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, 19, 16, 18]. 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 and distributed processors. The fundamental idea is to generate clauses that are inductive relative to stepwise reachability information.

Section 2 introduces the vocabulary to discuss the algorithm. Section 3 reviews how to generate an inductive subclause from a given clause, the technique on which this new work is based [4]. Then Section 4 presents the algorithm at a high level, while Section 6 describes it formally and proves its correctness.

An implementation of the algorithm, called `ic3` ("**I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness"), placed third at HWMCC'10 and is available for download at the author's website [3]. Section 7 discusses the critical optimizations. Section 8 empirically analyzes the runtime characteristics of a parallel version of `ic3`.

2 Definitions

A *finite-state transition system* $S : (\bar{x}, I, T)$ is described by a pair of propositional logic formulas: an initial condition $I(\bar{x})$ and a transition relation $T(\bar{x}, \bar{x}')$ over

a set of Boolean variables $\bar{x}$ and their next-state primed forms $\bar{x}'$ [6]. 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 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, denoted $s \models F$, or falsifies it, denoted $s \not\models F$. 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 (states satisfying P) 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 $s_k \not\models P$.

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 assertion F is inductive *relative to* another assertion G if condition (1) and a modified version of (2) hold: $G \wedge F \wedge T \Rightarrow F'$. An inductive *strengthening* of a safety property P is a formula F such that $F \wedge P$ is inductive.

3 Review of Inductive Generalization

Previous work introduced a technique for discovering a *minimal inductive subclause* of a given clause if one exists [4]. Such a clause (1) is a subclause of c, (2) is inductive (possibly relative to known or assumed information G), and (3) is minimal in that it does not contain any strict subclauses that are also inductive.

Inductive generalization of a cube s is the process of finding a minimal inductive subclause d of $\neg s$, if one exists. The resulting subclause over-approximates the set of reachable states while excluding s and all states that can reach s. In practice, a minimal inductive subclause is typically substantially smaller than the cube s from which it is extracted and excludes states that are not necessarily related to s by T, which is why we say that the inductive subclause generalizes that s is unreachable.

While the details of finding inductive subclauses are best left to the original paper [4], an informal description of how to do so is in order. Suppose that we wish to find a subclause $d \subseteq c_0$ that is inductive relative to G, if one 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 upon 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 [4].

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 [4]. Section 7 discusses optimizations to these procedures.

4 Informal Description

Consider a transition system $S : (\bar{x}, I, T)$ and a safety property P. Our 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 discussion.)

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.

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 $F_0, F_1, \ldots, F_k, F_{k+1} = P$ 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}) 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 step from violating P. What is the maximum i, $0 \leq i \leq k$, such that $\neg s$ is inductive relative to F_i? 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. 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 already inductive relative to F_i.

The relatively inductive clause c is now conjoined to each of $F_0, \ldots, F_{i+1}$. (Why to F_{i+1}? $F_i \wedge c \wedge T \Rightarrow c'$ holds by the construction of c, so doing so maintains property (4).) Notice that because c' is not, in general, simply an implicate of $F_i \wedge T$, c may actually provide new information to F_ℓ for some $\ell < i$, in addition to definitely strengthening F_{i+1}. That clauses are formed through inductive generalization is what distinguishes this algorithm from SAT-based symbolic model checking [15], and it is also what makes the algorithm effective.

(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 here or in Section 6.)

If $i \geq k - 1$, 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 states than s.

But it is of course possible that $i < k - 1$. In this case, s is still an F_k-state. How, then, can we proceed?

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}. (Otherwise there would exist an F_{i-1}-state u that is a predecessor to t; but by (4), t would then have to be an F_i-state.) And even if $i = 0$, t may nevertheless be inductive relative to some F_j, $0 \leq j \leq k$.

Hence we recur on t. The new goal is to produce a clause that is inductive relative to F_k and that eliminates t. And indeed, unless P is not invariant, a clause is eventually added to F_{i+1} that eliminates t, possibly after considering one or more predecessors of t. Then both s and t can be considered with respect to the now stronger 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 also 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. Doing so benefits the next major iteration.

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

5 Related Work

SAT-based unbounded model checking constructs clauses via quantifier elimination; additionally, for a safety property P, it computes the weakest inductive

strengthening of P [15]. In our algorithm, induction is a means not only for generalizing from states but also for abstracting the system based on the property.

Our algorithm can be seen as an instance of predicate abstraction/refinement [13, 5]: the minor iterations generate new predicates (clauses) while the major iterations propagate them. If the current clauses are insufficient for convergence to an inductive strengthening assertion, the next major iteration 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 [16]. 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 BMC queries for arbitrarily large k.

Our work could in principle be applied as a method of strengthening k-induction [19, 18, 1, 20]. 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.

6 Formal Presentation and Analysis

We present the algorithm and its proof of correctness simultaneously with annotated pseudocode in Listings 1.1-1.4 using the classic approach to program verification [12, 14]. 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 $S : (\bar{x}, I, T)$ 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 `check`(k) (Listing 1.2), which strengthens F_i for $1 \leq i \leq k$ so that F_i-states are at least

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 check(k):                                                        15
      return false                                                          16
    propagate(k)                                                            17
    if clauses(F_i) = clauses(F_{i+1}) for some 1 ≤ i ≤ k:                   18
      return true                                                           19
```

$k-i+1$ steps away from violating P. Next it calls `propagate`(k) (Listing 1.2) to propagate clauses forward through $F_1, F_2, \ldots, F_{k+1}$. If this propagation yields 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 `propagate`'s *post*(2), avoiding termination at line 19 requires that they be strictly increasing with level, which is 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 a given level k, `check`(k) (Listing 1.2) iterates until F_k excludes all states that can 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 F_n (for some $0 \leq n \leq k$) through a call to `inductive`(s, $k-2$, k) (Listing 1.3) and, second, pushing for a generalization at level k through a call to `push`($\{(n+1,\ s)\}$, k) (Listing 1.4). At the end of the iteration, F_k excludes s (assertion C). This progress implies that the loop can iterate at most as many times as there are possible states, yielding `check`'s ranking function.

The functions in Listing 1.3 perform inductive generalization relative to some F_i. If $min < 0$, s might have an I-state predecessor; hence the check at line 69.

The `push` algorithm (Listing 1.4) 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

Listing 1.2. The `check` and `propagate` functions

```
{ @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 check(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 := inductive(s, k-2, k)                                                37
      push({(n+1, s)}, k)                                                      38
      { @assert (C): s ⊭ F_k }                                                 39
    return true                                                                40
  except Counterexample:                                                         41
    return false                                                                 42
                                                                                 43
{ @pre:                                                                          44
   (1) A.1-3                                                                     45
   (2) ∀ 0 ≤ i ≤ k, F_i ∧ T ⇒ F'_{i+1}                                           46
   (3) ∀ i > k+1, |clauses(F_i)| = 0                                             47
  @post:                                                                         48
   (1) pre                                                                       49
   (2) ∀ 0 ≤ i ≤ k, ∀c ∈ clauses(F_i), if F_i ∧ T ⇒ c' then c ∈ F_{i+1} }        50
void propagate(k : level):                                                       51
  for i := 1 to k:                                                               52
    { @assert: ∀ 0 ≤ j < i, ∀c ∈ clauses(F_j), if F_j ∧ T ⇒ c' then c ∈ F_{j+1} } 53
    for each c ∈ clauses(F_i):                                                   54
      { @assert: pre }                                                           55
      if not sat(F_i ∧ T ∧ ¬c'):                                                 56
        clauses(F_{i+1}) := clauses(F_{i+1}) ∪ {c}                               57
```

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 `push` 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 `push` is established by the inductive assertions $D(2)$, which asserts that the set of states represented in *states* does not decrease

Listing 1.3. Stepwise-relative inductive generalization

```
{ @pre:                                                                    58
    (1) B                                                                  59
    (2) min ≥ −1                                                           60
    (3) if min ≥ 0 then ¬s is inductive relative to F_min                  61
    (4) there is a trace from s to a ¬P−state                              62
  @post:                                                                   63
    (1) B                                                                  64
    (2) min ≤ rv ≤ k, rv ≥ 0                                               65
    (3) s ⊭ F_{rv+1}                                                       66
    (4) ¬s is inductive relative to F_rv }                                 67
level inductive(s : state, min : level, k : level):                        68
  if min < 0 and sat(F_0 ∧ T ∧ ¬s ∧ s'):                                   69
    raise Counterexample                                                   70
  for i := max(1, min + 1) to k:                                           71
    { @assert:                                                             72
      (1) B                                                                73
      (2) min < i ≤ k                                                      74
      (3) ∀ 0 ≤ j < i, ¬s is inductive relative to F_j }                    75
    if sat(F_i ∧ T ∧ ¬s ∧ s'):                                             76
      generate(s, i − 1, k)                                                77
      return i − 1                                                         78
  generate(s, k, k)                                                        79
  return k                                                                 80
                                                                           81
{ @pre:                                                                    82
    (1) B                                                                  83
    (2) i ≥ 0                                                              84
    (3) ¬s is inductive relative to F_i                                    85
  @post: (1) B, (2) s ⊭ F_{i+1} }                                          86
void generate(s : state, i : level, k : level):                            87
  c := subclause of ¬s that is inductive relative to F_i                   88
  for j := 1 to i + 1:                                                     89
    { @assert: B }                                                         90
    clauses(F_j) := clauses(F_j) ∪ {c}                                     91
```

($states_{\mathsf{prev}}$ 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 1.1-1.4 and the termination arguments yield total correctness:

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

Listing 1.4. The `push` function

```
{ @pre:                                                                    92
   (1) B                                                                   93
   (2) ∀ (i,q) ∈ states, 0 < i ≤ k+1                                       94
   (3) ∀ (i,q) ∈ states, q ⊭ F_i                                           95
   (4) ∀ (i,q) ∈ states, ¬q is inductive relative to F_{i-1}               96
   (5) ∀ (i,q) ∈ states, there is a trace from q to a ¬P-state             97
  @post:                                                                   98
   (1) B                                                                   99
   (2) ∀ (i,q) ∈ states, q ⊭ F_k }                                         100
void push(states : (level, state) set, k : level):                         101
  while true:                                                              102
    { @rank: (k+1)2^|x̄|                                                    103
      @assert (D):                                                         104
        (1) pre                                                            105
        (2) ∀ (i,q) ∈ states_prev, ∃j ≥ i, (j,q) ∈ states }                106
    (n, s) := choose from states, minimizing n                             107
    if n > k: return                                                       108
    if sat(F_n ∧ T ∧ s'):                                                  109
      p := the predecessor extracted from the witness                      110
      { @assert (E): ∀ (i,q) ∈ states, p ≠ q }                             111
      m := inductive(p, n-2, k)                                            112
      states := states ∪ {(m+1, p)}                                        113
    else:                                                                  114
      m := inductive(s, n, k)                                              115
      { @assert (F): m+1 > n }                                             116
      states := states \ {(n, s)} ∪ {(m+1, s)}                             117
```

A variation exists that is perhaps more satisfying conceptually. Recall that `inductive` and `generate` (Listing 1.3) together generate a subclause of $\neg s$ that is inductive relative to F_i, where i is the greatest level for which $\neg s$ is inductive relative to F_i. 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 76 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]. The data are publicly available at `http://fmv.jku.at/hwmcc10`. The competition version of `ic3` is available at `http://ecee.colorado.edu/~bradleya`. The interested reader may use the `-v` option to generate verbose output that includes

all generated clauses and their levels as well as runtime statistics. We discuss the implementation details of `ic3` in this section.

We implemented the algorithm, AIG sweeping [7], and conversion of the transition relation to CNF based on technology mapping [9] in O'Caml. The preprocessor of MiniSAT 2.0 is applied to further simplify the transition relation [8, 9]. 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 [17]. The most significant modification was to change the main data structure and algorithm for BCP to be like MiniSAT [10]. Why did we use such an outdated library? ZChaff offers full 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 [11], 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 full incremental capability. The consequence is that ZChaff caused timeouts on the following benchmarks during HWMCC'10: `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 would clearly benefit from a better incremental solver.

We highlight important implementation decisions. The most significant optimization is to extract the unit clauses of an unsatisfiable core whenever possible. If d is a subclause of c and $F \wedge c \wedge T \Rightarrow d'$, then d is an inductive subclause as long as it also satisfies initiation. If the initial state is defined such that all latches are 0 (as in HWMCC'10 [3]) and d does not satisfy initiation, simply restore a negative literal from c. This situation occurs in the following contexts: (1) in the `inductive` 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}; (2) in the down algorithm [4], from the (final) unsatisfiable query indicating an inductive subclause; (3) in the `up` algorithm; and (4) in `propagate`, during propagation of clauses between major iterations.

In the implementation of inductive generalization (algorithm MIC [4]), we use a threshold to end the search for a minimal inductive subclause. If down is applied to three subclauses of c, each formed by removing one randomly chosen literal, without success, then c is returned. While c may not be minimal — that is, some $d \subset c$ may also be inductive — it is typically sufficiently strong; and the search is significantly faster.

We use a stepwise cone of influence (COI) [2] to reduce initial cubes: if a state s is i steps away from a violating state, the initial clause $c \subseteq \neg s$ is pruned to contain only the literals corresponding to non-input latches in the i-step COI. While j such that c is inductive relative to F_j may be less than ℓ such that $\neg s$ is inductive relative to F_ℓ, the generated clause is more relevant in explaining why states similar to s are unreachable.

We reduce clauses across levels by subsumption between major iterations: clause c at level i subsumes clause d at level j if c subsumes d and $i \geq j$.

To minimize memory usage, a single SAT manager is used for computing consecution. 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 equivalences are then propagated across the k-step approximations between major iterations. We added this analysis because we found that a few benchmarks are easily solved once key equivalences are discovered, yet the pure analysis is poor at discovering these equivalences. The simulations make this analysis inexpensive 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, the order in which literals are considered should not be arbitrary and static. An arbitrary static ordering can yield poor results. We tried various heuristics for dynamically and intelligently ordering the literals; none were particularly effective. The one that we used in the competition version of `ic3` orders the literals according to their occurrence in the *states* set of `push`: the negation of literals that appear more frequently are preferred, as a clause with such literals is relevant to many of the states in *states*. That said, 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 performance gains.

Besides time and memory data, other interesting attributes include (1) the maximum k, (2) the number of generated clauses, (3) the number of clauses in the final proof, and (4) the number of SAT calls. Here are typical approximate data for several benchmarks, where the benchmarks are ordered according to increasing runtime and the data are presented in the order specified above: `kenflashp01`, 2, 26, 23, 200; `intel006`, 9, 2K, 760, 27K; `intel055`, 18, 2K, 350, 27K; `pdtvisns3p00`, 14, 4K, 1K, 135K; `nusmvreactorp2`, 140, 18K, 1500, 700K; `bjrb07amba10andenv`, 6, 450, 260, 11K. Notice how the maximum k for `nusmvreactorp2` is so much greater than for the other benchmarks, suggesting that long traces must be examined to establish its property inductively. Another reason why a benchmark can exhibit a relatively large maximum k is that the design has a long initialization phase before the actual interesting behavior becomes apparent. Many of the `intel` benchmarks exhibit this behavior: the analysis seems to really begin when k reaches 35. The reader is invited to download `ic3` and use the `-v` option to explore these quantities further.

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 informs a central server via the tuple (c, i) 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 `push` phases, each process `push`es a subset of the clauses based on hashing modulo the number of total processes, and the processes proceed in lockstep, level by level. There are additional communications necessary to 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 rediscovery occurs in practice at the beginning and ending of major iterations.

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, 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 explanation, however, is simply that the 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 indicate that this reduction in variance cannot explain all of the performance gain. In particular, the standard eight-process parallel version is significantly 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.

Unfortunately, saturation is also possible: at some number of processes, the rate of co-discovery of (redundant) information is such that additional processes do not improve runtime. For example, the performance that benchmarks 31 and

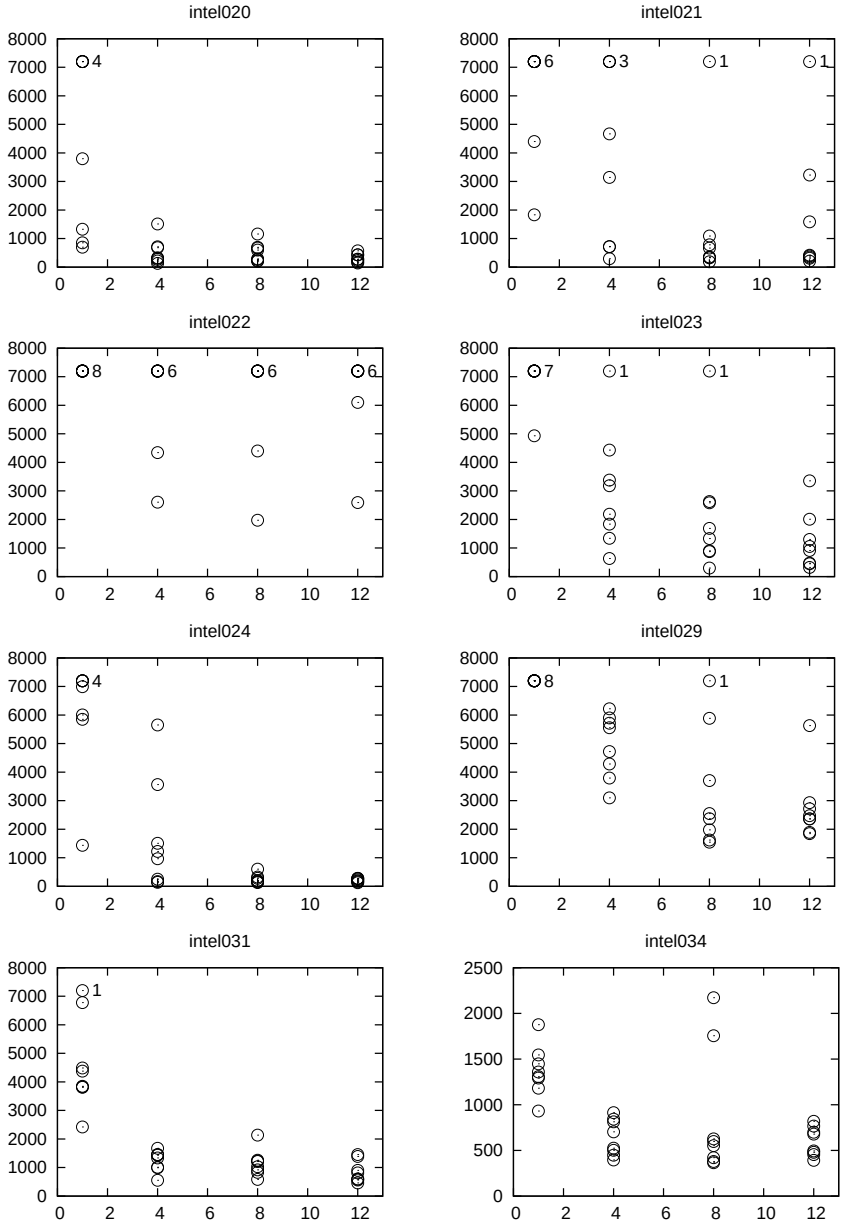

Fig. 1. Number of communicating processes *vs.* time (in seconds)

34 gain from the four-process configuration is not improved upon with additional processes. Fortunately, the additional processes did not yield worse performance.

Given these results, we ran the twelve-process communicating configuration on the benchmarks that `ic3` failed to solve during HWMCC'10. Of those, we extracted the benchmarks that were proved to be unsatisfiable and analyzed

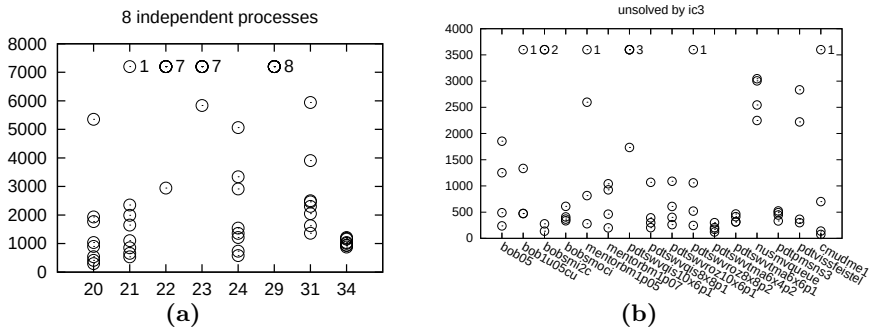

Fig. 2. Benchmarks *vs.* time

them four times each with a timeout of one hour (3600 seconds), producing the data in Figure 2**(b)** (which excludes data for the Intel benchmarks). Figures 1 and 2**(b)** indicate that simply providing more cores to `ic3` would likely yield at least twelve additional proofs within the competition time limit.

We do not provide data for satisfiable benchmarks because the algorithm is not particularly suited for finding counterexamples. We are investigating ideas based on inductive clause generation for addressing the satisfiable case.

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. It is amenable to simple yet effective parallelization (Section 8).

Why does this algorithm work so well? Here we are forced to speculate. Consider a clause c. Predecessors to c-states are likely to look similar to c-states — or even be c-states. If not all predecessors are c-states, they are perhaps sufficiently similar that dropping a few literals from c will yield an inductive clause d. This reasoning motivates the inductive generalization algorithm (Section 3), and it succeeds on some benchmarks that are hard for other methods [4]. However, on state-spaces that violate this observation, the method fails. The framework of stepwise sets $F_0, ..., F_k$ offers a new possibility: if inductive generalization fails relative to F_i, it can be attempted in the more restricted context of F_{i-1}. Subsequent discovery of additional clauses can yield a set of mutually (relatively) inductive clauses that are not independently (relatively) inductive. They are propagated forward together.

Because the algorithm eschews the unrolling of the transition relation, its demands on a SAT solver differ from those of other SAT-based methods. Handling many incremental queries well is more important than quickly solving large problems. A SAT solver ideal for this style of model checking would allow multi-threaded access. Threads would share a core set of constraints into which clauses could be added but not removed. Thread-local managers would handle pushes and pops in thread-local memory.

The current algorithm is most suited for finding proofs. Ongoing research includes exploring how inductive clause generation can be used to accelerate finding counterexamples. Another direction for research is to apply the ideas of stepwise-relative inductive generalization to an infinite-state setting.

References

1. Awedh, M., and Somenzi, F. Automatic invariant strengthening to prove properties in bounded model checking. In *DAC* (2006), ACM Press, pp. 1073–1076.
2. Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. Symbolic model checking without BDDs. In *TACAS* (London, UK, 1999), Springer-Verlag, pp. 193–207.
3. Biere, A., and Claessen, K. Hardware model checking competition. In *Hardware Verification Workshop* (2010).
4. Bradley, A. R., and Manna, Z. Checking safety by inductive generalization of counterexamples to induction. In *FMCAD* (2007).
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50*, 5 (2003), 752–794.
6. Clarke, E., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, 2000.
7. Eén, N. Cut sweeping. Tech. rep., Cadence, 2007.
8. Eén, N., and Biere, A. Effective preprocessing in SAT through variable and clause elimination. In *SAT* (2005).
9. Eén, N., Mishchenko, A., and Sörensson, N. Applying logic synthesis for speeding up SAT. In *SAT* (2007), pp. 272–286.
10. Eén, N., and Sörensson, N. An extensible SAT-solver. In *SAT* (2003).
11. Eén, N., and Sörensson, N. Temporal induction by incremental SAT solving. In *BMC* (2003).
12. Floyd, R. W. Assigning meanings to programs. In *Symposia in Applied Mathematics* (1967), vol. 19, American Mathematical Society, pp. 19–32.
13. Graf, S., and Saidi, H. Construction of abstract state graphs with PVS. In *CAV* (June 1997), O. Grumberg, Ed., vol. 1254 of *LNCS*, Springer, pp. 72–83.
14. Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (October 1969), 576–580.
15. McMillan, K. L. Applying SAT methods in unbounded symbolic model checking. In *CAV* (2002), vol. 2404 of *LNCS*, Springer-Verlag, pp. 250–264.
16. McMillan, K. L. Interpolation and SAT-based model checking. In *CAV* (2003), vol. 2725 of *LNCS*, Springer, pp. 1–13.
17. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an Efficient SAT Solver. In *DAC* (2001).
18. Moura, L. D., Ruess, H., and Sorea, M. Bounded model checking and induction: From refutation to verification. In *CAV* (2003), Springer-Verlag, pp. 14–26.
19. Sheeran, M., Singh, S., and Stålmarck, G. Checking safety properties using induction and a SAT-solver. In *FMCAD* (2000), pp. 127–144.
20. Vimjam, V. C., and Hsiao, M. S. Fast illegal state identification for improving SAT-based induction. In *DAC* (2006), ACM Press, pp. 241–246.