Boolean Satisfiability Solvers and Their Applications in Model Checking

Article *in* Proceedings of the IEEE · August 2015

DOI: 10.1109/JPROC.2015.2455034

3 authors, including:

Yakir Vizel
Princeton University
26 PUBLICATIONS **301** CITATIONS

SEE PROFILE

INVITED PAPER

Boolean Satisfiability Solvers and Their Applications in Model Checking

By Yakir Vizel, Georg Weissenbacher, and Sharad Malik, *Fellow IEEE*

ABSTRACT | Boolean satisfiability (SAT)—the problem of determining whether there exists an assignment satisfying a given Boolean formula—is a fundamental intractable problem in computer science. SAT has many applications in electronic design automation (EDA), notably in synthesis and verification. Consequently, SAT has received much attention from the EDA community, who developed algorithms that have had a significant impact on the performance of SAT solvers. EDA researchers introduced techniques such as conflict-driven clause learning, novel branching heuristics, and efficient unit propagation. These techniques form the basis of all modern SAT solvers. Using these ideas, contemporary SAT solvers can often handle practical instances with millions of variables and constraints. The continuing advances of SAT solvers are the driving force of modern model checking tools, which are used to check the correctness of hardware designs. Contemporary automated verification techniques such as bounded model checking, proof-based abstraction, interpolation-based model checking, and IC3 have in common that they are all based on SAT solvers and their extensions. In this paper, we trace the most important contributions made to modern SAT solvers by the EDA community, and discuss applications of SAT in hardware model checking.

KEYWORDS | IC3; interpolation; model checking; proofs; satisfiability solving

I. INTRODUCTION

At the turn of the last century, a new generation of Boolean satisfiability (SAT) solvers such as Grasp [62] and Chaff [61] brought about a leap in the performance and scalability of satisfiability checkers for propositional formulas. This breakthrough was made possible by novel search techniques such as clause learning, clever branching heuristics, and carefully-engineered data structures. The impressive advances in the field of SAT solving have revolutionized a range of applications in electronic design automation (EDA), such as formal equivalence checking, model checking and formal verification of hardware, and automatic test pattern generation.

Hardware model checking [26], [27], [75], a technique to automatically determine whether a hardware design satisfies a given specification, particularly benefited from the advances of SAT solvers. SAT solvers boosted the scalability of symbolic model checking [12], an important hardware model checking technique. In symbolic model checking, sets of states and transition relations of circuits are represented as formulas to avoid a computationally expensive enumeration of explicit states. Historically, the first symbolic model checkers used binary decision diagrams (BDDs) [16], a data structure to represent formulas, to encode sets of states [17]. Later, BDDs were replaced with SAT solvers in order to increase scalability.

Manuscript received December 7, 2014; revised April 15, 2015; accepted June 16, 2015. The work of Y. Vizel and S. Malik was supported in part by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. The work of G. Weissenbacher was supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and the Vienna Science and Technology Fund (WWTF) under the Vienna Research Groups for Young Investigators Grant VRG11-005.

Y. Vizel and **S. Malik** are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: yvizel@princeton.edu; sharad@princeton.edu).

G. Weissenbacher is with TU Wien, 1040 Vienna, Austria (e-mail: georg.weissenbacher@tuwien.ac.at).

Digital Object Identifier: 10.1109/JPROC.2015.2455034

Bounded model checking (BMC) relies on SAT solvers to exhaustively check hardware designs up to a limited depth [12]. The inherently incomplete BMC approach was soon followed by complete SAT-based model checking techniques such as *k*-induction [77], image computation methods based on quantifier elimination [63] or quantified Boolean formulas (QBFs) [73], and the combination of automatic test pattern generation (ATPG) with SAT techniques [51]. As a result, the verification of industrial-sized circuit designs became feasible, and model checking was adopted by major chip manufacturers to ensure design correctness [8], [43]. An excellent survey of the state of the art and an empirical and industrial evaluation of SAT-based verification techniques a decade ago are provided by [1] and [73], respectively.

The boundaries of modern SAT solvers and model checkers have been pushed ever since. The aim of this paper is to provide a self-contained overview of contemporary model checking algorithms and their underlying SAT techniques, highlighting the advances since [1] and [73]. Our focus is on finite-state model checking exclusively; we refer the reader to [36] and [53] for surveys of software verification techniques.

Section II is dedicated to satisfiability solvers and their extensions, which form the basis of the model checking techniques covered in Section III. The capability of solvers to generate refutation proofs (Section II-D), provide Craig interpolants (Section II-E), and to solve SAT instances incrementally (Section II-C) enables novel verification techniques such as proof-based abstraction (Section III-D), interpolation-based model checking (Section III-E), and recently IC3 (Section III-F). Recent research combines these model checking techniques (Section III-G) in order to leverage the advantages of both approaches. In return, the increasingly larger designs verified by these tools result in large and difficult SAT instances, thus enriching the benchmark sets of the annual SAT competition (http://www.satcompetition.org). In this way, model checking drives the development of ever faster solvers.

II. SATISFIABILITY SOLVING

The *Boolean Satisfiability Problem* (abbreviated as SAT) is the problem of determining whether there exists an assignment that satisfies a given propositional formula. SAT [31] is the prototypical NP-complete problem and of tremendous importance in computer science [13]. Paradoxically and despite its theoretical complexity, SAT is often considered to be tractable. While this is a bold claim in general, it seems to hold for many well-structured propositional formulas derived from industrial applications. The focus of this section are the techniques underlying contemporary SAT engines (Section II-B and C) and extensions such as proof logging (Section II-D) and interpolation (Section II-E), which are at the core of the model checking techniques presented in Section III.

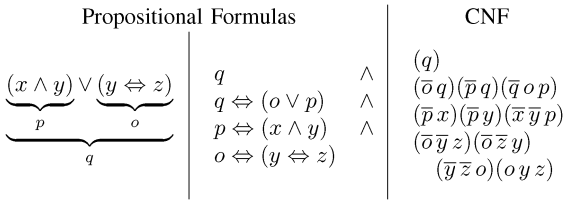

Fig. 1. *Tseitin's satisfiability-preserving transformation.*

A. Propositional Formulas and Basic Techniques

A propositional formula φ is built from a set V of Boolean variables, the logical constants T and F (denoting true and false, respectively), the logical connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and $\neg$ (denoting conjunction, disjunction, implication, bi-implication, and negation, respectively), and parenthesis. The syntax and semantics of formulas are defined as usual; we refer the reader to [13] for details. We use $\mathrm{Vars}(F)$ to denote the Boolean variables occurring in a formula F. An *assignment* maps the variables V to logical values (T or F). An assignment *satisfies* a formula φ if φ evaluates to T when the variables $\mathrm{Vars}(\varphi)$ are assigned according to the assignment. A formula is satisfiable if there exists at least one assignment satisfying it, and unsatisfiable otherwise. A formula is valid if and only if its negation is unsatisfiable.

A SAT solver is a program that determines whether or not a given formula is satisfiable. Many contemporary solvers expect the input instances to be formulas in *conjunctive normal form* (CNF). A formula in CNF is a conjunction of *clauses*, which in turn are disjunctions over the *literals* $\{x, \neg x | x \in V\}$. We use $\overline{x}$ to abbreviate the negation $\neg x$, and omit the operator $\vee$ in clauses and $\wedge$ in formulas whenever it is clear from the context. The subformula $(\overline{o}\,\overline{y}z)(\overline{o}\,\overline{z}y)$ in Fig. 1, for instance, corresponds to $(\neg o \vee \neg y \vee z) \wedge (\neg o \vee \neg z \vee y)$. The disjunction of two clauses C and D, denoted by $C \vee D$, is a clause containing all literals of C and D. If D contains only one literal t (i.e., D is a unit clause), we write $C \vee D$ as $C \vee t$. The empty clause (denoted by $\square$) corresponds to the Boolean value F.

Propositional formulas can be transformed into CNF by repeated application of rewriting rules such as distributivity, double negation elimination, and De Morgan's laws. Such a transformation, however, may lead to an exponential increase in formula size, a problem that can be avoided by constructing a formula in CNF that is not logically equivalent to the original formula, but preserves its satisfiability. Tseitin's transformation [80] recursively replaces each subformula $\varphi \circ \psi$ (where $\circ$ is an arbitrary logical operation) of the original formula with a fresh propositional identifier o and adds the constraint $o \Leftrightarrow (\varphi \circ \psi)$. The resulting formula is a conjunction of constraints of the form $o \Leftrightarrow (p \circ q)$, each of which can be represented by at most four clauses. This process is illustrated in Fig. 1. Optimizations of Tseitin's transformation take the

structure of the original formula (such as the polarity of subformulas [72]) into account, resulting in smaller CNF instances.

The translation into CNF, however, may introduce unnecessary (functionally dependent) atoms. This problem is addressed by *preprocessing* techniques that reduce the size of the formula by eliminating dependent variables by means of substitution [38], for instance. More such techniques are covered in [11] and our tutorial on SAT and extensions [60].

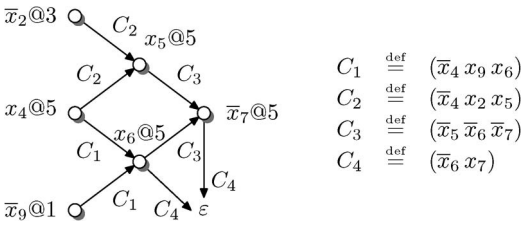

Fig. 3. ***Implication graph and conflict analysis.***

B. Conflict-Driven Clause Learning

SAT solvers search for a satisfying assignment of a formula given in CNF. The influential DPLL-algorithm (introduced by Davis, Putnam, Logemann, and Loveland [34]) performs a case split on the truth values of variables. Whenever the solver encounters a variable assignment in which one of the clauses of the formula evaluates to F, it *backtracks* and changes the most recent assignment until all assignments have been explored.

Chronological backtracking, as described above, has since been superseded by backjumping and conflict-driven clause learning (CDCL) [62], [78]. The CDCL algorithm (an outline of which is presented in Fig. 2) avoids the repeated exploration of conflicting variable assignments by caching the causes of failures in the form of learned clauses.

By iteratively assigning values to variables, the solver systematically explores partial assignments. Under a given partial assignment, each clause is in one of the following states:

- *Satisfied*: At least one of the literals of the clause evaluates to T.
- *Conflicting*: All literals are assigned to F, and therefore the clause cannot be satisfied.
- *Unit*: All but one of the literals are assigned, but the clause is not satisfied. Consequently, the remaining literal must be assigned in a way that satisfies the clause, i.e., the value of the corresponding variable is *implied*.
- In all other cases the clause is *unresolved*.

During the search, a *trail* represents the current partial assignment. The algorithm tracks the reason for each assignment to a variable: The value of a variable can be the result of a *decision* by the solver, or it can be implied by a clause that is unit (under the current partial assignment). In the former case, the solver associates a new *decision level* (represented by a counter) with the assignment. Implied decisions retain the current decision level. The trail ensures that no decision level is entered with the same partial assignment twice.

① In case of conflict at decision level 0: report UNSAT
② Repeat:
- ❶ if all variables assigned return SAT
- ❷ Make decision
- ❸ Propagate constraints
- ❹ No conflict? Go to ❶
- ❺ If decision level = 0 return UNSAT
- ❻ Analyse conflict and compute backtracking level
- ❼ Add conflict clause
- ❽ Backtrack to computed level and go to ❸

Fig. 2. ***CDCL algorithm.***

The solver stores decisions and their implications in an *implication graph*, from which it derives learned clauses in case of a conflict (step ❻ in Fig. 2). Each node of the implication graph represents an assignment and the respective decision level. In this context, $x@n$ is short for $x = \mathsf{T}$ at decision level n, and $\overline{x}@n$ denotes $x = \mathsf{F}$ at decision level n. Each directed edge represents a clause that is unit and implies the assignment represented by the node the edge points to. An implication graph is a *conflict graph* if it contains two implications resulting in a conflicting assignment.

Fig. 3 shows a conflict graph for the clauses C_1–C_4. The graph shows the implications at the current decision level 5 and the decisions at levels 1 and 3 ($\overline{x}_9@1$ and $\overline{x}_2@3$, respectively) causing these implications. The conflict node ε indicates the conflict under the given assignment, and its incoming edges are annotated with the conflicting clause C_4. This conflict stems from the fact that C_4 disagrees with C_1 and C_3 on the implied literals x_6 and $\overline{x}_7$, respectively.

In case a conflict is encountered, the solver proceeds to extract from the conflict graph a set of assignments and decisions causing the conflict. The decisions and assignments at and prior to the current decision level in the partial conflict graph constitute an obvious choice: for instance, the decisions $\overline{x}_9@1$ and $\overline{x}_2@3$ and the assignment $x_4@5$ in Fig. 3. Adding a *learned* clause $(x_2\ \overline{x}_4\ x_9)$ to the clause database prevents the solver from revisiting the same conflicting constellation of assignments. Note that $x_4@5$ need not necessarily be a decision in our example: It is sufficient to choose an assignment at the current level (different from the conflict node) that lies on all paths from the most recent decision to the conflict node ε. Such a node is called a *unique implication point* (UIP) [62], [78] and determines which single literal from the current decision level appears in the learned clause.

The decision levels contributing literals to the learned clause then determine the backtracking level. By undoing

all decisions up to (but excluding) the second highest decision level occurring in the learned clause (Fig. 2, step ❽), all literals of the clause (added in step ❼) except one are assigned. Such a clause is called *asserting* and enables immediate unit propagation [61], [62], [78]. The UIP that is closest to the conflict results in a higher backjump level [2], [62], [78] is called the *first UIP* and is preferred in practice.

By caching variable values erased from the trail and (re)using these saved literals in future decisions, *progress saving* (or phase saving) can avoid work repetition caused by far nonchronological backtracking [69].

An in-depth discussion of clause learning is provided in [70]. Competitive SAT solvers incorporate numerous additional heuristics and optimizations, some of which are discussed below.

The propagation of implied literals described above is known as *Boolean constraint propagation* (BCP) and is contingent on the efficient detection of unit clauses. To detect clauses that are unit, it is sufficient to *watch* the first two literals of each clause [41], [42], [61]. Whenever one of the *watched literals* is assigned to F, it is swapped with a literal not yet assigned in that clause. If there is no such literal and the clause is not satisfied, then the clause must be unit. The advantage of watched literals is that no action is required when decisions are undone.

In practice, the order of decisions (made in step ❷) is of paramount importance to the performance of the solver. *Variable State Independent Decaying Sum* (VSIDS) [61] is a popular heuristic that associates a weight with each literal that is increased whenever a learned clause containing the literal is added. The computational overhead of this technique is low since only the weight of literals occurring in learned clauses needs to be updated. Choosing a literal can be done efficiently, and using a floating point number to represent the weight adds accuracy and avoids ties [41]. The weights indicate the priority of a literal and are periodically divided by a constant, leading to a bias towards literals involved in recent conflicts.

Periodic restarts of the search process pose a key difference between CDCL solvers and the DPLL algorithm [6], [50], making the former exponentially more powerful (see also Section II-D). Restarts can be enforced by imposing a bound on the number of decisions or conflicts, for instance. After each restart, these bounds are increased until they are large enough to ensure completeness of the solver. For pure Las Vegas algorithms, optimal restart strategies achieving the minimum expected running time exist, given full knowledge of the distribution of the solver running time (which is a random variable) [59]. Luby *et al.* [59] prove that a universal sequence of bounds of the form 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, 1, ... results in a strategy whose performance is within a logarithmic factor of the optimal strategy. In practice, these results do not immediately apply to CDCL solvers, which retain learned clauses across restarts. Depending on the benchmarks, dynamic and adaptive restart strategies (see [3], [4], [9], [55], and [79], for instance) can have a significant impact on the performance.

Since the trail guarantees the completeness of the solver, learned clauses can be safely discarded to save memory and increase performance. Keeping "relevant" clauses, however, can be highly beneficial for guiding the search process. In Glucose [3], for instance, the importance of clauses is determined by computing the number of different decision levels in the learned clause (the *Literal Block Difference*).

Some of the leading SAT solvers also apply formula rewriting and simplification rules as an integral part of the CDCL search [52]. Such SAT solvers are called *inprocessing*.

C. Incremental SAT Solving

In many applications, solvers are required to answer sequences of similar queries (see Section III). Restarting the solver *from scratch*, however, would result in the loss of learned clauses and partial assignments. To avoid the overhead of rediscovering previously learned facts, many solvers support *incremental* queries. Incremental solvers allow for the subsequent modification of already solved instances by: 1) adding or retracting assumptions about the values of literals, or 2) adding or removing clauses.

Assumptions are conjunctions of unit clauses, which are added as decisions at the beginning of the trail. Consequently, assumptions can be retracted by simply undoing the corresponding decisions [24], [40], [41]. For unsatisfiable instances, incremental solvers provide a subset of the assumptions that is inconsistent with the given formula.

The CDCL algorithm (Fig. 2 in Section II-B) naturally supports augmenting formulas in CNF with additional clauses. Removing (or *popping*) clauses requires additional bookkeeping, however, since learned clauses derived from obsolete clauses need to be discarded as well. Satire [86] and zChaff [61] provide partial support for the removal of clauses by creating a new *context* on a stack when a clause is *pushed*. A subsequent pop discards the corresponding context. PicoSAT [10] enables the removal of a clause C by adding an activation literal a to C. The literal a is initially set to F using an assumption; accordingly, $C \vee a$ corresponds to C. If a is later set to T, $C \vee a$ is satisfied and effectively ignored by the solver.

D. Clausal Refutation Proofs

Contemporary solvers are sophisticated artifacts of engineering and therefore not necessarily free of errors. Accordingly, certificates enabling the efficient validation of the output of a solver are desirable. A satisfying assignment provided by a solver is evidence of the satisfiability of the input instance and can be checked easily. To certify unsatisfiability, solvers are required to log additional information to enable the construction of a *refutation proof*.

Propositional refutations comprise a sequence of *resolution* steps. The resolution rule states that an assignment

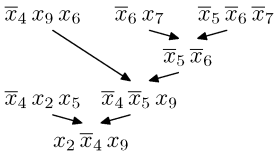

Fig. 4. ***Resolution proof for learned clause.***

satisfying the antecedents $(C \vee x)$ and $(D \vee \overline{x})$ (where x is called the *pivot*) also satisfies the *resolvent* $(C \vee D)$

$$\frac{C \vee x \qquad \overline{x} \vee D}{C \vee D}\ [\mathrm{Res}].$$

We use $\mathrm{Res}((C \vee x), (D \vee \overline{x}), x)$ to denote the resolvent of the clauses $(C \vee x)$ and $(D \vee \overline{x})$ with the pivot x.

Each learned clause in a CDCL solver is the consequence of a sequence of resolution steps, as illustrated by the following example. Consider the conflict graph in Fig. 3. By subsequently resolving on the conflicting literals in the reverse order in which they were assigned (as witnessed by the trail), we obtain

$$C_5 = \mathrm{Res}(C_4, C_3, x_7) = (\overline{x}_5\ \overline{x}_6)$$
$$C_6 = \mathrm{Res}(C_5, C_1, x_6) = (\overline{x}_4\ \overline{x}_5\ x_9).$$

The clause C_6 disagrees with C_2 on the implied literal x_5. The resolvent of these clauses is $C_7 = \mathrm{Res}(C_6, C_2, x_5) = (x_2 \overline{x}_4 x_9)$. C_7 contains a *single* literal $(\overline{x}_4)$ assigned at decision level 5 while still conflicting with the current partial assignment. Stopping to resolve at this point yields the learned clause corresponding to the first UIP.

The learned clause in the example above is a consequence of clauses of the original instance and previously learned clauses. This sequence of resolution steps can be represented as a directed acyclic graph (DAG), as illustrated in Fig. 4.

Since each learned clause is justified by a sequence of resolution steps—including the final empty clause—logging all resolution steps and intermediate consequences during the execution of the solver results in a refutation proof [88]. This approach, however, results in a relatively high overhead since not all learned clauses are reused. Alternatively, it is possible to reconstruct a resolution proof *offline* from the sequence of learned clauses [44]. Each resolution sequence generated by a CDCL solver has the following properties [6]:

- *Regularity*: Each pivot is resolved upon at most once.
- *Linearity*: Each intermediate clause in a sequence is obtained by resolving the preceding intermediate clause with an initial or previously learned clause.
- *Tree-likeness*: Each intermediate clause is used exactly once in the sequence and never outside.

A resolution derivation with these properties is called trivial [6]. The properties of trivial resolution derivations enable us to *discard* the intermediate clauses. Since each learned clause is a consequence of initial and learned clauses, its negation is inconsistent with the learned clauses preceding it and the original formula. Negating C_7 in the example above yields the unit clauses $(\overline{x}_2)(x_4)(\overline{x}_9)$, for instance. By propagating these literals, BCP can efficiently establish the inconsistency with the clauses C_1, C_2, C_3, and C_4, and reconstruct the proof in Fig. 4. Note that neither the order of the pivots nor the intermediate clauses of the sequence are required for the proof construction.

Proofs of unused learned clauses can be pruned by starting the reconstruction process with the final (empty) clause [44] and propagating unit clauses backwards. (Conversely, forward checking validates each learned clause.) This process can be accelerated by augmenting clauses discarded by the solver with deletion information [48] since clauses, once discarded, can be ignored during proof construction.

Clausal proofs [44], (D)RUP proofs [48], and proofs stored in the `TraceCheck`-format[1] retain only learned clauses and the order in which they were encountered. Full-resolution proofs can then be constructed by means of BCP.

Restrictions on the order of pivots or the structure of resolution proofs can have a significant impact on the size of the resulting proofs. Notably, different solving algorithms simulate proof systems of different power.

- The original DP algorithm [35] results in ordered resolution proofs (imposing a fixed ordering on the variables along any path to the empty clause in the proof).
- The DPLL algorithm [34] yields tree resolution proofs, i.e., it does not reuse any previously derived clauses.
- CDCL (given the right heuristics and unlimited restarts) is capable of generating proofs that are at most polynomially larger than the smallest proofs obtained by general resolution [71], i.e., the proof system underlying CDCL in combination with the above-mentioned restrictions *p-simulates* general resolution.

Accordingly, CDCL deploys a proof system that is exponentially more powerful than the system underlying DPLL [6], [50] since none of the above-mentioned restricted systems p-simulates general resolution.

Besides validating the output of solvers and generating resolution proofs, the trimming techniques discussed

[1]http://fmv.jku.at/tracecheck/README.tracecheck

above [44], [48] also enable the extraction of *unsatisfiable cores* (UCs). A UC is a subset of the clauses of the original instance for which there is no satisfying assignment. A UC is minimal if removing any of its clauses makes it satisfiable. Given a resolution refutation of a formula, the set of all initial clauses in the refutation forms a UC. Algorithms for deriving minimal UCs from clausal proofs are presented in [7] and [48]. Intuitively, a (minimal) UC provides a concise reason for the unsatisfiability of a SAT instance, which can be exploited to derive abstractions of circuits (see Section III-D).

E. Craig Interpolation

Craig's interpolation theorem [33] is a seminal theoretical result combining model and proof theory. Interpolation is a prime example of a theoretical result that has found numerous unexpected practical applications in computer science, such as the approximation of reachable states in model checking (see Section III-E). The theorem states that for every pair of first-order logic formulas A and B such that $A \Rightarrow B$, there exists a formula I (the *interpolant*) with the following properties: 1) $A \Rightarrow I$; 2) $I \Rightarrow B$; and 3) the function and predicate symbols occurring in I occur in A as well as in B. In the context of verification and propositional logic, McMillan's formulation [64] is more common.

Definition 1 (Interpolant): Let A and B be quantifier-free propositional formulas whose conjunction $A \wedge B$ is unsatisfiable. An *interpolant* is a quantifier-free propositional formula I such that $A \Rightarrow I$ is valid, $I \wedge B$ is unsatisfiable, and the variables occurring in I are a subset of the variables common to A and B.

Intuitively, if A represents reachable states and B unsafe or bad states, then the interpolant I safely overapproximates A, a property frequently used in the context of fixed-point detection (as discussed in detail in Section III-E).

The existence of propositional interpolants as defined in Definition 1 is an immediate consequence of Craig's theorem. Moreover, Huang, Krajíček, and Pudlák [49], [56], [74], and later McMillan [64], showed that propositional interpolants can be derived from resolution refutations in polynomial time. While their interpolation algorithms have subsequently been generalized and unified [37], we restrict our presentation to the algorithm by Huang, Krajíček, and Pudlák, which is sufficient for all practical purposes.

Pudlák [74] presents interpolants as "separators"; the interpolant I evaluates to T under assignments that make A true, and to F under assignments making B true (in accordance with Definition 1). This intuition is illustrated in Fig. 5. The refutation on the left in Fig. 5 proves that conjunction of the formulas $A \stackrel{\text{def}}{=} (\overline{x}_0)(x_0\ x_2)(\overline{x}_1\ \overline{x}_2)$ and $B \stackrel{\text{def}}{=} (\overline{x}_2)(x_1\ x_2)$ is unsatisfiable. $I \stackrel{\text{def}}{=} \overline{x}_1$ is an interpolant for A and B. I is F if $x_1 = \mathsf{T}$, and if we replace x_1 accordingly in the original proof, we obtain a refutation for A under the chosen substitution (second subfigure from the left in Fig. 5). Similarly, substituting F for x_1 in the proof and B yields a refutation. The value of the shared variable x_1 acts as a switch between these two proofs (illustrated by the multiplexer on the right of Fig. 5). This insight allowed Pudlák to formulate an interpolation that annotates each node in the proof with a partial interpolant.

Clauses from A or B respectively constitute the base case and are annotated with the partial interpolants T and F accordingly. Resolution steps with a *shared* pivot x occurring in both A and B introduce a "multiplexer" choosing the partial interpolant I_1 as interpolant if x is false, and I_2 otherwise. The graph on the far right of Fig. 5 is the original proof with all clauses replaced by their respective partial interpolants. The annotation of the sink node (corresponding to the empty clause) constitutes an interpolant for the formulas A and B.

In the context of software verification, interpolants have been generalized to sequences of formulas [65] representing subsequent transition. An interpolation sequence overapproximates reachable states at the respective points in such a formula (see Section III-E2).

Definition 2 (Interpolation Sequence): Let $\langle A_1, A_2, \ldots, A_n \rangle$ be an ordered sequence of propositional formulas such that the conjunction $\bigwedge_{i=1}^{n} A_i$ is unsatisfiable. An interpolation sequence is a sequence of formulas $\langle I_0, I_1, \ldots, I_n \rangle$ such that all of the following conditions hold.

1) $I_0 = \mathsf{T}$ and $I_n = \mathsf{F}$.
2) For every $0 \leq j < n$, $I_j \wedge A_{j+1} \Rightarrow I_{j+1}$ is valid.
3) For every $0 < j < n$, it holds that the variables in I_j are a subset of the common variables of $\langle A_1, \ldots, A_j \rangle$ and $\langle A_{j+1}, \ldots, A_n \rangle$.

Interpolation sequences can be computed in an iterative manner by computing the interpolants $I_j\ (1 \leq j \leq n)$ for $A \stackrel{\text{def}}{=} I_{j-1} \wedge A_j$ and $B \stackrel{\text{def}}{=} \bigwedge_{i=j+1}^{n} A_i$ using Pudlák's algorithm. Alternatively, a sequence interpolant can also be computed

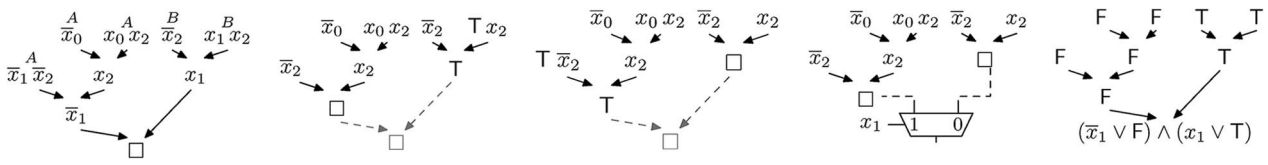

Fig. 5. ***Interpolant $\overline{x}_1$ acts as a "separator" for the resolution refutation (left); partial interpolants for the refutation proof (far right).***

more efficiently by a single traversal of a given refutation proof [23], [76], [81].

Interpolants are not unique, and the interpolants computed by different algorithms are not necessarily equivalent. Properties of the interpolants (such as logical strength) generated by different algorithms have been studied extensively in [37], [54], [76], and [84]. The generation of interpolants from resolution proofs and clausal proofs in CNF has been addressed in [45] and [83], respectively. Which logical or structural properties of interpolants are desirable is highly application-dependent, however, and in many cases an open research problem.

III. APPLICATIONS OF SAT IN MODEL CHECKING

Model checking [26], [27], [75] is an automated verification technique for checking whether a given system satisfies a desired property. The system is usually described as a finite-state model in the form of a state transition graph. The specification is given as a temporal logic formula. Unlike testing or simulation-based verification, model checking tools are exhaustive in the sense that they traverse *all* behaviors of the system, and either confirm that the system behaves correctly or present a *counterexample*.

Model checking has been successfully applied to verifying hardware and software systems. Its main limitation, however, is the *state explosion problem* that arises due to the huge state space of real-life systems. The size of the model induces high memory and time requirements that may make model checking not applicable to large systems. Much of the research in this area is dedicated to increasing model checking applicability and scalability.

The first significant step in this direction was the introduction of BDDs [16] into model checking. A BDD is a data structure representing a propositional formula, and the use of BDDs to *symbolically* (rather than explicitly) represent sets of states led to a significant performance improvement. BDD-based *symbolic model checking* (SMC) [17] enabled model checking of real-life hardware designs with several hundred state elements. Current design blocks with well-defined functionality, however, typically have many thousands of state elements. To handle designs of that scale, researchers deployed SAT solvers, starting with the invention of SAT-based BMC [12], a technique that targets the detection of counterexamples.

In this section, we focus on SAT-based model checking algorithms that are based on *abstraction*, *interpolation*, and *IC3*. We show how the advancements in SAT led to the development of state-of-the-art model checking algorithms and to the applicability of model checking to large design blocks. It is safe to say that due to these advancements, SAT-based model checking is nowadays a part of the design methodology applied by industry [1], [8], [43].

Before describing some of the most successful SAT-based model checking algorithms, we introduce some basic definitions and notation.

A. Preliminaries

In this paper, we discuss hardware verification and focus on finite-state transition systems that can be used to model sequential circuits. Given a set of Boolean variables V, V induces a set of states $S \stackrel{\text{def}}{=} \mathbb{B}^{|V|}$, and a state $s \in S$ is an assignment to V and can be represented as a conjunction of literals that are satisfied in s. More generally, a formula over V represents the set of states in which it is satisfiable.

Definition 3: A *finite-state transition system* is a tuple $M \stackrel{\text{def}}{=} \langle V, I, T, P \rangle$, where V is a set of Boolean variables, $I(V)$ and $P(V)$ are formulas over V describing the initial states and safe states, respectively. The *transition relation* $T(V, V')$ is a formula over the variables V and their primed counterparts $V' = \{v' | v \in V\}$, representing starting states and successor states of the transition, respectively.

Given a formula F over V, we use F' to denote the corresponding formula in which all variables $v \in V$ have been replaced with their counterparts $v' \in V'$. In the context of multiple steps of the transition system, we use V^i instead of V' to denote the variables in V after i steps. Given a formula F over V^i, the formula $F[V^i \leftarrow V^j]$ is identical to F except that for each variable $v \in V$, each occurrence of v^i in F is replaced with v^j. This substitution allows us to change the execution step to which a formula refers.

We also use substitution for formulas and subformulas. Let $F(V)$ and $H(V)$ be formulas over V and let $G(V)$ be a subformula of F. $F[G \leftarrow H]$ denotes the formula obtained by replacing all occurrences of the subformula G in F with H.

A path of length k in a transition system $M = \langle V, I, T, P \rangle$ is described by the following formula:

Formula 1. $\text{path}^{i,j} \stackrel{\text{def}}{=} T(V^i, V^{i+1}) \wedge \ldots \wedge T(V^{j-1}, V^j)$ where $0 \leq i < j$ and $j - i = k$. An *initial* path of length k is defined using the formula $I(V^0) \wedge \text{path}^{0,k}$.

The notation introduced above illustrates how a traversal of the state space of a transition system M can be reduced to satisfiability. In essence, an initial *path* formula of a transition system describes all possible executions of length k of the circuit when starting from the initial condition. Thus, it also represents all states that can be reached in k transitions of M.

Most SAT-based model checking algorithms are, explicitly or implicitly, based on a traversal of the state space using the above formula. Central to all these algorithms is the notion of a Forward Reachability Sequence, defined in the following:

Definition 4: A *forward reachability sequence* (FRS) of length k with respect to a transition system $M = \langle V, I, T, P \rangle$,

denoted $\bar{F}_{[k]}$, is a sequence $\langle F_0, \ldots, F_k \rangle$ of propositional formulas over V such that the following holds.

- $F_0 = I$.
- $F_i \wedge T \Rightarrow F'_{i+1}$ for $0 \leq i < k$.

A reachability sequence $\bar{F}_{[k]}$ is *monotonic* if $F_i \Rightarrow F_{i+1}$ for $0 \leq i < k$, and is *safe* if $F_i \Rightarrow P$ for $0 \leq i \leq k$. The individual propositional formulas F_i are called *elements* or *frames* of the sequence.

An element F_i in an FRS $\bar{F}_{[k]}$ represents an overapproximation of states reachable in i steps of the transition system. If the FRS is monotonic, then F_i is an overapproximation of all states reachable in at most i steps. Monotonic FRSs arise: 1) in the context of BDD-based model checking [17], where the set of reachable states is iteratively increased until either a fixed point is reached or a counterexample is detected; and 2) in the IC3 algorithm [14], discussed in detail in Section III-F.

Definition 5 (Inductive Invariant, Consecution): A set of states characterized by the formula F is *inductive* (satisfies *consecution*, respectively) if $F \wedge T \Rightarrow F'$ holds. F is inductive *relative* to a formula G if $G \wedge F \wedge T \Rightarrow F'$ holds. F satisfies *initiation* if $I \Rightarrow F$, i.e., if F comprises all initial states. F is *safe* with respect to P if $F \Rightarrow P$.

We say that F is an *inductive invariant* if it satisfies initiation and is inductive.

Lemma 1: Let $M = \langle V, I, T, P \rangle$ be a transition system and let F be a propositional formula representing a set of states. If F is an inductive invariant that implies P (i.e., $F \Rightarrow P$ is valid), then P holds in M and M is said to be *safe*.

The following lemma highlights the relationship between inductive invariants (Definition 5) and FRS (Definition 4).

Lemma 2: Let M be a transition system $\langle V, I, T, P \rangle$ and let $\bar{F}_{[k]}$ be an FRS. Let us define $F \stackrel{\text{def}}{=} \bigvee_{j=0}^{i} F_j$ where $0 \leq i < k$. Then, F is an inductive invariant if $F_{i+1} \Rightarrow F$. In addition, if $\bar{F}_{[k]}$ is *safe*, then $F \Rightarrow P$ holds, and thus M is safe.

B. Overview of Model Checking Algorithms

As mentioned above, model checking algorithms aim to establish the safety of a given transition system, or provide a counterexample if the system is not safe. A model checking algorithm is *complete* if it is able to either provide a counterexample or prove the absence of counterexamples of any length, and *sound* if it does not provide a wrong answer.

In the following, we provide a brief overview of the principles and mechanics underlying a number of widely used model checking techniques (illustrated in Figs. 6–9). The technical details are provided in Section III-C–G.

a) BMC: The success of BMC [12] is based on its ability to find counterexamples. BMC is based on the exploration of bounded paths in a transition system M. To this end, BMC *unwinds* the transition relation T—illustrated in Fig. 6 and explained in Section III-C—in order to determine whether the property P can be violated after k steps.

Complete SAT-based model checking algorithms, on the other hand, are predominantly based on a search for an

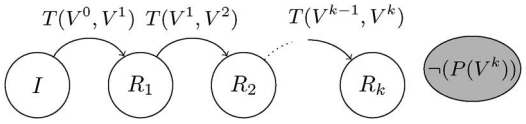

Fig. 6. *BMC checks whether a property P can be violated in k steps by encoding reachable sets of states $(R_1, \ldots, R_k)$ as a SAT instance. BMC does not identify repeatedly visited states and can therefore not determine whether the property holds for arbitrarily many steps.*

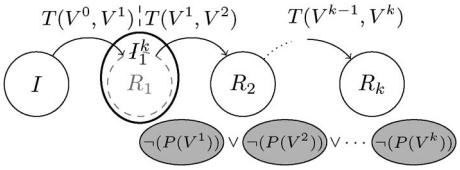

Fig. 7. *Interpolation-based model checking partitions an unsatisfiable BMC instance into a formula A representing initial states and the first step, and a formula B representing the states from which a property P can be violated within k − 1 steps. The interpolant I_1^k safely overapproximates all states reachable in a single step and is used to extend the FRS.*

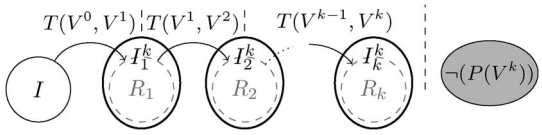

Fig. 8. *Interpolation sequence-based model checking partitions an unsatisfiable BMC instance into k + 1 partitions (with the last one representing the "bad" states), resulting in the interpolants $I_1^k, \ldots I_k^k$. Each I_i^k overapproximates the states reachable in i steps, and states in I_{i+1}^k are reachable from I_i^k in a single step.*

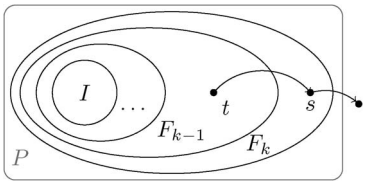

Fig. 9. *IC3 maintains a monotonic sequence of frames $F_1, \ldots, F_k$ that overapproximate the states reachable in up to k steps. The approximation is iteratively refined by eliminating from frame F_i states t that are predecessors of a state in ¬P, but have themselves no predecessor in F_{i-1} (i.e., ¬t is inductive relative to F_{i-1} and $F_{i-1} \wedge \neg t \wedge T \Rightarrow \neg t'$ holds).*

inductive invariant. A popular approach is k-induction [77], which aims to find a bound k such that all states reachable via an initial path $I(V^0) \wedge \text{path}^{0,k}$ are safe, and that whenever P holds in k consecutive steps of the transition system, then P also holds in the subsequent step. The model checking algorithms discussed below search for inductive invariants by means of FRS computation and the use of Lemma 2. In case an inductive invariant that implies P is found, M is reported to be safe.

b) Interpolation-based model checking: Interpolation-based model checking algorithms [64] also explore bounded paths in M, but use interpolation to synthesize an inductive invariant during the exploration. As illustrated in Figs. 7 and 8, interpolants I_i^k derived from unsatisfiable BMC instances safely overapproximate reachable states (as suggested in Section II-E). The resulting formulas I_i^k are then incorporated into a *safe* FRS that is maintained by the model checker and gradually refined until either an inductive invariant or a counterexample is found. The strength of interpolation-based model checking is in its ability to compute concise overapproximations of reachable states, thus accelerating fixed-point convergence.

c) IC3: The IC3 [14] technique (where IC3 is short for Incremental Construction of Inductive Clauses for Indubitable Correctness) and Property Directed Reachability [39] (PDR) differ from the above-mentioned algorithms in so far as they do not unroll the transition relation. IC3 maintains a *monotonic safe* FRS (as illustrated in Fig. 9), which is incrementally refined by eliminating states that can be proven unreachable by means of consecution checks (Definition 5) over subsequent frames. IC3's focus on single steps of the transition relation enables the efficient and targeted generation of relatively inductive clauses. Unlike interpolation-based techniques, IC3 does not depend on the unpredictable result of an interpolation engine.

Proof-based Abstraction (PBA), while in itself not a model checking algorithm, uses cores of unsatisfiable BMC instances to remove unnecessary details from M, thus simplifying the transition system for a subsequent complete model checking step (see Section III-D).

We now describe each of the above algorithms in detail.

C. Bounded Model Checking

As explained in Section II, a SAT solver either finds a satisfying assignment for a propositional formula or proves its absence. Using this terminology, BMC [12] determines whether a transition system has a counterexample of a given length k or proves its absence. BMC uses a SAT solver to achieve this goal. Given a transition system M, BMC translates the question "Does M have a counterexample of length k?" into a propositional formula and uses a SAT solver to determine if the formula is satisfiable or not. If the solver finds a satisfying assignment, a counterexample exists and is represented by the assignment. If the SAT solver proves that no satisfying assignment exists, then BMC concludes that no counterexample of length k exists.

Given a transition system $M = \langle V, I, T, P \rangle$, BMC is an iterative process for checking P in all initial paths up to a given bound on the length. Given a bound k, BMC either finds a counterexample of length k or less for P, or concludes that there is no such counterexample. In order to search for a counterexample of length k, the following propositional formula is built:

Formula 2. $\varphi^k \stackrel{\text{def}}{=} I(V^0) \wedge \text{path}^{0,k} \wedge (\neg P(V^k))$.

φ^k is then passed to a SAT solver that searches for a satisfying assignment. If there exists a satisfying assignment for φ^k, then the property is violated since there exists a path of length k violating P. In order to conclude that there is no counterexample of length k or less, BMC iterates all lengths from 0 up to the given bound k. In each iteration, a SAT procedure is invoked.

The formula φ^k represents all paths of length k in the transition system that reach a bad state at step k. Recall that a formula of an initial path of length k represents, *implicitly*, all states reachable up to k steps. Since this representation is only implicit, BMC cannot compute an FRS and therefore cannot find an inductive invariant. Verification is obtained only if k exceeds the length of the longest path among all shortest paths from an initial state to some state in the transition system. In practice, it is hard to compute this bound, and even when known, it is often too large to handle [30]. Thus, the main drawback of this approach is its incompleteness. BMC can only prove the absence of counterexamples of length up to k, but cannot guarantee that there is no counterexample of arbitrary length.

D. Proof-Based Abstraction

Abstraction [25] is a widely-used method to mitigate the state explosion problem. Since the root of the problem is the need of model checking algorithms to exhaustively traverse the entire state space of the system, abstraction aims at reducing the state space by removing irrelevant details from the system. The irrelevant details of the system are usually determined by analyzing the checked property and finding which parts of the system are not necessary for the verification or refutation of that property.

A well-known SAT-based abstraction technique is PBA [66]. One of the main advantages of SAT solvers is their ability to "zoom in" on the part that makes a CNF formula unsatisfiable. This part is referred to as the unsatisfiable core (UC) of the formula (see Section II-D). If an unsatisfiable CNF formula is a set of clauses, then its UC is an unsatisfiable subset of this set of clauses. PBA uses the UC of an unsatisfiable BMC formula to derive an abstraction.

Let us assume that the formula φ^k is unsatisfiable. Let us define the set $V_a = \{v | v^i \in \text{Vars}(UC(\varphi^k)), 0 \leq i \leq k\}$ as the set of variables from the transition system that appears in the UC of φ^k. Clearly $V_a \subseteq V$. The abstract transition system M_a is derived from M by making all variables

$v \in V \setminus V_a$ nondeterministic (i.e., leaving them unconstrained). This abstraction, in the above context, is usually referred to as a "visible variables" abstraction [57].

PBA is based on the BMC loop. At each iteration, a BMC formula φ^k is checked. If the formula is satisfiable, then a counterexample is found. Otherwise, a UC is extracted, and an abstract transition system M_a is computed. This abstract model can then be passed to a complete model checking algorithm, e.g., a BDD-based algorithm [67], for verification. If the property is proved using the abstract model, then the algorithm terminates concluding that the property holds. Otherwise, a counterexample is found. Note that if a counterexample is found in M_a, it may not exist in M (due to the abstraction). A counterexample that exists in M_a but not in M is referred to as *spurious*. In the case of a spurious counterexample, PBA executes the next iteration of the BMC loop with a larger k.

1) Other Abstraction Techniques: PBA can be viewed as a top–down approach. It first considers the concrete transition system M, and after verifying that no counterexample exists up to a specific length, it derives an abstract model. A different approach is Counter-Example Guided Abstraction Refinement (CEGAR) [28], [29]. Unlike PBA, this approach can be viewed as bottom–up. It starts with a coarse abstract model M_a and then uses spurious counterexamples and M to refine M_a. This process continues until either M_a is proved safe or a real counterexample is found.

In the description above, we show how abstraction is used to remove state variables from M. If we consider a circuit, this amounts to removing state elements from that circuit. Other approaches suggest targeting the internal gates of a circuit rather than its state elements (see [68], for instance).

E. Interpolation and Model Checking

In this section, we introduce two complete SAT-based model checking algorithms that use *interpolation*. The two algorithms use *interpolation* [32] and *interpolation sequences* [54] (as described in Section II-E) that, when combined with BMC, can provide complete model checking algorithms.

1) Interpolation-Based Model Checking (ITP): ITP [64] is a complete SAT-based model checking algorithm that relies on interpolation to compute the FRS. More precisely, interpolation is used to overapproximate the reachable states in a transition system M. Like PBA, ITP is based on BMC. When a BMC query is unsatisfiable, an interpolant is extracted from the proof of unsatisfiability. The interpolant represents an overapproximation of reachable states.

Before going into details, let us first revisit BMC. As we have shown, BMC formulates the question: "Does M have a counterexample of length k?" as a propositional formula φ^k (Formula 2). In a similar manner, BMC can also be formulated using the question "Does M have a counterexample of length i such that $1 \leq i \leq k$?" by using the following propositional formula:

Formula 3. $\psi^k \stackrel{\text{def}}{=} I(V^0) \wedge \text{path}^{0,k} \wedge (\bigvee_{i=1}^{k} \neg P(V^i))$.

In the original description of ITP [64], the above formula is used. ITP uses nested loops where the inner loop computes a *safe* FRS by repeatedly checking formulas of the form ψ^k with a fixed k, and the outer loop increases the bound k when needed. The safe FRS is computed inside the inner loop by extracting interpolants from unsatisfiable BMC formulas. Let us now describe the nested loops in more detail.

- **Inner Loop**: In general, the inner loop checks a fixed-bound BMC formula. At the first iteration, ψ^k is checked. If this BMC formula is satisfiable, then a counterexample exists and the algorithm terminates. If it is unsatisfiable, then the following (A, B) pair is defined:
  - $A \stackrel{\text{def}}{=} I(V^0) \wedge T(V^0, V^1)$;
  - $B \stackrel{\text{def}}{=} \text{path}^{1,k} \wedge (\bigvee_{i=1}^{k} \neg P(V^i))$.

  Following Definition 1, an interpolant I_1^k is extracted (cf. Fig. 7). The interpolant represents an overapproximation of the states reachable from I after one transition $(A \Rightarrow I_1^k)$. In addition, no counterexample can be reached from I_1^k in $k-1$ transitions or less ($I_1^k \wedge B$ is unsatisfiable), which also guarantees that $I_1^k \Rightarrow P$. (Note that if instead of ψ^k, φ^k would have been used, the interpolant would not necessarily satisfy P.) Thus, the sequence $\langle I, I_1^k[V^1 \leftarrow V] \rangle$ is a valid safe FRS. In the subsequent iterations, the formula $\psi^k[I \leftarrow I_{j-1}^k]$ is checked, where j is the iteration of the inner loop. Thus, in the jth iteration, if $\psi^k[I \leftarrow I_{j-1}^k]$ is unsatisfiable, an interpolant I_j^k is extracted with respect to the (A, B) pair where $A = I_{j-1}^k(V^1 \leftarrow V^0) \wedge T(V^0, V^1)$ and B is as before. Following this definition, I_j^k is an overapproximation of states reachable from I_{j-1}^k in one transition and $\langle I, I_1^k, \ldots, I_j^k \rangle$ is a safe FRS. The inner loop terminates either when the BMC formula it checks is satisfiable, or when an inductive invariant is found. In the latter case, the algorithm terminates concluding that the transition system is safe. In the former case, there are two cases to handle: If the BMC formula is satisfiable in the first iteration, a counterexample exists and the algorithm terminates, otherwise, the control is passed back to the outer loop, which increases k.
- **Outer Loop**: After the first iteration of the inner loop, overapproximated sets of reachable states are used as the initial condition of the checked BMC formulas. Thus, in case such a BMC formula becomes satisfiable, it is not clear if it is due to the existence of a counterexample or due to the overapproximation. When a BMC formula that uses an overapproximated set of states as the set of initial states becomes satisfiable, the control goes back to the outer loop that increases the bound k used for the BMC queries. Increasing k

helps to either find a real counterexample or to increase the precision of the overapproximation. Note that B represents all bad states and all states that can reach a bad state in $k-1$ transitions or less. Therefore, when k is increased, the precision of the computed interpolant is also increased. For a sufficiently large k, the approximation obtained through interpolation becomes precise enough such that the inner loop is guaranteed to find an inductive invariant if the system is safe [64], leading to the termination of the algorithm.

2) Interpolation Sequence-Based Model Checking: In [81], an interpolation sequence-based (ISB) algorithm is suggested for the computation of a *safe* FRS as part of the main BMC loop. Unlike ITP, ISB is integrated in BMC's main loop (avoiding nested loops). ISB starts with a $\langle F_0 = I\rangle$ as an FRS. It then operates just like BMC. In its first iteration, it solves φ^1. If the formula is satisfiable, a counterexample is found and the algorithm terminates. Otherwise, an interpolation sequence is extracted for $A_1 = I \wedge T$ and $A_2 = \neg P'$. In this case, the sequence contains the interpolant I_1^1. ISB then defines $F_1 = I_1^1[V^1 \leftarrow V]$, and the result is a safe FRS $\langle F_0, F_1\rangle$ (recall Definition 2).

Let us assume that the algorithm now executes the kth iteration. At the kth iteration, the FRS is $\langle F_0, \ldots, F_{k-1}\rangle$ and φ^k is checked. The goal of the kth iteration is to extend the FRS with a new element F_k. If φ^k is satisfiable, a counterexample is found and the algorithm terminates. In case it is unsatisfiable, an interpolation sequence $\langle I_1^k, \ldots, I_k^k\rangle$ is extracted with respect to $A_1 = I(V^0) \wedge T(V^0, V^1)$, $A_i = T(V^{i-1}, V^i)$ for $2 \leq i \leq k$ and $A_{k+1} = \neg P(V^k)$ (see Fig. 8). This interpolation sequence is used to extend the FRS. The ith element of the existing FRS is updated by defining $F_i = F_i \wedge I_i^k[V^i \leftarrow V]$ for $1 \leq i < k$ and F_k to be I_k^k ($F_k = I_k^k[V^k \leftarrow V]$). The result is a safe FRS of length k. At the end of the kth iteration, if an inductive invariant is found (Lemma 2), the algorithm terminates concluding that M is safe. Otherwise, the next iteration is executed.

A detailed description of ISB appears in [19] and [81], and a detailed comparison to ITP appears in [81].

F. IC3

The introduction of IC3 [14] has signified a change in the way SAT-based model checking is perceived. Usually referred to as the "monolithic" approaches, interpolation-based techniques, and even PBA, use the SAT solver as a blackbox that can either find a satisfying assignment or generate a proof of unsatisfiability. The proof of unsatisfiability represents a generalization of a bounded proof into a candidate inductive invariant. While this allows these approaches to utilize the strength of state-of-the-art SAT solvers, it gives them no control over the performed generalization and no way to control the "inductiveness" of the generated candidate.

IC3, however, waives some of the strengths of the SAT solver and, in return, gains control over the generation of the candidate inductive invariant. This is achieved by employing a very specific search strategy. IC3's search strategy is based on a backward search that starts from the unsafe (or "bad") states in $\neg P$. The algorithm maintains a monotonic safe FRS $F_0, \ldots, F_k$, where each frame F_i overapproximates the set of states reachable from I in up to i steps of T. In addition, IC3 maintains a queue of states s occurring in the FRS from which a bad state is reachable (via a sequence of steps from s to a state in $\neg P$, which we call a bad suffix). At each iteration, IC3 picks a state s from the queue, prioritizing states in frames with lower indices. Assume that s occurs in F_k (as in Fig. 9). Then, IC3 tries to find a one-step predecessor to s in F_{k-1} (e.g., state t in Fig. 9) in an attempt to extend the bad suffix until an initial state is reached. If the bad suffix is found to be unreachable (i.e., no predecessor t exists), then IC3 blocks the suffix using a process called *inductive generalization*. The generalization technique yields a clause that is inductive relative to F_{k-1} and blocks s, which is then used to strengthen the frames $F_0, \ldots, F_k$ of the FRS. The algorithm terminates if either a counterexample is found or a frame is determined to be an inductive invariant that proves the property.

Notably, the SAT queries made by IC3 involve only a single step of the transition relation. Each state s is represented by a conjunction of literals over V whose only satisfying assignment corresponds to s; accordingly, its negation $\neg s$ is a clause. Consequently, the SAT queries performed by IC3 are computationally cheap (in comparison to ITP). In addition, IC3 relies heavily on incremental solving (see Section II-C).

In the following, we describe IC3 in more detail. Given a transition system $M = \langle V, I, T, P\rangle$, the IC3 algorithm [14] iteratively refines and extends a monotonic safe FRS where the frames are in CNF.

In each iteration, the algorithm performs one of two actions.

- If no unsafe state is reachable from F_k (i.e., $F_k \wedge T \Rightarrow P'$), the algorithm extends the sequence with an additional frame $F_{k+1} = P$. F_{k+1} becomes the new *frontier*, and the resulting sequence is a monotonic safe FRS. In addition, for each frame F_i, $0 \leq i \leq k$, IC3 propagates clauses c in F_i forward to F_{i+1} if $F_i \wedge T \Rightarrow c'$ (where c' is obtained by replacing all variables in c with their primed counterparts). IC3 ensures that the clauses in each frame F_{i+1} (for $0 \leq i < k$) are a subset of the clauses of its predecessor F_i, enabling efficient syntactic checks for the equality of frames. If during the propagation it is discovered that $F_{i+1} = F_i$ for some i, the algorithm terminates since F_i is an inductive invariant proving the safety of the transition system.
- If $F_k \wedge T \not\Rightarrow P'$, an unsafe state is reachable from F_k. A predecessor s of an unsafe state can be extracted from a satisfying assignment of $F_k \wedge T \wedge \neg P'$ (see Fig. 9). The state s constitutes a *counterexample to*

induction (CTI) since it demonstrates that F_k is not inductive. Note that F_{k-1} does not contain s since its unsafe successor state would otherwise be reachable from F_{k-1}, violating the properties of a monotonic safe FRS. Subsequently, IC3 will try to prove the unreachability of s from F_{k-1} and adds the tuple $\langle s, k-1\rangle$ to a priority queue used to store proof obligations, accordingly.

A proof obligation $\langle s, i\rangle$ with $i = 0$ and a predecessor of s in I represents a violation of the property P and is reported. Otherwise, the algorithm checks whether the clause $\neg s$ is inductive relative to F_i by means of the *consecution query* $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$. If this attempt fails, a new CTI t (a predecessor of s, as illustrated in Fig. 9) can be extracted from a satisfying assignment of $F_i \wedge T \wedge s'$. Since the query $t \wedge T \Rightarrow s'$ holds by construction, t can be generalized to a partial assignment u by obtaining an unsatisfiable core of $t \wedge T \wedge \neg s'$ and dropping all literals of t not contained in the core [22]. The resulting *lifted* cube is added as a new proof obligation $\langle u, i-1\rangle$. A similar optimization can be achieved using ternary simulation [39].

If $\neg s$ is inductive relative to F_i, on the other hand, s and its successors are unreachable and can be removed from the priority queue. To accelerate convergence, IC3 deploys a *generalization* algorithm [47] to find a clause $c \subseteq \neg s$ such that $F_i \wedge c \wedge T \Rightarrow c'$, which is added to all frames F_j, $0 \leq j \leq i+1$.

IC3 owes its performance to numerous optimizations implemented in addition to its clever inductive generalization technique. For instance, IC3 attempts to *push* clauses c blocking a CTI that surfaced in frame F_i forward to frames F_j with $j > i$, in an attempt to avoid a reencounter with the same CTI in later frames. Furthermore, inductive invariants can be detected syntactically thanks to the CNF structure of the frames.

G. Bringing Interpolation and IC3 Together

Interpolation-based model checking and IC3, described in the previous sections, are two of the most successful methods for SAT-based model checking. The generalization techniques deployed in interpolation-based model checking and IC3, however, are very different: interpolation relies on a proof of unsatisfiability of a BMC instance and is usually referred to as a "monolithic" approach, while IC3 uses single-step queries and is referred to as "incremental."

The interpolation-based approach does not pose restrictions on the SAT solver's search strategy, thus leveraging advances in SAT and in interpolation. As a result, however, the technique does not offer much control over generalization. It is at the mercy of the choices made by the SAT solver, which provides a particular resolution proof, and of the procedure used to generate the interpolant. Furthermore, interpolants tend to be large, which poses additional limitation on their use.

Unlike interpolation-based methods, IC3 manages both the search for the counterexample as well as generalization [47] directly. Conceptually, IC3 is based on a backward search and can be seen as a SAT solver with a specific search strategy that is based on the BMC structure of the problem [5]. Execution traces are extended step by step, and *inductive generalization* is used to block suffixes that cannot be extended further.

While IC3 offers many advantages compared to the interpolation-based methods, including incremental solving and fine-grained control over generalization, it is limited to a fixed *local* search strategy that can be inefficient. The work in [5] tries to overcome the locality of IC3 by applying unit propagation (BCP) to all frames in a global manner. Yet, it still maintains IC3's backward search by forcing a specific decision order.

To remedy the weaknesses of each approach, some techniques [82], [83] combine ideas from both interpolation and IC3. We briefly discuss two recent approaches here and refer the reader to the papers for a more detailed description.

1) Interpolation With CNF Interpolants: As mentioned in Section II-E, interpolants are not unique and can vary in structure. In [83], the authors describe an efficient algorithm for computing interpolants in CNF. The algorithm uses both the resolution refutation generated by the SAT solver, and IC3-style inductive generalization to compute an interpolant. At first, a formula I_w that approximates an interpolant is derived from a resolution refutation. Due to the approximation, a second phase is needed to transform the approximated interpolant into an actual interpolant. Recall from Section III-E that in the context of ITP the following partitioning is used:

- $A \stackrel{\text{def}}{=} F(V) \wedge T(V, V^1)$, where $F(V)$ is a propositional formula representing a set of states;
- $B \stackrel{\text{def}}{=} \text{path}^{1,k} \wedge (\bigvee_{i=1}^{k} \neg P(V^i))$ representing bad states and their pre-image up to $k-1$ steps.

The approximated interpolant I_w is guaranteed to be in CNF, and satisfies all properties of an interpolant except that it is not necessarily inconsistent with B. I_w is then iteratively refined by eliminating all states s that satisfy $I_w \wedge B$. Since $A \wedge B$ is known to be unsatisfiable, these states can be blocked using IC3's generalization mechanism. The resulting formula is an interpolant in CNF, which is then used in the context of interpolation-based model checking [64] to compute a monotonic FRS in CNF. The model checking algorithm (called CNF-ITP) is modified to take advantage of the fact that the monotonic FRS is in CNF by incorporating ideas of IC3 (such as pushing clauses forward in the FRS).

2) Avy—An Interpolating IC3: The authors of [82] combine sequence interpolation [81] with IC3 to create Avy.

Like IC3 and CNF-ITP, AVY computes a monotonic FRS in CNF, but starts from an interpolation sequence obtained from a BMC query (as in Fig. 8). AVY then uses IC3 to transform the interpolants into CNF. Intuitively, the interpolant I_1^k can be used to form a transition system $M_1 \stackrel{\text{def}}{=} \langle V, I, T, (I \vee I_1^k)\rangle$, such that the first frame F_1 of the corresponding IC3 instance satisfies $I \Rightarrow F_1$ and $F_1 \Rightarrow (I \vee I_1^k)$. F_1 is then in turn used to construct transition system $M_2 \stackrel{\text{def}}{=} \langle V, F_1, T, (F_1 \vee I_2^k)\rangle$, and IC3 is used to construct a formula F_2 with $F_1 \Rightarrow F_2$ and $F_2 \Rightarrow (F_1 \vee I_2^k)$, and so on.

The result of this process is a safe monotonic FRS $\langle I, F_1, F_2, \ldots\rangle$ that is identical, in characteristics, to the monotonic FRS computed by IC3. Thus, AVY can make use of all enhancements that make IC3 efficient without imposing any restrictions on the SAT solver's search strategy. Reference [82] shows that AVY is complementary to IC3.

H. Hardware Model Checking Competition

The hardware model checking competition (HWMCC) annually evaluates contenders for the best (academic) model checking tool. The Web page http://fmv.jku.at/hwmcc provides an up-to-date empirical evaluation of verification algorithms.

In the single safety property track of the recent edition of the HWMCC,[2] the winning verification tools were ABC [15] and V3[3] (both of which incorporate a range of techniques including BDD-based model checking, BMC, interpolation, and IC3), followed by IIMC,[4] which is based on IC3. NUXMV [21], which also incorporates multiple algorithms including sequence interpolation, k-induction, and IC3, and AVY,[5] ranked second in the UNSAT category and described in Section III-G, were tied sixth.

The prevalence of tools implementing multiple algorithms shows that there is no clear-cut winner in terms of model checking techniques. While IC3 certainly stands out (considering the fact that IIMC implements only this approach), no single algorithm can typically solve all problem instances [20]. This gives rise to *portfolio* solvers, which in the domain of SAT solvers have proven very successful [87]. Portfolio solvers either run different algorithms in parallel, or use heuristics to select a promising algorithm. We refer the reader to [18] for details.

IV. CONCLUSION

SAT solving and hardware model checking have a history of mutually beneficial development. Techniques such as CDCL and efficient BCP developed by the EDA community revolutionized the field of satisfiability checking. In turn, the resulting performance boost of SAT solvers arguably marked the coming of age of model checking, enabling its application to industrial-size designs. Driven by initiatives such as the SAT competition and the HWMCC, and vivid exchange and collaboration between academia and the industrial research community, the trend of ever-improving performance and scalability in both fields seems unbroken. Recent work, for instance, combines abstraction and IC3 for model checking word-level designs [58], [85]. One of the frontiers in both fields not addressed in this paper is parallelism—the IC3 algorithm "lends itself to a parallel implementation" [14] (realized in IIMC), and parallel SAT solving is a promising field that still holds significant challenges [46]. We are confident that 10 years from now we will get to report on another decade of successful symbiosis of SAT solving and model checking. ■

[2]http://fmv.jku.at/hwmcc14cav/
[3]http://dvlab.ee.ntu.edu.tw/~publication/V3/
[4]http://ecee.colorado.edu/wpmu/iimc/
[5]http://arieg.bitbucket.org/avy/

Acknowledgment

The authors thank A. Biere, M. Chan, U. Egly, and the anonymous reviewers for their helpful comments.

REFERENCES

[1] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment," in *Correct Hardware Design and Verification Methods*, vol. 3725, *LNCS*. New York, NY, USA: Springer-Verlag, 2005, pp. 254–268.
[2] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais, "A generalized framework for conflict analysis," in *Theory and Applications of Satisfiability Testing*, vol. 4996, *LNCS*. New York, NY, USA: Springer-Verlag, 2008, pp. 21–27.
[3] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. IJCAI*, 2009, pp. 399–404.
[4] G. Audemard and L. Simon, "Refining restarts strategies for SAT and UNSAT," in *Constraint Programming*, vol. 7514, *LNCS*. New York, NY, USA: Springer-Verlag, 2012, pp. 118–126.
[5] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu, "Efficient modular SAT solving for IC3," in *Proc. FMCAD*, 2013, pp. 149–156.
[6] P. Beame, H. Kautz, and A. Sabharwal, "Towards understanding and harnessing the potential of clause learning," *J. Artif. Intell. Res.*, vol. 22, no. 1, pp. 319–351, Dec. 2004.
[7] A. Belov, M. Heule, and J. Marques-Silva, "MUS extraction using clausal proofs," in *Theory and Applications of Satisfiability Testing*, vol. 8561, *LNCS*. New York, NY, USA: Springer-Verlag, 2014, pp. 48–57.
[8] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal, "Model checking at IBM," *Formal Methods Syst. Design*, vol. 22, no. 2, pp. 101–108, 2003.
[9] A. Biere, "Adaptive restart strategies for conflict driven SAT solvers," in *Theory and Applications of Satisfiability Testing*, vol. 4996, *LNCS*. New York, NY, USA: Springer-Verlag, 2008, pp. 28–33.
[10] A. Biere, "PicoSAT essentials," *J. Satisfiability, Boolean Modeling Comput.*, vol. 4, no. 2–4, pp. 75–97, 2008.
[11] A. Biere, "Preprocessing and inprocessing techniques in SAT," in *Proc. HVC*, vol. 7261, *LNCS*, 2011, p. 1.
[12] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1579, *LNCS*. New York, NY, USA: Springer-Verlag, 1999, pp. 193–207.
[13] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, vol. 185, Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press, Feb. 2009.
[14] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking and Abstract Interpretation*, vol. 6538, *LNCS*. New York, NY, USA: Springer-Verlag, 2011, pp. 70–87.
[15] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, vol. 6174, *LNCS*. New York, NY, USA: Springer-Verlag, 2010, pp. 24–40.
[16] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model

checking:10^{20} states and beyond," in *Proc. LICS*, 1990, pp. 428–439.
[18] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," *Formal Methods Syst. Design*, vol. 39, no. 2, pp. 205–227, 2011.
[19] G. Cabodi, S. Nocco, and S. Quer, "Interpolation sequences revisited," in *Proc. DATE*, 2011, pp. 316–322.
[20] G. Cabodi, M. Palena, and P. Pasini, "Interpolation with guided refinement: Revisiting incrementality in sat-based unbounded model checking," in *Proc. FMCAD*, 2014, pp. 43–50.
[21] R. Cavada *et al.*, "The nuXmv symbolic model checker," in *Computer Aided Verification*, vol. 8559, *LNCS*. New York, NY, USA: Springer-Verlag, 2014, pp. 334–342.
[22] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Proc. FMCAD*, 2011, pp. 135–143.
[23] A. Cimatti, A. Griggio, and R. Sebastiani, "Efficient generation of Craig interpolants in satisfiability modulo theories," *Trans. Comput. Logic*, vol. 12, no. 1, p. 7, 2010.
[24] K. Claessen and N. Sörensson, "New techniques that improve MACE-style finite model finding," in *Proc. Model Comput.—Principles, Algor., Appl.*, 2003.
[25] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," in *Proc. POPL*, 1992, pp. 343–354.
[26] E. Clarke, O. Grumberg, and D. Peled, Model Checking. Cambridge, MA, USA: MIT Press, Dec. 1999.
[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.
[28] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, vol. 1855, *LNCS*. New York, NY, USA: Springer-Verlag, 2000, pp. 154–169.
[29] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.
[30] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman, "Completeness and complexity of bounded model checking," in *Verification, Model Checking and Abstract Interpretation*, vol. 2937, *LNCS*. New York, NY, USA: Springer-Verlag, 2004, pp. 85–96.
[31] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. ACM STOC*, 1971, pp. 151–158.
[32] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *J. Symb. Logic*, vol. 22, no. 3, pp. 250–268, 1957.
[33] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," *J. Symb. Logic*, vol. 22, no. 3, pp. 269–285, 1957.
[34] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, Jul. 1962.
[35] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–214, Jul. 1960.
[36] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Trans. CAD Integrated Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.
[37] V. D'Silva, M. Purandare, G. Weissenbacher, and D. Kroening, "Interpolant strength," in *Verification, Model Checking and Abstract Interpretation*, vol. 5944, *LNCS*. New York, NY, USA: Springer-Verlag, 2010, pp. 129–145.
[38] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Theory and Applications of Satisfiability Testing*, vol. 3569, *LNCS*. New York, NY, USA: Springer-Verlag, 2005, pp. 102–104.
[39] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proc. FMCAD*, 2011, pp. 125–134.
[40] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.
[41] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, vol. 2919, *LNCS*. New York, NY, USA: Springer-Verlag, 2004, pp. 333–336.
[42] A. V. Gelder, "Generalizations of watched literals for backtracking search," in *Proc. ISAIM*, 2002. [Online]. Available: http://rutcor.rutgers.edu/~amai/aimath02/; http://rutcor.rutgers.edu/~amai/aimath02/PAPERS/32.ps
[43] R. Gerth, "Model checking if your life depends on it: A view from intel's trenches," in *Model Checking and Software Verification*, vol. 2057, *LNCS*. New York, NY, USA: Springer-Verlag, 2001, p. 15.
[44] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Proc. DATE*, 2003, pp. 886–891.
[45] A. Gurfinkel and Y. Vizel, "DRUPing for interpolants," in *Proc. FMCAD*, 2014, DOI: 10.1109/FMCAD.2014.6987601.
[46] Y. Hamadi and C. M. Wintersteiger, "Seven challenges in parallel SAT solving," *AI Mag.*, vol. 34, no. 2, pp. 99–106, 2013.
[47] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *Proc. FMCAD*, 2013, pp. 157–164.
[48] M. Heule, W. A. Hunt, Jr., and N. Wetzler, "Trimming while checking clausal proofs," in *Proc. FMCAD*, 2013, pp. 181–188.
[49] G. Huang, "Constructing Craig interpolation formulas," in *Computing and Combinatorics*, vol. 959, *LNCS*. New York, NY, USA: Springer-Verlag, 1995, pp. 181–190.
[50] J. Huang, "The effect of restarts on the efficiency of clause learning," in *Proc. IJCAI*, 2007, pp. 2318–2323.
[51] M. K. Iyer, G. Parthasarathy, and K. Cheng, "SATORI—A fast sequential SAT engine for circuits," in *Proc. ICCAD*, 2003, pp. 320–325.
[52] M. Järvisalo, M. Heule, and A. Biere, "Inprocessing rules," in *Proc. IJCAR*, vol. 7364, *LNCS*, 2012, pp. 355–370.
[53] R. Jhala and R. Majumdar, "Software model checking," *Comput. Surveys*, vol. 41, no. 4, 2009.
[54] R. Jhala and K. L. McMillan, "Interpolant-based transition relation approximation," in *Computer Aided Verification*, vol. 3576. New York, NY, USA: Springer-Verlag, 2005, pp. 39–51.
[55] H. A. Kautz, E. Horvitz, Y. Ruan, C. P. Gomes, and B. Selman, "Dynamic restart policies," in *Proc. AAAI/IAAI*, 2002, pp. 674–681.
[56] J. Krajíček, "Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic," *J. Symb. Logic*, vol. 62, no. 2, pp. 457–486, 1997.
[57] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton, NJ, USA: Princeton Univ. Press, 1994.
[58] S. Lee and K. A. Sakallah, "Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction," in *Computer Aided Verification*, vol. 8559, *LNCS*. New York, NY, USA: Springer-Verlag, 2014, pp. 849–865.
[59] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of Las Vegas algorithms," in *Proc. ISTCS*, 1993, pp. 128–133.
[60] S. Malik and G. Weissenbacher, "Boolean satisfiability solvers: Techniques and extensions," in *Software Safety and Security—Tools for Analysis and Verification*. Amsterdam, The Netherlands: IOS Press, 2012.
[61] S. Malik, Y. Zhao, C. F. Madigan, L. Zhang, and M. W. Moskewicz, "Chaff: Engineering an efficient SAT solver," in *Proc. DAC*, 2001, pp. 530–535.
[62] J. A. P. Marques-Silva and K. A. Sakallah, "GRASP—A new search algorithm for satisfiability," in *Proc. ICCAD*, 1996, pp. 220–227.
[63] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *Computer Aided Verification*, vol. 2404, *LNCS*. New York, NY, USA: Springer-Verlag, 2002, pp. 250–264.
[64] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, vol. 2725, *LNCS*. New York, NY, USA: Springer, 2003, pp. 1–13.
[65] K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification*, vol. 4144, *LNCS*. New York, NY, USA: Springer, 2003, pp. 123–136.
[66] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2619, *LNCS*. New York, NY, USA: Springer-Verlag, 2003, pp. 2–17.
[67] K. L. McMillan, "The SMV system," Carnegie Mellon University, Tech. Rep. CMU-CS-92-131, 1992.
[68] A. Mishchenko *et al.*, "GLA: Gate-level abstraction revisited," in *Proc. DATE*, 2013, pp. 1399–1404.
[69] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing*, vol. 4501, *LNCS*. New York, NY, USA: Springer-Verlag, 2007, pp. 294–299.
[70] K. Pipatsrisawat and A. Darwiche, "On modern clause-learning satisfiability solvers," *J. Autom. Reasoning*, vol. 44, no. 3, pp. 277–301, 2010.
[71] K. Pipatsrisawat and A. Darwiche, "On the power of clause-learning sat solvers as resolution engines," *Artif. Intell.*, vol. 175, no. 2, pp. 512–525, 2011.
[72] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *J. Symb. Comput.*, vol. 2, no. 3, pp. 293–304, 1986.
[73] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *Softw. Tools Technol. Transfer*, vol. 7, no. 2, pp. 156–173, 2005.
[74] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *J. Symb. Logic*, vol. 62, no. 3, pp. 981–998, 1997.
[75] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Proc. Int. Symp. Programming*, 1982, pp. 337–351.
[76] S. F. Rollini, O. Sery, and N. Sharygina, "Leveraging interpolant strength in model checking," in *Computer Aided Verification*, vol. 7358, *LNCS*. New York, NY, USA: Springer-Verlag, 2012, pp. 193–209.
[77] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design*, vol. 1954, *LNCS*. New York, NY, USA: Springer-Verlag, 2000, pp. 108–125.

[78] J. P. M. Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Comput.*, vol. 48, no. 5, pp. 506–521, May 1999.
[79] C. Sinz and M. Iser, "Problem-sensitive restart heuristics for the DPLL procedure," in *Theory and Applications of Satisfiability Testing*, vol. 5584, *LNCS*. New York, NY, USA: Springer-Verlag, 2009, pp. 356–362.
[80] G. Tseitin, "On the complexity of proofs in propositional logics," in *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, vol. 2, J. Siekmann and G. Wrightson, Eds. New York, NY, USA: Springer-Verlag, 1983, Originally published 1970.
[81] Y. Vizel and O. Grumberg, "Interpolation-sequence based model checking," in *Proc. FMCAD*, 2009, pp. 1–8.
[82] Y. Vizel and A. Gurfinkel, "Interpolating property directed reachability," in *Computer Aided Verification*, vol. 8559, *LNCS*. New York, NY, USA: Springer-Verlag, 2014, pp. 260–276.
[83] Y. Vizel, V. Ryvchin, and A. Nadel, "Efficient generation of small interpolants in CNF," in *Computer Aided Verification*, vol. 8044, *LNCS*. New York, NY, USA: Springer-Verlag, 2013, pp. 330–346.
[84] G. Weissenbacher, "Interpolant strength revisited," in *Theory and Applications of Satisfiability Testing*, vol. 7317, *LNCS*. New York, NY, USA: Springer-Verlag, 2012, pp. 312–326.
[85] T. Welp and A. Kuehlmann, "QF_BV model checking with property directed reachability," in *Proc. DATE*, 2013, pp. 791–796.
[86] J. Whittemore, J. Kim, and K. A. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Proc. ACM DAC*, 2001, pp. 542–545.
[87] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res.*, vol. 32, pp. 565–606, 2008.
[88] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proc. IEEE DATE*, 2003, p. 10880.

ABOUT THE AUTHORS

Yakir Vizel received the Ph.D. degree in computer science from The Technion, Haifa, Israel, in 2014.

He is a Postdoctoral Research Associate with Princeton University, Princeton, NJ, USA, working under the supervision of Prof. Sharad Malik. Previously, he worked in the electronic design automation (EDA) industry for 10 years developing model checking and formal verification solutions. His research interests include model checking, abstraction, satisfiability solving, and interpolation.

Georg Weissenbacher received the doctorate degree in computer science from the University of Oxford, Oxford, U.K, in 2010.

He is an Assistant Professor at TU Wien, Vienna, Austria. He spent two years as a Postdoctoral Research Associate with Princeton University, Princeton, NJ, USA. Funded by a Vienna Research Groups for Young Investigators grant of the Vienna Science and Technology Fund (WWTF), his research revolves around satisfiability solving and interpolation. Further, it includes software model checking as well as the localization and explanation of post-silicon faults and concurrency bugs.

Sharad Malik (Fellow, IEEE) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology (IIT), New Delhi, India, in 1985, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, CA, USA, in 1987 and 1990, respectively.

Currently, he is the George Van Ness Lothrop Professor of Engineering with Princeton University, Princeton, NJ, USA, and the Chair of the Department of Electrical Engineering. Previously, he served as the Director of the Keller Center for Innovation in Engineering Education, Princeton University, from 2006 to 2011, and the Director of the multi-university Gigascale Systems Research Center from 2009 to 2012. His research focuses on design methodology and design automation for computing systems. His research in functional timing analysis and propositional satisfiability has been widely used in industrial electronic design automation tools.

Prof. Malik is a Fellow of ACM. He has received the DAC Award for the most cited paper in the 50-year history of the conference in 2013, the CAV Award for fundamental contributions to the development of high-performance Boolean satisfiability solvers in 2009, the ICCAD Ten Year Retrospective Most Influential Paper Award in 2011, the Princeton University President's Award for Distinguished Teaching in 2009, as well as several other research and teaching awards. In 2009, he received the IIT Delhi Distinguished Alumni Award.