$                                  ▷ Decision level
4  while (not AllVariablesAssigned($\varphi, \nu$))
5        do $(x, v)$ = PickBranchingVariable($\varphi, \nu$) ▷ Decide stage
6           $dl \leftarrow dl + 1$                    ▷ New decision: update decision level
7           $\nu \leftarrow \nu \cup \{(x, v)\}$
8           ▷ Deduce stage
9           if (UnitPropagation($\varphi, \nu$) == CONFLICT)
10             then $\beta$ = ConflictAnalysis($\varphi, \nu$) ▷ Diagnose stage
11                  if ($\beta < 0$)
12                     then return UNSAT
13                     else Backtrack($\varphi, \nu, \beta$)
14                          $dl \leftarrow \beta$  ▷ Backtracking: update decision level
15 return SAT
```

the following techniques:

- Identification of unique implication points (UIPs) [25] (see Example 2).
- Memory efficient lazy data structures [31]. Lazy data structures required essentially no effort during backtracking. Moreover, during propagation, only a fraction of a variable's clauses are updated.
- Adaptive branching heuristics, usually derived from the Variable State Independent Decaying Sum (VSIDS) heuristic [31]. The VSIDS heuristic associates a weight with each variable. The weights are regularly divided by a constant, and each is incremented when the variable participates in a conflict.
- Integration of search restarts, by using some completeness criterion [18,1].
- Implementation of clause deletion policies [17].

Because modern backtrack search SAT solvers learn clauses, it is straightforward to track all the learnt clauses, and use these clauses for constructing a resolution refutation (or unsatisfiability proof) of the original formula [38].

3 Resolution Proofs

This section addresses a number of SAT-related concepts, which are required for the use of interpolants in SAT-based model checking. For this purpose, we review *proof traces*, *unsatisfiable cores* and *unsatisfiability proof.*

As mentioned in the previous section, CDCL SAT solvers learn clauses. For

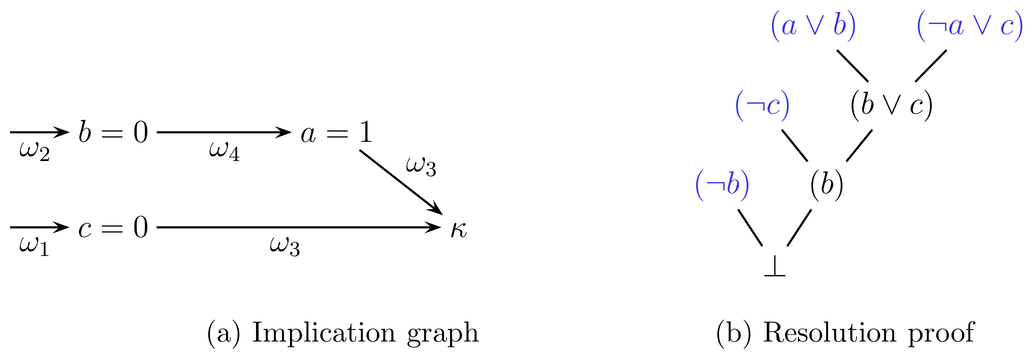


Fig. 2. Example of resolution proof

unsatisfiable instances, the original clauses and the learned clauses can be used to generate a resolution-based unsatisfiability proof [38]. Modern SAT solvers can be instructed for generating a *proof trace*, which associates with each learned clause ω, all the clauses that explain the creation of ω [38]. Observe that it suffices to associate only the clauses used for learning the clauses, as each resolution step is a *trivial resolution step* [2]. For unsatisfiable instances of SAT, the set of clauses associated with each learnt clause represents part of a resolution proof.

Given a proof trace Γ, where the final traced clause is the empty clause $\perp$, we can identify, in linear time on the size of the proof trace, a subset of the original set of clauses which is itself unsatisfiable [38]. This subset is referred to as an *unsatisfiable core*.

Moreover, and given a proof trace Γ, generated by a SAT solver, it is possible to create a resolution-based unsatisfiability proof in time and size linear on the size of the proof trace. For the purposes of the remainder of the paper, unsatisfiability proofs are represented as graphs.

Definition 1 (Unsatisfiability Proof [29]) *A proof of unsatisfiability Π for a set of clauses φ is a directed acyclic graph (V_Π, E_Π), where V_Π is a set of clauses, such that:*

- *For every $\omega \in V_\Pi$, either*
  - *$\omega \in \varphi$, and ω is a root, or*
  - *ω has two predecessors, ω_1 and ω_2, such that ω is the resolvent of ω_1 and ω_2 (the variable v used for resolving ω_1 with ω_2 is referred to as the* pivot *variable of the resolution step), and*
- *the empty clause $\perp$ is the unique leaf.*

Example 3 *Consider the CNF formula $\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 = (\neg c) \wedge (\neg b) \wedge (\neg a \vee c) \wedge (a \vee b)$. Figure 2(a) shows the single implication graph (at decision level 0), from which the empty clause is learnt. Observe that edges labelled ω_1 and ω_2 do not have input vertices, because each is a unit clause. One possible proof trace is the sequence of clauses $\langle \omega_3, \omega_4, \omega_1, \omega_2 \rangle$, denoting* *the order in which clauses are analyzed during clause learning. Figure 2(b) shows the associated resolution proof, obtained from the proof trace by trivial resolution steps [2]. For this example, ω_3 is resolved with ω_4, the result of which is then resolved with ω_1 and, finally, the result is resolved with ω_2.*

4 Craig Interpolants

Assume a propositional formula $\psi_A(Y, X)$, defined over the sets of variables Y and X, and a propositional formula $\psi_B(Y, W)$, defined over the sets of variables Y and W. If $\psi_A(Y, X) \wedge \psi_B(Y, W)$ is unsatisfiable, then there exists a propositional formula $\psi_P(Y)$, such that:

(1) $\psi_P(Y)$ is defined over the set of common variables Y.
(2) $\psi_A(Y, X) \rightarrow \psi_P(Y)$ is a tautology.
(3) $\psi_B(Y, W) \wedge \psi_P(Y)$ is unsatisfiable.

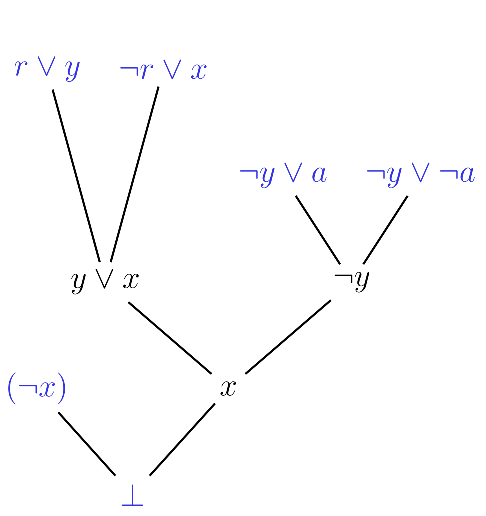
The propositional formula $\psi_P(Y)$ is referred to as an *interpolant* for $\psi_A(Y, X)$ and $\psi_B(Y, W)$ [11]. Recent work has shown that an interpolant can be constructed in linear time on the size of a resolution refutation of $\psi_A(Y, X) \wedge \psi_B(Y, W)$ [22,33].

In what follows we outline McMillan's interpolant construction [29], even though Pudlák's construction [33] could also be considered. Regarding the propositional formulas $\psi_A(Y, X)$ and $\psi_B(Y, W)$, and associated CNF formulas, respectively $\varphi_A(Y, X, U)$ and $\varphi_B(Y, W, V)$, the variables in set Y are referred to as *global* variables, whereas the variables in sets X and U are *local* to $\varphi_A(Y, X, U)$, and the variables in sets W and V are *local* to $\varphi_B(Y, W, V)$. Further, let $g(\omega)$ denote the disjunction of literals corresponding to global variables in clause ω (recall that, when necessary, the disjunction of literals can also be interpreted as a set of literals).

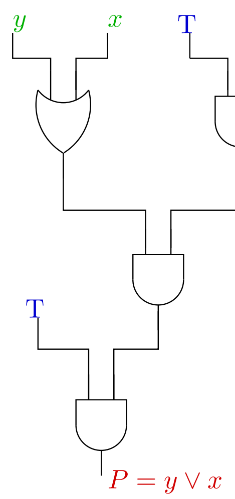
The interpolant is obtained from the resolution proof. The literals from variables common to ψ_A and ψ_B are kept in the clauses of ψ_A. Each clause in ψ_B is replaced with $\top$. Moreover, each resolution node produces a gate. For resolution nodes corresponding to variables that only exist in ψ_A, then an OR gate is used. Otherwise, an AND gate is used. The generation of the interpolant can be formalized as follows.

Definition 2 (Interpolant [29]) *Let (φ_A, φ_B) be a pair of clause sets and let Π be a proof of unsatisfiability of $\varphi_A \cup \varphi_B$, with leaf vertex $\perp$. For each vertex $\omega \in V_\Pi$, let ψ_ω be a Boolean formula, such that:*

- *If ω is a root then*



(a) Resolution proof



(b) Interpolant

Fig. 3. Computation of interpolant

  · *if* $\omega \in \varphi_A$ *then* $\psi_\omega = g(\omega)$,
  · *else* $\psi_\omega =$ True
- *else, let* ω_1, ω_2 *be the predecessors of* ω *and let* v *be their pivot variable*
  · *if* v *is local to* φ_A, *then* $\psi_\omega = \psi_{\omega_1} \vee \psi_{\omega_2}$,
  · *else* $\psi_\omega = \psi_{\omega_1} \wedge \psi_{\omega_2}$

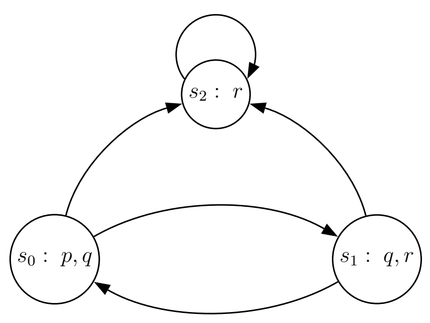
The Π*-interpolant of* (φ_A, φ_B), *denoted* $\textsc{Itp}(\Pi, \varphi_A, \varphi_B)$ *is* $\psi_\perp$.

A simple proof that $\psi_\perp$ is indeed an interpolant for the pair ψ_A and ψ_B can be found in [33,29]. The rationale is that $\psi_\perp$ is either 0 or, if it takes value 1, then ψ_B must take value 0. Moreover, it is simple to conclude that the interpolant $\textsc{Itp}(\Pi, \varphi_A, \varphi_B)$ has size linear on the size of the unsatisfiability proof.

Example 4 *Figure 3 illustrates the computation of an interpolant, with* $A = (r \vee y) \wedge (\neg r \vee x)$ *and* $B = (\neg y \vee a) \wedge (\neg y \vee \neg a) \wedge (\neg x)$. *The three vertices associated with* B *are set to* T, *whereas each vertex corresponding to each clause* ω *in* A *is set* $g(\omega)$.

5 Model Checking

Given a set of propositional symbols Σ, a Kripke structure is defined as a 4-tuple $\mathcal{M} = (S, I, T, L)$, where S is a (countable) set of states, $I \subseteq S$ is a set of initial states, $T \subseteq S \times S$ is a transition relation, and $L : S \to \mathcal{P}(\Sigma)$ is a labelling function, where $\mathcal{P}(\Sigma)$ denotes the powerset over the set of propositional symbols. For the purposes of this paper, S is assumed to be finite.



$$\begin{aligned} S &= \{s_0, s_1, s_2\} \\ I &= \{s_0\} \\ T &= \{(s_0, s_1), (s_1, s_0), (s_0, s_2), (s_1, s_2), (s_2, s_2)\} \\ L &= \{(s_0, \{p, q\}), (s_1, \{q, r\}), (s_2, \{r\})\} \end{aligned}$$

Fig. 4. State transition system

Temporal logics allow describing properties of systems. Two propositional temporal logics are widely used: Linear-Time Logic (LTL) and Computation-tree Logic (CTL) [9]. This paper assumes LTL, but the algorithms described are also valid for CTL.

LTL is defined as an extension of propositional logic. Besides the standard connectives of propositional logic (i.e. $\wedge$, $\vee$, $\rightarrow$ $\neg$, $\top$, $\bot$, and parentheses), the following temporal operators are defined:

- X denotes the *next* operator, and is used to describe a property in the next time step.
- F denotes some *future* state, and is used to describe a property in some future time step.
- G denotes *all* future states, and is used to describe a property that holds in all future time steps (or *globally*).
- Other operators include the until (U) operator, the weak until (W) operators and the release (R) operator.

The semantics of LTL can be easily described from the syntax [20]. Given a Kripke structure $\mathcal{M} = (S, I, T, L)$, model checking consists in deciding whether φ holds in some state $s \in S$. We say that $\mathcal{M}, s \vDash \varphi$ holds if, given $\mathcal{M}$, φ holds in some state $s \in S$.

Example 5 *Consider the state transition system shown in Figure 4. The set of propositional symbols is $\Sigma = \{p, q, r\}$ and the set of states is $S = \{s_0, s_1, s_2\}$. The transition relation is given by $T = \{(s_0, s_1), (s_1, s_0), (s_0, s_2), (s_1, s_2), (s_2, s_2)\}$. Finally the labelling function is given by $\{(s_0, \{p, q\}), (s_1, \{q, r\}), (s_2, \{r\})\}$.*

Consider the LTL formula $\varphi = \mathrm{F}\ r$. We can conclude that this formula is true in state s_0, because in any sequence of states starting from s_0 it is eventually true that r holds. Hence, we say that $\mathcal{M}, s_0 \vDash \varphi$.

Model checking algorithms can be characterized as explicit-state or implicit-state (or symbolic) [9]. Explicit state model checking algorithms represent explicitly the states of the transition relation, whereas symbolic model checking algorithms do not. Initial symbolic model checking algorithms were based on

Binary Decision Diagrams (BDDs) [27]. Over the last decade, a number of alternatives based on Boolean Satisfiability (SAT) have been proposed [5,34,28,29].

Even though LTL is a relatively rich logic, most work on SAT-based model checking assumes *safety* properties G ψ_S, where ψ_S is a purely propositional formula. On the one hand, most practical model checking problems need to guarantee that for all time steps, nothing wrong happens (e.g. an invalid or error state is not reached), and so G ψ_S captures this condition. On the other hand, it is unclear how to guarantee completeness with SAT-based approaches for arbitrary temporal formulas. Moreover, all existing complete SAT-based model checking solutions assume safety properties.

The remainder of the paper considers model checking of LTL safety properties G ψ_S. For simplicity, a Kripke structure $\mathcal{M} = (S, I, T, L)$ with a finite set of states will be represented by the 3-tuple $M = (I, T, F)$, where I is a predicate representing the initial states, T is a predicate representing the transition relation, and F is a predicate representing the failing property (i.e. $F = \neg\psi_S$), defined on state variables. Moreover, the predicates I, T or F assume the underlying Kripke structure $\mathcal{M} = (S, I, T, L)$ and associated target formula ψ_S. As mentioned above, for simplicity the predicates associated with the characteristic functions of the components of the Kripke structure are represented with the same letters, I, T and F.

6 SAT-Based Bounded Model Checking

This section overviews the work on using SAT in bounded model checking (BMC). As mentioned earlier in the paper, bounded model checking focuses on safety properties G ψ_S, denoting that ψ_S must hold globally. The solution to address this problem with SAT is to consider the complement $F\ \neg\psi_S$, representing the condition that ψ_S will not hold in some state reachable from an initial state. The condition $\neg\psi_S$ will be referred to as the failing property, and represented with a predicate F. Bounded model checking consists of iteratively unfolding the transition relation, while checking whether the failing property holds. The generic Boolean formula associated with SAT-based BMC is the following [4,5,35]:

$$\textsc{Bmc}(M, r, s, t) = I(Y_r) \wedge \left(\bigwedge_{r \le i < t} T(Y_i, Y_{i+1}) \right) \wedge \left(\bigvee_{s \le i \le t} F(Y_i) \right) \tag{6}$$

r denotes a lower index in the number of the states considering. For now $r = 0$, but this will change when describing interpolant-based UMC algorithms. s denotes the time step from which the failing property is checked for. Finally, t

Algorithm 2 Organization of BMC

```
BMC($M = (I, T, F)$, $\lambda$, $\iota$, $\mu$)
1  $j \leftarrow 0$
2  $k \leftarrow \lambda$
3  while $k \leq \mu$
4      do $\varphi \leftarrow$ CNF(BMC($M, 0, j, k$), $W$)
5         if SAT($\varphi$)
6            then return false ▷ Found counterexample
7         $k \leftarrow k + \iota$
8  return true
```

denotes the last time step for the unfolding considered. Formula (6) represents the unfolding of the transition relation for $t - r$ time steps, where $I(Y_r)$ represents the initial state (at time step r), $T(Y_i, Y_{i+1})$ represents the transition relation between states Y_i and Y_{i+1}, and $F(Y_i)$ represents the failing property at time step i. Given the Boolean formula $\textsc{Bmc}(M, r, s, t)$, it is straightforward to generate a CNF formula φ, by applying either Tseitin's [36] transformation or the structure preserving transformation [32], and by using additional auxiliary Boolean variables. This formula can then be evaluated by a SAT solver.

The typical organization of BMC for safety properties is illustrated in Algorithm 2. The details regarding the sets of variables associated with each propositional formula are omitted, but are clear from the context. Constants λ, ι, and μ represent, respectively, a lower bound on the unfolding of the transition relation, the unfolding increment to be used at each iteration of the algorithm, and an upper bound on the unfolding of the transition relation. Experimental evidence has confirmed SAT-based BMC to be an extremely competitive technique, that has been widely applied in industrial settings [4,10,15].

In order to describe the use of interpolants in UMC, the following predicates are extensively used:

$$\textsc{Unfold}(M, r, s) = I(Y_r) \wedge \left(\bigwedge_{r \leq i < s} T(Y_i, Y_{i+1}) \right) \tag{7}$$

Equation (7) represents the unfolding of the transition system for $s - r$ time steps, with $s \geq r$. The first state (represented with state variables Y_r) must be one of the initial states. Each set of variables Y_i represents a state reached after $i - r$ time steps, starting from one of the initial states.

$$\textsc{Tran}(M, s, t) = \bigwedge_{s \leq i < t} T(Y_i, Y_{i+1}) \tag{8}$$

Equation (8) captures the transition relation for $t - s$ time steps, with $t \geq s$.

$$\textsc{Fail}(M, u, v) = \left(\bigwedge_{u \leq i < v} T(Y_i, Y_{i+1}) \right) \wedge \left(\bigvee_{u \leq i \leq v} F(Y_i) \right) \tag{9}$$

Equation (9) represents the transition relation for the last $v - u$ time steps, with $v \geq u$, during which the failing property is checked for.

Hence, we can express the BMC formula in terms of these predicates:

$$\begin{aligned} \text{BMC}(M, r, s, t) &= \textsc{Unfold}(M, r, s) \wedge \textsc{Fail}(M, s, t) \\ &= \textsc{Unfold}(M, r, r) \wedge \textsc{Tran}(M, r, s) \wedge \textsc{Fail}(M, s, t) \end{aligned} \tag{10}$$

7 SAT-Based Unbounded Model Checking

A key difficulty with BMC is its inability for proving that there is no counterexample for a given safety property G ψ_S. Unless the recurrence (or the reachability) diameter [4] of an automaton is known, it is not possible to precompute the value of the upper bound (μ) used in Algorithm 2; in the case the recurrence diameter is known, BMC becomes complete. In general the recurrence diameter of an automaton is not known, and so BMC is incomplete. As a result, in recent years different approaches have been proposed for ensuring the completeness of SAT-based model checking. We refer to these approaches as *Unbounded Model Checking* (UMC) [28,29]. The first UMC SAT-based approach was proposed by Sheeran et al. in [34] and extended in [6]. Additional techniques include [8,28,16,30,29,19]. The induction-based approach of Sheeran et al. [34] requires unfolding the transition relation for the largest simple path (i.e. a path without cycles) between any two reachable states in the worst case. However, the largest simple path between any two reachable states can be exponentially larger than the reachability diameter. Alternatively, Chauhan et al. [8] and Glusman et al. [16] propose refinement techniques based on elimination of invalid counterexamples provided by an abstracted representation of the system. Another approach based on iterative abstraction is proposed by Gupta et al. in [19]. More recently, McMillan and Amla [30] propose the use of proof-based abstraction, even though the proposed approach is not fully SAT-based. Finally, McMillan proposed the use of interpolants [29], which requires unfolding the transition relation by at most the largest shortest path between any two states.

The next sections summarize the most widely used unbounded model checking approaches, namely induction [34] and the use of interpolants [29].

Algorithm 3 Induction-based UMC algorithm

```
UMC($M = (I, T, F)$)
1  $k \leftarrow 0$
2  while true
3        do if not $\textsc{Sat}(I(Y_0) \wedge \textsc{LoopFree}(M, 0, k))$ ▷ Check fixed point
4              then return true
5           if not $\textsc{Sat}(\textsc{LoopFree}(M, 0, k) \wedge \textsc{Fail}(M, k, k))$ ▷ Check fixed point
6              then return true
7           if $\textsc{Sat}(I(Y_0) \wedge \textsc{Tran}(M, 0, k) \wedge \textsc{Fail}(M, k, k))$
8              then return false ▷ Found counterexample
9           $k \leftarrow k + 1$
```

7.1 Induction-Based Unbounded Model Checking

Sheeran et al. proposed the first complete approach for SAT-based UMC [34]. In order to present this UMC solution, let us introduce a predicate that holds true for paths with no repeated states in the transition system:

$$\textsc{LoopFree}(M, r, s) = \textsc{Tran}(M, r, s) \wedge \bigwedge_{r \leq i < j \leq s} (Y_i \neq Y_j) \tag{11}$$

Algorithm 3 outlines the induction-based UMC algorithm of Sheeran et al. The existence of a counterexample is tested in line 7. Moreover, the induction step is tested in lines 3 and 5. If for a given k there can be no loop free paths of length k starting from an initial state, and a counterexample has not yet been found, then a counterexample *cannot* be found. Similarly, if for a given k there can be no loop free paths of length k reaching a failing property, and a counterexample has not yet been found, then a counterexample *cannot* be found. Further improvements to induction-based UMC, including the use of incremental SAT, are described in [15].

7.2 Interpolant-Based Unbounded Model Checking

Recent work on SAT-based unbounded model checking has addressed the use of interpolants [29], with promising experimental results. This section reviews McMillan's interpolant-based UMC algorithm [29].

The definition of the BMC propositional formula is modified slightly with respect to (6):

$$\begin{aligned} \textsc{Pref}(M, r, s) &= I(Y_r) \wedge \left(\textstyle\bigwedge_{r \leq i < s} T(Y_i, Y_{i+1})\right) \\ &= \textsc{Unfold}(M, r, s) \end{aligned} \tag{12}$$

Algorithm 4 Interpolant-based UMC algorithm

```
UMC($M = (I, T, F)$)
 1  $k \leftarrow 0$
 2  if Sat($I(Y_0) \wedge F(Y_0)$)
 3     then return false ▷ Counterexample found
 4  while true
 5        do $status$ = CheckFixedpoint($M, k$)
 6           if status = false
 7              then return false ▷ Counterexample found
 8              else  if status = true
 9                       then return true ▷ Property proved
10           $k \leftarrow k + 1$ ▷ status is abort; unfold further
```

$$\begin{aligned}\textsc{Suff}(M, r, s, t) &= \left(\bigwedge_{r \leq i < t} T(Y_i, Y_{i+1})\right) \wedge \left(\bigvee_{s \leq i \leq t} F(Y_i)\right) \\ &= \textsc{Tran}(M, r, s) \wedge \textsc{Fail}(M, s, t)\end{aligned} \tag{13}$$

Hence, the BMC formula becomes:

$$\textsc{Bmc}(M, r, s, t) = \textsc{Pref}(M, -1, r) \wedge \textsc{Suff}(M, r, s, t) \tag{14}$$

The idea of re-expressing the Bmc condition is to allow computing interpolants after one time step [29]. Hence, the prefix Pref is used for computing the states after one time step, which is then used for computing abstractions of the reachable states. The suffix Suff represents everything else. Observe that the failing property is only checked for in the suffix.

Suppose that $\textsc{Bmc}(M, r, s, t)$ is unsatisfiable, and let Π be a resolution proof. Moreover, let $r = 0$, let A represent the CNF encoding of $\textsc{Pref}(M, -1, 0)$ and B represent the CNF encoding of $\textsc{Suff}(M, 0, s, t)$. Finally, compute the interpolant $P = \textsc{Itp}(\Pi, A, B)$. Then, by definition of interpolant, P verifies the following conditions:

(1) P is expressed only in terms of the common variables of A and B, i.e. Y_0.
(2) $\textsc{Pref}(M, -1, r) \rightarrow P(Y_0)$ is a tautology.
(3) $P(Y_0) \wedge \textsc{Suff}(M, r, s, t)$ is unsatisfiable.

Given that $\textsc{Pref}(M, -1, r) \rightarrow P(Y_0)$ is a tautology, we can conclude that $P(Y_0)$ represents an abstraction of the states reachable in one time step.

The SAT-based model checking algorithm can be organized into two main phases: a BMC loop, where the circuit is unfolded, and a fixed point checking step, that tests both the existence of a counterexample and of a fixed point in the set of reachable states. Observe that the second phase requires the iterative computation of interpolants until a fixed point is reached or a

Algorithm 5 Fixed point identification in SAT-based UMC

```
CheckFixedpoint($M = (I, T, F), k$)
 1  $R \leftarrow I$
 2  while true
 3        do $M' \leftarrow (R, T, F)$ ▷ Update abstract transition relation
 4           $A \leftarrow$ Cnf(Pref($M', -1, 0$), $W_1$)
 5           $B \leftarrow$ Cnf(Suff($M', 0, 0, k$), $W_2$)
 6           $(isSAT, \Gamma) \leftarrow$ Sat($A \cup B$) ▷ If unsat, $\Gamma$ represents a proof trace
 7           if $isSAT$
 8              then if $R = I$
 9                      then return false ▷ Found counterexample
10                      else  return abort ▷ Need to unfold further
11           ▷ $A \cup B$ is unsat
12           $\Pi \leftarrow$ UnsatProof($\Gamma$) ▷ Generate resolution proof from proof trace
13           $P \leftarrow$ Itp($\Pi, A, B$) ▷ Generate interpolant from resolution proof
14           $R' \leftarrow P(Y_0/Y)$ ▷ Compute abstraction of reachable states
15           $C \leftarrow$ Cnf($\neg R, W_3$)
16           $D \leftarrow$ Cnf($R', W_4$)
17           $(isSAT, \Gamma) \leftarrow$ Sat($C \cup D$) ▷ Check if new states in abstraction $R'$
18           if not $isSAT$
19              then return true ▷ Failing property cannot be reached
20           $R \leftarrow R \vee R'$ ▷ Update abstraction of reachable states
```

true or (possibly) false counterexample is identified. The organization of the BMC loop is outlined in Algorithm 4, whereas the organization of fixed point checking step is outlined in Algorithm 5.

For the BMC loop there is no upper bound on the number of unfoldings, since the algorithm is now complete. The increment of k is not required to be 1. In fact, feedback from the fixed point checking procedure can be used for increasing k by values larger than 1 [23]. In addition, observe that the fixed point checking procedure consists of iterative computation of interpolants, where for iteration m the interpolant represents an abstraction of the reachable states in m time steps [29]. At each iteration of the UMC fixed point checking procedure, the existence of a fixed point is tested. The fixed point is reached when the abstraction of the reachable states in m time steps contains only states already included in the abstractions of the reachable states in less than m time steps.

In Algorithm 5 the abstraction of the reachable states after k iterations is R. Moreover, the new abstraction of reachable states, using R as the set of initial states, is R' and denotes an abstraction of the states reachable in $k + 1$ time steps. A fixed point in the set of reachable states occurs when $\forall_Y R'(Y) \rightarrow R(Y)$. This condition can be checked for with a SAT solver. The first step is to negate the formula, to obtain $\exists_Y R'(Y) \wedge \neg R(Y)$, which is true for states when R' holds and R does not hold. If this formula is satisfiable, then R'

contains states not yet contained in R, and so R' is added to R (see line 20). Otherwise, if the formula is unsatisfiable, then any state in R' is also a state in R, and so a fixed point has been reached. Observe that P is originally expressed in terms of variables Y_0, and so must be re-expressed in terms of generic variables Y, as shown in line 13 of Algorithm 5.

7.3 Recent Improvements

Recent work addressed optimizations to the UMC core algorithms [15,23,7,24]. The work on induction [15] focused on the use of incremental SAT, whereas the work on interpolants [23,7,24] addressed a number of issues, from the identification of more effective interpolants to alternative fixed point conditions.

One of the proposed improvements involves rescheduling the BMC loop [23]. Suppose the current unfolding size consists of K time steps. Moreover, assume the interpolant iteration procedure is executed I times, until a (possibly) false counterexample is identified. According to the definition of computed interpolants, this means that the target property cannot be satisfied within $K+I-1$ time steps. As a result, the property cannot be satisfied for any unfolding with size less than or equal to $K+I-1$ time steps. Hence, instead of a fixed policy of incrementing the size of the unfolding by 1 time frame (or some other constant), we can safely consider the size of the next unfolding to be $K+I$ time steps. The potential gains introduced with rescheduling can be significant. Assume a transition system and safety property such that a counterexample can be identified with an unfolding of T time steps. Moreover, assume that the BMC loop increases the unfolding by 1 time frame each time, that the initial unfolding size is 1, and that the interpolant iteration procedure runs for $T-K$ iterations for an unfolding size of K time steps (observe that if a counterexample exists, then we cannot iterate the computation of interpolants more than $T-K$ times). In this case, rescheduling guarantees that the UMC step is invoked only once, and so the number of times the SAT solver is invoked is $2+2\times(T-1)=O(T)$. In contrast, without rescheduling, the number of times the SAT solver is invoked is $T+2\times\sum_{i=1}^{T-1}(T-i)=O(T^2)$.

8 Conclusions

Symbolic model checking of finite-state transition systems is one of the most successful applications of modern SAT solvers. This paper provides a survey of the uses of SAT in model checking, focusing on the most successful approaches, the original bounded model checking work, and two alternative approaches for unbounded model checking, namely induction and interpolants.

Recent work addressed optimizations to the core algorithms [15,23,7,24]. The work on induction [15] focused the use of incremental SAT. The work on interpolants [23,7,24] addressed a number of issues, from the identification of more effective interpolants to alternative fixed point conditions.

A recent competition of hardware model checking algorithms [3] suggests that the most effective algorithms for model checking are currently based on interpolants, with induction representing a viable alternative. The feedback from the competition is likely to bring further improvements to SAT-based model checking algorithms.

Acknowledgments

The reviewers provided insightful comments in earlier versions of this paper. This work is partially supported by EU projects IST/033709 and ICT/217069 and by EPSRC grant EP/E012973/1.

References

[1] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 489–494, September 2000.

[2] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[3] A. Biere. Hardware model checking competition, July 2007. http://fmv.jku.at/hwmcc/.

[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Advances in Computers*, chapter Bounded Model Checking. Academic Press, 2003.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, March 1999.

[6] P. Bjesse and K. Claesen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, pages 372–389, 2000.

[7] G. Cabodi, M. Murciano, S. Nocco, and S. Quer. Stepping forward with interpolants in unbounded model checking. In *International Conference on Computer-Aided Design*, pages 772–778, 2006.

[8] 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 *Formal Methods in Computer-Aided Design*, pages 33–51, 2002.

[9] E. M. Clarke, O. Grumberg, and A. Peled. *Model Checking.* MIT Press, 1999.

[10] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer-Aided Verification*, pages 436–453, 2001.

[11] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

[13] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, July 1960.

[14] N. Een and N. Sörensson. An extensible SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, May 2003.

[15] N. Een and N. Sörensson. Temporal induction by incremental SAT solving. In *Workshop on Bounded Model Checking*, volume 89 of *ENTCS*, 2003.

[16] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Vardi. Multiple-counterexample guided iterative abstraction refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 176–191, April 2003.

[17] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Design, Automation and Testing in Europe Conference*, pages 142–149, March 2002.

[18] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI Conference on Artificial Intelligence*, pages 431–437, July 1998.

[19] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *International Conference on Computer-Aided Design*, pages 416–423, November 2003.

[20] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge University Press, 2nd edition, 2004.

[21] H.-J. Kang and I.-C. Park. SAT-based unbounded symbolic model checking. In *Design Automation Conference*, pages 840–843, June 2003.

[22] J. Krajicek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *Journal of Symbolic Logic*, 62(2):457–486, 1997.

[23] J. Marques-Silva. Improvements to the implementation of interpolant-based model checking. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 367–370, October 2005.

[24] J. Marques-Silva. Interpolant learning and reuse in SAT-based model checking. *Electr. Notes Theor. Comput. Sci.*, 174(3):31–43, 2007.

[25] J. Marques-Silva and K. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[26] J. Marques-Silva and K. Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

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

[28] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer-Aided Verification*, pages 250–264, July 2002.

[29] K. L. McMillan. Interpolation and SAT-based model checking. In *Computer-Aided Verification*, pages 1–13, 2003.

[30] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 2–17, April 2003.

[31] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.

[32] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[33] P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone circuit computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.

[34] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT solver. In *Formal Methods in Computer-Aided Design*, pages 108–125, 2000.

[35] O. Strichman. Tuning SAT checkers for bounded model checking. In *Computer-Aided Verification*, pages 480–494, July 2000.

[36] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.

[37] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *AAAI Conference on Artificial Intelligence*, pages 155–160, July 1988.

[38] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Testing in Europe Conference*, pages 10880–10885, March 2003.