# SAT-Based Model Checking without Unrolling

Aaron R. Bradley

Dept. of Electrical, Computer & Energy Engineering
University of Colorado at Boulder
Boulder, CO 80309
bradleya@colorado.edu

**Abstract.** A new form of SAT-based symbolic model checking is described. Instead of unrolling the transition relation, it incrementally generates clauses that are inductive relative to (and augment) stepwise approximate reachability information. In this way, the algorithm gradually refines the property, eventually producing either an inductive strengthening of the property or a counterexample trace. Our experimental studies show that induction is a powerful tool for generalizing the unreachability of given error states: it can refine away many states at once, and it is effective at focusing the proof search on aspects of the transition system relevant to the property. Furthermore, the incremental structure of the algorithm lends itself to a parallel implementation.

## 1 Introduction

Modern SAT-based model checkers unroll the transition relation and thus present the SAT solver with large problems [2,24,21,23]. We describe a new SAT-based model checking algorithm that does not unroll the transition relation, that is nevertheless complete, that is competitive with the best available model checkers [3], and that can be implemented to take advantage of parallel computing environments. The fundamental idea is to generate clauses that are inductive relative to stepwise reachability information.

When humans analyze systems, they produce a set of lemmas — typically inductive properties — that together imply the desired property. Each lemma holds *relative to* some subset of previously proved lemmas in that this prior knowledge is invoked in proving the new lemma [19]. A given lemma usually focuses on just one aspect of the system. Typically, early lemmas discuss variable domains, properties about module-local data structures, and so on, while later lemmas address more global aspects of the system. Yet even the later lemmas are fairly easy to prove since prior information heavily constrains the state space. When a lemma is too difficult to prove directly, a skilled human verifier typically searches for additional supporting lemmas.

In contrast, standard model checkers employ monolithic strategies. Many iteratively compute pre- or post-images precisely [7,20] or approximately [21]; others unroll the transition relation [2,24,23]. Section 5 provides further discussion of related work.

R. Jhala and D. Schmidt (Eds.): VMCAI 2011, LNCS 6538, pp. 70–87, 2011.

This paper describes a model checking algorithm for safety properties whose strategy is not monolithic but rather closer to that of a human, albeit a particularly industrious one. It produces lemmas in the form of clauses that are inductive *relative to* previous lemmas and stepwise assumptions. At convergence, a subset of the generated lemmas comprise a 1-step inductive strengthening of the given property. Humans construct lemmas of a more general form than clauses, and in this respect, the analogy breaks down; but the incremental construction of lemmas to form a proof is similar. Section 3 covers induction in further depth, reviews previous work on generating inductive clauses [6], and motivates the use of stepwise assumptions. Sections 4 and 6 discuss the algorithm in detail.

The implementation of the algorithm, `ic3` ("**I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness"), has certain runtime characteristics that support the comparison to a human strategy (see Table 1 of Section 7). A typical run of `ic3` on a nontrivial problem executes many tens of thousands of SAT queries that test whether various formulas are 1-step inductive, and it produces hundreds or thousands of intermediate lemmas. Each SAT query is trivial compared to the queries made in other techniques [20,2,24,23]: `ic3` successfully employs ZChaff [22], a decidedly non-state-of-the-art solver, yet one that provides efficient incremental support. While a human would likely produce fewer but higher-level lemmas, the overall pattern of applying relatively low reasoning power to produce many lemmas is familiar to the experienced human verifier. In HWMCC'10, `ic3` ranked third, placing it among sophisticated, multi-engine model checkers. It is available for download at the author's website [3].

The human verifier analogy can be taken one step further. Just as large verification problems benefit from the attention of several humans, they also benefit from the attention of multiple cooperating `ic3` instances. In particular, because the *relative to* relation among lemmas is a partial order and typically not a linear order, `ic3` can take advantage of multi-core and distributed computing environments. Each instance shares only its lemmas and not the work that went into producing them, and it incorporates the lemmas that the other processes generate. Section 8 demonstrates empirically that this parallel implementation is effective and that, in particular, processes do not duplicate work too frequently.

## 2 Definitions

A *finite-state transition system* $S : (\bar{i}, \bar{x}, I, T)$ is described by a pair of propositional logic formulas: an initial condition $I(\bar{x})$ and a transition relation $T(\bar{i}, \bar{x}, \bar{x}')$ over a set of input variables $\bar{i}$, internal state variables $\bar{x}$, and the next-state primed forms $\bar{x}'$ of the internal variables [9]. Applying prime to a formula, $F'$, is the same as priming all of its variables.

A state of the system is an assignment of Boolean values to all variables $\bar{x}$ and is described by a *cube* over $\bar{x}$, which, generally, is a conjunction of literals, each *literal* a variable or its negation. An assignment $s$ to all variables of a formula $F$ either satisfies the formula, $s \models F$, or falsifies it, $s \not\models F$. If $s$ is interpreted as a state and $s \models F$, we say that $s$ is an *$F$-state*. A formula $F$ *implies* another formula $G$, written $F \Rightarrow G$, if every satisfying assignment of $F$ satisfies $G$.

A *clause* is a disjunction of literals. A subclause $d \subseteq c$ is a clause $d$ whose literals are a subset of $c$'s literals.

A *trace* $s_0, s_1, s_2, \ldots$, which may be finite or infinite in length, of a transition system $S$ is a sequence of states such that $s_0 \models I$ and for each adjacent pair $(s_i, s_{i+1})$ in the sequence, $s_i, s'_{i+1} \models T$. That is, a trace is the sequence of assignments in an execution of the transition system. A state that appears in some trace of the system is *reachable*.

A safety property $P(\bar{x})$ asserts that only $P$-states are reachable. $P$ is *invariant* for the system $S$ (that is, $S$-invariant) if indeed only $P$-states are reachable. If $P$ is not invariant, then there exists a finite *counterexample* trace $s_0, s_1, \ldots, s_k$ such that $s_k \not\models P$.

## 3 Applying Induction Incrementally

For a transition system $S : (\bar{i}, \bar{x}, I, T)$, an *inductive* assertion $F(\bar{x})$ describes a set of states that (1) includes all initial states: $I \Rightarrow F$, and that (2) is closed under the transition relation: $F \wedge T \Rightarrow F'$. The two conditions are sometimes called *initiation* and *consecution*, respectively. An inductive *strengthening* of a safety property $P$ is a formula $F$ such that $F \wedge P$ is inductive. Standard symbolic model checkers [7,20] and interpolation-based model checkers [21] compute inductive strengthenings of a given safety property $P$. Upon convergence, iterative post-image, respectively pre-image, computation yields the strongest, respectively weakest, inductive strengthening of $P$, while approximate methods yield strengthenings of intermediate strength.

Induction need not be applied in a monolithic way, however. One can construct a sequence of inductive assertions, each inductive *relative to* (a subset of) the previous assertions [19]. An assertion $F$ is inductive *relative to* another assertion $G$ if condition (1) holds unchanged: $I \Rightarrow F$, and a modified version of (2) holds: $G \wedge F \wedge T \Rightarrow F'$. The assertion $G$ reduces the set of states that must be considered so that an assertion $F$ that is not inductive on its own (because $F \wedge T \not\Rightarrow F'$) may be inductive relative to $G$.

The value of using induction in an incremental fashion is that constructing a sequence of simple lemmas is often easier than constructing a strengthening all at once. Of course, translating human intuition into a model checking algorithm is difficult if not impossible, so one typically fixes a domain of assertions [10].

In previous work, we introduced a technique for discovering relatively inductive clauses [6]. We review it here as motivation for the algorithm that we subsequently introduce. The main idea is to augment the following naive model checker: enumerate states that can reach a violation of the asserted property $P$ and conjoin their negations to $P$ until an inductive assertion is produced. For each such state $s$, the method searches for an *inductive generalization* $c \subseteq \neg s$ to conjoin to $P$ instead. Such a subclause (1) is inductive relative to known or assumed reachability information ($P$, previous inductive generalizations, and the negations of considered states without inductive generalizations) and (2) is minimal in that it does not contain any strict subclauses that are also inductive.

In practice, such a *minimal inductive subclause* is substantially smaller than $\neg s$ and excludes states that are not necessarily related to $s$ by $T$.

While the method succeeds on some hard benchmarks [4], it sometimes enters long searches for the next relatively inductive clause, for a state may not have an inductive generalization even if it is unreachable. It is this problem of search that motivated investigation into a more effective way to use inductive clause generation. The new approach de-emphasizes global information that is sometimes hard to discover in favor of stepwise information that is easy to discover. In particular, stepwise assumptions guarantee that an unreachable state always has a *stepwise-relative inductive generalization*, one that asserts that $s$ and many similar states are unreachable for some number of steps. Because we consider finite state systems, these stepwise-relative inductive generalizations eventually become truly inductive.

Since finding minimal inductive subclauses remains a core subprocedure in the new approach, an informal description of it is in order. To find a subclause $d \subseteq c = c_0$ that is inductive relative to $G$, if such a clause exists, first consider consecution: $G \wedge c_0 \wedge T \Rightarrow c_0'$. If both this implication and initiation hold, $c_0$ is itself inductive. Otherwise, a counterexample state $s$ exists. Form the clause $c_1 = c_0 \cap \neg s$ by keeping only the literals that $c_0$ and $\neg s$ share. Iterate this process until it converges to some clause $c_i$. If $c_i$ satisfies initiation, then let $d = c_i$; otherwise, $c_0$ does not have an inductive subclause. This process is called the down algorithm [6].

Now $d \subseteq c_0$ is inductive, but it is not necessarily minimal — and in practice it is large. Form $d_1$ by dropping some literal of $d$, and apply down to $d_1$. If down succeeds, the result is a smaller inductive subclause; if it fails, try again with a different literal. Continue until no literal can be dropped from the current inductive subclause. The result is a minimal inductive subclause of $c_0$. This process is called the MIC algorithm; it can be accelerated using the up algorithm [6]. Section 7 discusses optimizations to these procedures.

## 4 Informal Description

Consider a transition system $S : (\bar{i}, \bar{x}, I, T)$ and a safety property $P$. The algorithm decides whether $P$ is $S$-invariant, producing an inductive strengthening if so or a counterexample trace if not.

Let us first establish the core logical data structure. The algorithm incrementally refines and extends a sequence of formulas $F_0 = I, F_1, F_2, \ldots, F_k$ that are over-approximations of the sets of states reachable in at most $0, 1, 2, \ldots, k$ steps. While major iterations of the algorithm increase $k$, minor iterations can refine any $i$-step approximation $F_i$, $0 < i \leq k$. Each minor iteration conjoins one new clause to each of $F_0, \ldots, F_j$ for some $0 < j \leq k$, unless a counterexample is discovered. (Adding a clause to $F_0 = I$ is useless, but it simplifies the exposition.)

Assuming that any clause conjoined to $F_0, \ldots, F_j$ over-approximates $j$-step reachability, this simple description implies that the sequence always obeys the following properties: (1) $I \Rightarrow F_0$ and (2) $F_i \Rightarrow F_{i+1}$ for $0 \leq i < k$. Actually,

letting clauses($F_i$) be the set of clauses that comprise $F_i$, (2) can be more strongly expressed as (2') clauses($F_{i+1}$) $\subseteq$ clauses($F_i$) for $0 \leq i < k$. The algorithm guarantees two other relationships: (3) $F_i \Rightarrow P$ for $0 \leq i \leq k$, and (4) $F_i \wedge T \Rightarrow F'_{i+1}$ for $0 \leq i < k$. If ever clauses($F_i$) = clauses($F_{i+1}$), then these properties imply that $F_i$ is an inductive strengthening of $P$.

With this logical data structure and its intended invariants in mind, we now turn to the workings of the algorithm. Initially the satisfiability of $I \wedge \neg P$ and $I \wedge T \wedge \neg P'$ are checked to detect 0- and 1-step counterexamples. If none exist, $F_1$ is set to $P$.

Now let us suppose that we are in major iteration $k > 0$, so that sequence $F_0, F_1, \ldots, F_k$ satisfies properties (1)-(4). Is it the case that $F_k \wedge T \Rightarrow P'$?

Suppose so. Then the extended sequence $F_0, F_1, \ldots, F_k, P = F_{k+1}$ satisfies properties (1)-(4). We can move onto major iteration $k+1$. Additionally, for any clause $c \in F_i$, $0 \leq i \leq k$, if $F_i \wedge T \Rightarrow c'$ and $c \notin$ clauses($F_{i+1}$), then $c$ is conjoined to $F_{i+1}$. If during the process of propagating clauses forward it is discovered that clauses($F_i$) = clauses($F_{i+1}$)[1] for some $i$, the proof is complete: $P$ is invariant.

Now suppose not: $F_k \wedge T \not\Rightarrow P'$. There must exist an $F_k$-state $s$ that is one transition away from violating $P$. What is the maximum $F_i$ (that is, the weakest stepwise assumption), $0 \leq i \leq k$, such that $\neg s$ is inductive relative to it? If $\neg s$ is not even inductive relative to $F_0$, then $P$ is not invariant, for $s$ has an $I$-state predecessor. But if $P$ is invariant, then $\neg s$ must be inductive relative to some $F_i$.[2] We then apply inductive generalization to $s$: a minimal subclause $c \subseteq \neg s$ that is *inductive relative to* $F_i$ is extracted as described in Section 3. Because the inductive generalization is performed relative to $F_i$, this process must succeed. After all, $\neg s$ is itself inductive relative to $F_i$. The clause $c$ is conjoined to each of $F_0, \ldots, F_{i+1}$. (Why to $F_{i+1}$? Because $F_i \wedge c \wedge T \Rightarrow c'$ holds.)[3]

If $i = k-1$ or $i = k$, then $c$ was conjoined to $F_k$, eliminating $s$ as an $F_k$-state. Subsequent queries of $F_k \wedge T \Rightarrow P'$ must either indicate that the implication holds or produce different counterexample states than $s$. But it is possible that $i < k-1$. In this case, $s$ is still an $F_k$-state.

Consider this question: Why is $\neg s$ inductive relative to $F_i$ but not relative to $F_{i+1}$? There must be a predecessor, $t$, of $s$ that is an $F_{i+1}$-state but not an $F_i$-state. Now if $i = 0$, $t$ may have an $I$-state as a predecessor, in which case $P$ would not be invariant. But if $i > 0$, then because of property (4), $\neg t$ must be inductive relative to at least $F_{i-1}$. And even if $i = 0$, $\neg t$ may nevertheless be inductive relative to some $F_j$.

[1] Notice that this syntactic check avoids checking semantic equivalence of potentially complex formulas.

[2] In fact, $\neg s$ is inductive relative to $F_{k-2}$, if not a later stepwise approximation. For suppose not; then there would exist an $F_{k-2}$-state $t$ that is a predecessor to $s$, so that by (4), $s$ would be a $F_{k-1}$-state. But then an $F_{k-1}$-state could reach a violation in one transition, contradicting (3) and (4).

[3] In practice, because $c$ may actually be inductive relative to $F_j$ for some $j > i$ even though $\neg s$ is not, we attempt to push it forward as far as possible, that is, until $F_j \wedge c \wedge T \Rightarrow c'$ but $F_{j+1} \wedge c \wedge T \not\Rightarrow c'$. However, this variation complicates the discussion, so we do not consider it further.

Hence we recur on $t$. The new subgoal is to produce a subclause of $\neg t$ that is inductive relative to $F_i$, eliminating $t$ at $F_{i+1}$. Unless $P$ is not invariant, such a clause is eventually added to $F_{i+1}$, possibly after considering one or more predecessors of $t$. Then $s$ can be considered with respect to the strengthened over-approximation $F_{i+1}$. This process of considering predecessors recursively continues until $\neg s$ is finally inductive relative to $F_k$ (unless a counterexample trace is discovered first). In practice, it is worthwhile to find subclauses inductive relative to $F_k$ for every other state considered during the recursion, as the resulting clauses may be mutually inductive but not independently inductive.

With $s$ no longer an $F_k$-state, $F_k \wedge T \Rightarrow P'$ can be considered again.

## 5 Related Work

SAT-based unbounded model checking constructs clauses via quantifier elimination; for a safety property $P$, it computes the weakest inductive strengthening of $P$ [20]. In our algorithm, induction is a means not only to construct clauses by generalizing from states, but also to abstract the system based on the property.

Our algorithm can be viewed from the perspective of predicate abstraction/refinement [17,8]: the minor iterations generate new predicates (clauses) while the major iterations propagate them forward through the stepwise approximations (that is, add $c \in F_i$ to $F_{i+1}$ if $F_i \wedge T \Rightarrow c'$). If the current clauses are insufficient for convergence to an inductive strengthening of $P$, the next sequence of minor iterations generates new clauses that enable propagation to continue at least one additional step.

The stepwise over-approximation structure of $F_0, F_1, F_2, \ldots, F_k$ is similar to that of interpolation-based model checking (ITP), which uses an interpolant from an unsatisfiable $K$-step BMC query to compute the post-image approximately [21]. All states in the image are at least $K-1$ steps away from violating the property. A larger $K$ refines the image by increasing the minimum distance to violating states. In our algorithm, if the frontier is at level $k$, then $F_i$, for $0 \leq i \leq k$, contains only states that are at least $k-i+1$ steps from violating the property. As $k$ increases, the minimum number of steps from $F_i$-states to violating states increases. In both cases, increasing $k$ (in ours) or $K$ (in ITP) sufficiently for a correct system yields an inductive assertion. However, the algorithms differ in their underlying "technology": ITP computes interpolants from $K$-step BMC queries, while our algorithm uses inductive generalization of cubes, which requires only 1-step induction queries for arbitrarily large $k$.

Our work could in principle be applied as a method of strengthening $k$-induction [24,23,1,25]. However, $k$-induction would simply eliminate the states that are easiest to inductively generalize — since they have short predecessor chains — so we do not recommend this combination.

The method described in this paper first appeared in a technical report [5].

**Listing 1.1.** The main function

```
{ @post: rv iff P is S−invariant }                                    1
bool prove():                                                         2
  if sat(I ∧ ¬P) or sat(I ∧ T ∧ ¬P′):                                 3
    return false                                                      4
  F_0 := I, clauses(F_0) := ∅                                         5
  F_i := P, clauses(F_i) := ∅ for all i > 0                           6
  for k := 1 to ...:                                                  7
    { @rank: 2^|x̄| + 1                                                8
      @assert (A):                                                    9
        (1) ∀ i ≥ 0, I ⇒ F_i                                          10
        (2) ∀ i ≥ 0, F_i ⇒ P                                          11
        (3) ∀ i > 0, clauses(F_{i+1}) ⊆ clauses(F_i)                  12
        (4) ∀ 0 ≤ i < k, F_i ∧ T ⇒ F′_{i+1}                           13
        (5) ∀ i > k, |clauses(F_i)| = 0 }                             14
    if not strengthen(k):                                             15
      return false                                                    16
    propagateClauses(k)                                               17
    if clauses(F_i) = clauses(F_{i+1}) for some 1 ≤ i ≤ k:             18
      return true                                                     19
```

## 6 Formal Presentation and Analysis

We present the algorithm and its proof of correctness simultaneously with annotated pseudocode in Listings 1.1-1.5 using the classic approach to program verification [16,18]. In the program text, *@pre* and *@post* introduce a function's pre- and post-condition, respectively; *@assert* indicates an invariant at a location; and *@rank* indicates a ranking function represented as the maximum number of times that the loop may iterate. As usual, a function's pre-condition is over its parameters while its post-condition is over its parameters and its return value, rv. For convenience, the system $S : (\bar{i}, \bar{x}, I, T)$ and property $P$ are assumed to be in scope everywhere. Also, some assertions are labeled and subsequently referenced in annotations. All assertions are inductive, but establishing the ranking functions requires additional reasoning, which we provide below.

Listing 1.1 presents the top-level function `prove`, which returns `true` if and only if $P$ is $S$-invariant. First it looks for 0-step and 1-step counterexample traces. If none are found, $F_0, F_1, F_2, \ldots$ are initialized to assume that $P$ is invariant, while their clause sets are initialized to empty. As a formula, each $F_i$ for $i > 0$ is interpreted as $P \wedge \bigwedge \text{clauses}(F_i)$. Then it constructs the sequence of $k$-step over-approximations starting with $k = 1$. On each iteration, it first calls `strengthen`$(k)$ (Listing 1.2), which strengthens $F_i$ for $1 \leq i \leq k$ so that $F_i$-states are at least $k - i + 1$ steps away from violating $P$, by assertions $A(2)$ and `strengthen`'s *post*$(2)$. Next it calls `propagateClauses`$(k)$ (Listing 1.3) to propagate clauses forward through $F_1, F_2, \ldots, F_{k+1}$. If this propagation yields

**Listing 1.2.** The `strengthen` function

```
{ @pre:                                                                    20
   (1) A                                                                   21
   (2) k ≥ 1                                                               22
  @post:                                                                   23
   (1) A.1–3                                                               24
   (2) if rv then ∀ 0 ≤ i ≤ k, F_i ∧ T ⇒ F'_{i+1}                          25
   (3) ∀ i > k + 1, |clauses(F_i)| = 0                                     26
   (4) if ¬rv then there exists a counterexample trace }                   27
bool strengthen(k : level):                                                28
  try:                                                                     29
    while sat(F_k ∧ T ∧ ¬P'):                                              30
      { @rank: 2^{|x̄|}                                                     31
        @assert (B):                                                       32
          (1) A.1–4                                                        33
          (2) ∀ c ∈ clauses(F_{k+1}), F_k ∧ T ⇒ c'                         34
          (3) ∀ i > k + 1, |clauses(F_i)| = 0 }                            35
      s := the predecessor extracted from the witness                      36
      n := inductivelyGeneralize(s, k − 2, k)                              37
      pushGeneralization({(n + 1, s)}, k)                                  38
      { @assert (C): s ⊭ F_k }                                             39
    return true                                                            40
  except Counterexample:                                                   41
    return false                                                           42
```

**Listing 1.3.** The `propagateClauses` function

```
{ @pre:                                                                    43
   (1) A.1–3                                                               44
   (2) ∀ 0 ≤ i ≤ k, F_i ∧ T ⇒ F'_{i+1}                                     45
   (3) ∀ i > k + 1, |clauses(F_i)| = 0                                     46
  @post:                                                                   47
   (1) pre                                                                 48
   (2) ∀ 0 ≤ i ≤ k, ∀c ∈ clauses(F_i), if F_i ∧ T ⇒ c' then c ∈ F_{i+1} }   49
void propagateClauses(k : level):                                          50
for i := 1 to k:                                                           51
  { @assert: ∀ 0 ≤ j < i, ∀c ∈ clauses(F_j), if F_j ∧ T ⇒ c' then c ∈ F_{j+1} }52
    for each c ∈ clauses(F_i):                                             53
      { @assert: pre }                                                     54
      if not sat(F_i ∧ T ∧ ¬c'):                                           55
        clauses(F_{i+1}) := clauses(F_{i+1}) ∪ {c}                         56
```

any adjacent levels $F_i$ and $F_{i+1}$ that share all clauses, then $F_i$ is an inductive strengthening of $P$, proving $P$'s invariance.

While the assertions are inductive, an argument needs to be made to justify the ranking function. By $A(3)$, the state sets represented by $F_0, F_1, \ldots, F_k$ are nondecreasing with level. Given `propagateClauses`'s *post*(2), avoiding termination at line 19 requires that they be strictly increasing with level, which is

**Listing 1.4.** Stepwise-relative inductive generalization

```
{ @pre:                                                                    57
    (1) B                                                                  58
    (2) min ≥ −1                                                           59
    (3) if min ≥ 0 then ¬s is inductive relative to F_min                  60
    (4) there is a trace from s to a ¬P−state                              61
  @post:                                                                   62
    (1) B                                                                  63
    (2) min ≤ rv ≤ k, rv ≥ 0                                               64
    (3) s ⊭ F_{rv+1}                                                       65
    (4) ¬s is inductive relative to F_rv }                                 66
level inductivelyGeneralize(s : state, min : level, k : level):           67
  if min < 0 and sat(F_0 ∧ T ∧ ¬s ∧ s'):                                   68
    raise Counterexample                                                   69
  for i := max(1, min+1) to k:                                             70
    { @assert:                                                             71
      (1) B                                                                72
      (2) min < i ≤ k                                                      73
      (3) ∀ 0 ≤ j < i, ¬s is inductive relative to F_j }                    74
    if sat(F_i ∧ T ∧ ¬s ∧ s'):                                             75
      generateClause(s, i−1, k)                                            76
      return i−1                                                           77
  generateClause(s, k, k)                                                  78
  return k                                                                 79
                                                                           80
{ @pre:                                                                    81
    (1) B                                                                  82
    (2) i ≥ 0                                                              83
    (3) ¬s is inductive relative to F_i                                    84
  @post: (1) B, (2) s ⊭ F_{i+1} }                                          85
void generateClause(s : state, i : level, k : level):                      86
  c := subclause of ¬s that is inductive relative to F_i                   87
  for j := 1 to i+1:                                                       88
    { @assert: B }                                                         89
    clauses(F_j) := clauses(F_j) ∪ {c}                                     90
```

impossible when $k$ exceeds the number of possible states. Hence, $k$ is bounded by $2^{|\bar{x}|} + 1$, and, assuming that the called functions always terminate, `prove` always terminates.

For level $k$, `strengthen`($k$) (Listing 1.2) iterates until $F_k$ excludes all states that lead to a violation of $P$ in one step. Suppose $s$ is one such state. It is eliminated by, first, inductively generalizing $\neg s$ relative to some $F_i$ through a call to `inductivelyGeneralize`($s, k-2, k$)[4] (Listing 1.4) and, second, pushing for a generalization at level $k$ through a call to `pushGeneralization`($\{(n+1, s)\}$, $k$) (Listing 1.5). At the end of the iteration, $F_k$ excludes $s$ (assertion $C$). This

[4] Note that $\neg s$ is inductive relative to $F_{k-2}$ by $A(2)$ and $A(4)$.

**Listing 1.5.** The `pushGeneralization` function

```
{ @pre:                                                                    91
   (1) B                                                                   92
   (2) ∀ (i,q) ∈ states, 0 < i ≤ k+1                                       93
   (3) ∀ (i,q) ∈ states, q ⊭ F_i                                           94
   (4) ∀ (i,q) ∈ states, ¬q is inductive relative to F_{i-1}               95
   (5) ∀ (i,q) ∈ states, there is a trace from q to a ¬P-state             96
  @post:                                                                   97
   (1) B                                                                   98
   (2) ∀ (i,q) ∈ states, q ⊭ F_k }                                         99
void pushGeneralization(states : (level, state) set, k : level)           100
  while true:                                                             101
    { @rank: (k+1)2^|x̄|                                                   102
      @assert (D):                                                        103
        (1) pre                                                           104
        (2) ∀ (i,q) ∈ states_prev, ∃j ≥ i, (j,q) ∈ states }               105
    (n, s) := choose from states, minimizing n                            106
    if n > k: return                                                      107
    if sat(F_n ∧ T ∧ s'):                                                 108
      p := the predecessor extracted from the witness                     109
      { @assert (E): ∀ (i,q) ∈ states, p ≠ q }                            110
      m := inductivelyGeneralize(p, n-2, k)                               111
      states := states ∪ {(m+1, p)}                                       112
    else:                                                                 113
      m := inductivelyGeneralize(s, n, k)                                 114
      { @assert (F): m+1 > n }                                            115
      states := states \ {(n, s)} ∪ {(m+1, s)}                            116
```

progress implies that the loop can iterate at most as many times as there are possible states, yielding `strengthen`'s ranking function.

The functions in Listing 1.4 perform inductive generalization relative to some $F_i$. If $min < 0$, $s$ might have an $I$-state predecessor, which is checked at line 68.

The `pushGeneralization` algorithm (Listing 1.5) is the key to "pushing" inductive generalization to higher levels. The insight is simple: if a state $s$ is not inductive relative to $F_i$, apply inductive generalization to its $F_i$-state predecessors. The complication is that this recursive analysis must proceed in a manner that terminates despite the presence of cycles in the system's state graph. To achieve termination, a set *states* of pairs $(i, s)$ is maintained such that each pair $(i, s) \in states$ represents the knowledge that (1) $s$ is inductive relative to $F_{i-1}$, and (2) $F_i$ excludes $s$. The loop in `pushGeneralization` always selects a pair $(n, s)$ from *states* such that $n$ is minimal over the set. Hence, none of the states already represented in *states* can be a predecessor of $s$ at level $n$.

Formally, termination of `pushGeneralization` is established by the inductive assertions $D(2)$, which asserts that the set of states represented in *states* does not decrease ($states_{\mathsf{prev}}$ represents *states*'s value on the previous iteration or, during the first iteration, upon entering the function); $E$, which asserts that the new state $p$ is net yet represented in *states*; and $F$, which asserts that the level

associated with a state can only increase. Given that each iteration either adds a new state to *states* or increases a level for some state already in *states* and that levels peak at $k+1$, the number of iterations is bounded by the product of $k+1$ and the size of the state space.

Listings 1.1-1.5 and the termination arguments yield total correctness:

**Theorem 1.** *For finite transition system* $S : (\bar{i}, \bar{x}, I, T)$ *and safety property* $P$*, the algorithm terminates, and it returns* **true** *if and only if* $P$ *is* $S$*-invariant.*

A variation exists that is perhaps more satisfying conceptually. Recall that `inductivelyGeneralize` and `generateClause` (Listing 1.4) together generate a subclause of $\neg s$ that is inductive relative to $F_i$, where $F_i$ is the weakest stepwise assumption relative to which $\neg s$ is inductive. It is possible to find the highest level $j \geq i$ for which $\neg s$ has a subclause that is inductive relative to $F_j$ even if $\neg s$ is not itself inductive relative to $F_j$ (in which case $j > i$). However, in practice, this variation requires more time on designs with many latches. Whereas the unsatisfiable core of the query $F_{i-1} \wedge T \wedge \neg s \wedge s'$ at line 75 can be used to reduce $s$, often significantly, before applying inductive generalization (see Section 7), no such optimization is possible for the variation.

## 7 Single-Core Implementation

Our submission to HWMCC'10, `ic3`, placed third in the "unsatisfiable" category, third overall, and solved 37 more benchmarks than the 2008 winner [3].[5] We discuss the implementation details of `ic3` in this section.

We implemented the algorithm, AIG sweeping [11], and conversion of the transition relation to CNF based on technology mapping [13] in OCaml. The preprocessor of MiniSAT 2.0 is applied to further simplify the transition relation [12,13]. The time spent in preprocessing the transition relation is amortized over thousands to millions of 1-induction SAT instances in a typical analysis.

One implementation choice that may seem peculiar is that we used a modified version of ZChaff for SAT-solving [22]. The most significant modification was to change the main data structure and algorithm for BCP to be like MiniSAT [14]. We chose ZChaff, which is considered to be outdated, because it offers efficient incremental functionality: clauses can be pushed and popped, which is necessary for finding an inductive subclause. While this functionality can be simulated in more recent solvers [15], each push/pop iteration requires a new literal. Given that hundreds to thousands of push/pop cycles occur *per second* in our analysis, each involving clauses, it seems that the amount of garbage that would accumulate in the simulated approach would be prohibitive. Thus we elected to use a library with built-in incremental capability. The consequence is that ZChaff caused timeouts on the following benchmarks during HWMCC'10:

[5] The data are available at `http://fmv.jku.at/hwmcc10`. The competition binary and an open source version of `ic3` are available at `http://ecee.colorado.edu/~bradleya`

`bobaesdinvdmit`, `bobsmfpu`, `bobpcihm`, and `bobsmminiuart`. Otherwise, the percentage of time spent in SAT solving varies from as low as 20% to as high as 85%. Benchmarks on which SAT solving time dominates could benefit from a faster solver.

We highlight important implementation decisions. The most significant optimization is to extract the unit clauses of an unsatisfiable core whenever possible. Consider the unsatisfiable query $F \wedge c \wedge T \wedge \neg c'$; the unsatisfiable core can reveal a clause $d \subset c$ such that $F \wedge c \wedge T \wedge \neg d'$ is also unsatisfiable. The clause $d$ is an inductive subclause if it satisfies initiation. If the initial state is defined such that all latches are 0 (as in HWMCC'10) and $d$ does not satisfy initiation, `ic3` simply restores a negative literal from $c$. This optimization applies in the following contexts: (1) in the `inductivelyGeneralize` algorithm, from the unsatisfiable query that indicates that $\neg s$ is inductive relative to $F_i$ when $\neg s$ is not inductive relative to $F_{i+1}$ (Listing 1.4, line 75); (2) in the down algorithm [6], from the (final) unsatisfiable query indicating an inductive subclause; (3) in the `up` algorithm; and (4) in `propagateClauses`, during propagation of clauses between major iterations (Listing 1.3, line 55).

In the implementation of inductive generalization (algorithm MIC [6]), we use a threshold to end the search for a minimal inductive subclause. If down is applied unsuccessfully to three subclauses of $c$, each formed by removing one randomly chosen literal, then $c$ is returned. While $c$ may not be minimal — that is, some $d \subset c$ may also be (relatively) inductive — it is typically sufficiently strong; and the search is significantly faster.

We use a stepwise cone of influence (COI) [2] to reduce cubes: if a state $s$ is $i$ transitions away from violating $P$, the initial clause $c \subseteq \neg s$ is set to contain only state variables of the $i$-step COI; the transition relation is unchanged for practical reasons. The generated clause is more relevant with respect to $P$ in explaining why states similar to $s$ are unreachable, although $c$ may only be inductive relative to a stronger stepwise assumption than $\neg s$.

Subsumption reduces clause sets across levels between major iterations: if clause $c$ at level $i$ subsumes clause $d$ at level $j \leq i$, then $d$ is removed.

For memory efficiency, one SAT manager is used for computing consecution at all levels. A level-specific literal is added to each generated clause. Clauses at and above level $i$ are activated when computing consecution relative to $F_i$.

An initial set of simulation runs yields candidate equivalences between latches. These candidate equivalences are then logically propagated across the stepwise approximations between major iterations. Some benchmarks are easily solved once key equivalences are discovered, and while the pure analysis is poor at discovering them, propagation easily finds them. Simulation make this analysis inexpensive even when it is not effective. This binary clause analysis fits well with the overall philosophy of generating stepwise-relative inductive clauses.

When searching for inductive subclauses, using an arbitrary static ordering of literals to consider for removal yields poor results. We tried various heuristics for dynamically ordering the literals, but none were particularly effective. The competition version of `ic3` prefers the negations of literals that appear frequently

**Table 1.** Runtime data for selected benchmarks from HWMCC'10 [3]

| Benchmark | Result | Time (s) | # queries | \|proof\| | $k$ |
|---|---|---|---|---|---|
| bjrb07amba10andenv | unsat | 260 | 12238 | 262 | 7 |
| bob3 | unsat | 10 | 44058 | 865 | 7 |
| boblivea | unsat | 5 | 34884 | 652 | 14 |
| boblivear | unsat | 4 | 34547 | 668 | 14 |
| bobsmnut1 | unsat | 9 | 20530 | 554 | 15 |
| intel007 | unsat | 30 | 31250 | 1382 | 6 |
| intel044 | sat | 303 | 578982 | 92 | 57 |
| intel045 | sat | 316 | 596539 | 124 | 49 |
| intel046 | sat | 223 | 431123 | 78 | 44 |
| intel047 | sat | 293 | 561304 | 82 | 52 |
| intel054 | unsat | 56 | 147986 | 1459 | 19 |
| intel055 | unsat | 9 | 28302 | 385 | 15 |
| intel056 | unsat | 15 | 63877 | 649 | 19 |
| intel057 | unsat | 21 | 72925 | 731 | 18 |
| intel059 | unsat | 11 | 47840 | 558 | 17 |
| intel062 | unsat | 301 | 389065 | 3372 | 26 |
| nusmvbrp | unsat | 5 | 55281 | 306 | 27 |
| nusmvreactorp2 | unsat | 51 | 308627 | 779 | 116 |
| nusmvreactorp6 | unsat | 178 | 753335 | 1723 | 119 |
| pdtvisns3p00 | unsat | 11 | 4428 | 465 | 12 |
| pdtvisns3p01 | unsat | 27 | 104750 | 1109 | 10 |
| pdtvisns3p02 | unsat | 21 | 85812 | 680 | 12 |
| pdtvisns3p03 | unsat | 21 | 80810 | 745 | 12 |
| pdtvisns3p04 | unsat | 115 | 281812 | 1783 | 14 |
| pdtvisns3p05 | unsat | 135 | 326604 | 2033 | 13 |
| pdtvisns3p06 | unsat | 13 | 55016 | 631 | 9 |
| pdtvisns3p07 | unsat | 84 | 228175 | 1631 | 11 |
| pj2017 | unsat | 233 | 74417 | 685 | 27 |

in the *states* set of `pushGeneralization`. A clause with such literals is relevant to many of the states in *states*. However, the only definite claim is that changing the variable ordering is superior to using an arbitrary static ordering. We have not investigated whether well-chosen static orderings might yield better performance.

While time and memory data for HWMCC'10 are already publicly available, Table 1 provides data particular to `ic3` for the benchmarks that `ic3` and at most two other entries solved. The table indicates the number of executed SAT queries (**# queries**); the size of the proof (|**proof**|), which is the number of clauses for unsatisfiable benchmarks and the length of the counterexample for satisfiable benchmarks; and the maximum value of $k$. Notice how widely the maximum $k$ value varies. The benefit of the work described in this paper over previous work [6] is particularly apparent for benchmarks with large $k$, as such benchmarks require generalizing the many states of long sequences simultaneously. Notice also the rate at which SAT queries are solved — several thousand per second — indicating that these queries are trivial compared to those posed by other SAT-based model checkers.

A variant of this algorithm emphasizes speed over quality in inductive clause generation. Rather than using "strong" induction to compute a minimal inductive subclause $c \subseteq d$ relative to $F_i$, it computes a prime implicate $\hat{c}$ of $F_i \wedge d \wedge T$, that is, a minimal subclause $\hat{c} \subseteq d$ such that $F_i \wedge d \wedge T \Rightarrow \hat{c}'$ holds. On the HWMCC'10 benchmark set, this variation solves 28 fewer unsatisfiable benchmarks and three fewer satisfiable benchmarks. Quality matters.

## 8 Parallel Implementation

Converting the implementation from sequential to parallel is straightforward. The overall model is of independent model checkers sharing information. Each time a process generates a clause $c$ at level $i$, it sends the tuple $(c, i)$ to a central server and receives in return a list of clause-level tuples generated since its last communication. To avoid one source of duplicated effort, it uses the new information to syntactically prune its *states* set. During `propagateClauses` calls, each process propagates a subset of the clauses based on hashing modulo the number of total processes, and the processes proceed in lockstep, level by level. Additional communications handle exceptional situations such as the discovery of a counterexample. Processes attempt to avoid discovering the same information simultaneously simply through exploiting the randomness in the ZChaff implementation, although co-discovery occurs in practice early and late in each major iteration.

How well does the parallel implementation scale with available cores? To investigate this question, we selected eight benchmarks from the competition that are difficult but possible for the non-parallel version: Intel benchmarks 20, 21, 22, 23, 24, 29, 31, and 34. We ran the non-parallel and parallel implementations on four Quad Core i5-750/2.66GHz/8MB-cache machines with 8GB, DDR3 non-ECC SDRAM at 1333MHz, running 64-bit Ubuntu 9.10. One process was arranged as a single process on an otherwise mostly idle machine; four processes were arranged as one process per machine; eight processes were arranged as two processes per machine; and twelve processes were arranged as three processes per machine. Unfortunately, (shared) memory latency increased significantly with the number of processes per machine so that the twelve-process configuration was not necessarily an improvement on the eight-process configuration in terms of the system-wide number of SAT problems solved per second.

Each benchmark was analyzed eight times by each configuration, with a timeout of two hours (7200 seconds). Figure 1 presents the results in eight graphs that plot running times against the number of processes. The numbers adjacent to dots at 7200 indicate the number of timeouts.

Every benchmark benefits from additional cores. One possible explanation, however, is simply that parallelism reduces variance. The high variability of the single-process implementation may be a result of "lucky" discoveries of certain clauses that yield major progress toward proofs. Runs that fail to make these discoveries early can take significantly longer than those that do. To explore this possibility, we set up the following configuration: eight non-communicating processes, where the first to finish causes the others to terminate. In other words, the minimum time is taken from eight independent runs, except that all are executed simultaneously, thus experiencing the memory latency of the eight-process communicating configuration. The results are shown in Figure 2**(a)**.

The data show that some performance gain can indeed be attributed to a reduction in variance. However, comparing Figures 1 and 2 for each benchmark indicates that this reduction in variance cannot explain all of the performance gain. In particular, the standard eight-process parallel version is significantly

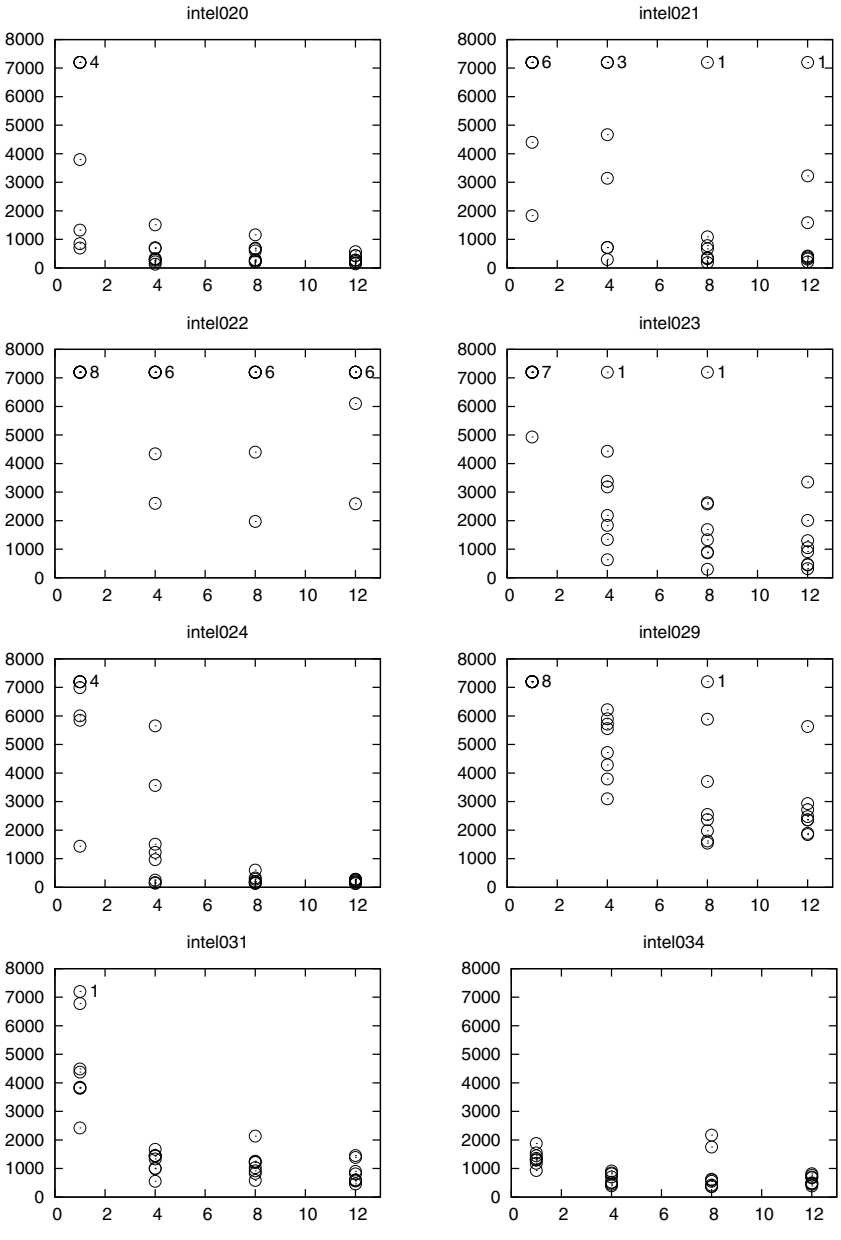


**Fig. 1.** Number of communicating processes *vs.* time (in seconds)

faster on benchmarks 23, 24, and 29. Except on benchmark 22, for which the data are inconclusive, it is faster on the other benchmarks as well. Therefore, communication is a significant factor in explaining superior performance.

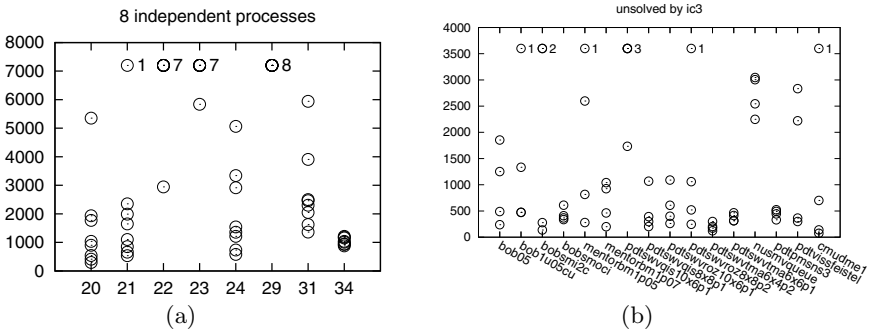


**Fig. 2.** Benchmarks *vs.* time

Unfortunately, saturation is also possible: at some number of processes, the rate of co-discovery of information is such that additional processes do not improve runtime. For example, the performance that benchmarks 31 and 34 gain from the four-process configuration is not improved upon with additional processes. However, the data do not indicate degrading performance, either.

Having established that communicating processes are superior to independent processes, we next tested if parallel `ic3` is superior to serial `ic3` in numbers of benchmarks solved in a given time, in particular the 900 seconds per benchmark allotted in HWMCC'10. We ran the twelve-process communicating configuration for one hour on each of the 105 benchmarks that `ic3` failed to solve during HWMCC'10 and then extracted the 16 benchmarks that were proved to be unsatisfiable, excluding the `intel` set of Figure 1. Analyzing these 16 benchmarks four times each with a timeout of one hour produced the data in Figure 2**(b)**. Figures 1 and 2**(b)** indicate that the twelve-process configuration would yield at least twelve additional proofs within 900 seconds.

## 9 Conclusion

The performance of `ic3` in HWMCC'10 shows that the incremental generation of stepwise-relative inductive clauses is a promising new approach to symbolic model checking. Furthermore, it is amenable to simple yet effective parallelization, a crucial characteristic given modern architectures.

Why does this algorithm work so well? Consider a clause $c$. Predecessors to $c$-states are likely to be or to look similar to $c$-states, to the extent that dropping a few literals from $c$ may yield an inductive clause $d$. This reasoning motivates the inductive generalization algorithm (Section 3). However, systems violate this observation to a varying extent. The stepwise sets $F_0, ..., F_k$ offer a new possibility: $c$, if invariant, is inductive relative to a stepwise assumption $F_i$. Subsequent discovery of additional clauses can yield a set of mutually (relatively) inductive clauses that are propagated forward together.

Ongoing research includes designing a thread-safe incremental SAT solver, in which threads share a common set of core constraints but have thread-local temporary constraints; investigating how inductive clause generation can accelerate finding counterexamples; and exploring how stepwise-relative inductive generalization can apply to the analysis of infinite-state systems.

**Acknowledgments.** I am grateful to Fabio Somenzi for many fruitful discussions. Arlen Cox provided the initial implementation of technology mapping-based CNF translation. This work was supported by NSF grant CCF 0952617.

## References

1. Awedh, M., Somenzi, F.: Automatic invariant strengthening to prove properties in bounded model checking. In: DAC, pp. 1073–1076. ACM Press, New York (2006)
2. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
3. Biere, A., Claessen, K.: Hardware model checking competition. In: Hardware Verification Workshop (2010)
4. Bradley, A.R.: Safety Analysis of Systems. PhD thesis, Stanford University (May 2007)
5. Bradley, A.R.: k-step relative inductive generalization. Tech. Rep., CU Boulder (March 2010), `http://arxiv.org/abs/1003.3649`
6. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD (2007)
7. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^20 states and beyond. Inf. Comput. 98(2), 142–170 (1992)
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5) (2003)
9. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM Press, New York (1977)
11. Eén, N.: Cut sweeping. Tech. rep., Cadence (2007)
12. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
13. Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007)
14. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
15. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In: BMC (2003)
16. Floyd, R.W.: Assigning meanings to programs. In: Symposia in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society, Providence (1967)
17. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

18. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
19. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
20. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
21. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
22. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC (2001)
23. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)
24. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
25. Vimjam, V.C., Hsiao, M.S.: Fast illegal state identification for improving SAT-based induction. In: DAC, pp. 241–246. ACM Press, New York (2006)