Model Checking: From BDDs to Interpolation

Orna GRUMBERG
Computer Science Department, Technion, Haifa, Israel

Abstract. In this paper we describe the development of model checking from BDD-based verification, through SAT-based bug finding, to Interpolation-based verification.

Model checking is an automatic approach to formally verifying that a given system satisfies a given specification. BDD-based *Symbolic Model Checking* (SMC) was the first to enable model checking of real-life designs with a few hundreds of state elements. Currently, SAT-based model checking is the most widely used method for verifying industrial designs. This is due to its ability to handle designs with thousands of state elements and more. Its main drawback, however, is its orientation towards "bug-hunting" rather than full verification.

In this paper we present two SAT-based approaches to full verification. The approaches combine BMC with *interpolation* or *interpolation-sequence* in order to compute an over-approximated set of the system's reachable states while checking that the specification is not violated. We compare the two methods both algorithmically and experimentally and conclude that they are incomparable.

Keywords. Model Checking, BDD-based Symbolic Model Checking (SMC), SAT-based Model Checking, Interpolation, Interpolation Sequence, Bounded Model Checking (BMC)

1. Introduction

Computerized systems dominate almost every aspect of our lives and their correct behavior is essential. *Model checking* [15] is an automated verification technique for checking whether a given system satisfies a desired property. Unlike testing or simulation based verification, model checking tools are exhaustive in the sense that they traverse *all* behaviors of the system, and either confirm that the system behaves correctly or present a *counterexample*.

Model checking has been successfully applied to verifying hardware and software systems. However, with the rapid increase in size and complexity of computerized systems, there is a constant need for a similar increase in verification capabilities.

In this paper we will survey several model checking techniques which can improve model checking applicability and scalability. We will start with the "old fashion" techniques of BDD-based Symbolic Model Checking (SMC) [9] and SAT-

based Bounded Model Checking (BMC) [7]. We will then proceed to using interpolation and interpolation-sequence for SAT-based model checking.

One of the main limitations of model checking is the *state explosion problem* which arises due to the huge state space of the checked systems. The size of the model induces high memory and time requirements that may make model checking infeasible. Traditionally, BDD-based methods are known to suffer from high memory requirements while SAT-based methods have high time requirements. Much research efforts have been invested along the years in trying to solve this problem.

The first significant solution was the introduction of BDDs [8] into model checking. BDD-based *Symbolic Model Checking* (SMC) [9] enabled model checking of real-life hardware designs with a few hundreds of state elements. However, current design blocks with well-defined functionality typically have thousands of state elements and more. To handle designs of that scale, SAT-based *Bounded Model Checking* (BMC) [7] has been developed. BMC is currently the most widely used method for formal verification of industrial designs. Its main drawback, however, is its orientation towards "bug-hunting" rather than full verification.

Several works extend BMC for full verification. [6] defines a *Reachability Diameter*, which sets a bound on the number of BMC iterations needed for full verification. This bound, however, is usually hard to compute. Moreover, the bound is often very large and therefore the resulting formulas are too large for a SAT solver to handle.

[35] suggests to use *Induction* for full verification. This method uses the BMC check as the induction base. Then, the induction step is determined by checking a second formula. The induction method works automatically mainly for simple local properties. For complex properties, the user has to come up with a good inductive invariant. A completely different approach is the *Proof-Based Abstraction* [32] which exploits BMC to determine an abstract model on which BDD-based model checking can be applied.

In this paper we present two SAT-based approaches to full verification which combine BMC with *interpolation* [17] or *interpolation-sequence* [24, 31]. The proposed methods compute an over-approximated set of the system's reachable states while checking that the specification is not violated. The process terminates with either a counterexample produced by BMC, or by reaching a fixpoint, indicating that no more reachable states will be found. In the latter case, since no violation of the formula has been encountered so far, it is guaranteed that the property holds.

The two approaches result in different traversals of the sets of reachable states, thus their convergence may differ. We compare them both on the algorithmic level and by running experiments. As is often the case with model checking techniques and tools, none is overall superior to the other.

In order to compare the methods experimentally we implemented them within Intel's verification tool. All experiments were conducted on models from Intel's Sandy Bridge micro-architecture. The checked properties are real specifications, used to verify those designs. The experiments compare various parameters of the two methods. In all our experiments, when a fixpoint could be reached only at a high bound, the interpolation-sequence based (ISB) algorithm performed better

than the interpolation based (IB) algorithm. The IB algorithm, on the other hand, performed better when a fixpoint could be reached at a low bound. Falsified properties always favored the ISB algorithm.

When describing the two methods we assume a safety property of the form AGq, where q is a propositional formula. This, however, does not restrict their generality since model checking of liveness properties can be reduced to handling safety properties [4]. Further, model checking of safety properties can be reduced to handling properties of the form AGq [25].

2. Model Checking

Temporal logic model checking [15] is an automatic approach to formally verifying that a given system satisfies a given specification. The system is often modelled by a finite state transition graph called *Kripke structure* and the specification is written in a *temporal logic*. Determining whether a model satisfies a given specification is often based on an exploration of the model's state space in a search for violations of the specification.

Definition 2.1. Given a set of atomic propositions AP, a *Kripke structure* M is the quadruple $M = (S, INIT, TR, L)$ where S is a finite set of states, $INIT \subseteq S$ is a set of initial states and $TR \subseteq S \times S$ is a *total* transition relation. That is, for every $s \in S$ there exists $s' \in S$ such that $(s, s') \in TR$. Additionally, $L : S \to \mathcal{P}(AP)$ is the labeling function which associates with every state $s \in S$ the set $L(s)$ of atomic propositions true in s.

A *path* in a Kripke structure M is a sequence of states $\pi = (s_0, s_1, \ldots)$ such that for all $i \geq 0$, $s_i \in S$ and $(s_i, s_{i+1}) \in TR$. The length of a path is denoted by $|\pi|$. If π is infinite then $|\pi| = \infty$. If $\pi = (s_0, s_1, \ldots, s_n)$ then $|\pi| = n$. A path is an *initial path* when $s_0 \in INIT$. We sometimes refer to a prefix of a path as a path as well.

A formula in *Linear Temporal Logic* (LTL) [15,34] is of the form $\mathbf{A}f$ where f is a path formula. A model M satisfies an LTL property $\mathbf{A}f$ if all infinite initial paths in M satisfy f. If there exists an infinite initial path not satisfying f, this path is defined to be a *counterexample*.

In this paper we consider a subset of LTL formulas of the form $\mathbf{AG}\, q$, where q is a propositional formula. As mentioned before, this does not restrict the generality of the suggested methods since model checking of liveness properties can be reduced to handling safety properties [4]. Further, model checking of safety properties can be reduced to handling properties of the form $\mathbf{AG}\, q$ [25].

The *model checking problem* is the problem of determining whether a given model satisfies a given property. Let M be a model, *Reach* be the set of reachable states in M, and $f = \mathbf{AG}\, q$ be a property. If for every $s \in Reach$, $L(s) \models q$ then the property holds in M. On the other hand, if there exists a state $s \in Reach$ such that $L(s) \models \neg q$ then there exists an initial path $\pi = s_0, s_1, \ldots, s_n$ such that $s_n = s$. The path π is a *counterexample* for the property f.

We would sometimes like to represent a Kripke structure by means of propositional formulas (or, equivalently, Boolean functions). In order to do so, we de-

fine a set of *Boolean state variables*, denoted V. When $|V| = n$, a state $s \in S$ is represented by a vector in the set $\{0,1\}^n$. Thus, s is a valuation of the state variables in V. A set of states S' can be represented by a formula η over V, where the satisfying assignments of η represent the states in S'. By abuse of notation we will refer to a formula η over V as a set of states and use the notion $s \in \eta$ for states represented by η. For some variable v, v' is used to denote the value of v after one time unit. The set of these variables is denoted by V'. In the general case V^i is used to denote the variables in V after i time units (thus, $V^0 \equiv V$). Let η be a formula over V^i, the formula $\eta[V^i \leftarrow V^j]$ is identical to η except that for each variable $v \in V$, v^i is replaced with v^j.

Model checking has been successfully applied in hardware verification, and is emerging as an industrial standard tool for hardware design . A partial list of tools for hardware verification includes SMV [28] and NuSMV [13], FormalCheck [21], RuleBase [2], and Forecast [19]. In recent years, several tools for model checking of software have been developed and applied to non-trivial examples. A partial list consists of SPIN [23], Bandera [16], Java PathFinder [22], SLAM, Bebop, and Zing [1], Blast [3], Magic [11], and CBMC [14]. An extensive overview of model checking algorithms can be found in [15].

The main technical challenge in model checking is the *state explosion* problem which occurs if the system is a composition of several components or if the system variables range over large domains.

An explicit state model checker is a program which performs model checking directly on a Kripke structure. SPIN [23] is an example of a successful tool of that kind. Large models are often handled implicitly, based on a symbolic representation of the Kripke structure by means of Boolean functions or propositional formulas. Two widely used such approaches are the BDD-based model checking [10,28] and the SAT-based bounded model checking [5], described in the following sections.

3. BDD-Based Model Checking

Ordered Binary Decision Diagrams (BDDs) [8] are canonical representations of Boolean functions. They are often concise in their memory requirements. Furthermore, most operations needed for model checking can be defined in terms of Boolean operations on Boolean functions and can be implemented efficiently with BDDs.

In BDD-based (also called *symbolic*) model checking (SMC) sets of states are represented by Boolean functions over the set of Boolean state variables V, as explained in the previous section. Those functions are represented by BDDs. The transition relation of the Kripke structure is also represented by a Boolean function (and thus by a BDD), over the sets V and V' of variables representing current and next states, respectively.

BDDs are sometimes, but not always, exponentially smaller than explicit representation of the corresponding Boolean functions. In such cases, symbolic verification is successful.

Since BDDs are particularly suitable for representing set of states, BDD-based algorithms are based on operations on sets, which are implemented by means of

operations on BDDs (or the corresponding Boolean function). For example, union and intersection of sets correspond to disjunction and conjunction of the BDDs representing those sets.

Two special operations are central to model checking. Given a set of states Q, the *image computation* computes the set of successors of states in Q:

$$Image(Q) := \{t \mid \exists s[TR(s,t) \ \wedge \ Q(s)]\}.$$

The *preimage computation* computes the set of the predecessors of states in Q:

$$Preimage(Q) := \{s \mid \exists t[TR(s,t) \ \wedge \ Q(t)]\}.$$

Unfortunately, in contrast to pure Boolean operations, these operations are not efficiently computable [28], and their computation is a major bottleneck in symbolic model checking.

We conclude this section by demonstrating a BDD-based algorithm for model checking the property $\mathbf{AG}\, q$, for some atomic proposition q. The algorithm manipulates sets of states represented as BDDs. It checks whether $M \models \mathbf{AG}\, q$ by computing the set of reachable states starting from the set of initial states. It iteratively applies the image computation in order to find the set of successors of the current set of states. If a reachable state that does not satisfy q is found, then the procedure returns "$M \not\models \mathbf{AG}\, q$". Otherwise, when all reachable states have been found, the procedure returns "$M \models \mathbf{AG}\, q$". We assume that the structure M is given by the following BDDs: $TR(V, V')$ is the BDD representing the transition relation, $INIT(V)$ represents the set of initial states, and for each $p \in AP$, $S_p(V)$ represents the set of states in M that satisfy p. Figure 1 presents the algorithm.

```
 1: procedure CheckAG(S_q)
 2:     Reach := INIT
 3:     New := INIT
 4:     while New ≠ ∅ do
 5:         if New ∩ ¬S_q ≠ ∅ then return "M ⊭ AG q"      // New ⊈ S_q
 6:         end if
 7:         New := Image(New)
 8:         New := New \ Reach
 9:         Reach := Reach ∪ New
10:     end while
11:     return "M ⊨ AG q"
12: end procedure
```

Figure 1. The procedure CheckAG for checking the formula $f = \mathbf{AG}\, q$

4. Bounded Model Checking

Many problems, including some versions of model checking, can naturally be translated into the *satisfiability* problem of the propositional calculus. The satisfi-

ability problem is known to be NP-complete. Nevertheless, modern SAT-solvers, developed in recent years, can handle formulas with several thousands of variables within a few seconds. SAT-solvers such as Grasp [26], Prover [36], Chaff [33], MiniSAT [18], and many others, are based on sophisticated learning techniques and data structures that accelerate the search for a satisfying assignment, if exists.

A SAT solver is a complete decision procedure that given a propositional formula, determines whether the formula is *satisfiable* or *unsatisfiable*. Most SAT solvers assume a formula in *Conjunctive Normal Form* (CNF), consisting of a conjunction of a set of clauses, each of which is a disjunction of propositional variables or their negation. A CNF formula is satisfiable if there exists a *satisfying assignment* for which every clause in the set is evaluated to $\top$. If the clause set is satisfiable then the SAT solver returns a satisfying assignment for it. If it is not satisfiable (unsatisfiable), meaning, it has no satisfying assignment, then modern SAT solvers produce a *proof of unsatisfiability* [32,39]. The proof of unsatisfiability has many useful applications. We will introduce one of them in the next section.

Below we describe a simple way to exploit satisfiability for bounded model checking of properties of the form $\mathbf{AG}\, q$, where q is a Boolean formula.

Bounded model checking (BMC) [7] is an iterative process for checking properties of a given structure up to a given bound. Let M be a Kripke structure and $f = AGq$ be the property to be verified. Given a bound k, BMC either finds a counterexample of length k or less for f in M, or concludes that there is no such counterexample. In order to search for a counterexample of length k the following propositional formula is built:

Formula 1. $\varphi^k_M(f) = INIT(V^0) \wedge TR(V^0, V^1) \wedge TR(V^1, V^2) \wedge \ldots \wedge TR(V^{k-1}, V^k) \wedge (\neg q(V^k))$

$\varphi^k_M(f)$ is then passed to a SAT solver which searches for a satisfying assignment. If there exists a satisfying assignment for $\varphi^k_M(f)$ then the property is violated, since there exists a path of M of length k violating the property. In order to conclude that there is no counterexample of length k or less, BMC iterates all lengths from 0 up to the given bound k. At each iteration a SAT procedure is invoked.

When M and f are obvious from the context we omit them from the formula $\varphi^k_M(f)$ denoting it as φ^k. The BMC algorithm is described in Figure 2.

The main drawback of this approach is its incompleteness. It can only guarantee that there is no counterexample of size smaller or equal to k. It cannot guarantee that there is no counterexample of size greater than k.

Thus, this method is mainly suitable for refutation. Verification is obtained only if the bound k exceeds the length of the longest path among all shortest paths from an initial state to some state in M. In practice, it is hard to compute this bound and even when known, it is often too large to handle. Several methods for full verification with SAT have been suggested, such as induction [35], ALL-SAT [12,20,29], and interpolation [27,30,38]. In the rest of the paper we will focus on SAT-based verification with interpolation.

```
1: function BMC(M,f,k)
2:     i := 0
3:     while i ≤ k do
4:         build φ^i_M(f)
5:         result = SAT(φ^i_M(f))
6:         if result = true then
7:             return cex      // returning the counterexample
8:         else
9:             i = i + 1
10:        end if
11:    end while
12:    return No cex for bound k
13: end function
```

Figure 2. Bounded model checking

5. Interpolation

In this section we introduce two notions, *interpolation* [17] and *interpolation-sequence* [24] that, when combined with BMC can provide full program verification.

Throughout the paper we denote the value *false* as $\bot$ and the value *true* as $\top$. For a formula X, $\mathcal{L}(X)$ is the set of variables appearing in X. For a set of formulas $\{X_1, \ldots, X_n\}$ we will use $\mathcal{L}(X_1, \ldots, X_n)$ to denote the variables appearing in $X_1, \ldots, X_n$. That is, $\mathcal{L}(X_1, \ldots, X_n) = \mathcal{L}(X_1) \cup \ldots \cup \mathcal{L}(X_n)$.

Definition 5.1. Let (A, B) be a pair of formulas such that $A \wedge B \equiv \bot$. The *interpolant* for (A, B) is a formula I such that:

- $A \Rightarrow I$.
- $I \wedge B \equiv \bot$.
- $\mathcal{L}(I) \subseteq \mathcal{L}(A) \cap \mathcal{L}(B)$.

The interpolant can be viewed as the part of A that is sufficient to contradict B. As mentioned above, modern SAT solvers produce a *proof of unsatisfiability* if the checked formula is unsatisfiable. An interpolant can be extracted from a proof of unsatisfiability [30], where different proofs yield different interpolants.

A similar notion can be defined when we have a sequence of formulas whose conjunction is unsatisfiable.

Definition 5.2. Let $\Gamma = \{A_1, A_2, \ldots, A_n\}$ be a set of formulas such that $\bigwedge \Gamma \equiv \bot$. That is $\bigwedge \Gamma = A_1 \wedge \ldots \wedge A_n$ is unsatisfiable. An *interpolation-sequence* for Γ is a set $\{\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_n\}$ such that:

1. $\mathcal{I}_0 \equiv \top$ and $\mathcal{I}_n \equiv \bot$
2. For every $0 \leq j < n$ it holds that $\mathcal{I}_j \wedge A_{j+1} \Rightarrow \mathcal{I}_{j+1}$
3. For every $0 < j < n$ it holds that $\mathcal{L}(\mathcal{I}_j) \subseteq \mathcal{L}(A_1, \ldots, A_j) \cap \mathcal{L}(A_{j+1}, \ldots, A_n)$

Computing an interpolation-sequence for a sequence of formulas is done in the following way: for each $\mathcal{I}_i$, $0 < i < n$, the sequence of formulas is partitioned

in a different way such that $\mathcal{I}_i$ is the interpolant for the formulas $A(i) = \bigwedge_{j=1}^{i} A_j$ and $B(i) = \bigwedge_{j=i+1}^{n} A_j$. In fact, all interpolants $\mathcal{I}_i$ in the sequence can be computed efficiently at once, by a single traversal of a given proof of unsatisfiability [38].

Theorem 5.3. *Let $\Gamma = \{A_1, A_2, \ldots, A_n\}$ be a set of formulas such that $\bigwedge \Gamma \equiv \bot$ and let Π be a proof of unsatisfiability for $\bigwedge \Gamma$. For every $1 \leq i < n$ let us define $A(i) = A_1 \wedge \ldots \wedge A_i$ and $B(i) = A_{i+1} \wedge \ldots \wedge A_n$. Let $\mathcal{I}_i$ be the interpolant for the pair $(A(i), B(i))$ extracted using Π then the set $\{\top, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n-1}, \bot\}$ is an interpolant sequence for Γ.*

6. Exploiting Interpolation-Sequence in Model Checking

In this section we present a SAT-based algorithm for full verification (sometimes also called *unbounded model checking* (UMC)), which combines BMC and interpolation-sequence [38]. BMC is used to search for counterexamples while the interpolation-sequence is used to produce over-approximated sets of reachable states and to check for termination.

Interpolation-sequence has been introduced and used in [24] and [31]. In [24] it is used for computing an abstract model based on predicate abstraction for software model checking. In [31] interpolation-sequence is used for software model checking and lazy abstraction and is applied to individual execution paths in the control flow graph. The method presented in this section exploits interpolation-sequence in a different manner. In particular, it is applied to the whole model for imitating SMC.

From this point and on, we will use M to denote the Kripke structure representing the model and $f = AGq$ for a propositional formula q, as the property to be verified.

In order to better understand the algorithm and the motivation behind it, we first review some basic concepts of symbolic model checking (SMC).

6.1. Revisiting Symbolic Reachability Analysis

SMC performs *forward reachability analysis* by computing sets of reachable states S_j where j is the number of transitions needed to reach a state in S_j when starting from the initial states. Further, for every $j \geq 1$, $S_j(V) \wedge TR(V, V') \equiv S_{j+1}(V')$. Once S_j is computed, if it contains states violating q, a counterexample of length i is found and returned. Otherwise, if $S_j \subseteq \bigcup_{i=1}^{j-1} S_i$ then a *fixpoint* has been reached, meaning that all reachable states have been found already. If no reachable state violates the property then the algorithm concludes that $M \models f$.

6.2. Interpolation-Sequence Based Model Checking (ISB)

The method presented in this section demonstrates how over-approximated sets, similar to S_i in their characteristics, can be extracted from BMC, based on interpolation-sequences.

Informally, we will use the notion of *fixpoint* when we can conclude that all *reachable* states in the model have been visited[1].

Note that, the interpolation-sequence exists for a bound N only when there is no counterexample of length N. In case a counterexample exists, BMC returns a counterexample and the interpolation-sequence is not needed.

Definition 6.1. A *BMC-partitioning* for φ^N is the set $\Gamma = \{A_1, A_2, \ldots, A_{N+1}\}$ of formulas such that $A_1 = INIT(V^0) \wedge TR(V^0, V^1)$, for every $2 \leq i \leq N$ $A_i = TR(V^{i-1}, V^i)$ and $A_{N+1} = \neg q(V^N)$. Note that $\varphi^N = \bigwedge_{i=1}^{N+1} A_i$ $(= \bigwedge \Gamma)$.

For a bound N, consider a BMC formula φ^N and its BMC-partitioning Γ. In case φ^N is unsatisfiable, its interpolation-sequence is denoted by $\bar{I}^N = (I_0^N, I_1^N, \ldots, I_{N+1}^N)$. Note that the BMC-partitioning for φ^N contains $N+1$ elements and therefore the interpolation-sequence contains $N+2$ elements where the first element and the last one are always $\top$ and $\bot$, respectively.

Next, we intuitively explain our method. We start with $N = 1$. Consider the formula φ^1 and its BMC-partitioning: $\{A_1, A_2\}$. In case φ^1 is unsatisfiable, there exists an interpolation-sequence of the form $\bar{I}^1 = (I_0^1 = \top, I_1^1, I_2^1 = \bot)$. By Definition 5.2, $S_1 \subseteq I_1^1$, where S_1 is the set of states reachable from the initial states in one transition. This is because $\top \wedge A_1 \Rightarrow I_1^1$ where $A_1 = INIT(V^0) \wedge TR(V^0, V^1)$. Also, $I_1^1 \wedge \neg q(V^1)$ is unsatisfiable, since $I_1^1 \wedge A_2 \Rightarrow \bot$, where $A_2 = \neg q(V^1)$. Therefore, $I_1^1 \models q$.

In the next BMC iteration, for $N = 2$, consider φ^2 and its BMC-partitioning $\{A_1, A_2, A_3\}$. In case φ^2 is unsatisfiable, we get $\bar{I}^2 = (\top, I_1^2, I_2^2, \bot)$. Here too, $S_1 \subseteq I_1^2$ and the states reachable from it in one transition are a subset of I_2^2 since $I_1^2 \wedge A_2 \Rightarrow I_2^2$. Also, $S_2 \subseteq I_2^2$ and $I_2^2 \models q$. Let us define the sets $I_1 = I_1^1 \wedge I_1^2$ and $I_2 = I_2^2$. These sets have the following properties, $S_1 \subseteq I_1$, $S_2 \subseteq I_2$, $I_1 \models q$ and $I_2 \models q$. Moreover, $I_1[V^1 \leftarrow V] \wedge TR(V, V') \Rightarrow I_2[V^2 \leftarrow V']$.

In the general case if φ^N is unsatisfiable then for every $1 \leq j \leq N$, $S_j \subseteq I_j^N$. If we now define $I_j = \bigwedge_{k=j}^{N} I_j^k$ then for every $1 \leq j \leq N$ we get:

- $I_j \models q$ since $I_j^j \models q$.
- $I_j \wedge TR(V, V') \Rightarrow I_{j+1}$ since $I_j^k \wedge TR(V^j, V^{j+1}) \Rightarrow I_{j+1}^k$ for every $1 \leq k \leq N$
- $S_j \subseteq I_j$ since $S_j \subseteq I_j^k$ for every $1 \leq k \leq N$.

As a result, the sets $I_1, I_2, \ldots, I_N$ can be used to determine if $M \models f$. Intuitively, the sets I_j are similar to the sets S_j computed by SMC except that they are over-approximations of S_j. Therefore, these sets can be used to imitate the forward reachability analysis of the model's state-space by means of an over-approximation. This is done in the following manner. BMC runs as usual with one extension. After checking bound N, if a counterexample is found, the algorithm terminates. Otherwise, the interpolation-sequence $\bar{I}^N$ is extracted and the sets I_j

[1]Since we compute over-approximated sets of reachable states, the computed sets are not monotonic. Therefore, we cannot define a monotonic function g for which the existence of a fixpoint is guaranteed.

for $1 \leq j \leq N$ are updated. If $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ for some $1 \leq j \leq N$, then we conclude that a fixpoint has been reached and all reachable states have been visited. Thus, $M \models f$. If no fixpoint is found, the bound N is increased and the computation is repeated for $N+1$.

Next, we explain why the ISB algorithm uses $I_j = \bigwedge_{k=j}^{N} I_j^k$ rather than I_j^N in its Nth iteration. Informally, the following facts are needed in order to guarantee the correctness of the algorithm. For every $1 \leq j \leq N$ we need:

1. I_j should satisfy q.
2. $I_j(V) \wedge TR(V, V') \Rightarrow I_{j+1}(V')$ for $j \neq N$.
3. $S_j \subseteq I_j$.

This means that the algorithm cannot be implemented using the extracted interpolation sequence $\bar{I}^N$ alone. This is because $\bar{I}^N$ does not satisfy condition (1): while $I_N^N \models q$, I_j^N for $j \neq N$, does not necessarily satisfy q. This can be remedied by conjuncting each I_j^N with I_j^j. However, now condition (2) no longer holds. Taking $I_j = \bigwedge_{k=j}^{N} I_j^k$ results in a set with all three properties.

Definition 6.2. If no counterexample of length N or less exists in M, then $I_j = \bigwedge_{k=j}^{N} I_j^k[V^j \leftarrow V]$ where I_j^k is the j-th element in the interpolation-sequence extracted for the BMC-partitioning of φ^k. The *reachability vector* is defined to be $\bar{I} = (I_1, I_2, \ldots, I_N)$.

The algorithms for updating the reachablility vector and checking for a fixpoint are described in Figure 3 and Figure 4, respectively. The complete model checking algorithm using the method described above is given in Figure 5.

It is important to note that a call to UpdateReachability changes all elements of the reachability vector. Therefore, the function FixpointReached cannot count on inclusion checks done in previous iterations and needs to search for a fixpoint at every point in $\bar{I}$. Moreover, it is not sufficient to check for inclusion of only the last element I_N of $\bar{I}$. Indeed, if for any $j \leq N$, $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ then all reachable states have been found already. However, the implication $I_N \Rightarrow \bigvee_{i=1}^{N-1} I_i$ might not hold due to additional *unreachable* states in I_N. This is because for all $1 \leq j < N$, I_{j+1} is an over-approximation of the states reachable from I_j and not the exact image (That is, $I_j \wedge TR(V, V') \Rightarrow I_{j+1}[V \leftarrow V']$ rather than $I_j \wedge TR(V, V') \equiv I_{j+1}[V \leftarrow V']$).

6.3. Correctness of the ISB algorithm

The following lemmas and definition formalize the above explanation and prove the correctness of the algorithm.

```
1: function UpdateReachable(Ī,Ī^k)
2:     j = 1
3:     while (j < k) do
4:         I_j = I_j ∧ I_j^k
5:         Ī[j] = I_j
6:         j = j + 1
7:     end while
8:     Ī[k] = I_k^k
9: end function
```

Figure 3. Updating the reachability vector

```
1: function FixpointReached(Ī)
2:     j = 2
3:     while (j ≤ Ī.length) do
4:         R = ⋁_{k=1}^{j−1} I_k
5:         φ = I_j ∧ ¬R // Negation of I_j ⇒ R
6:         if (SAT(φ) == false) then return true
7:         end if
8:         j = j + 1
9:     end while
10:    return false
11: end function
```

Figure 4. Checking if a fixpoint has been reached

Lemma 6.3. *If M does not have a counterexample of length N, then $S_j \subseteq I_j^N[V^j \leftarrow V]$ for every $1 \leq j \leq N$ and $I_N^N \models q$.*

Proof. M does not have a counterexample of length N. Therefore, the formula φ^N is unsatisfiable. Let $\bar{I}^N$ be the interpolation-sequence for the BMC-partitioning of φ^N. By Definitions 5.2 and 6.1, for $j = 1$, $\top \wedge \mathit{INIT}(V^0) \wedge \mathit{TR}(V^0, V^1) \Rightarrow I_1^1$. For each $2 \leq j \leq N$, $I_j^N \wedge \mathit{TR}(V^j, V^{j+1}) \Rightarrow I_{j+1}^N$. Hence, $S_j \subseteq I_j^N$. Definition 5.2 also state that $I_N^N \wedge \neg q(V^N) \Rightarrow \bot$ and therefore $I_N^N \models q$. □

Lemma 6.4. *If M does not have a counterexample of length N or less, then $S_j \subseteq I_j$ and $I_j \models q$ for every $1 \leq j \leq N$.*

Proof. For every $j \leq k \leq N$ by Lemma 6.3 $S_j \subseteq I_j^k$ and $I_j^j \models q$. Since I_j is the conjunction of I_j^k for every $j \leq k \leq N$, $S_j \subseteq I_j$ and $I_j \models q$. □

Lemma 6.5. *Let $\bar{I} = (I_1, I_2, \ldots, I_N)$ be the reachability vector. For every $1 \leq j < N$, $I_j \wedge \mathrm{TR}(V, V') \Rightarrow I_{j+1}[V \leftarrow V']$.*

Proof. Definition 6.2 and 5.2 imply that $I_j = \bigwedge_{k=j}^{N} I_j^k[V^j \leftarrow V]$ and that for every $j \leq k \leq N$, $I_{j-1}^k \wedge \mathit{TR}(V^{j-1}, V^j) \Rightarrow I_j^k$. We get $I_j \wedge \mathit{TR}(V, V') \Rightarrow I_{j+1}[V \leftarrow V']$. □

```
1: function ISB(M,f)
2:     k := 0
3:     result = BMC(M, f, 0)
4:     if (result == cex) then
5:         return cex
6:     end if
7:     Ī = ∅        // the reachability vector
8:     while (true) do
9:         k = k + 1
10:        result = BMC(M, f, k)
11:        if (result == cex) then
12:            return cex
13:        end if
14:        Ī^k = (⊤, I_1^k, ..., I_k^k, ⊥)
15:        UpdateReachable(Ī,Ī^k)
16:        if (FixpointReached(Ī) == true) then
17:            return true
18:        end if
19:    end while
20: end function
```

Figure 5. The ISB Algorithm

Theorem 6.6. *Assume there is no path of length N or less violating f in M. If there exist $1 < j \leq N$ such that $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$, then $M \models f$.*

Proof. By assumption, there is no path in M of length N or less that violates f. We now show that given $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ we can conclude that there is no path of any length violating f. Let $R = \bigvee_{i=1}^{j-1} I_i$. By assumption, $I_j \Rightarrow R$ and by Lemma 6.5, for every $1 \leq i < j$, $I_i \wedge TR(V^i, V^{i+1}) \Rightarrow I_{i+1}$. Thus, $R(V) \wedge TR(V, V') \Rightarrow R(V')$ (1). Moreover, for every $1 \leq i \leq j$ the formula $I_i \wedge \neg q$ is unsatisfiable (since $I_i \models q$ by Lemma 6.4). Hence, $R \wedge \neg q$ is unsatisfiable (2).

We can show by induction that all reachable states are in $R^* = R \vee INIT$. The base case handles initial states. This holds trivially by the definition of R^*. Now let us assume it holds for all states reachable in k steps. It should be proved for states reachable in $k+1$ steps. Let s_{k+1} be a state reachable in $k+1$ steps from an initial state. Let $\pi = s_0, s_1, \ldots, s_k, s_{k+1}$ be an initial path to s_{k+1}. By the induction hypothesis $s_k \in R^*$. From (1) we know that $R[V \leftarrow V^k] \wedge TR(V^k, V^{k+1}) \Rightarrow R[V \leftarrow V^{k+1}]$. Therefore, $s_{k+1} \in R^*$.

By assumption, $INIT \models q$ since there is no path of length N or less violating f. By that and (2), $R^* \models q$. Thus, the set of reachable states satisfy q which implies that $M \models f$.

□

Lemma 6.7. *Suppose* $M \models f$ *then there exists a bound* N *such that* $\bar{I} = (I_1, I_2, \ldots, I_N)$ *and there exists an index* $1 < j \leq N$ *such that* $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$.

Proof. The set of states S is finite. Let us define $N = j = |S| + 1$. $M \models f$ hence for every $0 \leq k \leq N$, φ^k is unsatisfiable. Thus, the interpolation-sequence $\bar{I}^k$ exists for every $0 \leq k \leq N$ and by that the reachability vector $\bar{I} = (I_1, I_2, \ldots, I_N)$ exists. Since $|S| < \infty$ we get $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$. □

Theorem 6.8. *There exists a path* π *of length* N *such that* π *violates* f *if and only if ISB terminates and returns cex.*

Proof. Assume that the minimal violating path is of length N. For $N-1$ there is no path in M violating f. By Theorem 6.6 we get that for every j such that $1 \leq j < N$, $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ does not hold. Therefore, the algorithm cannot terminate by returning *true* in the first $N-1$ iterations. When the algorithm reaches the N-th iteration, $BMC(M, f, N)$ will return a counterexample and the algorithm terminates. The other direction is immediate. □

Theorem 6.9. *For every model* M *and a property* $f = AGq$ *there exists* N *such that ISB terminates.*

Proof. If $M \models f$ it follows by Lemma 6.7 that the algorithm terminates and returns *true*. If there is a path in M that violates f, it follows by Theorem 6.8 that the algorithm terminates and returns *cex*. □

7. Interpolation Based Model Checking (IB)

In [30], *interpolation* has been suggested for the first time in order to obtain a SAT-based model checking algorithm for full verification.

The algorithm combines BMC and Craig's Interpolation [17]. Similarly to the ISB algorithm presented in the previous section, the interpolant is used to compute an over-approximation of the set of reachable states. However, the computation is done differently. As before, the algorithm concludes that the property holds when a fixpoint is reached during the computation of the reachable states and none of the computed state violates the property.

The following definition is useful in explaining the interpolation based algorithm. Recall that the verified property is of the form $f = AGq$.

Definition 7.1. For a set of states T, T is a S_j*-approximation* w.r.t N, where $1 \leq j \leq N$, if the following two conditions hold: $S_j \subseteq T$ and there is no path of length $(N-j)$ or less violating q, starting from a state $s \in T$. We write $S_j \preceq_N T$ to denote that T is a S_j-approximation w.r.t N.

The formula φ^k is used in BMC to represent a counterexample of length exactly k. This formula can be modified to represent a counterexample of length

function CHECKREACHABLE(M,f,k)
  $R = M.INIT$ // Initialize R - initial states of M
  if $(BMC(M, f, 1, k) == cex)$ **then**
    return cex
  end if
  $M' = M$
  repeat
    $A = J(V^0) \wedge TR(V^0, V^1)$
    $B = TR(V^1, V^2) \wedge \ldots \wedge TR(V^{k-1}, V^k) \wedge (\bigvee_{j=1}^{k} \neg q(V^j))$
    $J = SAT.getInterpolant(A, B)$
    if $J \subseteq R$ **then**
      return $fixpoint$
    end if
    $R = R \cup J$
    $M'.INIT = J$
  until $(BMC(M', f, 1, k) == cex)$
  return $abort$
end function

Figure 6. Computing reachable states using interpolation and BMC with a specific bound k

l for $1 \leq l \leq k$. We denote this formula by $\varphi^{1,k}$ and write $BMC(M, f, 1, k)$ when BMC runs on $\varphi^{1,k}$.

Formula 2. $\varphi^{1,k} = INIT(V^0) \wedge TR(V^0, V^1) \wedge TR(V^1, V^2) \wedge \ldots \wedge TR(V^{k-1}, V^k) \wedge (\bigvee_{j=1}^{k} \neg q(V^j))$

Consider the following partitioning for $\varphi^{1,k}$:

- $A = INIT(V^0) \wedge TR(V^0, V^1)$
- $B = \bigwedge_{i=1}^{k-1} TR(V^i, V^{i+1}) \wedge (\bigvee_{j=1}^{k} \neg q(V^j))$.

Clearly $\varphi^{1,k} \equiv A \wedge B$. Assume that $\varphi^{1,k}$ is unsatisfiable. By the interpolation theorem [17], there exists an interpolant J_1^k which, by Definition 5.1, has the following properties:

- J_1^k is defined over the variables of $\mathcal{L}(A) \cap \mathcal{L}(B)$, namely, V^1.
- $A \Rightarrow J_1^k$. Hence, $S_1 \subseteq J_1^k$.
- $J_1^k(V_1) \wedge B$ is unsatisfiable. This means that there is no path of length $k-1$ or less, starting from J_1^k, which violates q.

By the above we get that $S_1 \preceq_k J_1^k$. We can now proceed by replacing the initial states of M with the computed interpolant J_1^k. BMC is reinvoked with the same bound k and with the modified model $M' = (S, J_1^k[V^1 \leftarrow V], TR, L)$ in which the initial states are J_1^k. A new interpolant J_2^k is then extracted. J_2^k satisfies $S_2 \preceq_{k+1} J_2^k$. It is important to notice that J_1^k now satisfies $S_1 \preceq_{k+1} J_1^k$

since the BMC run on M' did not find a counterexample of length k starting from a state in J_1^k. In the general case we replace $INIT$ with J_i^k and get J_{i+1}^k.

Figure 6 presents, for a given bound k, the computation of an over-approximated set of reachable states. Note that after L iterations of the main loop in CheckReachable we get L interpolants and for every $1 \leq i \leq L$, $S_i \preceq_{k+L} J_i^k$. All computed states are collected in R. If at any iteration, the interpolant J is contained in R, then all reachable states have been found with no violation of f. CheckReachable then returns "$fixpoint$".

On the other hand, if a counterexample is found on a modified model, then CheckReachable(M,f,k) is aborted and CheckReachable(M,f,$k+1$) is initiated. Recall that the counterexample has been obtained on an over-approximated set of states and therefore might not represent a real counterexample in the original model. In case a real counterexample exists, it will be found during a BMC run on the original model M for a larger bound.

In [37], an optimization for CheckReachable is suggested. If the current bound is k and at the L-th iteration a counterexample is found, then CheckReachable is reinvoked with bound $k+L$ (rather than $k+1$). This is possible since M is known not to have a counterexample of length $k+L-1$. The usefulness of this heuristic highly depends on the type of property that is checked. On the one hand, if the property is false, this heuristic indeed results in a better performance. On the other hand, for true properties, this approach may hurt performance since a fixpoint could have been found at a lower bound than $k+L$ (e.g. $k+1$).

8. Comparing Interpolation-Sequence Based MC to Interpolation Based MC

In the previous sections we presented two model checking algorithms which combine BMC and interpolation: the Interpolation-Sequence Based (*ISB*) [38] and the Interpolation Based (IB) [30]. In this section we analyze the differences between the algorithms. In the next section we compare them experimentally.

Both methods compute an over-approximation of the set of reachable states. However, their state traversal is different. As a result, none is better than the other in all cases. In specific cases, though, one may converge faster.

Several technical details distinguish between ISB and IB. First, the formulas from which the interpolants are extracted are different. For a given bound N, ISB uses the formula φ^N while IB uses $\varphi^{1,N}$.

Second, the approximated sets are computed in different manners. ISB computes the sets I_j incrementally and refines them after each iteration of BMC, as part of the BMC loop. IB, on the other hand, recomputes the interpolants whenever the bound is incrememnted (that is, whenever CheckReachable is called with a greater bound).

Third, ISB can be viewed as an addition to the BMC loop. At each application of BMC (with a different bound), the addition includes the extraction of an interpolation-sequence and the check if a fixpoint has been reached. Indeed, after N iterations of the BMC loop in ISB, there are N over-approximated sets of states, $I_1, \ldots, I_N$ satisfying, for each $1 \leq j \leq N$, $S_j \preceq_N I_j$.

SMC	ISB	IB
$\{S_1, \ldots, S_N\}$	$\{I_1, I_2, \ldots, I_N\}$ $S_i \preceq_N I_i$ After checking bounds 1 to N	$\{J_1^1, J_2^1, \ldots, J_N^1\}$ $S_i \preceq_N J_i^1$ N iterations at bound 1, if possible
$\{S_1, \ldots, S_{N+L}\}$	$\{I_1, \ldots, I_L, \ldots, I_{N+L}\}$ $S_i \preceq_{N+L} I_i$ After checking bounds 1 to $N+L$	$\{J_1^N, J_2^N, \ldots, J_L^N\}$ $S_i \preceq_{N+L} J_i^N$, $(1 \leq i \leq L)$ L iterations at bound N, if possible

Table 1. The correlation between the interpolants computed by ISB and IB to the sets computed by SMC

On the other hand, IB consists of two nested loops. The outer loop increments the bounds while the inner loop computes over-approximated sets of reachable states. If the outer loop is at some bound $N > 1$ and the inner loop performs L iterations then there are L sets of states $J_1^N, \ldots, J_L^N$, each satisfying $S_i \preceq_{N+L} J_i^N$ $(1 \leq i \leq L)$. Table 1 summarizes the above differences.

In summary, IB can compute, at a given bound N, as many sets as needed as long as no counterexample is found (not necessarily a real counterexample). On the other hand, for bound N, ISB can only compute N sets. However, it does not need recurrent BMC calls for each bound (only one is needed). Thus, we can conclude that in cases IB can compute all the needed sets at a low bound it performs better than ISB. However, for examples where the needed sets can only be computed using higher bounds, ISB has an advantage. This fact is reflected in the experimental results.

As mentioned before, when a counterexample exists the over-approximated sets of reachable states are not needed. If a property is violated then there exists a minimal bound N for which a violating path of length N exists. Both algorithms have to reach this bound in order to find the counterexample. Here, ISB has a clear advantage over IB. This is because after each BMC run on the original model, IB executes at least one additional BMC run on a modified model. Thus, IB invokes at least two BMC runs for each bound from 1 to $N-1$. Clearly, the second BMC run is more demanding than the inclusion check performed by ISB. In all our experiments, this kind of properties always favored ISB.

9. Implementation Details and Experimental Results

9.1. Implementation Details

Both the ISB and the IB algorithms were implemented within Intel's verification system using a SAT-based model checker which is based on Intel's in-house SAT solver *Eureka*. The interpolants are represented by a data-structure similar to an And-Inverter Graph (AIG) and are simplified and optimized using known methods such as constant propagation and sharing of redundant expressions.

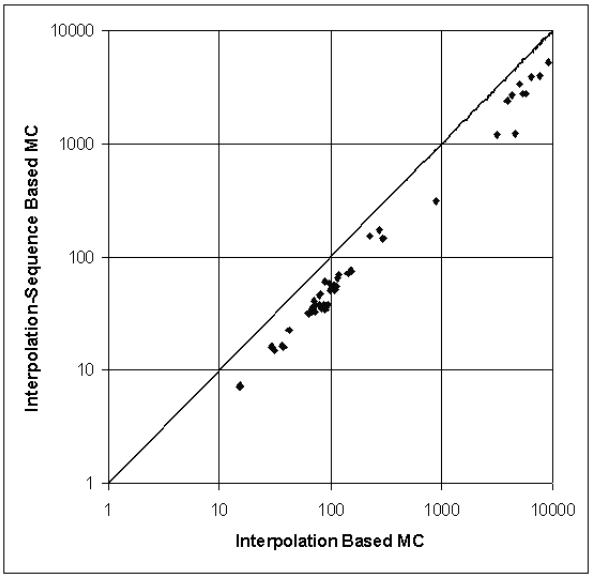

Figure 7. Runtime of falsified properties on Intel's recent micro-architecture.

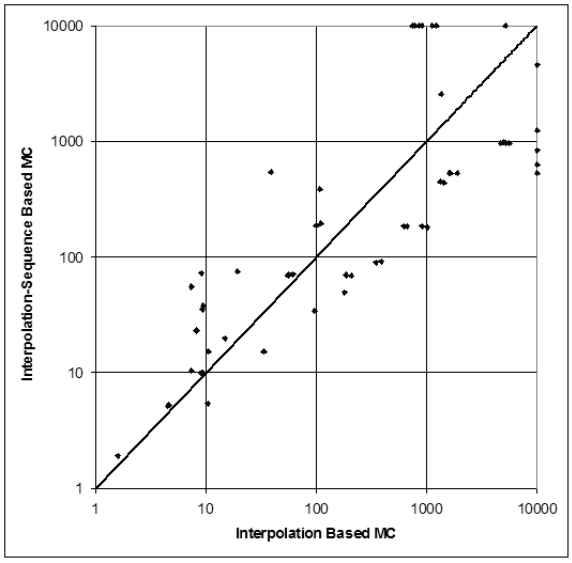

Figure 8. Runtime of verified properties on Intel's recent micro-architecture.

9.2. Experimental Results

The two algorithms have been checked on various models taken from two of Intel's recent CPU designs. The characteristics of the checked models appear in Table 3. The 136 properties chosen for the experiments were all real safety properties used to verify the correctness of the designs. The cone of influence for the properties

Name	♯Vars	B	B_{IB}	♯I	♯I_{IB}	♯BMC	♯BMC$_{IB}$	Time [s]	Time$_{IB}$ [s]
f_1	3406	16	15	136	80	16	80	970	5518
f_2	1753	9	8	45	40	9	40	91	388
f_3	1753	7	6	28	28	7	28	49	179
f_4	1753	16	15	136	94	16	94	473	1901
f_5	3406	6	5	21	13	6	13	68	208
f_6	1761	2	1	3	2	2	2	5	4
f_7	3972	3	1	6	3	3	3	19	14
f_8	2197	3	1	6	3	3	3	10	7
f_9	1629	23	6	276	39	23	39	2544	1340
f_{10}	4894	5	1	15	3	5	3	635	101

Table 2. Verified properties and their running parameters.
Unindexed columns refer to the ISB algorithm; columns indexed with IB refer to the IB algorithm. ♯*Vars* stands for the number of state variables in the cone of influence. B - bound at convergence, ♯I - number of interpolants computed, ♯BMC - number of calls to the BMC algorithm, and *Time[s]* - the runtime in seconds.

contains thousands of state variables and tens of thousands of gates and signals. The properties vary in that some are *true* and some are *false*. During all checks, a timeout of 10,000 seconds has been set. Experiments were conducted on systems with a dual core Xeon 5160 processors (Core 2 micro-architecture) running at 3.0GHz (4MB L2 cache) with 32GB of main memory. Operating system running on the system is Linux SUSE.

Figure 7 and Figure 8 show the runtime in seconds for the two algorithms. Each point represents a property from the set of chosen properties. The X axis represents runtime for IB while the Y axis represents the runtime using ISB. We can see that the results vary. Figure 7 shows the runtime for the falsified properties. Figure 8 shows the runtime for the verified properties. All falsified properties (total of 67) favor ISB. There are five properties that can be verified by ISB and not by IB (due to timeout) and two properties that can be falsified using ISB while cannot be falsified using IB. On the other hand, there are seven properties that cannot be verified by ISB but can be verified by IB. The rest of the properties (57 total) are all verified by both algorithms.

A more accurate analysis of the algorithms is shown in Table 2 that presents running parameters (number of state variables in the cone of influence, bound at convergence, number of interpolants computed, number of calls to BMC and runtime) on various properties for both IB and ISB. For some cases, even though IB converges at a lower bound, and computes less interpolants than ISB, ISB still converges faster by means of runtime. This is due to the fact that BMC calls are computationally heavier than the extraction of the interpolants. Since IB issues more calls to BMC than ISB in these cases, the influence on its runtime is noticeable. Through all our experiments, when convergence for IB could be achieved only at high bounds, ISB always performed better while for convergence at lower bounds, IB performs better. This result is supported by the analysis presented in the previous section.

Name	♯ Latches	♯ Inputs	♯ Gates
M_1	3611	3	84570
M_2	4968	2079	133255
M_3	12806	402	89392
M_4	1672	459	11195
M_5	19213	305	146717

Table 3. Models used for testing

10. Conclusion

We presented two methods which use interpolation for SAT-based unbounded model checking. The experiments show a clear advantage of ISB when the properties are falsified. In case of verified properties, the results vary: some favor ISB while others favor IB.

Further investigation in needed in order to obtain a better understanding of the difference between the methods and to characterize the type of properties (when the properties are true) suitable for each of the methods.

Acknowledgment

The author would like to thank Yael Meller and Yakir Vizel for careful reading and useful comments.

References

[1] T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *In Proceedings of the Workshop on Advances in VErification (WAVE)*, July 2000.

[2] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In *proceedings of the 33rd Design Automation Conference (DAC'96)*, pages 655–660. IEEE Computer Society Press, June 1996.

[3] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with Blast. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 2–18, 2005.

[4] A. Biere and C. Artho. Liveness checking as safety checking. In *FMICS02*.

[5] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *proceedings of the 36rd Design Automation Conference (DAC'99)*. IEEE Computer Society Press, June 1999.

[6] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*, volume 58 of *Advances in Computers*. Elsevier, 2003.

[7] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking Without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS. Springer.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on Computers*, C-35(8):677–691, 1986.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–171, June 1992. Special Issue: Selections from 1990 IEEE Symposium on Logic in Computer Science.

[11] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[12] P. Chauhan, E. M. Clarke, and D. Kroening. Using SAT based image computation for reachability analysis. Technical Report CMU-CS-03-151, Carnegie Mellon University, School of Computer Science, 2003.

[13] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[14] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[15] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, December 1999.

[16] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[17] W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[18] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[19] R. Fraer, G. Kamhi, Z. Barukh, M.Y. Vardi, and L. Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In *12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, Chicago, USA, July 2000.

[20] O. Grumberg, A. Schuster, and A. Yadgar. Reachability Using a Memory-Efficient All-Solutions SAT Solver. In *Fifth Inernation Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, November 2004.

[21] Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.–Feb. 1990.

[22] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[23] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editors, 1991.

[24] R. Jhala and K.L. McMillan. Interpolant-Based Transition Relation Approximation. In *17th International Conference on Computer Aided Verification (CAV'05)*, LNCS 3576, Edinburgh, July 2005.

[25] O. Kupferman and M.Y. Vardi. Model checking of safety properties. In *Computer-Aided Verification (CAV'99)*, Lecture Notes in Computer Science. Springer-Verlag.

[26] J.P. Marques-Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[27] K. McMillan. Applications of craig interpolation to model checking. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 1–12, Edinburgh, Scotland, April 2005. Springer.

[28] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[29] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification*, 2002.

[30] K.L. McMillan. Interpolation and SAT-based Model Checking. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, Bolder, Colorado, 2003.

[31] K.L. McMillan. Lazy Abstraction with Interpolants. In *18th International Conference on Computer Aided Verification (CAV'06)*, LNCS 4144, Seattle, August 2006.

[32] K.L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*,

volume 2619 of *LNCS*, pages 331–346, Warsaw, Poland, April 2003.

[33] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *39th Design Aotomation Conference (DAC'01)*, 2001.

[34] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science (FOCS'77)*, 1977.

[35] M. Sheeran, S. Singh, and G. Staalmarck. Checking safety properties using induction and a SAT-solver. In *Third International Conference on Formal methods in Computer-Aided Design (FMCAD'00)*, Austin, Texas, November 2000.

[36] M. Sheeran and G. Staalmarck. A tutorial on stalmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1), January 2000.

[37] João P. Marques Silva. Improvements to the implementation of interpolant-based model checking. In *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference (CHARME'05)*, volume 3725 of *Lecture Notes in Computer Science*, pages 367–370, Saarbrücken, Germany, 2005. Springer.

[38] Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *Ninth International Conference on Formal methods in Computer-Aided Design (FMCAD'09)*, pages 1–8, Austin, Texas, 2009.

[39] Lintao Zhang and Sharad Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10880–10885. IEEE Computer Society, 2003.