Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Survey of Recent Advances in SAT-Based Formal Verification

Mukul R Prasad[1], **Armin Biere**[2], **Aarti Gupta**[3]

[1] Fujitsu Labs. of America, Sunnyvale, CA, USA
[2] Johannes Kepler University, Linz, Austria
[3] NEC Labs. America, Princeton, NJ, USA

The date of receipt and acceptance will be inserted by the editor

Abstract Dramatic improvements in SAT solver technology over the last decade, and the growing need for more efficient and scalable verification solutions have fueled research in verification methods based on SAT solvers. This paper presents a survey of the latest developments in SAT-based formal verification, including incomplete methods such as bounded model checking, and complete methods for model checking. We focus on how the surveyed techniques formulate the verification problem as a SAT problem, and how they exploit crucial aspects of a SAT solver, such as application-specific heuristics and conflict-driven learning. Finally, we summarize the noteworthy achievements in this area so far, and note the major challenges in making this technology more pervasive in industrial design verification flows.

Key words: Verification, SAT, Model Checking, QBF, ATPG

1 Introduction

Functional verification of digital hardware designs has become one of the most expensive and time-consuming components of the current product development cycle. Symbolic model checking based on BDDs [22,72] have come a long way since their introduction more than a decade ago. However, they are still incapable of handling the largest problems encountered in current industrial practice. Reduction in feature size coupled with the recent move towards IP-based design has led to dramatic increases in the size and complexity of systems that are being designed, thereby posing new challenges for functional verification methods. Hence there is a growing need to investigate and develop more robust and scalable verification methods based on novel and alternative technologies.

Verification methods based on SAT solvers have recently emerged as a promising solution. Dramatic improvements in SAT solver technology over the past decade have led to the development of several powerful SAT solvers [45,71,77,105]. Verification methods based on these solvers have been shown to push the envelope of functional verification in terms of both capacity and efficiency, as reported in several academic and industrial case studies [4,16,19,31]. This has fueled further interest and intense research activity in the area of SAT-based verification.

This paper surveys the recent developments in SAT-based formal verification techniques and methodologies. The work surveyed falls primarily in the category of property verification or model checking methods since such has been the focus of most recent works on SAT-based verification. For other verification applications of SAT methods, such as combinational equivalence checking, the interested reader is referred to [47,68].

Additionally, there is an interesting body of work based on applying SAT to richer types of specifications and logics, which, due to lack of space can not be covered in this short survey. Here is a list of recent relevant topics, which may serve as a starting point for the interested reader: quantifier free fragments of first order logic [9,87,100], Presburger Arithmetic [97], monadic second order logic [6], object oriented software specifications [60].

1.1 Organization

The survey is organized as follows. Section 2 briefly reviews the SAT problem, basic SAT algorithms and advanced features of modern SAT solvers, and model checking. Section 3 discusses work on *bounded model checking (BMC)* including ways of strengthening SAT-based BMC with BDD-based analysis and several industrial case studies comparing SAT-BMC with traditional BDD-based symbolic model checking. Section 4 reviews techniques that implement complete methods for model checking based on state-space search, inductive reasoning and abstraction-refinement.

Recently there have been some successful attempts at using sequential ATPG tools for model checking. These are surveyed in Section 5. Another recent development has been the use of *Quantified Boolean Formulae (QBF)* solvers, a generalization of SAT, to solve model checking problems. The state-of-the-art in QBF solving and its applications to verification are discussed in Section 6. We conclude the paper in Section 7 with a summary of the major achievements in SAT-based verification to date and some thoughts on the future prospects and challenges for SAT-based verification.

2 Background

2.1 The Boolean Satisfiability Problem

The Boolean Satisfiability (SAT) problem is a well-known constraint satisfaction problem, with many applications in the fields of VLSI Computer-Aided Design and Artificial Intelligence. Given a propositional formula φ, the Boolean Satisfiability problem posed on φ is to determine whether there exists a variable assignment under which φ evaluates to true. Such an assignment, if it exists, is called a *satisfying assignment* for φ and φ is called *satisfiable*. Otherwise φ is said to be *unsatisfiable*. The SAT problem is known to be NP-Complete [42]. However, in practice, there has been tremendous progress in SAT solvers technology over the years, summarized in a recent survey [107]. Earlier work in the context of theorem proving is covered in [63].

Most SAT solvers use a *Conjunctive Normal Form (CNF)* representation of the Boolean formula. In CNF, the formula is represented as a conjunction of clauses, each clause is a disjunction of literals, and a literal is a variable or its negation. Note that in order for the formula to be satisfied, each clause must also be satisfied, *i.e.*, evaluate to true. There exist polynomial algorithms (*e.g.*, [81,98]) to transform an arbitrary propositional formula into a *satisfiability equivalent* CNF formula that is satisfiable if and only if the original formula is satisfiable. Similarly a Boolean circuit may be encoded as a satisfiability equivalent CNF formula using the method of [65]. Alternatively, for SAT applications arising from the circuit domain, the SAT solver may be modified to work directly on the Boolean circuit representation.

2.2 SAT Solvers

Most modern SAT solvers are based on the *Davis-Putnam-Logemann-Loveland (DPLL) algorithm* [32,33], which performs a branching search with backtracking. The DPLL algorithm is sound and complete, *i.e.*, it finds a solution if and only if the formula is satisfiable. In this section, we summarize the main features of modern DPLL-based SAT solvers. This provides the context for enhancements targeted at verification applications, discussed in the rest of the paper.

Probabilistic SAT solvers, including WALKSAT [85] and GSAT [86], are based on stochastic local search instead of DPLL. They have their strength on random SAT instances, but in practice do not work well on structured instances obtained from real verification problems.

```
sat-solve()
  if preprocess() = CONFLICT then
    return UNSAT;
  while TRUE do
    if not decide-next-branch() then
      return SAT;
    while deduce() = CONFLICT do
      blevel ⇐ analyze-conflict();
      if blevel = 0 then
        return UNSAT;
      backtrack (blevel);
    done;
  done;
```

Figure 1. DPLL-based SAT Solver

The basic skeleton of DPLL-based SAT solvers is shown in Figure 1, adapted from the GRASP work [71]. The initial step consists of some preprocessing, during which it may be discovered that the formula is unsatisfiable. The outer loop starts by choosing an unassigned variable, and a value to assign to it (decide-next-branch). If no such variable exists, a solution has been found. Otherwise, the variable assignments deducible from this decision are made (using deduce), through a procedure called *Boolean Constraint Propagation (BCP)*. It typically consists of iterative application of the *unit clause rule*, which is invoked whenever a clause becomes a unit clause, *i.e.*, all but one of its literals are false and the remaining literal is unassigned. According to the rule, the last unassigned literal is *implied* to be true – this avoids the search path where the last literal is also false, since such a path cannot lead to a solution. A *conflict* occurs when a variable is implied to be true as well as false. If no conflict is discovered during BCP, then the outer loop is repeated, by choosing the next variable for making a decision. However, if a conflict does occur, *backtracking* is performed within an inner loop in order to undo some decisions and their implications. If all decisions need to be undone (*i.e.*, the backtracking level *blevel* is 0), the formula is declared unsatisfiable since the entire search space has been exhausted.

The original DPLL algorithm used chronological backtracking, *i.e.*, it would backtrack up to the most recent decision, for which the other value of the variable had not been tried. However, modern SAT solvers use *conflict analysis* techniques (shown as (analyze-conflict) in the figure), to analyze the reasons for a conflict. Conflict analysis is used to perform *conflict-driven learning* and *conflict-driven backtracking*, which were incorporated independently in GRASP [71] and rel-sat [11]. Conflict-driven learning consists of adding *conflict clauses* to the formula, in order to avoid the same conflict in the future. Conflict-driven backtracking allows non-

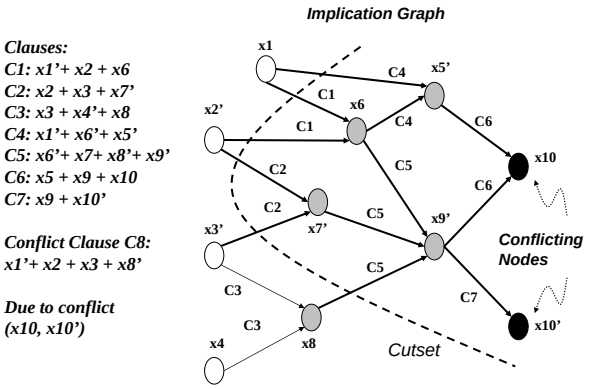

Figure 2. Conflict Analysis using an Implication Graph

chronological backtracking, *i.e.*, up to the closest decision which caused the conflict. These techniques greatly improve the performance of the SAT solver on structured problems.

The essential component of conflict analysis is an *implication graph* [71,106], which captures the current state of the SAT solver. A small example of an implication graph is shown in Figure 2, where the original SAT problem consists of clauses $C1-C7$, as shown on the left. In an implication graph, nodes represent assignments to variables. For example, node *x1* represents $x1=1$, and node $x5'$ represents $x5=0$.

Edges in an implication graph represent clauses, which cause implications due to source nodes on sink nodes. For example, when $x1=1$ and $x2=0$, clause *C1* causes an implication $x6=1$. This is shown as two edges – between *x1* and *x6*, and between $x2'$ and *x6* – both marked with clause *C1* as shown. Nodes with no incoming edges, such as *x1*, denote decision assignments (shown as white nodes in the figure). A *conflict* is indicated when there are two nodes in the graph with opposite values assigned to the same variable. In this example, a conflict is indicated by nodes *x10* and $x10'$, which are called conflicting nodes. Conflict analysis takes place by following back the edges from the conflicting nodes, up to any edge cutset which separates the conflicting nodes from the decision nodes. An example cutset is shown by the dashed line in the figure. A conflict clause is derived from the variables feeding into the chosen cutset to capture the reasons for the conflict. It also corresponds to a resolution on all the clauses associated with the edges traversed up to the cutset. In this example, conflict clause *C8* is derived as shown, corresponding to the observation that a partial assignment $(x1=1,x2=0,x3=0,x8=1)$ always leads to a conflict. For conflict-driven learning, the derived clause *C8* is added to the clause database in order to avoid the same conflict in the future.

Many other advances have been made in these basic components which comprise the DPLL-based SAT solver – decision engine (heuristics for choosing decision variables and values), deduction engine (data structures and heuristics for performing BCP and detecting conflicts), diagnosis engine (heuristics for conflict-driven learning). Some of these are described in the remainder of this section.

2.2.1 CNF-based SAT Solvers

An interesting property of CNF representations was first exploited by Zhang in the SATO SAT solver [105], to improve the performance of BCP. It proposed the use of head and tail pointers to point to non-false literals in the list representation of a clause, and maintained the *strong invariant* that all literals before the head pointer, and all literals after the tail pointer, are false. Clearly, detection of a unit clause during BCP becomes easy, *i.e.*, when the head and tail pointers coincide on an unassigned literal. The main advantage is that the clause status is updated *only* when either of the head/tail literals is assigned a false value during BCP. In particular, this eliminates an update when any of the other literals in the clause is assigned a value. When the head/tail literal is assigned a false value during BCP, the associated pointer needs to be moved to another non-false literal if it exists. This is facilitated by the strong invariant. However, during backtracking, the head/tail pointers may need to be moved back again, in order to maintain the strong invariant.

A different tradeoff was proposed by Moskewicz *et al.* in the Chaff SAT solver [77]. Its BCP scheme, known as *two literal watching with lazy update*, is also based on tracking only two literals per clause during BCP. However, Chaff maintains a *weak invariant*, whereby the two watched literals are required to be non-false, but there is no ordering requirement with respect to other false literals. Again, detection of a unit clause during BCP is easily performed by checking whether both watched pointers coincide, and whether clause updates on assignment to other literals are eliminated. Note that due to the weaker invariant, more work than SATO may be required during BCP, to search for a non-false literal when one of the two watched literals is assigned a false value. However, the weaker invariant ensures that no additional work is required during backtracking. This tradeoff has been shown to work better in practice.

Chaff also proposed a useful decision heuristic that prioritizes the literals that appear in recent conflict clauses. Recall that conflict clauses are added due to conflict-driven learning, which is very beneficial for SAT solvers on structured problems. This was taken a step further by Goldberg and Novikov in the BerkMin SAT solver [45], which prioritizes all literals involved in the conflict analysis, and not just those that appear in the conflict clause. The performance improvement due to these decision heuristics is additional testament to the importance of conflict-driven learning in practice.

More recently, additional information recorded during conflict analysis has been used very effectively to provide a proof when a formula is determined to be unsatisfiable by the SAT solver. This proof can be independently checked to verify the SAT solver itself [46,109]. These techniques can also be

easily adapted to identify a subset of clauses from the original problem, called the *unsatisfiable core* [109, 75], such that these clauses are sufficient for implying unsatisfiability. The use of such techniques in verification applications are described in more detail in Section 4.

2.2.2 Circuit-based SAT Solvers

SAT has many applications in the logic circuit domain, such as automatic test pattern generation (ATPG), verification, timing analysis, *etc.* The Boolean reasoning problem is typically derived from the circuit structure. This has also led to interest in circuit-based SAT solvers [38, 44, 64]. These work directly on the circuit structure, and use circuit specific heuristics to guide the search. In general, attempts to include circuit structure information into CNF-based SAT solvers have been unsuccessful due to significant overhead.

Among verification applications, recently Kühlmann *et al.* [64] focused on SAT techniques for a simple, uniform gate-level representation of circuits, and their integration with other useful techniques like BDD sweeping and dynamic circuit transformation. Circuit-based BCP is performed by a single table lookup per gate, in contrast to CNF-based updates for potentially three equivalent clauses. This improves the BCP performance. However, there is no effective way to perform conflict-driven learning. The bottleneck is that conflict clauses correspond to large *OR-tree* circuits.

2.2.3 Hybrid SAT Solvers

More recently, there has been an effort by Ganai *et al.* [41] to combine the relative benefits of CNF-based and circuit-based SAT solvers. In particular, their hybrid SAT solver incorporates efficient circuit-based BCP techniques, along with conflict analysis techniques of CNF-based solvers. The original circuit problem is represented as a simple gate-level netlist, while the learned conflict clauses are represented in CNF. The BCP engine consists of table lookups for the gates, and a Chaff-style two-literal watching scheme for conflict clauses. Note that since the clauses for a simple gate are short clauses (3-literals or less), a single table lookup is cheaper than multiple clause updates. On the other hand, since conflict clauses tend to be much longer, a two-literal watching scheme (which avoids multiple updates) is more useful than multiple table lookups for its many literals. This enables consistent speedups in the BCP performance. Their hybrid representation of the Boolean problem allows exploitation of both circuit-based and CNF-based decision heuristics.

Another effort by Lu *et al.* [69] used these ideas along with additional conflict-driven learning, in order to improve the SAT solver performance. The idea is to use cheaper methods (such as simulation) to find candidate pairs of co-related signals in the given circuit. Then the inequivalence of the co-related signals is added as a constraint to the SAT problem. Since this constraint is likely to be conflicting, it provides additional opportunities for the SAT solver to perform conflict-driven learning, which can potentially improve its performance on larger problems.

2.3 Model Checking

With the introduction of *bounded model checking* [15] it became clear that SAT can be used for *model checking* [27]. One can even argue, that currently one of the main driving forces behind SAT research is its application to model checking. The purpose of this section is to give a short overview on the history and terminology of model checking. More details can be found in the text book [28] or the survey [30].

The target of model checking is the verification of sequential properties of dynamic systems. A dynamic system has a state component which changes over time. Typical systems are sequential circuits, which contain delay elements, such as flip-flops and latches. Verification of sequential circuits is also the main application area of this survey.

Model checking, in the first place, is only applicable to finite systems. However, if suitable *finite abstractions* can be found, then some classes of infinite systems can be checked as well. Applications of infinite model checking are real-time systems, modeled as timed automata [3], or even system software [8]. Further, model checking has also been applied successfully in the context of telecommunication protocols and cache coherence protocols. It must be noted, that SAT has mainly been used for model checking sequential circuits. However, it is apparent that the verification of more general systems can also benefit from SAT technology.

Sequential properties are usually represented in temporal logic [36]. Formulas of temporal logic try to express system behavior over time. There are various variants of temporal logic, such as *Linear Temporal Logic (LTL)* or *Computation Tree Logic (CTL)*, which usually require dedicated algorithms. In this paper we focus on *simple safety properties*, also often called invariants, written in CTL as $\mathbf{AG}p$. This formula specifies, that for all executions paths, globally in all states along the path, the property p holds. Alternatively, it states the property that $\neg p$, read as *not p*, which could be some catastrophic system state, can not be reached. Note that for finite systems, many practically relevant properties can be translated into simple safety properties [84].

Originally [27] model checking used an explicit representation of states. A typical implementation [55] of this type of *explicit model checking* stores individual states in a large hash table, memorizing the states reached during a depth first traversal of the state space. Since the number of states of even small systems can be very large, *e.g.*, a 128 bit shift register has 2^{128} states, this method does not scale, in particular for sequential circuits. One solution to this so called *state explosion problem* is *symbolic model checking* [72], which operates on sets of states instead of individual states and represents sets of states symbolically in a compact form.

In the past, Binary Decision Diagrams (BDDs) [21] and variants have frequently been used as representation for sets of states. They also allow efficient computation of Boolean

```
model-check^μ_forward (I, T, B)
    S_C = ∅; S_N = I;
    while S_C ≠ S_N do
      S_C = S_N;
      if B ∩ S_C ≠ ∅ then
        return "found error trace to bad states";
      S_N = S_C ∪ Img(S_C);
    done;
    return "no bad state reachable";
```

Figure 3. Forward least fixpoint algorithm for safety properties

```
model-check^ν_backward (I, T, G)
    S_C = "all states"; S_P = G;
    while S_C ≠ S_P do
      S_C = S_P;
      S_P = S_C ∩ AX(S_C);
    done;
    if I ⇒ S_C then return "only good states reachable";
    else return "found error trace to bad states";
```

Figure 4. Backward greatest fixpoint algorithm for safety properties

operations. In particular BDDs allow an efficient implementation of the image operation *Img*, which lies at the core of the breadth first search in symbolic model checking. It calculates the states reachable in one step via the transition relation T from the current set of states S_C, by implicitly conjoining the BDD representing S_C with the BDD representing T and projecting the result onto the next state variables Y (after eliminating the current state variables X and primary input variables W).

$$Img(Y) \equiv \exists X,W.\ S_C(X) \wedge T(X,Y,W) \tag{1}$$

In the context of circuits, we additionally assume that the transition relation is deterministic. As shown above, it may however depend on primary inputs, encoded by a vector W of Boolean variables, which also need to be quantified during image computation. In the terminology of program verification, *Img* calculates the strongest post condition of a given predicate.

A basic algorithm for symbolic model checking simple safety properties can then be formulated as in Figure 3. It represents sets of states symbolically, and searches in breadth first order from the initial states to the bad states. Let B be the set of bad states, in which p does not hold, and I the set of initial states.

This *forward model checking* algorithm, starts at the initial states and searches forward along the transition relation. In the literature one can also find *backward model checking* algorithms. They rely on a dual operation to the *Img* operation *PreImg*, or equivalently the CTL operator **EX**. It calculates the set of previous states S_P that may reach the given set of current states S_C in one step:

$$PreImg(X) \equiv \exists Y,W.\ S_C(Y) \wedge T(X,Y,W)$$

A *backward model checking* algorithm can be obtained from the forward algorithm by, in essence, exchanging B with I and *Img* with *PreImg*. In practice, forward traversal usually is much faster [58,57,54,17]. The reason may be, that unreachable states do not have to be visited and BDDs behave much better. However, not all temporal properties, for instance $\mathbf{EX}p \wedge \mathbf{EX}q$ or $\mathbf{AG}\ \mathbf{EX}p$, can be handled with *Img* computation only. In certain cases backward traversal is better. For instance, if the property p is an inductive invariant, as defined in Section 4.2. In this case the backward fixpoint computation terminates after one *PreImg* computation. A general strategy is to try backward and forward traversal in parallel.

Both symbolic model checking algorithms presented so far can be interpreted as calculating a least fixpoint [22]. Dual formulations exist for greatest fix points. For backward traversal, the CTL operator **AX** (also known as the weakest pre condition operator *wp*) replaces *PreImg*:

$$\mathbf{AX}(X) \equiv \forall Y,W.\ T(X,Y,W) \rightarrow S_C(Y)$$

It calculates the set of previous states S_P that lead to a state in the current set of states S_C, independent of the values at the primary inputs. A backward model checking algorithm for simple safety properties, based on greatest fix point calculation and on the **AX** operator, can be formulated as in Figure 4. Here, G denotes the set of *good states*, *i.e.*, the states in which p holds.

SAT technology can be used for implementing all parts of these algorithms. One option is to unroll the loop in model-check$^{\mu}_{\text{forward}}$ only a finite number of times, omitting the termination checks. This, in essence, is the main idea behind bounded model checking, the topic of the next section. We will come back to backward traversal calculating greatest fix points in Section 4.1.2.

3 Bounded Model Checking

Bounded Model Checking based on SAT methods was introduced by Biere *et al.* in [14,15,26] and is rapidly gaining popularity as a complementary technique to BDD-based symbolic model checking. Given a temporal logic property $\mathcal{P}$ to be verified on a finite transition system M, the essential idea is to search for counter-examples to $\mathcal{P}$ in the space of all executions of M whose length is bounded by some integer k.

The problem is formulated by constructing the following propositional formula:

$$\varphi^k = I \wedge \bigwedge_{i=0}^{k-1} T_i \wedge (\neg \mathcal{P}^k) \tag{2}$$

where I is the characteristic function for the set of initial states of M, T_i is the characteristic function of the transition

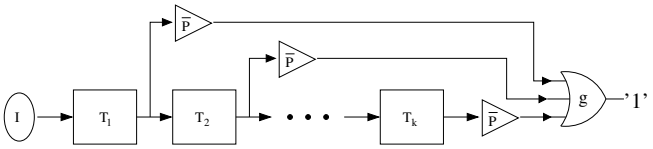

Figure 5. Bounded Model Checking

relation of M for time step i. Thus, the formula $I \wedge \bigwedge_{i=0}^{k-1} T_i$ precisely represents the set of all executions of M of length k or less, starting with a legal initial state. $\neg\mathcal{P}^k$ is a formula representing the condition that $\mathcal{P}$ is violated by a bounded execution of M of length k or less. Hence, φ^k is satisfiable if and only if there exists an execution of M of length k or less that violates property $\mathcal{P}$. φ^k is typically translated to CNF and solved by a conventional SAT solver.

The formula $\neg\mathcal{P}^k$ may be used to express both safety and liveness properties. Liveness properties of the form $\mathbf{AF}p$ are checked by having $\neg\mathcal{P}^k$ represent a loop within a bounded execution of length at most k, such that p is violated on each state in the loop. However, the more common application of BMC is for the purpose of checking safety properties of the form $\mathbf{AG}p$ (p is some propositional expression). In this case expression (2) reduces to:

$$\varphi^k = I \wedge \bigwedge_{i=0}^{k-1} T_i \wedge (\bigvee_{i=0}^{k} \neg P_i) \tag{3}$$

where P_i is the expression p in time step i. Thus, this formula can be satisfied if and only if for some $i \quad (i \leq k)$ there exists a reachable state in time step i in which p is violated. Figure 5 shows a circuit representation of this equation, where the block $\overline{P}$ denotes a combinational circuit block computing $\neg P_i$ as a function of the state variables of time step i.

A typical application of BMC consists of iteratively executing the above formulation for increasing values of k until either a property violation is discovered or some user specified limit on k or the computing resources (memory, runtime) is exceeded.

Recent research has improved upon both the technology and methodology of the basic BMC method described above in several ways. These improvements are discussed below.

3.1 Structural Pruning during CNF Generation

Many techniques use some kind of structural processing to generate a more compact CNF for the BMC problem, with the hope that the resulting SAT problem is easier for the SAT solver to solve.

The *bounded cone of influence (BCOI)* reduction [16] is an improvement on the classical *cone of influence (COI)* reduction used in traditional model checking. The intuition is that over a bounded time interval we need not consider every state variable in the classical COI at every time step. Specifically, in Figure 5, the BCOI reduction would extract the transitive fanin cone of the gate g and construct the BMC-CNF only from this sub-circuit. In our experience the BCOI reduction is cheap and easy to apply and can occasionally provide significant improvements over the simple COI reduction.

Ganai *et al.* [39] use binary AND-INVERTER graphs [64] to represent the transition relation of the system as well as the unrolled transition relation used for the BMC problem (Figure 5). The graph is compressed, as it is built, by using an efficient functional hashing scheme across two levels of logic, as well as term re-writing techniques. The CNF for the BMC problem is generated from this compressed representation. SAT results from earlier BMC runs are used to set appropriate $\overline{P}$ nodes (Figure 5) to 0 and then re-hash the circuit graph to obtain further compression. Such techniques work extremely well in practice especially if the logic level circuit used for the verification has been generated through a quick on-the-fly synthesis from an RTL description.

3.2 Decision Variable Ordering of the SAT solver

Variable ordering has long been recognized as a key determinant of the performance of SAT solvers. The earliest works on SAT-BMC were based on SAT solvers such as GRASP and SATO which used variable ordering heusristics such as the *DLIS* heuristic [70]. Strichman [95] proposed a static variable ordering scheme specifically targeted for BMC problems which improved upon the default DLIS ordering. The static order was generated from a BFS-like traversal of the unrolled circuit graph used for BMC.

However, recent results [88] show that the conflict-driven variable ordering heuristics used in modern SAT solvers (*e.g.*, the VSIDS heuristic in zchaff [77]) outperform any *fully* static BMC-specific variable ordering scheme, such as the one proposed in [95]. A slight tuning of these heuristics for the BMC problem [88] can further enhance the performance. On the other hand, BMC tools using circuit-based SAT solvers, *e.g.*, [41, 59, 64] essentially use some variant of the *J-frontier justification* heuristic popularly used in sequential ATPG tools.

While the above heuristics work fairly well for a SAT solver in a BMC setting, they do not specifically exploit any key aspects of the BMC problem to customize and target the SAT search for BMC. Since the SAT solver's runtime dominates the overall performance of the BMC tool, this topic could be an interesting avenue for future research.

3.3 Addition of Constraints to the SAT problem

The technique of learning *conflict clauses* during search has dramatically enhanced the efficacy of modern SAT solvers. Motivated by this, several other specialized static and dynamic learning techniques have been developed for the BMC problem. The learned constraints can be added as CNF clauses to the SAT problem being solved, with the hope of speeding up the solution process.

The technique of *constraints sharing* [96] proposed by Strichman is based on the observation that since BMC is an iterative process whereby the problem is repeatedly solved

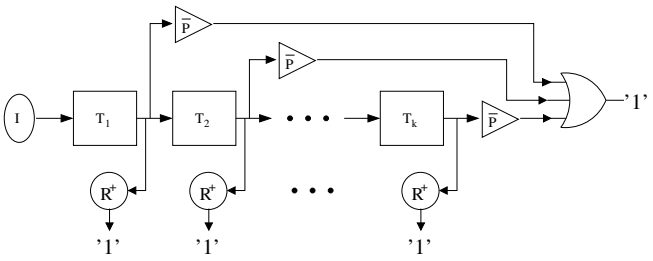

Figure 6. Improving BMC Using Reachability Over-approximation

for increasing values of the bound k, conflict clauses learned by the SAT solver in one run can potentially be used for subsequent runs instead of having to re-learn them. Specifically, any conflict clause derived *exclusively* from the sub-formula $\Phi_k = I \wedge \bigwedge_{i=0}^{k-1} T_i$ can be re-used (*i.e.*, added a-priori to the CNF) in future BMC runs with higher values of k. This technique is a specific instance of *incremental satifiability* techniques, with applications in BMC [92] and other general classes of SAT problems [61, 102]. Generally, this technique has been found to offer speed-ups of upto $2\times$ or more with negligible overhead.

A related technique called *constraints replication* [95] first identifies conflict clauses c, derived from the sub-formula $\bigwedge_{i=0}^{k-1} T_i$ alone, and creates new clauses by replacing literals of c by their time-frame shifted versions, which are then added a priori to CNFs of subsequent BMC runs. This technique is not very effective in practice, mainly due to the large overhead caused by addition of too many replicated clauses.

Recent work by Gupta *et al.* [49] proposed learning conflict clauses from BDDs, and adding them dynamically to the problem during the SAT search. The learned clauses correspond to paths to the $'0'$ terminal in a BDD representation, denoting unsatisfiable assignments on the path variables. These BDDs are created on-the-fly for heuristically selected small regions (*i.e.*, sub-circuits) in the unrolled design for BMC. They proposed several heuristics to keep the overhead low, while increasing the usefulness of the added clauses, and demonstrated significant speedups in BMC performance.

Another technique that draws upon BDD technology is the work of Cabodi *et al.* [23]. The basic idea is to use BDD-based approximate reachability analysis to *quickly* compute a *succinct* and coarse over-approximation, R^+ of the reachable state-space of a design. The BDD representing the characteristic function of R^+ is then asserted as constraints on the transition boundary between each successive pair of time-frames $i, i+1$, as shown in Figure 6. The BDDs are converted to CNF constraints which are conjoined with the BMC formulation of Equation 2. This technique does indeed have an overhead and is therefore useful primarily for larger, more difficult BMC problems. In such cases speed-ups of upto an order of magnitude have been observed.

3.4 Methodology Improvements to BMC

Although, BMC is by its intent an incomplete, bug-finding method rather than a complete verification method, a given property can be certified to be true if no counter-examples are found through BMC, upto the *sequential depth* of the circuit[15]. The sequential depth of a circuit is length of the *longest* of the shortest-paths from the initial state(s) to other reachable states of the system.

There have been a few attempts at computing or estimating the sequential depth of a circuit, to use as a target depth for BMC. Yen*et al.* [104] proposed a heuristic method based on a sampling of the state-space through random simulation. However, since the method can report an under-approximation or an over-approximation of the true sequential depth it does not provide a viable solution. Mneimneh and Sakallah [76] formulate the problem as a logical inference problem on Quantified Boolean formulas (QBF) (see also Section 6) and present a SAT-based procedure for solving the generated QBF. Unfortunately, this technique although precise, does not offer a scalable solution. Baumgartner *et al.* [10] present a structural approach based on traversing the circuit netlist to identify components with known sequential depth and using these to compute the overall sequential depth. Despite the above attempts, the problem of efficiently computing or tightly over-approximating the sequential depth of industrial size, arbitrary sequential circuits largely remains an open problem.

It is well known that different propositional encodings of the same problem can result in dramatically different runtimes on a given SAT solver. The approach of *Binary Time-frame Expansion* proposed by Fallah [37] provides a different propositional encoding of the check for violation of the property in various time-frames of an unrolled circuit. The proposed encoding has been demonstrated to improve the SAT solver runtimes over the traditional formulation of Eq. 2 provided the BMC instance is sufficiently deep (typically $k \geq 100$).

3.5 Industrial Application of BMC

Several successful attempts at applying SAT-based BMC technology to industrial problems have been reported over the past few years. The original proponents of BMC reported a case study [16] where they applied BMC based on the SAT solvers SATO [105] and GRASP [71] to verify safety properties on 5 control units from the *PowerPC*^{TM} microprocessor. BMC was found to significantly outperform the BDD-based CMU SMV model checker for several of the benchmarks. Bjesse *et al.* [19] reported a significant increase in bug-finding speed and efficiency by their application of SAT-BMC (based on GRASP and CAPTAIN PROVE [90] SAT solvers), to check safety properties in the memory sub-system of the Alpha microprocessor.

A recent comprehensive analysis with respect to the performance and capacity of BMC is presented in [31]. The authors compare Intel's BDD-based model checker, *Forecast*

(adapted for BMC) with a SAT-based BMC tool, *Thunder* on several benchmarks taken from Intel's Pentium 4 processor. Their evaluation yields an interesting tie between the performance of *untuned* Thunder and *tuned* Forecast. They conclude that the real productivity gains from SAT-based BMC are obtained by obviating the need for user ingenuity and tuning effort that would be needed to obtain a *comparable* performance from a BDD-based BMC. They also report success in using SAT-based BMC on large benchmarks, that are well beyond the capacity of BDD-based tools.

A more recent study [4] compares the performance of BDD-based, SAT-based and explicit state BMC on a wide variety of industrial property checking benchmarks including both safety and liveness properties on hardware and software designs. Interestingly, they conclude that SAT-BMC is most effective at finding bugs at shallow depths (< 50) while BDD-based methods should be the method of choice for finding deep counter-examples. They also find that explicit-state BMC based on random simulation can give comparable performance to SAT-BMC in finding shallow, easier bugs for safety properties.

The general understanding and consensus in the community is that SAT-BMC tools require minimal tuning effort and work particularly well on large designs where bugs need to be searched at shallow to medium depths. In other instances it *may be possible* to extract comparable or better performance from BDD-based model checkers or other algorithms.

4 SAT-based Unbounded Model Checking

In this section we describe verification efforts that have used SAT solvers for unbounded symbolic reachability analysis, *i.e.*, methods that can prove the correctness of a property on a design as well as find counter-examples for failing properties. The method may or may not be complete. The surveyed methods fall into three categories. The first set of techniques have their roots in BDD-based symbolic state space search where the use of BDDs has been partially or completely replaced with SAT solvers. The second category comprises methods based on inductive reasoning. Inductive techniques are sound but usually incomplete in that they may not be able to prove every correct property. The third category of methods are abstraction-refinement frameworks, where SAT-based BMC is used primarily for abstraction or refinement, and is supplemented by other techniques for obtaining proofs on smaller abstract models. These frameworks also provide completeness, and offer better scalability due to effective use of abstraction. In principle, completeness can also be achieved by making the transition from SAT to QBF as is explained in Section 6.

4.1 SAT-based State Space Search

Due to the success of SAT solvers in bounded model checking, there has been growing interest in their use for *unbounded* model checking. Here, the crucial non-trivial operation is quantifier elimination, which converts a QBF to a propositional Boolean formula. This is shown below for the image operation, which forms the computational core of symbolic methods for forward model checking, as explained in Section 2.3.

$$S_N(Y) = \exists X, W, Z.\ S_C(X) \wedge T(X, Y, W, Z) \tag{4}$$

In this equation, the variable sets X, Y, W, Z, denote the present state, next state, input, and internal (needed for a CNF representation) variables, respectively; and S_N, S_C, and T denote the next states, the current states, and the transition relation, respectively.

4.1.1 Combination of SAT with Decision Diagrams

Abdulla *et al.* [1] formulate the checks for property satisfaction and fixpoints as SAT problems, to be solved by standard SAT solvers. The SAT problems comprise combinations of formulas S_*, representing sets of states. These are obtained by using rewriting rules for eliminating the existential quantifier in the image/pre-image operations (shown in Equation 4). The most effective rule is an *inlining rule*, which substitutes an expression for a variable to be quantified; while the most expensive is rewriting the existential quantification as a disjunction, which can result in a size blowup. They use *Reduced Boolean Circuits (RBCs)* to represent the Boolean formulas, which can be exponentially more succinct than BDDs, but are semi-canonical. A similar effort was made by Williams *et al.* [103] to use SAT solvers for CTL model checking. They too used a substitution rule very effectively for elimination of the existential quantifier. They used *Boolean Expression Diagrams (BEDs)* [5], which are closely related to RBCs, for representation of the Boolean formulas,. In addition to using standard SAT solvers to check satisfiability of BEDs, they also used the conversion of BEDs to standard BDDs. Since this conversion can blow up in practice, they used various heuristics to reduce the size of BEDs.

A different approach was taken by Gupta *et al.* [52], which integrates BDD-based techniques tightly into the SAT decision procedure. They represent the transition relation T in CNF, and the set of reachable states S_* as BDDs. For image computation, quantifier elimination is performed by using SAT techniques to enumerate all solutions to the CNF formula, and by projecting each solution on the set of image variables (Y). The search for solutions is also constrained by the BDD for S_P, using a technique called *BDD Bounding*, whereby any partial solution in SAT which is inconsistent with the BDD is regarded as a conflict. This technique is also used effectively to avoid repeating image set solutions by bounding against the current S_N. They also generate BDD-based subproblems on-the-fly, under a partially explored path in SAT. Though their procedure can be used to perform cube enumeration in SAT alone, the use of BDD subproblems is highly beneficial in handling large designs. This image computation procedure was enhanced in [51] by adding circuit structure information to the CNF formula, in order to dynamically detect and remove redundant clauses. Partition-based

SAT decision heuristics [53] were used to further improve its performance.

4.1.2 Purely SAT-based Techniques

An approach using purely SAT-based techniques was proposed by McMillan [73], for performing backward symbolic model checking (see Figure 4 in Section 2.3). It is based on computing the CNF formula equivalent to $\mathbf{AX}p$, where p is an arbitrary Boolean formula, by enumerating all satisfying assignments using a SAT solver. Variables are universally quantified by simply dropping the associated literals from the resulting CNF. Note that this forms the dual of projection for existentially quantified variables in a Disjunctive Normal Form using cubes, as used by other researchers, *e.g.*, [52,80]. Each satisfying cube is used to derive a *blocking clause* which contributes to the set of solutions, and is also added to the current database of clauses in order to avoid repetition of the solutions. The procedure for deriving a blocking clause exploits circuit structure information to re-arrange the implication graph (described in Section 2.2) when a solution (*i.e.*, a satisfying assignment) is found by the SAT solver. This re-arrangement can be viewed as a *cube enlargement* technique, which allows a larger solution cube to be captured in each enumeration by the SAT solver. The overall approach works well for designs where the sets of states can be represented compactly in CNF, and where cube enumeration with blocking clauses does not blow up.

Another model checking approach based on use of SAT techniques and *Craig interpolants* has been proposed in [74]. Given an unsatisfiable Boolean problem, and a proof of unsatisfiability derived by a SAT solver, a Craig interpolant can be efficiently computed to characterize the interface between two partitions of the Boolean problem. In particular, when no counterexample exists for depth k in BMC, *i.e.*, the SAT problem for depth k is found to be unsatisfiable, a Craig interpolant is used to obtain an over-approximation of the set of states reachable from the initial state in 1 step (or any fixed number of steps). This provides an approximate image operator, which can be used iteratively to compute an over-approximation of the set of reachable states, *i.e.*, till a fixpoint is obtained. If at any point, the over-approximate set is found to violate the given property, then the depth k is increased for BMC, till either a true counterexample is found, or the over-approximation converges without violating the property. The main advantage of the interpolant-based method is that it does not require an enumeration of satisfying assignments by the SAT solver. Indeed, the proof of unsatisfiability is used to efficiently compute the interpolant, which serves directly as the over-approximate state set. In practice too, this method has been shown to work better than other BDD-based and SAT-based complete methods. However, if the focus is only on finding bugs, *e.g.*, *falsification*, then, in the current version, it can not be faster than BMC alone.

More recently, a SAT-based quantification technique using circuit cofactoring has been proposed by Ganai *et al.* [40]. They too use a SAT solver to enumerate solutions, but they use circuit cofactoring after each enumeration to capture a larger set of new state cubes per enumeration, in comparison to cube-wise enumeration techniques. Note that in general a cofactor can capture not just a single cube, but several cubes. This is greatly beneficial in reducing the total number of solutions enumerated by SAT, sometimes by several orders of magnitude, in comparison to approaches based on blocking clauses (described above). They also use an efficient circuit graph representation for the solution states [64], which is more robust than CNF-based or BDD-based representations, and use a hybrid SAT solver [41] to directly work on these representations. Ganai et al.'s quantification technique can be used to compute exact image/pre-image state sets, unlike the interpolant-based technique (described above) which computes approximate state sets. It has been used in SAT-based unbounded symbolic model checking to handle many difficult industry examples, which could not be handled by either BDDs or blocking-clause based SAT approaches.

4.2 SAT-based Inductive Reasoning

Inductive reasoning can be a cheap and efficient means of verifying properties, rather than simply finding counter-examples as in BMC. Inductive reasoning has previously been used, with some success, for various verification problems, including property checking using technologies such as BDDs. The inductive proof for verifying a property $\mathcal{P} = \mathbf{AG}p$ can be derived using a SAT solver by checking the formulas ϕ_{base} (the base case) and ϕ_{induc} (the induction step) for unsatisfiability.

$$\begin{aligned} \phi_{base} &= I \wedge \neg P_0 \\ \phi_{induc} &= P_k \wedge T(k,k+1) \wedge (\neg P_{k+1}) \end{aligned} \tag{5}$$

If ϕ_{induc} is unsatisfiable the property $\mathcal{P}$ is called an *inductive invariant*. Both formulas, if unsatisfiable, provide a sufficient (but not necessary) condition for verifying $\mathcal{P}$. However, the above form of induction, known as *simple induction*, is not powerful enough to verify many properties.

Two recent works [18,89] have proposed the use of more powerful forms of induction known as *induction with depth* and *unique states induction* to verify safety properties. For induction with depth n the formulas of Equation 5 become:

$$\begin{aligned} \phi^n_{base} &= I \wedge \left(\bigwedge_{i=0}^{n-1} T(i,i+1) \right) \wedge \bigvee_{i=0}^{n} \neg P_i \\ \phi^n_{induc} &= \left(\bigwedge_{j=k}^{k+n} P_j \right) \wedge \left(\bigwedge_{i=k}^{k+n} T(i,i+1) \right) \wedge \neg P_{k+n+1} \end{aligned} \tag{6}$$

Essentially, induction with depth corresponds to strengthening the induction hypothesis, by imposing the original induction hypothesis (P_k in ϕ_{induc}, Equation 5) on n consecutive time-frames. This can be further strengthened by requiring that the states appearing on each time-frame be unique

(*unique states induction*). This restriction results in a complete method for simple safety properties. However, the induction length may be as long as the recurrence diameter [15], which in most cases is much longer than the sequential depth. Further, the number of constraints needed to enforce the state uniqueness is quadratic in the depth of unrolling, *i.e.*, the induction depth, resulting in very large CNFs. In recent work [35], Eén *et al.* partly address this issue by proposing an iterative method for induction. The induction hypothesis starts off without any uniqueness constraints, which are gradually added in successive iterations till the induction proof goes through. The efficiency of the method is further improved by using an *incremental SAT* mechanism that allows sharing of conflict clauses (recorded by the SAT solver) between successive iterations of induction.

Another variant of this line of research is the work by Gupta *et al.* [48], which is similar to the work by Cabodi *et al.* [23], discussed in Section 3.3. As in [23], BDD-based techniques are used to *efficiently* compute a *succinct* over-approximation R^+ of the reachable states of a design. This is used to strengthen the induction hypothesis by imposing R^+ as an additional reachability invariant. In particular, it constrains the state values that are allowed to appear at the starting state of the induction step (or at the interfaces between each successive pair of time-frames). Note that in contrast to [23], the constraints here are not redundant, but are added to strengthen the induction hypothesis, which might be too weak with the property alone. This frequently allows induction proofs to go through successfully. A related line of research is based on generating an inductive invariant to be used as over-approximation for the reachable states in the context of sequential equivalence checking [18,99,94].

One of the original papers on SAT-BMC [16] had proposed the use of simple induction as a cheap and simple first pass to apply to all property checking instances before resorting to more comprehensive verification/falsification methods. The above powerful variants of induction undoubtedly enlarge the range of properties verifiable through inductive reasoning. At the same time they can produce very large SAT formulas which are very resource intensive to solve. Hence the real utility of these methods would only be brought out by a good verification methodology that uses them with the right trade-off between verification power and efficiency, and in the right balance with BDD-based verification techniques. Recent work by Li *et al.* [67] points in this direction as well. In this work the authors use SAT-based unique-states induction with depth as the model checking method in an abstraction refinement framework (discussed in the next section). They observe that the efficacy of SAT-based induction is considerably enhanced when used within such a framework. Further, even within this framework the SAT-based induction exhibits complementary strengths compared to a traditional BDD-based model checker, underscoring the need for a combined proof technique.

4.3 SAT-based Abstraction-Refinement Frameworks

In order to handle large designs, there has been a great deal of interest in the use of abstraction and refinement techniques for verification. Most efforts are refinement-based approaches, where starting from a small abstract model of the concrete design, counterexamples found on these models are used to refine them iteratively until either a conclusive result is obtained by conservative model checking, or the resources are exhausted [79]. One of the first attempts to use SAT solvers for counterexample guided abstraction refinement (CEGAR) was described by Clarke *et al.* [29]. In their approach, the SAT solver is used to check whether a counterexample trace found during model checking of the abstract model is spurious or not, by effectively checking its satisfiability on the concrete design. If it is spurious, ILP (integer linear programming) and machine learning techniques are used to perform the refinement. In a subsequent effort [25], they used SAT-based techniques for performing this refinement as well. In particular, they proposed heuristics using the SAT decision scores to pick refinement candidates among hidden (abstracted away) latches. A more interesting technique used ideas similar to a SAT solver's proof of unsatisfiability, in order to identify latches that are *sufficient* to exclude the spurious counterexample.

Another recent method for counterexample guided abstraction refinement has been proposed by Wang *et al.* [101]. They use BDDs to represent multiple abstract counterexamples, which are checked for satisfiability on the concrete design using a SAT solver interfaced with BDD constraints [48]. Rather than refining each counterexample individually, they propose a game-theoretic refinement procedure, that attempts to exclude multiple counterexamples simultaneously. In practice, their method performs better than other methods based on refining a single counterexample at a time.

One reason for the popularity of counterexample guided abstraction refinement approaches has been a lack of techniques that could extract relevant information from a relatively large concrete design. This is changing now with the use of proof analysis techniques for SAT solvers. These techniques can be easily used to identify a set of clauses from the original problem, called the *unsatisfiable core* [109,75], such that the clauses are sufficient for implying unsatisfiability. These unsatisfiable cores form the basis of two recent independent efforts on abstraction methods using SAT-based BMC [75,50]. In both methods, an abstract model is obtained from the unsatisfiable core, identified from an unsatisfiable BMC instance at depth k. This abstract model has the useful property that it does not have any counterexamples of depth less than or equal to k. The basis for abstraction is the intuition that after k is large enough, the corresponding abstract model may exclude counterexamples of all lengths. The usefulness of the abstraction stems from the empirical evidence that for typical verification applications, the unsatisfiable cores and the corresponding abstract models are much smaller than the concrete designs. There are minor differences in the abstraction methods used by these two ap-