Checking Safety by Inductive Generalization of Counterexamples to Induction

Aaron R. Bradley and Zohar Manna

Computer Science Department
Stanford University
Stanford, CA 94305-9045
Email: {arbrad, manna}@cs.stanford.edu

***Abstract*—Scaling verification to large circuits requires some form of abstraction relative to the asserted property. We describe a safety analysis of finite-state systems that generalizes from *counterexamples to the inductiveness of the safety specification* to *inductive invariants*. It thus abstracts the system's state space relative to the property. The analysis either strengthens a safety specification to be inductive or discovers a counterexample to its correctness. The analysis is easily made parallel. We provide experimental data showing how the analysis time decreases with the number of processes on several hard problems.**

I. Introduction

We describe a safety analysis for finite-state systems that incrementally strengthens a correct specification to be inductive. Each iteration of the analysis chooses a *counterexample to the inductiveness of the specification*: a state that explains why the specification is not yet inductive. It then generalizes from the assumption that this state is unreachable to an inductive invariant that proves that many states, including the counterexample to induction, are unreachable. Thus, through directed invariant generation, it accomplishes an abstraction of the state space of the system relative to the specification.

The main idea of our analysis is the following. Suppose the given safety property is not inductive. Then there exists a *counterexample to induction* (CTI): a state that can lead to a violation of the property. The analysis attempts to generate a single inductive invariant in the form of a clause that eliminates this state and many other states as well. If it succeeds, the generated inductive clause strengthens the property. Otherwise, the given property is extended to assert that the CTI state is also unreachable. After many iterations of this process, either the incrementally strengthened property becomes inductive or a counterexample trace is found. Section III presents the analysis in detail, and Section VII discusses the analysis more generally.

The application of induction as the basis for generalization is the distinguishing feature of this analysis. From one CTI, the analysis generalizes to an inductive clause that proves that many states are unreachable: the CTI state itself, all the states that can reach it, and *many other states that cannot reach it*. This generalization beyond the states that can reach the CTI contrasts with standard preimage computation.

The success of our approach depends on two points. First, we require a fast method of generating inductive clauses. We address this challenge in Section IV. Second, small inductive clauses must exist in practice. The empirical evidence of Section V suggests that they do. In Section VI, we discuss related techniques and the potential for hybrid approaches.

II. Preliminaries

A. Propositional Logic

Let us review a few useful notations and definitions of propositional logic. A *literal* ℓ is a propositional variable x or its negation $\neg x$. A *clause* c is a disjunction of literals. The size $|c|$ of clause c is its number of literals. A subclause $d \sqsubseteq c$ is a disjunction of a subset of literals of c. A formula in *conjunctive normal form* (CNF) is a conjunction of clauses.

We write $\varphi[\overline{x}]$ to indicate that the formula φ has variables $\overline{x} = \{x_1, \ldots, x_n\}$. An *assignment* s associates a truth value $\{\mathsf{true}, \mathsf{false}\}$ with each variable x_i of $\overline{x}$. $\varphi[s]$ is the truth value of φ on assignment s, and s is a *satisfying assignment* of φ if $\varphi[s]$ is true. A *partial assignment* t need not assign values to every variable.

Finally, a formula φ *implies* a formula ψ if every satisfying assignment (that assigns a truth value to every variable of φ and ψ) of φ also satisfies ψ. In this case, we say that the implication $\varphi \Rightarrow \psi$ holds. The implication $\varphi \Rightarrow \psi$ holds iff (if and only if) the formula $\varphi \wedge \neg\psi$ is unsatisfiable.

B. Transition Systems

We model finite-state systems as Boolean transition systems.

Definition 1 (Boolean Transition System) A *Boolean transition system* $\mathcal{S}\colon \langle \overline{x}, \theta, \rho \rangle$ has three components:

- a set of propositional *variables* $\overline{x} = \{x_1, \ldots, x_n\}$, which hold the state of the system;
- an *initial condition*, a propositional formula $\theta[\overline{x}]$, which describes in which states the system can start;
- and a *transition relation*, a propositional formula $\rho[\overline{x}, \overline{x}']$, which describes how the system evolves in each step of execution.

This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, and by NAVY/ONR contract N00014-03-1-0939. The first author was additionally supported by a Sang Samuel Wang Stanford Graduate Fellowship.

In $\rho[\overline{x}, \overline{x}']$, primed variables $\overline{x}'$ represent the values of the variables $\overline{x}$ in the next state.

The semantics of a finite-state system are given by its set of computations.

Definition 2 (State & Computation) A *state* s of Boolean transition system $\mathcal{S}$ is an assignment of the variables $\overline{x}$. A *computation* $\sigma : s_0, s_1, s_2, \ldots$ is a sequence of states such that

- s_0 satisfies the initial condition: $\theta[s_0]$ is true;
- and for each $i \geq 0$, s_i and s_{i+1} are related by ρ: $\rho[s_i, s_{i+1}]$ is true.

A state of $\mathcal{S}$ is *reachable* if it is in some computation of $\mathcal{S}$. We are interested in determining for a given system $\mathcal{S}$ and formula φ if every reachable state of $\mathcal{S}$ satisfies φ. If so, then φ is $\mathcal{S}$-*invariant*, or is an *invariant* of $\mathcal{S}$. If not, then there exists a computation with some state s that falsifies φ; the prefix of the computation up to s is a *counterexample trace*.

Rather than examining each of the possibly infinitely many computations of $\mathcal{S}$, however, we apply induction to determine if φ is $\mathcal{S}$-invariant.

Definition 3 (Inductive Invariant) A formula φ is $\mathcal{S}$-*inductive* if

- it holds initially: $\theta \Rightarrow \varphi$, (initiation)
- and it is preserved by ρ: $\varphi \wedge \rho \Rightarrow \varphi'$. (consecution)

In the latter formula, φ' abbreviates $\varphi[\overline{x}']$: each variable is replaced by its corresponding primed variable.

A formula φ is $\mathcal{S}$-inductive *relative to* a $\mathcal{S}$-inductive formula ψ if $\theta \Rightarrow \varphi$ as before, and $\psi \wedge \varphi \wedge \rho \Rightarrow \varphi'$.

The two implications — the base case and the inductive case, respectively — are the *verification conditions* for $\mathcal{S}$ and φ. When $\mathcal{S}$ is obvious from the context, we omit it from $\mathcal{S}$-inductive and $\mathcal{S}$-invariant.

III. Finite-State Inductive Strengthening

Given transition system $\mathcal{S} \colon \langle \overline{x}, \theta, \rho \rangle$ and specification formula Π, is Π $\mathcal{S}$-invariant? Proving that Π is inductive answers the question affirmatively. However, Π is often invariant yet not inductive. A standard approach in this case is to find a formula χ such that $\Pi \wedge \chi$ is inductive; χ is called a *strengthening assertion* [1]. There are many approaches to finding a strengthening assertion for finite-state systems (see Section VI). We describe an approach based on generating many clauses, each of which is inductive relative to the previously-generated clauses.

A. Inductive Clauses

First, let us consider how to find a single clause that is inductive relative to some formula ψ. Later, we show how to chain the discovery of such clauses together to decide whether a system $\mathcal{S}$ meets its specification Π. Our presentation is self-contained but follows the ideas of abstract interpretation [2].

Consider an arbitrary clause c that need not be inductive. It induces the *subclause lattice* $L_c : \langle 2^c, \sqcap, \sqcup, \sqsubseteq \rangle$ in which

- the elements 2^c are the subclauses of c;
- the elements are ordered by the subclause relation $\sqsubseteq$: in particular, the top element is c itself, and the bottom element is the formula false ("false");
- the join operator $\sqcup$ is simply disjunction;
- the meet operator $\sqcap$ is defined as follows: $c_1 \sqcap c_2$ is the disjunction of the literals common to c_1 and c_2.

L_c is complete; by the Knaster-Tarski theorem [3], [4], every monotone function on L_c has a least and a greatest fixpoint.

Consider the monotone nonincreasing function $\mathsf{down}(L_c, d)$ that, given the subclause lattice L_c and the clause $d \in 2^c$, returns the (unique) largest subclause $e \sqsubseteq d$ such that the implication $\psi \wedge e \wedge \rho \Rightarrow d'$ holds. In other words, it returns the best under-approximation in L_c to the weakest precondition of d. If the greatest fixpoint $\bar{c}$ of $\mathsf{down}(L_c, c)$ satisfies the implication $\theta \Rightarrow \bar{c}$ of initiation, then it is the largest subclause of c that is inductive relative to ψ. Section IV-A describes how to find $\bar{c}$ with a number of satisfiability queries linear in $|c|$.

Large inductive clauses are undesirable, however, because they provide less information than smaller clauses: they are satisfied by more states. We want to find a *minimal inductive subclause* of c, an inductive subclause that does not itself contain any strict subclause that is inductive. Therefore, we examine a fixpoint of an *extensive* function on *inductive* subclause lattices, which are lattices whose top elements are inductive. Constructing an inductive subclause lattice requires first computing the greatest fixpoint of down on a larger subclause lattice.

To that end, consider the (nondeterministic) function $\mathsf{implicate}(\varphi, c)$ that, given formula φ and clause c, returns a minimal subclause $d \sqsubseteq c$ such that $\varphi \Rightarrow d$ if such a clause exists, and returns $\top$ ("true") otherwise. This minimal subclause d is known as a *prime implicate* [5], [6]. There can be exponentially many such minimal subclauses, yet there may not be any prime implicate if $\varphi \not\Rightarrow c$. Section IV-B presents an optimal implementation of $\mathsf{implicate}$.

Using $\mathsf{implicate}$, we can find a subclause of c that best approximates θ: $b : \mathsf{implicate}(\theta, c)$. Consider the subclause *sublattice* $L_{b,c}$ of L_c that has top element c and bottom element b. Consider also the extensive operation $\mathsf{up}(L_{b,c}, d)$ that, for element d of $L_{b,c}$, returns a minimal subclause $e \sqsubseteq c$ such that the implications $\psi \wedge d \wedge \rho \Rightarrow e'$ and $d \Rightarrow e$ hold. In other words, it computes $e' : \mathsf{implicate}((\psi \wedge d \wedge \rho) \vee d',\ c')$, a best over-approximation in $L_{b,c}$ of the disjunction of d and the strongest postcondition of d.

The operation up is a function on $L_{b,c}$ — it maps an element of $L_{b,c}$ to an element of $L_{b,c}$ — precisely when the top element c is inductive. In this case, a fixpoint $\bar{c}$ is an inductive subclause of c that is small in practice. However, it is not necessarily a minimal inductive subclause, as different runs of $\mathsf{implicate}$ result in different fixpoints, some of which may be strict subclauses of others.

Now for a given clause c, compute the greatest fixpoint of down on L_c to discover inductive clause $\bar{c}$ and its corresponding inductive sublattice $L_{\bar{c}}$. Compute $b : \mathsf{implicate}(\theta, c)$ to identify the inductive sublattice $L_{b,\bar{c}}$ whose bottom element over-approximates θ. Finally, compute a fixpoint of up on $L_{b,\bar{c}}$ to find a small inductive subclause $\bar{d}$ of c. In practice, $\bar{d}$ is small but need not be minimal.

A brute-force recursive technique finds a minimal inductive subclause. First apply the procedure described above to c to find $\bar{d}$. Then recursively treat each immediate strict subclause of $\bar{d}$, of which there are $|\bar{d}|$. A clause $\bar{d}$ is a minimal inductive subclause of c precisely when each of these recursive calls fails to find an inductive strict subclause of $\bar{d}$. We call this procedure $\mathsf{MIC}(\mathcal{S}, \psi, c)$. It returns a minimal subclause of c that is $\mathcal{S}$-inductive relative to ψ, or $\top$ if no such clause exists.

B. FSIS: Generalizing from Counterexamples to Induction

Having developed the algorithm MIC to discover a minimal inductive subclause of a given clause c, the remaining consideration is which clause c to provide it. We use negations of *counterexamples to induction* (CTIs): Π-states that can lead to $\neg\Pi$-states. This choice guides MIC to discover inductive clauses that are relevant for proving that Π is $\mathcal{S}$-invariant, implicitly abstracting the state-space of $\mathcal{S}$ relative to Π. Hence, rather than finding a CNF representation of the exact set of reachable states of $\mathcal{S}$, we expect to find a much smaller formula χ such that $\Pi \wedge \chi$ is inductive and represents a larger set of states that all satisfy Π.

Suppose that χ is a conjunction of previously-generated clauses that are $\mathcal{S}$-inductive relative to Π, but $\Pi \wedge \chi$ is not $\mathcal{S}$-inductive. Why is $\Pi \wedge \chi$ not inductive? One possibility is that initiation fails: the formula $\theta \wedge \neg\Pi$ is satisfied by some state s. In this case, Π is not $\mathcal{S}$-invariant.

Another possibility is that consecution fails: the formula $\Pi \wedge \chi \wedge \rho \wedge \neg\Pi'$ is satisfied by some pair of states (s, s'). That is, it is possible for $\mathcal{S}$ to transition from state s, which satisfies $\Pi \wedge \chi$, to state s', which satisfies $\neg\Pi'$ and thus violates the specification Π. State s is a CTI.

Since s is an assignment of truth values to variables of $\mathcal{S}$, it can be viewed as a conjunction of literals: if s assigns x_i to be true, the conjunction contains x_i; if s assigns x_i to be false, the conjunction contains $\neg x_i$. Call this conjunction $\hat{s}$. Now, noticing that $\neg\hat{s}$ is a clause, compute $\mathsf{MIC}(\mathcal{S},\ \Pi \wedge \chi,\ \neg\hat{s})$. If MIC returns $\top$ because $\hat{s}$ does not contain a subclause that is inductive relative to $\Pi \wedge \chi$, then update Π to indicate that proving that state s is unreachable is a subgoal of proving the invariance of the original Π: $\Pi := \Pi \wedge \neg\hat{s}$. Otherwise, MIC returns $\bar{c}$, an inductive generalization of $\neg\hat{s}$: $\bar{c}$ is inductive relative to $\Pi \wedge \chi$ and excludes state s and many other states. Update χ accordingly: $\chi := \chi \wedge c$.

Eventually, either Π grows so that $\theta \wedge \neg\Pi$ is satisfiable, disproving the specification, or $\Pi \wedge \chi$ becomes inductive. For χ is always inductive relative to Π; and if the formula $\Pi \wedge \chi \wedge \rho \wedge \neg\Pi'$ is unsatisfiable, then Π is inductive relative to χ.

Which states does an inductive generalization of $\neg\hat{s}$ exclude? It clearly excludes state s. It also excludes all states that can reach s, for it would not be inductive otherwise. However, even the largest inductive subclause of $\neg\hat{s}$ excludes all states that can reach s, while MIC discovers significantly smaller inductive subclauses in practice. Hence, MIC generalizes the argument that s and those states that can reach s are unreachable to prove that many other states are also unreachable. This generalization contrasts with methods that compute the preimage of a CTI [7]–[9].

This analysis is naturally made parallel. By simply using a randomized decision procedure to obtain the CTIs, each process is likely to analyze a different part of the state-space. Processes need only communicate discovered inductive clauses and — depending on implementation choices (see Section V) — CTIs that do not yield inductive clauses.

We call this procedure $\mathsf{FSIS}(\mathcal{S}, \Pi)$, for *finite-state inductive strengthening*. It returns an inductive strengthening of Π if Π is $\mathcal{S}$-invariant; otherwise, it extracts from the subgoals conjoined to Π a counterexample trace.

C. One-Step Cone of Influence

One need not consider a full assignment s as a CTI. Computing a *one-step cone of influence* (COI) — the variables that can possibly impact the truth value of Π in the next state — yields a partial assignment t that describes a set of states, rather than just a single state s, that can lead to a violation of Π. Applying MIC to the resulting clause $\neg\hat{t}$ focuses it on excluding all of these states, rather than proving that just the state s is unreachable for a reason that is unrelated to why the other states are unreachable.

IV. Algorithms

This section develops the algorithms introduced in Section III-A to discover minimal inductive subclauses.

A. Computing the Largest Inductive Subclause

Recall that the function $\mathsf{down}(L_c, d)$ computes the largest subclause $e \sqsubseteq d$ such that the implication $\psi \wedge e \wedge \rho \Rightarrow d'$ holds. A straightforward method of computing the greatest fixpoint of down in L_c — which iteratively computes under-approximations to the weakest precondition — can require $\Omega(|c|^2)$ satisfiability queries. For systems with many variables, this quadratic cost is prohibitively expensive.

We describe a method that requires a linear number of queries. Consider checking if the implication $\psi \wedge c \wedge \rho \Rightarrow c'$ holds. If it does, and if the implication $\theta \Rightarrow c$ of initiation also holds, then c is inductive relative to ψ. If it does not, then the formula $\psi \wedge c \wedge \rho \wedge \neg c'$ is satisfied by some assignment (s, s') to the unprimed and primed variables. Let $\hat{s}$ be the conjunction of literals corresponding to s; and let $\neg\hat{t}$ be the best over-approximation to $\neg\hat{s}$ in L_c, which is the largest clause with literals common to c and $\neg\hat{s}$. Then compute the new clause $d : c \sqcap \neg\hat{t}$. In other words, d has the literals common to c and $\neg\hat{s}$. Now recurse on d.

If at any point during the computation, initiation does not hold, then report failure.

This algorithm, which we call $\mathsf{LIC}(L_c, c)$, computes the largest inductive subclause of the given clause c.

Proposition 1 (Largest Inductive Subclause) *The fixpoint of the iteration sequence computed by* $\mathsf{LIC}(L_c, c)$ *is the* *largest subclause of c that satisfies consecution. If it also satisfies initiation, then it is the largest inductive subclause of c. Finding it requires solving at most $O(|c|)$ satisfiability queries.*

Proof: Let the computed sequence be $c_0 = c,\ c_1,\ \ldots, c_k$, where the fixpoint c_k satisfies consecution. Notice that for each $i > 0$, $c_i \sqsubseteq c_{i-1}$ by construction. Suppose that $e \sqsubseteq c$ also satisfies consecution, yet it is not a subclause of c_k. We derive a contradiction.

Consider position i at which $e \sqsubseteq c_i$ but $e \not\sqsubseteq c_{i+1}$; such a position must occur by the existence of e. Now partition c_i into two clauses, $e \vee f$; f contains the literals of c_i that are not literals of e. Since consecution does not yet hold for c_i, the formula $\psi \wedge (e \vee f) \wedge \rho \wedge \neg(e' \vee f')$ is satisfiable. Case splitting, one of the following two formulae is satisfiable:

$$\psi \wedge e \wedge \rho \wedge \neg e' \wedge \neg f' \tag{1}$$

$$\psi \wedge \neg e \wedge f \wedge \rho \wedge \neg e' \wedge \neg f' \tag{2}$$

Formula (1) is unsatisfiable because e satisfies consecution by assumption. Therefore, formula (2) is satisfied by some assignment (s, s'). Now, because $\neg e[s]$ evaluates to true, we know that $e \sqsubseteq \neg\hat{s}$ (where $\hat{s}$ is the conjunction of literals corresponding to assignment s); but then $e \sqsubseteq c_{i+1} = c_i \sqcap \neg\hat{s}$, a contradiction.

The linear bound on the number of satisfiability queries follows from the observation that each iteration (other than the final one) must drop at least one literal of c. ■

We thus have an algorithm for computing the largest inductive subclause of a given clause with only a linear number of satisfiability queries.

In practice, during one execution of MIC, the clauses that are found not to contain inductive clauses should be cached to preclude the future exploration of its subclauses.

B. Computing Prime Implicates

Recall that the function $\mathsf{up}(L_c, d)$ computes a minimal subclause $e \sqsubseteq c$ such that the implication $(\psi \wedge d \wedge \rho) \vee d' \Rightarrow e'$ holds. As explained in Section III-A, the crucial part of implementing up is implementing an algorithm for finding minimal implicates: $\mathsf{implicate}(\varphi, c)$ should return a minimal subclause of c such that $\varphi \Rightarrow c$ holds, or $\top$ if no such subclause exists. We focus on implicate in this section.

In fact, we consider a more general problem. Consider a set of objects S and a predicate $p : 2^S \mapsto \{\mathsf{true}, \mathsf{false}\}$ that is monotone on S: if $p(S_0)$ is true and $S_0 \subseteq S_1 \subseteq S$, then also $p(S_1)$ is true. We assume that $p(S)$ is true; this assumption can be checked with one preliminary query. The problem is to find a minimal subset $\bar{S} \subseteq S$ that satisfies p: $p(\bar{S})$.

The correspondence between this general problem and $\mathsf{implicate}(\varphi, c)$ is direct: let S be the set of literals of c and p be the predicate that is true for $S_0 \subseteq S$ precisely when $\varphi \Rightarrow \bigvee S_0$, where $\bigvee S_0$ is the disjunction of the literals of S_0.

A straightforward and well-known algorithm for finding a minimal satisfying subset of S requires a linear number of queries to p. Drop an element of the given set. If the remaining set satisfies p, recurse on it; otherwise, recurse on the given set, remembering never to drop that element again.

```
let rec min p S₀ = function
  | []     → S₀
  | h :: t → if p(S₀ ∪ t)
             then min p S₀ t
             else min p (h :: S₀) t
let minimal p S = min p [] S
```

Fig. 1. Linear-time minimal

```
let rec split (ℓ, r) = function
  | []           → (ℓ, r)
  | h :: []      → (h :: ℓ, r)
  | h₁ :: h₂ :: t → split (h₁ :: ℓ, h₂ :: r) t
let split S₀ = split ([], []) S₀
let rec min p sup S₀ =
  if |S₀| = 1
  then S₀
  else let ℓ₀, r₀ = split S₀ in
       if p(sup ∪ ℓ₀)
       then min p sup ℓ₀
       else if p(sup ∪ r₀)
            then min p sup r₀
            else let ℓ = min p (sup ∪ r₀) ℓ₀ in
                 let r = min p (sup ∪ ℓ) r₀ in
                 ℓ ∪ r
let minimal p S = min p [] S
```

Fig. 2. Optimal minimal

Figure 1 describes this algorithm precisely using an O'Caml-like language. It treats sets as lists. S_0 contains the required elements of S that have already been examined; if there are not any remaining elements, return S_0. Otherwise, the remaining elements consist of $h :: t$ — a distinguished element h (the "head") and the other elements t (the "tail"). If $p(S_0 \cup t)$ is true, h is unnecessary; otherwise, it is necessary, so add it to S_0. We provide these details to prepare the reader to understand an algorithm that makes the optimal number of queries to p.

We can do exponentially better than always making a linear number of queries to p. Suppose we are given two disjoint sets, the "support" set sup and the set S_0, such that $p(sup \cup S_0)$ holds but $p(sup)$ does not hold. We want to find a minimal subset $\bar{S} \subseteq S_0$ such that $p(sup \cup \bar{S})$ holds. If S_0 has just one element, then that one element is definitely necessary, so return it. Otherwise, split S_0 into two disjoint subsets ℓ_0 and r_0 with roughly half the elements each (see Figure 2 for a precise description of split). Now if $p(sup \cup \ell_0)$ is true, immediately recurse on ℓ_0, using sup again as the support set. If not, but $p(sup \cup r_0)$ is true, then recurse on r_0, using sup again as the support set.

The interesting case occurs when neither $p(sup \cup \ell_0)$ nor $p(sup \cup r_0)$ hold: in this case, elements are required from both ℓ_0 and r_0. First, recurse on ℓ_0 using $sup \cup r_0$ as the support set. The returned set ℓ is a minimal subset of ℓ_0 that is necessary

@pre $p(sup \cup S_0) \;\wedge\; \neg p(sup)$
@post $V \subseteq S_0 \;\wedge\; p(sup \cup V)$
$\quad\quad \wedge\; \forall e \in V.\ \neg p(sup \cup V \setminus \{e\})$
`let rec` min $p\ sup\ S_0\ =$

Fig. 3. Annotated prototype of min, where V is the return value

for $p(sup \cup \ell \cup r_0)$ to hold. Second, recurse on r_0 using $sup \cup \ell$ (note: ℓ, not ℓ_0) as the support set. The returned set r is a minimal subset of r_0 that is necessary for $p(sup \cup \ell \cup r)$ to hold. Finally, return $\ell \cup r$, which is a minimal subset of S_0 for $p(sup \cup \ell \cup r)$ to hold.

Figure 2 gives a precise definition of this algorithm. To find a minimal subset of S that satisfies p, min is initially called with an empty support set ([]) and S.

Theorem 1 (Correct) *Suppose that S is nonempty, $p(S)$ is* true*, and $p(\emptyset)$ is* false*.*

1) min p [] S *terminates.*
2) *Let $\bar{S} =$* min p [] S*. Then $p(\bar{S})$ is* true*, and for each $e \in \bar{S}$, $p(\bar{S} \setminus \{e\})$ is* false*.*

Proof: The first claim is easy to prove: each level of recursion operates on a finite nonempty set that is smaller than the set in the calling context.

For the second claim, we make an inductive argument of correctness. We prove first that $p(\bar{S})$ is true. We then prove that for each $e \in \bar{S}$, $p(\bar{S} \setminus \{e\})$ is false. To prove these claims, we prove that five assertions are inductive for min. These assertions are summarized as function preconditions and function postconditions of min in Figure 3. Throughout the proof, let $V =$ min $p\ sup\ S_0$ be the return value.

For the first part of the second claim, we establish the following invariants:

1) $p(sup \cup S_0)$
2) $p(sup \cup V)$

Invariant (1) is a function precondition of min; invariant (2) is a function postcondition of min. Hence, the inductive argument for (1) establishes that it always holds upon entry to min, while the inductive argument for (2) establishes that it always holds upon return of min.

Invariants (1) and (2) are proved simultaneously. For the base case of (1), note that $p(\emptyset \cup S) = p(S)$, which is true by assumption. For the inductive case, consider that $p(sup \cup \ell_0)$ and $p(sup \cup r_0)$ are checked before the first two recursive calls; that $sup \cup r_0 \cup \ell_0 = sup \cup S_0$ for the third recursive call; and that $p(sup \cup r_0 \cup \ell)$ is true by invariant (2).

For the base case of invariant (2), we know at the first return of min that $p(sup \cup S_0)$ from invariant (1), and $V = S_0$. For the inductive case, consider that (2) holds at the next two returns by the inductive hypothesis; and that at the fourth return, $p(sup \cup \ell \cup r)$ holds by the inductive hypothesis of the prior line.

In the first call to min in minimal, $sup = \emptyset$; hence, $p(\bar{S}) = p(\emptyset \cup \bar{S}) =$ true by invariant (2).

To prove that $\bar{S}$ is minimal (that for each $e \in \bar{S}$, $p(\bar{S} \setminus \{e\})$ is false) for the second part of the second claim, consider the following invariants:

3) $V \subseteq S_0$
4) $\neg p(sup)$
5) $\neg p(sup \cup V \setminus \{e\})$ for $e \in V$

Invariant (4) is a function precondition, and invariants (3) and (5) are function postconditions.

For invariant (3), note for the base case that the first return of min returns $V = S_0$ itself; that the next two returns hold by inductive hypothesis; that $\ell \subseteq \ell_0$ and $r \subseteq r_0$ by inductive hypothesis; and, thus, that $V = \ell \cup r \subseteq \ell_0 \cup r_0 = S_0$.

For the base case of invariant (4), consider that $\neg p(\emptyset)$ by assumption. For the inductive case, consider that the first two recursive calls have the same sup as in the calling context and thus (4) holds by inductive hypothesis; that at the third recursive call, $\neg p(sup \cup r_0)$; and that at the fourth recursive call, $\neg p(sup \cup \ell_0)$ and, from (3), that $\ell \subseteq \ell_0$, so that $\neg p(sup \cup \ell)$ follows from monotonicity of p.

For the base case of invariant (5), consider that at the first return, $\neg p(sup)$ by invariant (4). Hence, the one element of S_0 is necessary. The next two returns hold by the inductive hypothesis. For the final return, we know by the inductive hypothesis that $\neg p(sup \cup \ell \cup r \setminus \{e\})$ for $e \in r$; hence, all of r is necessary. Additionally, from the inductive hypothesis, $\neg p(sup \cup r_0 \cup \ell \setminus \{e\})$ for $e \in \ell$, and $\neg p(sup \cup r_0 \cup \ell \setminus \{e\})$ implies that $\neg p(sup \cup r \cup \ell \setminus \{e\})$ by monotonicity of p and because $r \subseteq r_0$ by invariant (3); hence, all of ℓ is necessary.

In the first call to min at minimal, sup $= \emptyset$ and $V = \bar{S}$; hence, $\neg p(\bar{S} \setminus \{e\})$ for $e \in \bar{S}$ from invariant (5). ■

Theorem 2 (Upper Bound) *Let $\bar{S} =$* min p [] S*. Discovering $\bar{S}$ requires making $O\left((|\bar{S}| - 1) + |\bar{S}|\ lg\ \frac{|S|}{|\bar{S}|}\right)$ queries to p.*

Proof: Suppose that $|\bar{S}| = 2^k$ and $|S| = n2^k$ for some $k, n > 0$. Each element of $\bar{S}$ induces one divergence at some level in the recursion. At worst, these divergences occur evenly distributed at the beginning, inducing $|\bar{S}|$ separate binary searches over sets of size $\frac{|S|}{|\bar{S}|}$. Hence, $|\bar{S}| - 1$ calls to min diverge, while $|\bar{S}| \lg \frac{|S|}{|\bar{S}|}$ calls behave like in a binary search. Noting that each call results in at most two queries to p, we have the claimed upper bound in this special case, which is also an upper bound for the general case. (Adding sufficient "dummy" elements to construct the special case does not change the asymptotic bound.) ■

For studying the lower bound on the complexity of the problem, suppose that S has precisely one minimal satisfying subset.

Theorem 3 (Lower Bound) *Any algorithm for determining the minimal satisfying subset $\bar{S}$ of S must make $\Omega\left(|\bar{S}| + |\bar{S}| \lg \left(\frac{|S| - |\bar{S}|}{|\bar{S}|}\right)\right)$ queries to p.*

Proof: For the linear component, $|\bar{S}|$, consider deciding whether $\bar{S}$ is indeed minimal. Since all that is known is that p

```
let rec min f sup S_0 =
  if |S_0| = 1
  then (sup, S_0)
  else let ℓ_0, r_0 = split S_0 in
       let v, C = f(sup ∪ ℓ_0) in
       if v
       then min f (sup ∩ C) (ℓ_0 ∩ C)
       else let v, C = f(sup ∪ r_0) in
            if v
            then min f (sup ∩ C) (r_0 ∩ C)
            else let C, ℓ = min f (sup ∪ r_0) ℓ_0 in
                 let sup = sup ∩ C in
                 let r_0 = r_0 ∩ C in
                 let C, r = min f (sup ∪ ℓ) r_0 in
                 (sup ∩ C, (ℓ ∩ C) ∪ r)
let minimal f S =
  let _, S_0 = min f [] S in S_0
```

Fig. 4. Optimal minimal with additional information

is monotone over S, the information that $p(S_0)$ is false does not reveal any information about $p(S_1)$ when $S_1 \setminus S_0 \neq \emptyset$. Therefore, p must be queried for each of the $|\bar{S}|$ immediate strict subsets of $\bar{S}$.

For the other component, consider that any algorithm must be able to distinguish among $C(|S|, |\bar{S}|) = \frac{|S|!}{|\bar{S}|!(|S|-|\bar{S}|)!}$ possible results using only queries to p. Thus, the height of a decision tree must be at least $\lg C(|S|, |\bar{S}|)$. Using Stirling's approximation,

$$\begin{aligned} \lg \frac{|S|!}{|\bar{S}|!(|S|-|\bar{S}|)!} &\geq \lg |S|! - \lg |\bar{S}|! - \lg(|S| - |\bar{S}|)! \\ &\quad - o(\lg |\bar{S}| + \lg(|S| - |\bar{S}|)) \\ &= \Omega\left(|S| \lg\left(\frac{|S|}{|S|-|\bar{S}|}\right) + |\bar{S}| \lg\left(\frac{|S|-|\bar{S}|}{|\bar{S}|}\right)\right) . \end{aligned}$$

■

Hence, the algorithm is in some sense optimal. However, a set can have a number of minimal subsets exponential in its size. In this situation, the lower bound analysis does not apply.

In practice, one can often glean more information when executing the predicate p than just whether it is satisfied by the given set. For example, a decision procedure for propositional satisfiability (a "SAT solver") can return an *unsatisfiable core*. Hence, if $\psi \Rightarrow c$ holds ($\psi \wedge \neg c$ is unsatisfiable), the procedure might return a subclause $d \sqsubseteq c$ such that $\psi \Rightarrow d$ also holds. However, d need not be minimal. The algorithm of Figure 4 incorporates this extra information. Rather than a predicate p, it accepts a function f that returns two values: $f(S)$ returns the same truth value as $p(S)$; and if $p(S)$ is true, it also returns a subset $S_0 \sqsubseteq S$ such that $p(S_0)$ holds. This subset is used to prune sets appropriately. Additionally, min returns both the minimal set and a pruned support set to use on the other branch of recursion.

V. Experiments

A. Implementation

We implemented our analysis in O'Caml. We discuss important elements of our implementation.

1) SAT Solver: We instrumented `Z-Chaff` version "2004.11.15 Simplified" [10] to return original unit clauses that are leaves of the implication graph to aid in computing minimal implicates. We also refined its memory usage to allow tens of thousands of incremental calls. For parallel executions, we tuned `Z-chaff` to randomize some of its choices.

Conversion to CNF is minimized by caching the CNF version of the transition system within the SAT solver. Also, multiple versions of the transition relation are stored; each version corresponds to a particular slicing of the relation according to the one-step cone of influence.

2) Depth-First Search: Our implementation takes a depth-first approach: if it fails to find an inductive clause excluding a CTI, it focuses on this subgoal before again considering the rest of the given property.

3) Parallel Algorithm: Each process works mostly independently, relying on the randomness of the SAT solver to focus on different regions of the possible state space of the system. Upon discovery of an inductive clause, a process reports it to a central server and receives all other inductive clauses discovered by other processes since its last report. Because of the depth-first treatment of counterexample states, a process can report that a clause is inductive *under the assumption that subgoal states are unreachable*. If this assumption is incorrect, the process eventually discovers a counterexample trace. Otherwise, it eventually justifies this assumption with additional inductive clauses. However, other processes may finish before receiving these additional clauses. Hence, because only the last process to terminate receives all clauses, it is the only process that is guaranteed to have an inductive strengthening of the safety property.

B. Benchmarks

1) PicoJava II Set: We applied our analysis to the PicoJava II microprocessor benchmark set, previously studied in [11]–[13]. Each benchmark asserts a safety property about the instruction cache unit (ICU) — which manages the instruction cache, prefetches instructions, and partially decodes instructions — but includes the implementation of both the ICU and the instruction folding unit (IFU), which parses the byte stream from the instruction queue into instructions and divides them into groups for execution within a cycle. Including the IFU increases the number of variables in the cone-of-influence (COI) and complicates the combinatorial logic. Hence, for example, a static COI analysis is unhelpful. Of the 20 benchmarks, proof-based abstraction solved 18 [11] (it exhausted the available 512MB of memory on problems PJ_{17} and PJ_{18}), and interpolation-based model checking solved 19 [12], [13], each within their allotted times of 1000 seconds on 930MHz machines.

2) VIS Set: The second set of benchmarks are from the VIS distribution [14]. We applied the analysis to several valid properties of models that are difficult for standard k-induction (although easy for standard BDD-based model checking) [9]. k-induction with strengthening fails on Peterson and Heap

TABLE I
RESULTS FOR ONE PROCESS

Name	COI	Clauses	SAT queries	Time	Mem (MB)
PJ_2	306	6 (2)	202 (64)	38s (3s)	212 (9)
PJ_3	306	6 (3)	201 (78)	37s (3s)	213 (9)
PJ_5	88	159 (27)	12K (3.4K)	30s (9s)	50 (3)
PJ_6	318	414 (85)	32K (7.5K)	**1h30m (22m)**	589 (39)
PJ_7	67	63 (9)	4K (1K)	10s (2s)	41 (3)
PJ_8	90	70 (8)	3.5K (.8K)	13s (3s)	43 (3)
PJ_9	46	27 (5)	1K (.2K)	4s (1s)	35 (2)
PJ_{10}	54	6 (3)	213 (110)	6s (1s)	48 (1)
PJ_{13}	352	8 (6)	234 (149)	2m45s (1m9s)	379 (15)
PJ_{15}	353	145 (68)	6K (3.5K)	30m (17m)	493 (79)
PJ_{16}	290	241 (186)	18K (22K)	50m (1h10m)	539 (96)
PJ_{17}	211	1.2K (153)	337K (51K)	**16h20m (3h)**	1250 (110)
PJ_{18}	143	740 (152)	91K (23K)	**2h40m (50m)**	673 (83)
PJ_{19}	52	83 (11)	4K (.4K)	11m (5m)	237 (31)
PC_1	93	7 (4)	170 (105)	2m48s (1m)	360 (12)
PC_2	91	3 (0)	42 (1)	51s (4s)	335 (1)
PC_5	91	3 (0)	42 (1)	53s (4s)	335 (1)
PC_6	91	9 (4)	229 (109)	3m25s (1m18s)	377 (13)
PC_{10}	91	21 (10)	598 (260)	5m35s (1m47s)	370 (8)
HEAP	30	2.6K (237)	58K (60K)	**4h20m (45m)**	330 (25)
PET	16	4 (0)	140 (11)	2s (0s)	44 (0)

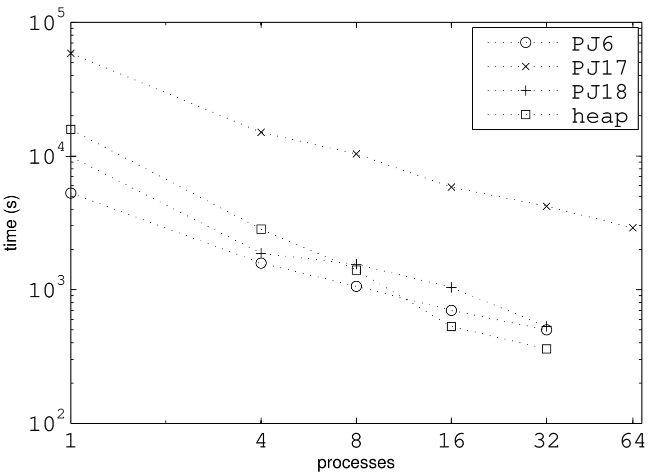

Fig. 5. Time for multiple processes

within 1800 seconds; but BDD-based model checking requires at most a few seconds for each [9].

C. Results

Table I reports results for executing one process on one processor of a 4×1.8GHz computer with 8GB of available memory. The analysis ran 16 times on each benchmark: Table I reports the number of variables in the cone of influence and the mean and standard deviation (in format *mean* (*std. dev.*)) for the number of discovered clauses, the number of SAT queries made, the required time, and the peak memory usage. Results are reported only for the nontrivial benchmarks: properties of benchmarks 0, 1, 4, 11, 12, and 14 of the PicoJava II set and benchmarks 3, 4, 7, 8, and 9 of the VIS PPC60x_2 set are inductive. The PicoJava II benchmarks are labeled PJ_x; the others are VIS benchmarks. All 20 of the PicoJava II benchmarks were solved; three required more than one hour.

Figure 5 reports results as a log-log plot for analyzing PicoJava II benchmarks 6, 17, and 18 and VIS benchmark HEAP with multiple processes on a cluster of computers with 4×1.8GHz processors and 8GB of memory. Results for one processor are the means from Table I. Times for 32 processes are as follows: PJ_6, 8m; PJ_{17}, 70m; PJ_{18}, 9m; and HEAP, 6m. PJ_{17} completed in 50m with 60 processes. All benchmarks completed within one hour with some number of processes.

The plot suggests that time decreases roughly linearly with more processes, but only HEAP trades processes for time almost perfectly, possibly because it requires the most clauses. Suboptimal scaling results from generating redundant clauses.

VI. RELATED WORK

A. Qualitative Comparisons

We compare the characteristics of several safety analyses: bounded model checking (BMC) [15], interpolation-based model checking (IMC) [12], [13], k-induction (kI) [7]–[9], [16], [17], predicate abstraction with refinement (CEGAR) [18], [19], and our analysis (FSIS). These analyses are fundamentally based on computing an inductive set that excludes all error states; they consider the property to prove during the computation; and they use a SAT solver as the main computational resource.

We now consider their differences.

1) Abstraction: IMC and CEGAR compute successively finer approximations to the transition relation. Each approximation causes a certain set of states to be deemed reachable. When this set includes an error state, IMC increments the k associated with its postcondition operator, solving larger BMC problems, while CEGAR learns a separating predicate. In contrast, BMC, kI, and FSIS operate on the full transition relation. kI strengthens by requiring counterexamples to induction to be ever longer paths. FSIS generalizes from CTIs to inductive clauses to exclude portions of the state space.

2) Use of SAT Solver: BMC, IMC, and kI pose relatively few but difficult SAT problems in which the transition relation is unrolled many times. CEGAR and FSIS pose many simple SAT problems in which the transition relation is not unrolled.

3) Intermediate Results: Each major iteration of IMC and CEGAR produces an inductive set that is informative even when it is not strong enough to prove the property. Each successive iteration of FSIS produces a stronger formula that excludes states that cannot be reached without previously violating the property. Intermediate iterations of BMC and kI are not useful, although exceptions include forms of strengthening, which we discuss in greater depth below [7]–[9], [17].

4) Parallelizable: Only FSIS is natural to make parallel. The difficulty of subproblems grows with successive iterations in BMC, IMC, and kI so that parallelizing across iterations is not useful. Each iteration of CEGAR depends on previously learned predicates. For these analyses, parallelization must be implemented at a lower level, perhaps in the SAT solver.

Differences suggest ways to combine techniques. For example, the key methods of FSIS and kI can be combined, and FSIS can serve as the model checker for CEGAR.

B. Other Related Work

Blocking clauses are used in SAT-based unbounded model checking [5]. Their discovery is refined to produce *prime* blocking clauses, requiring at worst as many SAT calls as literals [6]. Our minimal algorithm requires asymptotically fewer SAT calls. A similar algorithm has been proposed in a different context [20], but it handles only sets containing precisely one minimal satisfying subset.

Strengthening based on under-approximating the states that can reach a violating state s is applied in the context of k-induction [7]–[9], [17]. Quantifier-elimination [7], ATPG-based computation of the n-level preimage of s [8], and SAT-based preimage computation [9] are used to perform the strengthening. Inductive generalization can eliminate exponentially more states than preimage-based approaches.

VII. Discussion

Let us consider the methods of this paper more generally. The fundamental idea is to generalize from counterexamples to induction (CTIs) to simple inductive invariants. Together, the set of simple inductive invariants strengthens the specification to be inductive. Limiting the form of invariants controls computational costs, while using CTIs focuses the analysis on the safety specification. The structure of the analysis allows a parallel implementation.

Two questions are immediate. What are the CTIs? What is the abstract domain for invariant generation? When the invariant generation is based on propagation, as in this paper, these questions are linked: the abstract domain should be conjunctive *with respect to the CTI* so that the best over-approximation to the CTI in the domain is sufficiently precise.

For example, in FSIS, the CTIs are (partial) states that can lead to violations of the given property; and the domain consists of clauses of system variables. Clauses are conjunctive with respect to states like CTIs that ought to be unreachable. We thus start with Π and conjoin invariant clauses to exclude error states until CTIs no longer exist.

As another example, consider the dual analysis in which the set of reachable states is grown until it is inductive without including any $\neg\Pi$-states. Now, the CTIs are (partial) states that are reachable in one step from the currently reachable set; and the abstract domain is cubes, which are conjunctions of literals. Hence, the invariant cubes are combined through disjunction to grow the reachable set. Each round of invariant generation discovers a *minimal inductive subcube* of the cube defined by the CTI that includes only Π-states.

In another application, we explored inductive generalization from CTIs to affine inequalities [21]. In the domain of the analysis, invariant generation is not based on propagation.

Once the form of the CTI and the abstract domain are fixed, one desires to find a greatest inductive generalization to each CTI. Standard techniques suggest how to perform one direction of propagation in the abstract domain [2]. However, the other direction must suffer from the nondeterminism inherent in over- (under-) approximating in a disjunctive (conjunctive) domain, so that a greatest inductive generalization to the CTI need not be unique. For example, implicate is nondeterministic; and in the dual analysis, computing a best *implicant* is nondeterministic. The function minimal of Section IV-B is a general operator for performing forward (backward) propagation in disjunctive (conjunctive) domains.

This general perspective on the ideas of this paper suggests further work in the form of exploring other domains. Additionally, we intend to combine the method with k-induction. Finally, our analysis is motivated by a classically deductive approach to verification [1]. We are exploring analyses for other classes of temporal properties that are also motivated by classically deductive techniques.

Acknowledgments

The authors wish to thank Prof. A. Aiken, A. M. Bradley, Prof. E. Clarke, Prof. D. Dill, Dr. A. Gupta, Dr. H. Sipma, Prof. F. Somenzi, and the anonymous reviewers for their comments; and Prof. A. Aiken for the use of his computer cluster.

References

[1] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York: Springer-Verlag, 1995.

[2] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM Press, 1977, pp. 238–252.

[3] B. Knaster, "Un theoreme sur les fonctions d'ensembles," *Ann. Soc. Polon. Math.*, vol. 6, pp. 133–134, 1928.

[4] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[5] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking." in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 250–264.

[6] H. Jin and F. Somenzi, "Prime clauses for fast enumeration of satisfying assignments to boolean circuits," in *DAC*. ACM Press, 2005.

[7] L. de Moura, H. Ruess, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *CAV*, ser. LNCS. Springer, 2003.

[8] V. C. Vimjam and M. S. Hsiao, "Fast illegal state identification for improving SAT-based induction," in *DAC*. ACM Press, 2006.

[9] M. Awedh and F. Somenzi, "Automatic invariant strengthening to prove properties in bounded model checking," in *DAC*. ACM Press, 2006.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *DAC*, 2001.

[11] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples." in *TACAS*, ser. LNCS, vol. 2619. Springer, 2003, pp. 2–17.

[12] K. L. McMillan, "Interpolation and SAT-based model checking," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 1–13.

[13] ——, "Applications of Craig interpolants in model checking." in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 1–12.

[14] *VIS*. [Online]. Available: http://visi.colorado.edu/~vis

[15] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*. Springer, 1999, pp. 193–207.

[16] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, ser. LNCS, vol. 1954. Springer, 2000.

[17] R. Armoni, L. Fix, R. Fraer, S. Huddleston, N. Piterman, and M. Vardi, "SAT-based induction for temporal safety properties," in *BMC*, 2004.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[19] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word level predicate abstraction and refinement for verifying RTL verilog." in *DAC*, 2005.

[20] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *ESEC / SIGSOFT FSE*, 1999, pp. 253–267.

[21] A. R. Bradley and Z. Manna, "Verification constraint problems with strengthening," in *ICTAC*, ser. LNCS, vol. 3722. Springer, 2006.