Chapter 10
SAT-Based Model Checking

Armin Biere and Daniel Kröning

Abstract Modern satisfiability (SAT) solvers have become the enabling technology of many model checkers. In this chapter, we will focus on those techniques most relevant to industrial practice. In *bounded model checking* (BMC), a transition system and a property are jointly unwound for a given number k of steps to obtain a formula that is satisfiable if there is a counterexample for the property up to length k. The formula is then passed to an efficient SAT solver. The strength of BMC is *refutation*: BMC has been used to discover subtle flaws in digital systems. We cover the application of BMC to both hardware and software systems, and to hardware/software co-verification. We also discuss means to make BMC complete, including k-induction, Craig interpolation, abstraction refinement techniques, and inductive techniques with iterative strengthening.

10.1 Introduction

Modern satisfiability (SAT) solvers have become the core technology of many model checkers, greatly improving capacity when compared to BDD-based model checkers. In this chapter, we will focus on those SAT-based model-checking techniques that are most relevant to industrial practice. In SAT-based *bounded model checking* (BMC) [26], a symbolic representation of a transition system and a property are jointly unwound for a given number of steps k to obtain a formula that is satisfiable if there is a counterexample for the property up to length k. The formula is then passed to an efficient SAT solver.

The idea of using propositional SAT to encode and solve path constraints for transition systems was discussed before in the AI planning community. Originally Kautz and Selman [103] observed that direct encodings of planning problems into a propositional SAT problem outperformed the best planning algorithms by orders

A. Biere (✉)
Johannes Kepler University, Linz, Austria
e-mail: biere@jku.at

D. Kröning
University of Oxford, Oxford, UK

E.M. Clarke et al. (eds.), *Handbook of Model Checking*,
DOI 10.1007/978-3-319-10575-8_10

of magnitude. A more recent experimental survey of using SAT for planning can be found in [145].

The rationale for using BMC is based on the observation that SAT solvers are often able to solve much larger formulas than classical techniques based on binary decision diagrams (BDDs) [40] (see also Chap. 8 of this Handbook). It is now industrial practice to simply run BMC for a certain amount of time or up to a certain bound k, fixed for instance in the verification plan.

On the other hand, BDD-based techniques allow efficient implementations of quantifier elimination, which is crucial for termination checks in symbolic fix-point algorithms. The detection of the fix-point is essential to *prove* properties in general, but not necessary when aiming at *refutation*.

In this chapter, we cover the application of BMC to both hardware and software systems, and hardware/software co-verification. In its simplest form, BMC is incomplete, as bugs that are only exposed with more than k transitions are missed. These BMC-based techniques therefore either relinquish completeness, or have to rely on alternative ways to assert that a property holds in general for all bounds. The chapter therefore covers a range of SAT-based techniques that are able to establish a proof of correctness for the property for an unbounded depth.

This material has been covered more extensively in other tutorial-style publications and surveys before [69, 81, 141, 155], including two chapters [24, 110] in the *Handbook of Satisfiability* [28], by the same authors as this chapter. Thus, besides explaining some of the very basic ideas, the rather restricted amount of space available here is used to give pointers to existing important work on SAT-based model checking and elaborating on more recent publications.

The outline of the chapter is as follows. We begin with a description of how to perform BMC on an abstract description of the system, given in the form of a *transition system*. We then provide details on how to obtain formal models from industrial system description languages such as Verilog and ANSI-C, and how to encode these models and systems properties into a propositional formula. In particular, we show how model-checking problems for software and hardware can be encoded into satisfiability checking (SAT). The chapter concludes with a discussion of means to make BMC complete, including k-induction, Craig interpolation, and inductive techniques with iterative strengthening.

10.2 Bounded Model Checking on Kripke Structures

10.2.1 Kripke Structures

The behaviors of a program or circuit can be formally captured using a *Kripke Structure*, formally defined as follows.

Definition 1 (Kripke structure) A *Kripke Structure* is a (finite) set of states S, a set of initial states $I \subseteq S$, and a transition relation $T \subseteq S \times S$.

A *path* in a Kripke structure is a (possibly infinite) sequence of states $s_0, s_1, s_2, \ldots$ such that

- s_0 is an initial state, i.e., $s_0 \in I$, and
- there is a transition between any s_i and s_{i+1}, i.e., $(s_i, s_{i+1}) \in T$.

The states are typically valuations of a set of *state variables*, corresponding to latches and registers in circuits and program variables in software. In the case of a finite set of states we can always re-encode the Kripke structure to use propositional variables only. As a result, we obtain purely propositional predicates I and T. We use the set notation and the state predicates and relations interchangeably, i.e., the propositional formula $I(s_i)$ evaluates to true iff $s_i \in I$. Similarly, $T(s_i, s_{i+1})$ evaluates to true iff $(s_i, s_{i+1}) \in T$.

The key idea of bounded model checking is to construct a formula that is satisfiable if there exists a path that violates a given property. We now consider specific kinds of properties, and will distinguish *safety* and *liveness* properties.

10.2.2 Safety Properties

Properties are typically defined using a suitable *temporal logic*. We refer to Chap. 2 of this Handbook [140] for an introduction to temporal logics. We restrict the discussion in this chapter to properties given in *Linear Temporal Logic* (LTL). One benefit of this restriction is that counterexamples to LTL properties can always be given in the form of a path, as defined above. A full survey on ways to encode LTL in a BMC context together with an experimental comparison with BDD-based techniques is provided by Biere et al. [27]. Note that encodings differ in terms of compactness, ease of implementation, and of course SAT-solving efficiency.

We begin with LTL properties of the form $\mathbf{G}p$, where p is a state predicate. This property establishes that p is a global invariant of the system. A counterexample for a property of this kind can be given as a finite path that ends with a state s that satisfies $\neg p$. This gives rise to a straightforward condition for the existence of a counterexample path of length k:

$$\exists s_0, \ldots, s_k. \quad I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k) \tag{1}$$

The formula above contains three conjuncts. The first conjunct, $I(s_0)$, ensures that the state s_0 is one of the initial states. The second conjunct encodes the requirement that there is a transition from s_i to s_{i+1} for each $i \in \{0, \ldots, k-1\}$. This amounts to creating k replicas of the transition relation T. Finally, the conjunct $\neg p(s_k)$ asserts that the state s_k satisfies $\neg p$.

Note that the formula obtained in this way has only one level of (existential) quantification and thus corresponds to a propositional satisfiability problem. Most modern SAT solvers such as ZChaff [136] or MiniSAT [73] expect to receive the propositional formula in conjunctive normal form (CNF).[1] The transformation of the quantifier-free propositional formula into CNF is performed using the *Tseitin transformation* [151]. This transformation is linear-time, and results in an equisatisfiable formula in CNF. Numerous papers on more compact or more efficient variants of this step have been published, e.g., [45, 72, 153]. Further details on CNF encodings can also be found in the *Handbook of Satisfiability* [142]. See also the discussion on the relation between CNF-level preprocessing and encoding in [97].

10.2.3 Liveness Properties

We will consider further categories of system properties. The simplest type of liveness properties are *eventualities*, e.g., whether a particular state property is guaranteed to eventually hold. These properties are written as $\mathbf{F}p$ in LTL. The encoding of LTL formulas of this form is very similar to the encoding of $\mathbf{G}p$. We observe that counterexamples to properties of this form can always be given as a finite (possibly empty) prefix (called the *stem*) followed by a finite loop. All states on the path satisfy $\neg p$. This pattern can be encoded as follows:

$$\exists s_0, \ldots, s_k. \quad I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k-1} \neg p(s_i) \wedge \bigvee_{i=0}^{k-1} s_k = s_i \tag{2}$$

As described above, the formula can be converted into propositional logic, and can then be passed to a propositional SAT solver.

The translation of general LTL formulas is more complex. Techniques for performing this translation can be categorized as *syntactic* or *semantic* [58]. Syntactic translations follow the syntactic structure of the LTL property; instances include [26, 90, 91, 128, 139].

As an alternative *semantic translations* can be used, which are based on automata: the formula is transformed into a suitable kind of automaton that accepts counterexample paths. An instance is the translation of the LTL property φ into a Büchi automaton $M_{\neg\varphi}$ that accepts paths that satisfy $\neg\varphi$ [74, 152]. Counterexamples to φ then have the form of a path through the product of the Kripke structure and $M_{\neg\varphi}$ that contains infinitely many accepting states. A counterexample in a finite-state product is thus a loop that does not contain an accepting state. This condition can be encoded using a formula similar to Eq. (2). One key advantage of the automata-based encoding is that numerous minimization techniques can be applied to the automaton prior to building the BMC formula.

Semantic translations allow the use of sophisticated automata optimization techniques, but the space requirements might explode for larger formulas, due to explicit representation of potentially exponentially many states in the automata.

[1]There are now also non-clausal propositional SAT solvers, e.g., [95].

10.2.3.1 Liveness to Safety Translation

Besides these syntactic and semantic translations, a third approach to handle liveness is to encode liveness into safety (L2S) and then use model-checking algorithms for checking safety [25, 146]. This is particularly useful for techniques such as interpolation which only work for safety properties at this point. The L2S encoding actually increases the size of the model by a factor of two. Thus, it might be prohibitively expensive for BDD-based techniques, which are very sensitive to model size. However, even for BDD-based model checking there are cases where L2S is exponentially more efficient.

10.2.3.2 k-Liveness

More recently, a new approach for checking liveness was presented in [49] and independently discovered in [80] (see also [124]). In [49], the authors called it "k-liveness". Their implementation proved to be quite effective in the liveness track of the Hardware Model Checking Competition 2012 (HWMCC 2012). In this approach, liveness properties are assumed to be encoded as $\mathbf{FG}p$ properties. Then the approach tries to prove that a witness trace for such a property does not exist. In case of a finite-state system, a witness trace to $\mathbf{FG}p$ can be assumed to be an infinite path which ends in a loop, where the loop contains a state in which p holds. If $\mathbf{FG}p$ cannot be satisfied, then the prefix of any path satisfies p only an arbitrary (but finite) number of times.

The basic idea of the approach is to count the number of occurrences of p and then check that the count is smaller than a fixed bound k. Note that this turns the liveness-checking problem into a simple safety-checking problem. If p can only be satisfied at most k times, then $\mathbf{FG}p$ cannot be satisfied on any initialized path. If the safety check fails and a path is found on which p can be satisfied more than k times, the bound k is increased to say $k + 1$ and a new safety-checking problem for bound $k + 1$ is generated. If the property $\mathbf{FG}p$ does not hold for a finite-state system, then this process has to terminate after k reaches the number of states of the system. In practice the process terminates much earlier, in particular if combined with a method for extracting additional constraints [49]. In order to find violations of liveness properties, i.e., witness traces for formulas like $\mathbf{FG}p$, the approach has to rely on other techniques, such as those discussed above.

10.3 Bounded Model Checking for Hardware Designs

We will now cover techniques to translate system descriptions given in industrial system description languages into BMC instances. We begin with verification of designs given in hardware description languages (HDLs), which was one of the earliest applications of SAT-based BMC (see also Chap. 24 of this Handbook [77]).

10.3.1 Hardware Description Languages (HDLs)

In industrial practice, hardware designs are described by means of modeling languages. These include languages to describe schematics and net-lists at the lowest level. Higher levels of abstraction can be achieved by hardware description languages (HDLs) such as *VHDL* or *Verilog*.

The challenges in encoding models given in hardware description languages into SAT are mostly shared by all model-checking techniques for hardware; they affect BDD-based and SAT-based methods alike. Most HDLs have both *simulation semantics* and *synthesis semantics*. Designers rely heavily on simulation and build models with simulation semantics in mind. Simulation semantics are typically based on an *event queue*, resembling the data structures maintained by event-driven simulators. On the other hand, the synthesis semantics is closer to the actual hardware produced, and may uncover design flaws that go unnoticed during simulation.

10.3.2 BMC on Net-Lists

We will briefly elaborate on performing BMC using synthesis semantics. In this context, the BMC implementation will initially perform several stages of behavioral synthesis up to the point that a *net-list* is produced. A net-list is a collection of primitive elements. A typical way to represent net-lists is to use an *and-inverter graph* (AIG) [123], i.e., the net-list consists of “and” gates, inverters and memory elements referred to as registers.

Definition 2 A net-list N is a directed graph (V_N, E_N, τ_N) where V_N is a finite set of vertices, $E_N \subseteq V_N \times V_N$ is the set of directed edges and $\tau_N : V_N \to \{\text{AND}, \text{INV}, \text{REG}, \text{INPUT}\}$ maps a node to its type, where AND is an “and” gate, INV is an inverter, REG is a register, and INPUT is a primary input. The in-degree of a vertex of type AND is at least two, of type INV and REG is exactly one and of type INPUT is zero. Any cycle in N must contain at least one REG node.

As an example, consider the 3-bit counter whose Verilog module is shown in Fig. 1 (taken from [47]). The corresponding net-list is shown in Fig. 2. A node drawn as a box represents a REG. A circle-shaped node is an AND gate. An incoming edge of a node marked with a circle indicates negation.

A *state* of a net-list is a mapping of its registers to the Boolean values $\mathbb{B} = \{0, 1\}$. A net-list N with r registers gives rise to a Kripke structure $M = (S_N, I_N, T_N)$ where $S_N = \mathbb{B}^r$ is the set of states and T_N is the transition relation specifying what pairs of states are connected by transitions. The set I_N of *initial states* is determined by the values of the registers immediately after reset. In the above example, $I_N = \neg count[0] \wedge \neg count[1] \wedge \neg count[2]$. The state-transition diagram for the circuit is shown in Fig. 3. Note that S_N for the 3-bit counter consists of $2^3 = 8$ states.

Fig. 1 Verilog module of a counter

```
module counter(clk, count);
  input clk;
  output [2:0] count;
  reg [2:0] count;

  wire cin =
    ~count[0] & ~count[1] & ~count[2];

  initial count = 3'b0;

  always @ (posedge clk) begin
       count[0] <= cin;
       count[1] <= count[0];
       count[2] <= count[1];
  end
endmodule
```

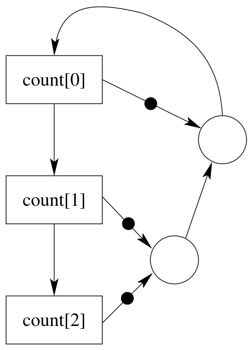

Fig. 2 Net-list for Fig. 1

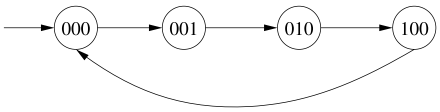

Fig. 3 State-transition diagram of the counter in Fig. 1

Unreachable states are not shown in Fig. 3. An algorithm for obtaining a transition relation for a net-list is given in [57].

The required property of a circuit can be given as part of the design description in languages such as *PSL* [76] or as a *System Verilog Assertion* [154]. A discussion of hardware specification languages can also be found in Chap. 24.

10.4 Bounded Model Checking for Software

We focus on BMC-like approaches to software verification; for a broader perspective on automated techniques for formal software verification, we refer the reader to a survey [69].

10.4.1 Monolithic Encodings

The most straightforward manner to implement BMC for software is to encode the transition relation of the program into a circuit representation, and then to perform BMC as described in Sect. 10.2.

1. The first step is to add a *program counter* (PC) to the set of state variables of the model. The program counter determines the instruction that is to be executed next.
2. Each instruction is turned separately into a transition relation. One way to obtain such a formula is to convert the arithmetic operators in the program into their circuit (net-list) equivalents. Arrays and pointers are treated as memories, using a large case split over the possible values of the address or a first-order array theory.

We will illustrate the second step by means of an example. Suppose that our program has three state variables named $\mathtt{x}$, $\mathtt{y}$, and $\mathtt{z}$, and suppose we wish to encode the following instruction, given in C syntax:

$$\mathtt{x} = \mathtt{y} + \mathtt{1};$$

Note that the equal sign = in the C program fragment above indicates an assignment, and not an equality relation. Following the usual convention, we will use $\mathtt{x}'$, $\mathtt{y}'$ and $\mathtt{z}'$ to denote the next-state values of the state variables. The transition relation for the statement above is then

$$\mathtt{x}' = \mathtt{y}' + 1 \quad \wedge \quad \mathtt{y}' = \mathtt{y} \quad \wedge \quad \mathtt{z}' = \mathtt{z}\,.$$

Note that in the above formula, the symbol $=$ denotes mathematical equality, and not assignment. Also note the second and third conjuncts: these constraints state the fact that the value of the program variables $\mathtt{y}$ and $\mathtt{z}$ is not changed by the instruction.

An unwinding using the "monolithic encoding" as described above with k steps permits all program paths that traverse k (or fewer) instructions to be explored. The size of this basic unwinding is k times the size of the program. For large programs, this is prohibitive, and thus, several optimizations have been proposed. These optimizations focus on reducing the size of the encoding by eliminating combinations of control-flow locations that do not correspond to paths through the program.

As an instance, in the case of sequential programs it is beneficial to merge all instructions within one basic block into a single big-step instruction. Each basic block of the program is converted into a formula by transforming it into *static single assignment* (SSA) form [3]. This reduces the number of control-flow locations. The model checker F-SOFT is reported to use an optimized monolithic encoding [94].

10.4.2 Path-Based Encodings

Instead of unwinding the entire transition relation, path-based software analyzers perform forward *symbolic execution* [105] or in general *symbolic simulation* alongside specific program paths up to a given depth. The resulting formula is then passed to the SAT solver [62]. This basic approach has a broad range of applications; e.g., it can be used to check arbitrary safety properties or to generate test vectors to achieve particular coverage goals. See [43] for a historical perspective on symbolic execution.

There are numerous approaches to prune the set of paths that are to be explored, or to heuristically choose a path that most likely leads to a particular goal [14]. Once a satisfying assignment is obtained, a counterexample can be extracted. There is also work on obtaining particularly desirable counterexamples, and attempts to use information from the BMC instance to explain the root cause of the error [85, 87].

In the most basic form, tools using path-based encodings explore precisely one path at a time. An advantage of this approach is that the formulas generated this way are often very simple, and can be solved effectively by modern solvers. However, this basic approach to path-based exploration suffers from the *path explosion problem*, as the number of paths through a program is exponential in the worst case. As an example, consider a loop with a branch in the body. The branching decision is potentially independent in each iteration of the loop, and thus, the program has 2^n distinct paths for n loop iterations.

A principal method to address the path explosion problem is *path merging*. The idea is to merge the formulas that correspond to two (or more) paths at points of reconverging control flow. As a result, the number of formulas is reduced, but the resulting formulas are larger and thus more difficult to solve for the SAT solver. This enables a trade-off between the number of formulas to solve and their relative difficulty.

At the extreme end, the CBMC bounded model checker *always* merges, and thus, generates only a *single* formula for a given unwinding bound k [51, 52, 108, 112]. This formula is linear in the size of the program and linear in k even if there is an exponential number of paths in the program. This corresponds to replicating the basic blocks along the path k times, followed by a transformation of the concatenation of these blocks into SSA form [3]. Other tools perform path merging heuristically in order to contain the total number of formulas.

10.4.3 Completeness for Bounded Programs

Bounded model checking, when applied as described above, is inherently incomplete, as it searches for property violations only up to a given bound and never returns "*No Errors*". Bugs that are deeper than the given bound are missed. Nevertheless, BMC can be used to *prove* liveness and safety properties on a particular class of programs if applied in a slightly different way. The class we consider here are programs that have a high-level worst-case execution time (WCET). Numerous programs are required to have this property, especially in the domain of safety-critical embedded software.

A high-level WCET is typically given by a bound on the maximum number of loop iterations and is usually computed via a simple syntactic analysis of loop structures. If the syntactic analysis fails, an iterative algorithm can be applied. First, a guess k for the bound on the number of loop iterations is made. The loop is then unrolled up to this bound k using BMC. The property that is checked is that any path exceeding k loop iterations is infeasible. If the property holds, k is established as a sound high-level WCET. Otherwise, there are paths in the program exceeding the bound, and a new guess for the bound is made [52, 112].

10.4.4 BMC for Multi-threaded Programs

The verification of concurrent software is primarily discussed in Chap. 18 of this Handbook [88]. We will thus only briefly mention those methods in which the use of SAT, or more general satisfiability modulo theories (SMT) (discussed in Chap. 11 of this Handbook [16]), is most prominent.

The basic approach described above also applies to concurrent software with interleaving semantics. In BMC for this scenario, path formulas with thread interleavings are built. Due to the potential for path explosion, numerous variants for restricting the search, path merging and compression have been considered [60, 78, 82, 143, 144]. An alternative to considering interleavings explicitly during the encoding is to build a formula in which the interleavings are encoded by means of clocks [2, 150]. Further constraint-based approaches to analyzing concurrent programs include [126, 149]. Concurrent programs can be reduced to sequential programs by applying a bound on the number of context switches [127, 143]. This transformation enables the application of analyzers for sequential programs as described above.

Verifiers for concurrent systems usually benefit from some form of partial-order reduction. Instances of BMC-based verifiers for concurrent systems that implement partial-order reduction are [70, 100, 101].

10.4.5 Bounded Model Checking for HW/SW Co-verification

The encodings described in Sect. 10.3.1 for hardware and Sect. 10.4 for software can be combined to form a single SAT instance, which enables the verification of systems that have both a hardware and a software component. This approach is the baseline for the broad area of “symbolic co-simulation” of two models, where one is written in C and the other is a hardware model in (for example) Verilog or SystemC.

A typical scenario is checking the correspondence between a “golden” hardware reference model and an RTL implementation. Another scenario is checking properties of software–hardware interaction, where the software is in C and the hardware is modeled in an HDL. There is a broad variety of styles in which ANSI-C programs or SystemC descriptions are used in these settings as (possibly partial) hardware specifications. In the special case of sequential equivalence checking between C and an HDL this is combined with heuristic insertions of *equivalence cut points*.

10.5 Encodings into Propositional SAT

In this section we elaborate on the original question of how to encode the model and the temporal specification into propositional SAT. Due to the widespread use of C to implement safety-critical software, model checking of C programs, even just for bug hunting, is an important application of formal verification. The challenge in making BMC work for a concrete programming language such as C is many-fold. First, programming languages have complex syntax and semantics which have to be parsed, analyzed and encoded. Reasoning about memory and in particular pointer arithmetic requires non-trivial decision procedures for arrays. In order to model the actual computation, including but not limited to modular arithmetic, bit-precise reasoning is indispensable.

10.5.1 Encoding Bit Vectors

At the core of SAT-based Model Checking is the encoding of word-level operations, which correspond to the evaluation of arithmetic expressions in programming languages or HDLs, into bit-level formulas. This task, also known as *bit-blasting*, is very similar to the synthesis of hardware models on the register transfer level (RTL) into net-lists. Alternatively, operations on the word-level can be modeled in the first-order *theory of bit vectors* (QF_BV).

As discussed in Chap. 11, there are various approaches to handle the bit-vector theory. Here we focus on bit-blasting. As examples we show the encoding of assignments, i.e., equality in BV, and addition of bit vectors. Other arithmetic and logical operations are treated in a similar way. Note that, in general, bit-blasting is an exponential procedure, if bit-widths, as is usually the case, are encoded logarithmically. This exponential explosion cannot be avoided, since the decision problem for full QF_BV is NEXPTIME complete [107].

After encoding models into bit vectors and bit vectors into propositional bit-level logic there remains a last step of encoding bit-level formulas into conjunctive normal form (CNF), the common input format of most SAT solvers.

In order to compactly represent formulas we need sharing. This means we use directed acyclic graphs (DAGs) or simply combinational circuits to represent generated bit-level formulas and not trees, which can be exponentially larger.

A bit-level data structure commonly used for this purpose is And-Inverter-Graphs (AIGs) [123]. AIGs are in essence representations of net-lists (Definition 2). In order to obtain a formula in CNF from an AIG it is possible to first translate the AIG into negation normal form (NNF), which at most doubles the size of the DAG, and then use the distributivity law to eliminate disjunctions over conjunctions. Each elimination of a disjunction is quadratic and thus this approach may lead to an exponential blow-up of the resulting CNF. As a consequence, translating an AIG into CNF by distribution is only feasible for small and shallow formulas. The common approach for translating formulas (and AIGs) into CNF is to use a Tseitin encoding and related optimizations, as discussed in Sect. 10.2.2.

10.5.2 Encoding Memory

Memory occurs in software but also in hardware models. The first-order *theory of arrays* is powerful enough to express most memory-related properties of practical interest. Therefore, decision procedures for the theory of arrays, as presented in Chap. 11, are essential for bounded model checking. We are mostly interested in bit-precise semantics. Thus for bounded model checking, we can focus on the quantifier-free fragment of *arrays over bit vectors* (QF_ABV).

Most of the time, memory in hardware can be handled by standard decision procedures for arrays. However, for software there are additional requirements. In particular, dynamic memory management has to be encoded.

10.5.3 Encodings with Under- and Over-approximation

The direct use of a SAT solver as cited earlier ("bit-blasting") is the conceptually simplest way to implement a bit-vector decision procedure. However, the bit-blasting approach can be too computationally expensive in practice, and there is a pressing need for better decision procedures for bit-vector arithmetic.

One frequently applied method to obtain faster decision procedures for bit-vector arithmetic and other theories is *abstraction*. The key insight is that in many cases, only a small part of the formula needs to be analyzed to conclude whether it is satisfiable or unsatisfiable. The goal of abstraction is to focus on this part of the formula.

Most decision procedures that employ abstraction implement either strict over- or under-approximations. In both cases, the desired result is a formula ϕ' that is easier to solve than the original formula ϕ.

An over-approximation of a decision problem permits more solutions than the original formula. A simple way to obtain an over-approximation for a satisfiability problem is to replace sub-formulas by new variables. In case an over-approximation ϕ' is found to be unsatisfiable, we can conclude that the original formula is unsatisfiable. Nothing, however, can be concluded if ϕ' is satisfiable, since the satisfying assignment for ϕ' need not be a satisfying assignment for ϕ.

Conversely, an under-approximation of a decision problem permits fewer solutions than the original formula. A simple way to obtain an under-approximation for a satisfiability problem is to add further constraints or to replace sub-formulas by constants. In case an under-approximation ϕ' is found to be satisfiable, we can conclude that the original formula is satisfiable. Nothing, however, can be concluded if ϕ' is unsatisfiable. A proof of unsatisfiability of ϕ' need not be a proof of unsatisfiability for ϕ.

Both over- and under-approximations can naturally be combined with forms of automated abstraction refinement, such as those pioneered in [55]. SMT solvers implementing DPLL(T) [15, 138, 147] can be seen as performing iterative refinement (strengthening) of an over-approximation. The array theory is a very typical instance of a fragment of first-order logic that is particularly suitable for over-approximation [120, 137]. Under-approximation is frequently applied in the case of expensive bit-vector arithmetic operations such as multiplication.

In order to obtain the strengths of both over- and under-approximation, *alternation* between the two schemes can be applied. This idea is particularly fruitful if each of the two phases provides refinement information for the other. An instance of this scheme for quantifier-free Presburger arithmetic has been presented in [114]; a variant for quantifier-free bit-vector arithmetic has appeared in [41]. It is also possible to combine over- and under-approximation in a *single* abstraction, thereby forming a *mixed* abstraction. The resulting formula in general neither implies nor is implied by the original formula [38, 39].

10.6 Complete Model Checking with SAT

As explained above, the search for a counterexample of fixed length is inherently incomplete, as means to conclude the absence of counterexamples of any length are missing. We now discuss methods that enable proofs that a given property holds for unbounded depth [7].

10.6.1 Completeness Thresholds

Intuitively, if we could search *deeply enough*, we could guarantee that we have examined all the relevant behavior of the bounded program, and that searching any deeper would only exhibit states that we have explored already. A depth that provides such a guarantee is called a *completeness threshold* [119]. The notion of completeness threshold is used to determine an upper bound on the length k of counterexamples that have to be tried before the property can be declared to hold.

Computing the smallest such threshold is as hard as the model-checking problem itself, and thus, one settles in practice for over-approximations. Techniques for obtaining completeness thresholds include structural analyses of the description of the transition system [20, 21, 109], and semantic analyses of the model and the property [5, 58, 115, 119].

The completeness threshold of a design can be lowered significantly by applying abstraction techniques such as localization reduction [125]. This idea has been exploited in a number of techniques [130, 135].

10.6.2 Image Computation with SAT

BDD-based model checkers perform forward or backward fixed-point iterations in order to determine the truth of a property given in temporal logic. The key step in this procedure is to compute a pre- or post-image of a given set of states with respect to the transition relation. Attempts have been made to emulate this fixed-point iteration using SAT solvers [1, 46, 131].

10.6.3 Basic Inductive Techniques

SAT-based techniques are well suited to check whether a given transition system satisfies a given *inductive invariant*. Recall that I denotes the initial state predicate, and that T denotes the transition relation. A state property P is *inductive* iff

1. P holds in the initial state, i.e., $I \Longrightarrow P$, and
2. P holds in all states reachable from states that satisfy P, i.e.,

$$\bigl(P(s) \wedge T\bigl(s, s'\bigr)\bigr) \Longrightarrow P\bigl(s'\bigr).$$

Observe that both conditions are quantifier-free and can therefore be checked effectively using the techniques we have described so far. The main practical problem is that a property that holds does not have to be inductive. Nothing can be concluded about P if the second condition fails. We now discuss techniques that attempt to address this case.

10.6.3.1 Strengthening the Inductive Argument

Induction can be made more likely to succeed when we check a state property P' that is stronger than the non-inductive property P. Numerous heuristics have been proposed to strengthen inductive arguments, both in the case of software and hardware models. Many initial methods relied on careful manual strengthening of properties to make them inductive, followed by automated heuristics [6].

10.6.3.2 Equivalence Reasoning

Another important preprocessing technique for bit-level model checking is based on iteratively computing the set of equivalent circuit nodes. This in particular includes the set of equivalent latches and registers. The pioneering work of van Eijk [75] consists of a greatest fixpoint computation of this equivalence relation. In essence it computes the largest equivalence relation among signals which is inductive, i.e., is preserved under the transition relation, and holds in the initial state. The resulting equivalence relation can then be used to simplify the model-checking problem by replacing equivalent nodes by representatives. An important related technique is SAT sweeping [122]. For a more complete set of references see [104].

10.6.3.3 Temporal Decomposition

Circuit nodes which are initialized to one specific constant value, true or false, and then never change, can be found in the same way. However, in many practical problems, nodes only stabilize after a certain number n of steps. In this situation, the original model-checking problem should be split into a bounded-model-checking problem for the first n steps, followed by checking a simplified model where the signals fixed after n steps are replaced by constants. This technique is called *temporal decomposition* and was introduced in [44]. Ternary simulation can be used to quickly compute an approximation of stabilizing signals.

10.6.3.4 k-Induction

An automated way to increase the strength of the inductive argument is to increase the depth of the unwinding, forming a formula that is very similar to a BMC instance. In k-induction, we first check that there is no counterexample of length k or less. We then check that no state reachable from a sequence of k-states that satisfy P violates P. Both checks can be performed effectively using a satisfiability decision procedure. The technique was first applied to hardware models [148], and then generalized to include software [64, 65]. The approach is also applicable to liveness properties, e.g., given in LTL, as ω-regular properties, or as Büchi automata [90, 91].

10.6.4 Craig Interpolation

Model checking with Craig interpolation [132] was the first robust complete SAT-based model-checking technique and is still considered to be one of the most effective techniques in practice. It uses an over-approximation of quantifier elimination, for image computation, which is obtained as an interpolation from a refutation of a BMC run between the first and the remaining states of the considered path [132]. The crucial part is an algorithm for extracting an interpolant from a resolution proof in linear time. The technique has been combined with other methods to reduce the complexity of the model, e.g., abstraction [129].

Interpolating decision procedures have been developed for numerous fragments of first-order logic, primarily with the goal of application to approximate loop invariants in program analyzers. An algorithm for interpolation in linear real arithmetic has been given in [133], for transitive relations in [156], and for full quantifier-free Presburger arithmetic in [36, 113]. An interpolating decision procedure for quantifier-free Presburger arithmetic with arrays is described in [37]. A full description of interpolation-based model checking is in Chap. 14 of this Handbook [134].

10.6.5 Iterative Inductive Strengthening

A failing inductive argument can be strengthened iteratively in a BMC-like setting, an idea exploited in the seminal algorithm IC3 [33, 34], also called *property-directed reachability checking* in [71]. As of 2013, IC3 is considered the most efficient single-engine model-checking technique for proving properties of bit-level models. In addition, it is also shown to be able to reach deep counterexamples. IC3 has been extended to full CTL, as demonstrated in [89], as well as to more general models [48, 93].

The basic idea of IC3 is to generate a relative inductive chain $F_0 \subseteq F_1 \subseteq \cdots \subseteq F_k$ of over-approximations of reachable states. "Relative inductive" means that all the successor states of F_i are in F_{i+1}. Starting with the initial state set $F_0 = I$ alone, the algorithm proceeds by either refining frontiers or by increasing k, which adds a new frontier. This process is repeated until the chain reaches a fix-point or a bad state is shown to be reachable.

The frontier sets F_i are refined by adding restrictions on states reachable in one step backward from a goal state, i.e., a bad state. These restrictions are expressed as clauses over state literals. In order to minimize their size, and speed up termination of IC3, the algorithm performs many incremental calls to a SAT solver. Initially only bad states are goal states, but after one step backward, the negation of an added clause becomes a goal too (unless the initial state is reached). These goals can thus be seen as partial models of the transition relation. Finding and minimizing these partial models is the most time-consuming part of the algorithm, and the current state of the art either uses SAT-based techniques [35] or uses ternary simulation [71].

In contrast to bounded model checking, IC3 requires many more calls to the SAT solver, typically in the range of thousands of SAT-solver calls per second. These calls, however, only check properties of one step, e.g., a single copy of the transition relation. This is a very different usage scenario for a SAT solver than in BMC. Further details and a discussion on lifting these ideas to SMT can be found in [34] or in the original publication on IC3 [33].

10.7 Abstraction Techniques Using SAT

10.7.1 Overview of Predicate Abstraction

Promoted by the success of the SLAM toolkit [8, 12, 13], *predicate abstraction* is currently the predominant abstraction technique in software model checking. Graf and Saïdi use *logical predicates* to construct an abstract domain by partitioning a program's state space [84]. The details of this procedure are described in Chap. 15 of this Handbook [99]. We focus on the use of SAT in this context.

In predicate abstraction, a sound approximation $\hat{R}$ of R is constructed using predicates over program variables. A predicate P partitions the states of a program into two classes: one in which P evaluates to true, and one in which it evaluates to false.

Each class is an *abstract state*. Let A and B be abstract states. A transition is defined from A to B (i.e., $(A, B) \in \hat{R}$) if there exists a state in A with a transition to a state in B. This construction yields an *existential abstraction* of a program, sound for reachability properties [56]. The abstract program corresponding to $\hat{R}$ is represented by a *Boolean program* [12, 13]; one with only Boolean data types, and the same control flow constructs as in C programs (including procedures). Together, n predicates partition the state space into 2^n abstract states, one for each truth assignment to all predicates.

10.7.2 Computing Abstractions with SAT

Abstractions are automatically constructed using a decision procedure to decide, for all pairs of abstract states A, B, and instructions Li, whether Li permits a transition from A to B. As n predicates lead to 2^n abstract states, this method requires $(2^n)^2$ calls to a decision procedure to compute an abstraction. In practice, a coarser but more efficiently computed *Cartesian Abstraction* (see for instance [11]) is obtained by constructing an abstraction for each predicate separately and taking the product of the resulting abstract relations.

The decision procedures are either SMT-based first-order logic theorem provers combined with theories such as machine arithmetic, for reasoning about the C programming language (e.g., ZAPATO [10] or SIMPLIFY [63]), or SAT-solvers, used to decide the satisfiability of a bit-level accurate representation of the formulas [53, 59, 120].

We now describe how an abstraction can be verified. Despite the presence of a potentially unbounded call stack, the reachability problem for sequential Boolean programs is decidable [42].[2]

The intuition is that the successor of a state is determined entirely by the top of the stack and the values of global variables, both of which take values in a finite set. Thus, for each procedure, the possible pairs of input-output values, called *summary edges*, is finite and can be cached and used during model checking [12, 79].

All existing model checkers for Boolean programs are symbolic. BDD-based tools suffer from scalability issues if the number of variables is very large. SAT-based methods scale significantly better, but cannot be used to detect fixed points. For this purpose, solvers for quantified Boolean formulas (QBF) must be used [83, 106]. However, the decision problem for QBF, a classical PSPACE-complete problem, faces the same scalability issues as BDDs. Most tools used in practice are therefore still based on BDDs, and the verification phase is often the bottleneck of predicate abstraction.

[2] In fact, all ω-regular properties are decidable for sequential Boolean programs [32].

10.7.3 Simulation with SAT

The reachability computation above may discover that an error state is reachable in the abstract program. Subsequently, a *simulation step* is used to determine whether the error exists in the concrete program or is *spurious*.

Symbolic simulation mentioned in Sect. 10.4.2, in which an abstract state is propagated through the sequence of program locations occurring in the abstract counterexample, is used to determine whether an abstract counterexample is spurious. If so, the abstraction must be *refined* to eliminate the spurious trace. This approach *does not* produce false error messages.

There are two sources of imprecision in the abstract model. *Spurious traces* arise because the set of predicates is not rich enough to distinguish between certain concrete states. *Spurious transitions* arise because the Cartesian abstraction may contain transitions not in the existential abstraction. Spurious traces are eliminated by adding additional predicates, obtained by computing the weakest precondition (or strongest postcondition) of the instructions in the trace. An alternative method is *Craig interpolation* [92]. Spurious transitions are eliminated by adding constraints to the abstract model. Such transitions are eliminated by restricting the valuations of the Boolean variables before and after the transition.

Various techniques to speed up the refinement and the simulation steps have been proposed. *Path slicing* eliminates from the counterexample instructions that do not contribute to a property violation [98]. *Loop detection* is used to compute the effect of arbitrary iterations of loops in a counterexample in a single simulation step [121]. The refinement step can be accelerated by adding statically computed invariants [22, 96], including those that eliminate a whole class of spurious counterexamples [23]. Proof-based refinement eliminates all counterexamples up to a certain length, shifting the computational effort from the verification to the refinement phase, and decreasing the number of iterations required [4].

10.7.4 Abstraction-Based Tools

The SATABS model checker uses SAT- or SMT-based abstraction, simulation and refinement [53, 54], and has also been combined with dynamic execution (testing) [86] and has been applied to concurrent software [18, 19, 157], including the scenario in which the number of threads is not bounded [102]. A proof-based technique to approximate images for bit-vector arithmetic has been proposed in [116]. Predicate abstraction has also been applied to hardware verification and HW/SW co-verification [111] and to SpecC [50] and SystemC models [29–31]. SLAM now also uses an SMT-based decision procedure [9], and experiments have been reported using a SAT-based decision procedure [59]. SAT-based checking has also been applied to the abstraction itself, i.e., to Boolean programs [17]. The LOOPFROG verifier uses SAT to compute a precise transformer for a given loop body and a given abstract domain [117, 118].

10.8 Outlook and Conclusions

We have given an overview of a broad range of SAT-based analysis techniques for both software and hardware, demonstrating the versatility of the approach.

The extension of techniques that rely on propositional SAT to the more general case of *Satisfiability Modulo Theories* (SMT) is often straightforward. The techniques described in Chap. 11 are therefore a very natural starting point for further development of the methods described here.

Early SAT-based methods have been restricted to bounded search, and are therefore typically applied for refutation, i.e., the generation of counterexamples. While bounded verification has been accepted as a useful paradigm in practical verification problems, research in recent years has extended this approach in a variety of ways to enable automated and scalable proofs for non-trivial systems.

Exciting avenues for future research include the generalization of the DPLL algorithm to rich natural domains [61] and the integration of abstraction-based methods implementing the abstract interpretation framework into SAT solvers over natural domains [66–68].

References

1. Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic reachability analysis based on SAT-solvers. In: Graf, S., Schwartzbach, M.I. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1785, pp. 411–425. Springer, Heidelberg (2000)
2. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
3. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Ferrante, J., Mager, P. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 1–11. ACM, New York (1988)
4. Amla, N., McMillan, K.L.: A hybrid of counterexample-based and proof-based abstraction. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 3312, pp. 260–274. Springer, Heidelberg (2004)
5. Awedh, M., Somenzi, F.: Proving more properties with bounded model checking. In: Alur, R., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 96–108. Springer, Heidelberg (2004)
6. Awedh, M., Somenzi, F.: Automatic invariant strengthening to prove properties in bounded model checking. In: Sentovich, E. (ed.) Design Automation Conf. (DAC), pp. 1073–1076. ACM, New York (2006)
7. Awedh, M., Somenzi, F.: Termination criteria for bounded model checking: extensions and comparison. Electron. Notes Theor. Comput. Sci. **144**(1), 51–66 (2006)
8. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Berbers, Y., Zwaenepoel, W. (eds.) European Conf. on Computer Systems (EuroSys), pp. 73–85. ACM, New York (2006)
9. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Bloem, R., Sharygina, N. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 35–42. IEEE, Piscataway (2010)

10. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: automatic theorem proving for predicate abstraction refinement. In: Alur, R., Peled, D. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 457–461. Springer, Heidelberg (2004)
11. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
12. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) Intl. Workshop on SPIN Model Checking and Software Verification. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
13. Ball, T., Rajamani, S.K.: Boolean programs: a model and process for software analysis. Tech. rep., Microsoft Research (2000)
14. Barner, S., Eisner, C., Glazberg, Z., Kroening, D., Rabinovitz, I.: ExpliSAT: guiding SAT-based software verification with explicit states. In: Yorav, K. (ed.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 4383, pp. 138–154. Springer, Heidelberg (2007)
15. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
16. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
17. Basler, G., Kroening, D., Weissenbacher, G.: SAT-based summarization for Boolean programs. In: Bosnacki, D., Edelkamp, S. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 4595, pp. 131–148 (2007)
18. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 64–78. Springer, Heidelberg (2009)
19. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Context-aware counter abstraction. Form. Methods Syst. Des. **36**(3), 223–245 (2010)
20. Baumgartner, J., Kuehlmann, A.: Enhanced diameter bounding via structural transformation. In: Design, Automation & Test in Europe Conf. and Exposition (DATE), pp. 36–41. IEEE, Piscataway (2004)
21. Baumgartner, J., Kuehlmann, A., Abraham, J.A.: Property checking via structural analysis. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 151–165. Springer, Heidelberg (2002)
22. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
23. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 300–309. ACM, New York (2007)
24. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press, Amsterdam (2009)
25. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: Cleaveland, R., Garavel, H. (eds.) Intl. ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS), pp. 160–177. Elsevier, Amsterdam (2002)
26. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
27. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Log. Methods Comput. Sci. **2**(5), 1–64 (2006)

28. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
29. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. In: Intl. Conf. on Computer-Aided Design (ICCAD), pp. 356–363. IEEE, Piscataway (2008)
30. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. ACM Trans. Des. Autom. Electron. Syst. **15**(3), 1–32 (2010)
31. Blanc, N., Kroening, D., Sharygina, N.: Scoot: a tool for the analysis of SystemC models. In: Ramakrishnan, C.R., Rehof, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 467–470. Springer, Heidelberg (2008)
32. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) Intl. Conf. on Concurrency Theory (CONCUR). LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
33. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
34. Bradley, A.R.: Understanding IC3. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 1–14. Springer, Heidelberg (2012)
35. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Formal Methods in Computer Aided Design (FMCAD), pp. 173–180. IEEE, Piscataway (2007)
36. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. In: Giesl, J., Hähnle, R. (eds.) Intl. Joint Conf. on Automated Reasoning (IJCAR). LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
37. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 88–102. Springer, Heidelberg (2011)
38. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: Formal Methods in Computer Aided Design (FMCAD), pp. 69–76. IEEE, Piscataway (2009)
39. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) Intl. Conf. on Computer Aided Systems Theory (EUROCAST). LNCS, vol. 5717, pp. 304–311. Springer, Heidelberg (2009)
40. Bryant, R.E.: Graph Based Algorithms for Boolean Function Manipulation. Trans. Comput. **C-35**(8), 677–691 (1986)
41. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
42. Büchi, J.R.: Regular canonical systems. Arch. Math. Log. **6**(3–4), 91–111 (1964)
43. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Commun. ACM **56**(2), 82–90 (2013)
44. Case, M.L., Mony, H., Baumgartner, J., Kanzelman, R.: Enhanced verification by temporal decomposition. In: Formal Methods in Computer Aided Design (FMCAD), pp. 17–24. IEEE, Piscataway (2009)
45. Chambers, B., Manolios, P., Vroon, D.: Faster SAT solving with better CNF generation. In: Design, Automation & Test in Europe (DATE), pp. 1590–1595. IEEE, Piscataway (2009)
46. Chauhan, P., Clarke, E.M., Kroening, D.: A SAT-based algorithm for reparameterization in symbolic simulation. In: Malik, S., Fix, L., Kahng, A.B. (eds.) Design Automation Conf. (DAC), pp. 524–529. ACM, New York (2004)
47. Chockler, H., Kroening, D., Purandare, M.: Computing mutation coverage in interpolation-based model checking. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **31**(5), 765–778 (2012)

48. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)
49. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Cabodi, G., Singh, S. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 52–59. IEEE, Piscataway (2012)
50. Clarke, E., Jain, H., Kroening, D.: Verification of SpecC using predicate abstraction. Form. Methods Syst. Des. **30**(1), 5–28 (2007)
51. Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Yasuura, H. (ed.) Asia and South Pacific Design Automation Conf. (ASPDAC), pp. 308–311. IEEE, Piscataway (2003)
52. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
53. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Form. Methods Syst. Des. **25**(2–3), 105–127 (2004)
54. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
55. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
56. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
57. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
58. Clarke, E.M., Kroening, D., Ouaknine, J., Strichman, O.: Computational challenges in bounded model checking. Softw. Tools Technol. Transf. **7**(2), 174–183 (2005)
59. Cook, B., Kroening, D., Sharygina, N.: Cogent: accurate theorem proving for program verification. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 296–300. Springer, Heidelberg (2005)
60. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) Intl. Conf. on Software Engineering (ICSE), pp. 331–340. ACM, New York (2011)
61. Cotton, S.: Natural domain SMT: a preliminary assessment. In: Chatterjee, K., Henzinger, T.A. (eds.) Intl. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, vol. 6246, pp. 77–91. Springer, Heidelberg (2010)
62. Currie, D.W., Hu, A.J., Rajan, S.P.: Automatic formal verification of DSP software. In: Micheli, G.D. (ed.) Design Automation Conf. (DAC), pp. 130–135. ACM, New York (2000)
63. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Tech. rep., HP Labs (2003)
64. Donaldson, A., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In: Jhala, R., Schmidt, D.A. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 6538, pp. 169–183. Springer, Heidelberg (2011)
65. Donaldson, A., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6015, pp. 280–295. Springer, Heidelberg (2010)
66. D'Silva, V., Haller, L., Kroening, D.: Satisfiability solvers are static analysers. In: Miné, A., Schmidt, D. (eds.) Intl. Symp. on Static Analysis (SAS). LNCS, vol. 7460, pp. 317–333. Springer, Heidelberg (2012)
67. D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: Giacobazzi, R., Cousot, R. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 143–154. ACM, New York (2013)

68. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric bounds analysis with conflict-driven learning. In: Flanagan, C., König, B. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 7214, pp. 48–63. Springer, Berlin (2012)
69. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **27**(7), 1165–1178 (2008)
70. Dubrovin, J., Junttila, T.A., Heljanko, K.: Exploiting step semantics for efficient bounded model checking of asynchronous systems. Sci. Comput. Program. **77**(10–11), 1095–1121 (2012)
71. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Bjesse, P., Slobodová, A. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 125–134. FMCAD, Austin (2011)
72. Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007)
73. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2003)
74. Eén, N., Sterin, B., Claessen, K.: A circuit approach to LTL model checking. In: Jobstmann, B., Ray, S. (eds.) Formal Methods in Computer Aided Design (FMCAD), pp. 53–60. IEEE, Piscataway (2013)
75. van Eijk, C.A.J.: Sequential equivalence checking based on structural similarities. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **19**(7), 814–819 (2000)
76. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)
77. Eisner, C., Fisman, D.: Functional specification of hardware via temporal logic. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
78. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Ball, T., Sagiv, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 411–422. ACM, New York (2011)
79. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electron. Notes Theor. Comput. Sci. **9**, 27–37 (1997)
80. Gan, X., Dubrovin, J., Heljanko, K.: A symbolic model checking approach to verifying satellite onboard software. Sci. Comput. Program. **82**, 44–55 (2014)
81. Ganai, M.K., Gupta, A.: SAT-Based Scalable Formal Verification Solutions. Springer, Heidelberg (2007)
82. Ghafari, N., Hu, A.J., Rakamaric, Z.: Context-bounded translations for concurrent software: an empirical evaluation. In: van de Pol, J., Weber, M. (eds.) Intl. Workshop on Model Checking Software (SPIN). LNCS, vol. 6349, pp. 227–244. Springer, Heidelberg (2010)
83. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 761–780. IOS Press, Amsterdam (2009)
84. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
85. Groce, A., Kroening, D.: Making the most of BMC counterexamples. Electron. Notes Theor. Comput. Sci. **119**, 67–81 (2005)

86. Groce, A., Kroening, D., Clarke, E.: Counterexample guided abstraction refinement via program execution. In: Davies, J., Schulte, W., Barnett, M. (eds.) Intl. Conf. on Formal Engineering Methods (ICFEM). LNCS, vol. 3308, pp. 224–238. Springer, Heidelberg (2004)
87. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with explain. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004)
88. Gupta, A., Kahlon, V., Qadeer, S., Touili, T.: Model checking concurrent programs. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
89. Hassan, Z., Bradley, A.R., Somenzi, F.: Incremental, inductive CTL model checking. In: Madhusudan, P., Seshia, S.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 7358, pp. 532–547. Springer, Heidelberg (2012)
90. Heljanko, K., Junttila, T.A., Keinänen, M., Lange, M., Latvala, T.: Bounded model checking for weak alternating Büchi automata. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 95–108. Springer, Heidelberg (2006)
91. Heljanko, K., Junttila, T.A., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
92. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 232–244. ACM, New York (2004)
93. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
94. Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M.K.: Model checking C programs using F-SOFT. In: Intl. Conf. on Computer Design (ICCD), pp. 297–308. IEEE, Piscataway (2005)
95. Jain, H., Clarke, E.M.: Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts. In: Design Automation Conf. (DAC), pp. 563–568. ACM, New York (2009)
96. Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 137–151. Springer, Heidelberg (2006)
97. Järvisalo, M., Biere, A., Heule, M.: Simulating circuit-level simplifications on CNF. J. Autom. Reason. **49**(4), 583–619 (2012)
98. Jhala, R., Majumdar, R.: Path slicing. In: Sarkar, V., Hall, M.W. (eds.) Conf. on Programming Language Design and Implementation (PLDI), pp. 38–47. ACM, New York (2005)
99. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
100. Jussila, T., Heljanko, K., Niemelä, I.: BMC via on-the-fly determinization. Electron. Notes Theor. Comput. Sci. **89**(4), 561–577 (2003)
101. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: an optimal symbolic partial order reduction technique. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009)
102. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6174, pp. 654–659. Springer, Heidelberg (2010)
103. Kautz, H.A., Selman, B.: Pushing the envelope: planning, propositional logic and stochastic search. In: Clancey, W.J., Weld, D.S. (eds.) National Conf. on Artificial Intelligence (AAAI), pp. 1194–1201. AAAI Press/MIT Press, Portland/Cambridge (1996)
104. Khasidashvili, Z., Nadel, A.: Implicative simultaneous satisfiability and applications. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 7261, pp. 66–79. Springer, Heidelberg (2011)

105. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
106. Kleine Büning, H., Bubeck, U.: Theory of quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 735–760. IOS Press, Amsterdam (2009)
107. Kovásznai, G., Fröhlich, A., Biere, A.: On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In: Fontaine, P., Goel, A. (eds.) Intl. Workshop on Satisfiability Modulo Theories (SMT), pp. 44–55 (2012). EasyChair
108. Kroening, D.: Application specific higher order logic theorem proving. In: Autexier, S., Mantel, H. (eds.) Proc. of the Verification Workshop (VERIFY), pp. 5–15 (2002)
109. Kroening, D.: Computing over-approximations with bounded model checking. Electron. Notes Theor. Comput. Sci. **144**(1), 79–92 (2006)
110. Kroening, D.: Software verification. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 505–532. IOS Press, Amsterdam (2009)
111. Kroening, D., Clarke, E.: Checking consistency of C and Verilog using predicate abstraction and induction. In: Intl. Conf. on Computer-Aided Design (ICCAD), pp. 66–72. IEEE/ACM, Piscataway/New York (2004)
112. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Design Automation Conf. (DAC), pp. 368–371. ACM, New York (2003)
113. Kroening, D., Leroux, J., Rümmer, P.: Interpolating quantifier-free Presburger arithmetic. In: Fermüller, C.G., Voronkov, A. (eds.) Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 6397, pp. 489–503. Springer, Heidelberg (2010)
114. Kroening, D., Ouaknine, J., Seshia, S., Strichman, O.: Abstraction-based satisfiability solving of Presburger arithmetic. In: Alur, R., Peled, D.A. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3114, pp. 308–320. Springer, Heidelberg (2004)
115. Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., Worrell, J.: Linear completeness thresholds for bounded model checking. In: Gopalakrishnan, G., Qadeer, S. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 6806, pp. 557–572. Springer, Heidelberg (2011)
116. Kroening, D., Sharygina, N.: Approximating predicate images for bit-vector logic. In: Hermanns, H., Palsberg, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 242–256. Springer, Heidelberg (2006)
117. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.: Loopfrog: a static analyzer for ANSI-C programs. In: Intl. Conf. on Automated Software Engineering (ASE), pp. 668–670. IEEE, Piscataway (2009)
118. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Cha, S.D., Choi, J., Kim, M., Lee, I., Viswanathan, M. (eds.) Intl. Symp. on Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 5311, pp. 111–125. Springer, Heidelberg (2008)
119. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 2575, pp. 298–309. Springer, Heidelberg (2003)
120. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg (2008)
121. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: Ball, T., Jones, R.B. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 4144, pp. 152–165. Springer, Heidelberg (2006)
122. Kuehlmann, A.: Dynamic transition relation simplification for bounded property checking. In: Intl. Conf. on Computer-Aided Design (ICCAD), pp. 50–57. IEEE/ACM, Piscataway/New York (2004)

123. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **21**(12), 1377–1394 (2002)
124. Kuismin, T., Heljanko, K.: Increasing confidence in liveness model checking results with proofs. In: Bertacco, V., Legay, A. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 8244, pp. 32–43. Springer, Heidelberg (2013)
125. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
126. Lahiri, S.K., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
127. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Form. Methods Syst. Des. **35**(1), 73–97 (2009)
128. Latvala, T., Biere, A., Heljanko, K., Junttila, T.A.: Simple bounded LTL model checking. In: Hu, A.J., Martin, A.K. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 3312, pp. 186–200. Springer, Heidelberg (2004)
129. Li, B., Somenzi, F.: Efficient abstraction refinement in interpolation-based unbounded model checking. In: Hermanns, H., Palsberg, J. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3920, pp. 227–241. Springer, Heidelberg (2006)
130. Li, B., Wang, C., Somenzi, F.: Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. Softw. Tools Technol. Transf. **7**(2), 143–155 (2005)
131. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
132. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A. Jr., Somenzi, F. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
133. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)
134. McMillan, K.L.: Interpolation and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
135. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
136. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conf. (DAC), pp. 530–535. ACM, New York (2001)
137. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: Formal Methods in Computer Aided Design (FMCAD), pp. 45–52. IEEE, Piscataway (2009)
138. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
139. Penczek, W., Wozna, B., Zbrzezny, A.: Bounded model checking for the universal fragment of CTL. Fundam. Inform. **51**(1–2), 135–156 (2002)
140. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2018)
141. Prasad, M., Biere, A., Gupta, A.: A survey on recent advances in SAT-based formal verification. Softw. Tools Technol. Transf. **7**(2), 156–173 (2005)

142. Prestwich, S.D.: CNF encodings. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 75–97. IOS Press, Amsterdam (2009)
143. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
144. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005)
145. Rintanen, J.: Planning as satisfiability: heuristics. Artif. Intell. **193**, 45–86 (2012)
146. Schuppan, V., Biere, A.: Efficient reduction of finite state model checking to reachability analysis. Softw. Tools Technol. Transf. **5**(1–2), 185–204 (2004)
147. Sebastiani, R.: Lazy satisfiability modulo theories. J. Satisf. Boolean Model. Comput. **3**(3–4), 141–224 (2007)
148. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A. Jr., Johnson, S.D. (eds.) Formal Methods in Computer Aided Design (FMCAD). LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
149. Sinha, N., Wang, C.: Staged concurrent program analysis. In: Roman, G.C., Sullivan, K.J. (eds.) Intl. Symp. on Foundations of Software Engineering (FSE), pp. 47–56. ACM, New York (2010)
150. Sinha, N., Wang, C.: On interference abstractions. In: Ball, T., Sagiv, M. (eds.) Symp. on Principles of Programming Languages (POPL), pp. 423–434. ACM, New York (2011)
151. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part II. Seminars in Mathematics, vol. 8, pp. 234–259 (1968). V.A. Steklov Mathematical Institute. English Translation, Consultants Bureau, pp. 115–125 (1970)
152. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
153. Velev, M.N.: Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In: Imai, M. (ed.) Asia and South Pacific Design Automation Conf. (ASPDAC), pp. 310–315. IEEE, Piscataway (2004)
154. Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, Heidelberg (2005)
155. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proc. IEEE **103**(11), 2021–2035 (2015). doi:10.1109/JPROC.2015.2455034
156. Weissenbacher, G., Kroening, D.: An interpolating decision procedure for transitive relations with uninterpreted functions. In: Namjoshi, K.S., Zeller, A., Ziv, A. (eds.) Intl. Haifa Verification Conference (HVC). LNCS, vol. 6405, pp. 150–168. Springer, Heidelberg (2009)
157. Witkowski, T., Blanc, N., Weissenbacher, G., Kroening, D.: Model checking concurrent Linux device drivers. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) Intl. Conf. on Automated Software Engineering (ASE), pp. 501–504. ACM, New York (2007)