Under consideration for publication in Formal Aspects of Computing

Property-Directed Incremental Invariant Generation

Aaron R. Bradley[1] and Zohar Manna[1]

[1]Department of Computer Science, Stanford University

Abstract. A fundamental method of analyzing a system such as a program or a circuit is invariance analysis, in which one proves that an assertion holds on all reachable states. Typically, the proof is performed via induction; however, an assertion, while invariant, may not be inductive (provable via induction). Invariant generation procedures construct auxiliary inductive assertions for strengthening the assertion to be inductive. We describe a general method of generating invariants that is incremental and property-directed. Rather than generating one large auxiliary inductive assertion, our method generates many simple assertions, each of which is inductive relative to those generated before it. Incremental generation is amenable to parallelization. Our method is also property-directed in that it generates inductive assertions that are relevant for strengthening the given assertion. We describe two instances of our method: a procedure for generating clausal invariants of finite-state systems and a procedure for generating affine inequalities of numerical infinite-state systems. We provide evidence that our method scales to checking safety properties of some large finite-state systems.

Keywords: Static analysis, Model checking, Invariant generation, Affine invariants, Polynomial invariants, Clausal invariants

1. Introduction

A safety assertion of a transition system such as a program or a sequential circuit asserts that all reachable states of the system satisfy the assertion; or, alternately, that certain states are unreachable. The fundamental technique for proving that a safety assertion is a property of a given system is induction: prove that (1) every initial state of the transition system satisfies the assertion; and that (2) if a state satisfies the assertion, then so does its successors [Flo67, MP95]. Unfortunately, there are many safety properties of systems that are not inductive — they cannot be proved through induction.

How, then, does one prove that such a safety assertion is indeed a property of its system? The inductive assertion method of proving a safety assertion Π of a system suggests finding an auxiliary assertion χ such that $\Pi \wedge \chi$ is inductive for the system [MP95, BM07a]. Typically, each of the two steps of the inductive argument are carried out by a theorem prover. The challenge, of course, is to find this auxiliary assertion. Methods for

Correspondence and offprint requests to: Aaron R. Bradley, arbrad@cs.stanford.edu

finding auxiliary inductive assertions are collectively known as *invariant generation procedures*. We review the main characteristics of these procedures and then introduce our approach to invariant generation.

1.1. Bottom-Up Invariant Generation

A standard method for finding inductive assertions is abstract interpretation [CC77]. It is a *bottom-up* technique in that it considers only the system and not any assertion that one would like to prove. An abstract interpretation over-approximates the semantics of a given transition system in some abstract domain by interpreting the effects of the initial condition and each transition of the system within this domain. An inductive assertion within the domain is a fixpoint of the equations given by the abstract semantics, and it over-approximates the reachable states of the system. Such an assertion may or may not be sufficient for proving the desired safety assertion. Examples of abstract domains include numerical domains such as linear equations [Kar76], intervals [CC77], polyhedra [CH78], octahedra [Min01], and template constraints [SSM05]; and memory shape domains [WSR02].

The classic implementation of abstract interpretation is to compute a forward propagation within the abstract domain according to the system semantics [CC77]. To guarantee termination of the forward propagation, many domains require a widening operator that guesses the limit of the forward propagation. In the examples given above, only the domain of linear equations guarantees termination of the analysis; the other domains require applying a widening operator in practice, sacrificing completeness.

An alternative implementation is to compute a fixpoint directly via a related constraint system [AW92, Aik99, CSS03]. For example, the duality theorem of linear programming [Sch86] can be exploited to construct a constraint system whose solutions are inductive affine inequalities [CSS03]. Affine inequalities are related to polyhedra in that each facet of a polyhedron is given by an affine inequality. However, the forward propagation-based and constraint-based analyses differ in that the former can compute an inductive assertion consisting of many affine inequalities [CH78], while the latter solves for only one or a small fixed number of inductive affine inequalities [CSS03]. Additionally, the constraint-based approach is complete for a given number of inequalities — it will find an inductive assertion of a set size if one exists — while forward propagation is not complete for any class of assertions because of the necessary application of widening. Practically, the two strategies for implementation have trade-offs: the dual-representation implementation of the polyhedral domain can require in practice an exponential amount of space, while constraint solvers based on linear-programming typically require little space. However, achieving efficiency in practice with the constraint-based approach requires giving up completeness: to maintain completeness requires solving nonlinear constraints [Tar51, Col75], inhibiting the scalability of an exact solver.

1.2. Top-Down Invariant Generation

In contrast to bottom-up invariant generation procedures, *top-down*, or *property-directed*, procedures focus on a given (non-inductive) safety assertion. A well-known example of a top-down procedure is BDD-based symbolic model checking of safety properties of finite-state systems [CE82, BCM$^+$92]. The typical fixpoint computation starts with the safety assertion and then iteratively strengthens it by computing and conjoining weakest preconditions. (The weakest precondition of a set of states S is the set of states that must reach a state of S in one step (or from which the system cannot progress).) A fixpoint is detected when the weakest precondition of the current intermediate assertion is implied by the intermediate assertion itself. If the fixpoint contains the initial states of the system, the safety assertion indeed holds, and the computed fixpoint is an inductive proof of this fact. Otherwise, the safety assertion is invalid for the system.

The advantage of a top-down invariant generation procedure is that it computes an auxiliary assertion that is directly useful for proving a desired safety assertion. However, an exact backward propagation, as in the BDD-based model checker described above, can suffer to the same degree as a forward propagation. In both cases, one can compute more information than is actually needed to prove a given safety assertion. In general, any domain that has elements with a representation exponential in some aspect of the assertion and system, such as the number of variables, has the potential of failing simply though generating too much information. Examples of such domains include BDDs and polyhedra.

What one desires is to compute an inductive over-approximation of the reachable states of a system that is strong enough to prove a given safety assertion. Such techniques exist. For example, predicate abstraction with counterexample-guided refinement [GS97, CGJ+00] computes successively refined abstractions of the system and its state space until the given safety assertion is proved. Interpolation-based model checking [McM03, McM05] uses the interpolant of an unsatisfiable bounded model checking [BCCZ99] instance to compute an over-approximation to the strongest postcondition. (The strongest postcondition of a set of states S is the set of states that can be reached in one step from states in S.)

1.3. Top-Down Incremental Invariant Generation

We propose an alternative approach for computing top-down invariants that is based on computing a sequence of simple inductive assertions, each of which is inductive relative to the assertions that appear before it, such that the conjunction of all assertions proves the given safety assertion. One assertion φ is inductive relative to another assertion ψ if the inductive step of the proof of φ holds under the assumption ψ. We call a sequence of such inductive assertions an *incrementally inductive sequence*. By "simple inductive assertions", we mean small assertions from a fixed domain. In Section 4, we show how to generate at each iteration a single inductive clause over the latches of a circuit. In Section 5, we show how to generate at each iteration a single inductive affine inequality (or a small fixed number of them) over the numerical variables of a system. Each such inductive assertion is weak; however, the conjunction of an incrementally inductive sequence can be strong, as we show empirically.

This approach to invariant generation is derived from the *incremental* approach that we typically use when analyzing a system by hand [MP95]. It is fundamentally limited: for a fixed domain, there can exist a subset of elements whose conjunction is inductive, yet for which there is no ordering that yields a sequence of incrementally inductive assertions. For example, in a system in which initially $x = 1$ and that at each step assigns $x := -x$, neither $-1 \le x$ nor $x \le 1$ is inductive, yet $-1 \le x \le 1$ is inductive. In even this simple system, an approach limited to producing one new inductive affine inequality at each step must fail; and, in general, an approach limited to producing a small fixed number of assertions at each step must fail on similar simple systems. Yet we show that in other small programs with relatively sophisticated behavior, the incremental approach succeeds; and we show that on circuits, the incremental approach can yield results superior to known approaches to model checking safety properties.

We consider *top-down* invariant generation procedures: each element of the computed incrementally inductive sequence should be useful in proving the given safety assertion. We propose using *counterexamples to induction* to guide the generation of new incrementally inductive assertions. Specifically, if the conjunction of the current sequence with the given safety assertion is not inductive, then there is a state that satisfies the sequence and the safety assertion and that has a successor that violates the safety assertion. This state is a counterexample to induction. The search for the next small inductive assertion then focuses on proving that this state is unreachable. For example, the procedure of Section 4 searches for an inductive clause that proves that the state is unreachable; while the procedure of Section 5 searches for an affine inequality that proves that the state is unreachable.

Of course, the very process of generating a simple inductive assertion generalizes beyond the single counterexample to induction. Indeed, in Section 4, much of the intellectual work focuses on finding a *minimal* (or *prime*) inductive clause: a clause that does not contain a strict subclause that is also inductive. Such a clause is a maximal inductive generalization (within the fixed domain of clauses) from the observation that the counterexample to induction should be unreachable. Consequently, from relatively few counterexamples to induction, one often obtains a small inductive proof of the entire safety assertion.

Immediately, one must ask: what if there is no simple inductive assertion that proves the unreachability of a given counterexample to induction? We know that the counterexample state should be unreachable — unless the given safety assertion is invalid for the system. Therefore, we simply augment the safety assertion to say that this state is also an error state. For finite-state systems, this failure step guarantees completeness of the method: the analysis either finds an auxiliary inductive assertion proving the safety assertion; or it augments the safety assertion until it includes an initial state, indicating that the assertion fails. For infinite-state systems, such as the systems we consider in Section 5, this augmentation simply avoids considering the particular counterexample to induction again. The resulting analysis may not terminate. But the intermediate incrementally inductive sequence can provide useful information, just as a bottom-up invariant generation procedure provides information even when it fails to prove a particular safety assertion.

Overall, then, each iteration of a top-down incremental invariant generation procedure works as follows:

1. Check if the safety assertion Π is inductive relative to the current incrementally inductive sequence χ_i. If so, then the safety assertion is a property of the system.
2. Otherwise, obtain as a counterexample to induction the state s.
3. Attempt to generate a simple incrementally inductive assertion φ_{i+1} that proves that s is unreachable.
4. Upon success, append the new inductive assertion to the incrementally inductive sequence: $\chi_{i+1} := \chi_i \wedge \varphi_{i+1}$.
5. Upon failure, augment the safety assertion to assert that state s, too, is an error state: $\Pi := \Pi \wedge \neg s$.

1.4. The Verification Team

The theorem prover or constraint solver used in Step 1 can be instrumented to return several counterexamples to induction, each of which yields a potentially different inductive assertion in Step 3. This feature is a basis for parallelizing the analysis. Each process independently solves for a counterexample to induction, relying on randomness — or, if necessary, the use of *blocking clauses* [McM02] or a similar device — to find a unique state. The process then attempts to generate an inductive assertion proving that the state is unreachable, which it then shares with the other processes.

This loose cooperation among processes is similar to the way in which a team of humans verifies a complex assertion for a system. In practice, humans tackle complex properties by making simpler relevant observations about the system in the form of lemmas. Members of a verification team make and prove such conjectures independently and then share the information in a library. The team succeeds at proving the assertion when it is inductive relative to the library of lemmas. We present evidence in Section 6 that this style of parallelization indeed scales well to many processes, at least when analyzing empirically difficult properties of large circuits on computer clusters available today.

This paper synthesizes two conference papers that introduce top-down incremental procedures for numerical programs [BM06] and for circuits [BM07b]. We present a uniform abstract setting for these procedures (Section 3). From this foundation, we describe the clause domain and analysis (Section 4) and the affine inequality domain and analysis (Section 5). For applications to hardware, we present empirical results (Section 6). Finally, we discuss related techniques (Section 7) and conclude (Section 8).

2. Preliminaries

2.1. Propositional Logic

For Section 4, it will be convenient to review several useful notations and definitions of propositional logic. Other concepts from first-order logic will be introduced as needed. 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, while a formula in *disjunctive normal form* (DNF) is a disjunction of conjunctions of literals.

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. Assignment s can be represented as a formula $\hat{s}$, which is a conjunction of literals that correspond to the truth values given by s.

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 if and only if the formula $\varphi \wedge \neg\psi$ is unsatisfiable.

2.2. Transition Systems

We model software and hardware uniformly in first-order logic as transition systems [MP95].

Definition 2.1 (Transition System). A *transition system* $\mathcal{S}\colon \langle \overline{x}, \theta, \rho \rangle$ contains three components:

```
int j, k;
@ j = 2 ∧ k = 0
while (···) do
  if (···)
  then j := j + 4;
  else j := j + 2;
       k := k + 1;
done
```

(a) Simple

```
int u, w, x, z;
@ x ≥ 1 ∧ u = 1 ∧ w = 1 ∧ z = 0
while (w ≤ x) do
  (z, u, w) := (z + 1, u + 2, w + u + 2);
done
```

(b) Sqrt

Fig. 1. Example loops

- a set of *system variables* $\overline{x} = \{x_1, \ldots, x_n\}$,
- a formula $\theta[\overline{x}]$ over variables $\overline{x}$ specifying the *initial condition*,
- and a formula $\rho[\overline{x}, \overline{x}']$ over variables $\overline{x}$ and $\overline{x}'$ specifying the *transition relation*.

Primed variables $\overline{x}'$ represent the next-state values of the variables $\overline{x}$.

Several restricted forms of transition systems will be of interest in this paper. In a *Boolean transition system*, all variables $\overline{x}$ range over $\mathbb{B} : \{\mathsf{true}, \mathsf{false}\}$. Boolean transition systems are appropriate for modeling hardware. In a *real-number transition system*, all variables range over $\mathbb{R}$. A *linear transition system* is a real-number transition system in which the atoms of θ and ρ are affine inequalities, while a *polynomial transition system* has polynomial inequality atoms. Linear and polynomial transition systems are useful for analyzing programs that emphasize numerical data (either explicitly or, for example, through size functions that map data structures to integers). Although the variables of these systems range over $\mathbb{R}$, every invariant is also an invariant when considering the variables to range over $\mathbb{Z}$.

Example 2.1 (Boolean). Consider the contrived Boolean transition system $\mathcal{S}_{\mathsf{B}}$ with variables

$$\overline{x} = \{x_0, x_1, x, y_0, y_1, y, z\} \ ,$$

initial condition

$$\theta: \ (x_0 \leftrightarrow \neg x_1) \ \wedge \ x \ \wedge \ (y_0 \leftrightarrow \neg y_1) \ \wedge \ y \ \wedge \ z \ ,$$

and transition relation

$$\rho: \ \left[\begin{array}{l} (x_0' \leftrightarrow \neg x_0) \ \wedge \ (x_1' \leftrightarrow \neg x_1) \ \wedge \ (x' \leftrightarrow x_0 \vee x_1) \\ \ \wedge \ (y_0' \leftrightarrow x \wedge \neg y_0) \ \wedge \ (y_1' \leftrightarrow x \wedge \neg y_1) \ \wedge \ (y' \leftrightarrow y_0 \vee y_1) \\ \ \wedge \ (z' \leftrightarrow x \wedge y) \end{array}\right] \ .$$

The intention is that x and y — and thus z — are always true. We assert this intention as the safety assertion $\Pi : z$. ■

Example 2.2 (Alternating). Consider the linear transition system $\mathcal{S}_1$:

$$\mathcal{S}_1 \ \langle \{x : \mathbb{Z}\}, \ x = -1, \ x' = -x \rangle \ .$$

We assert that x is always at least -1: $\Pi : x \geq -1$. ■

Example 2.3 (Simple). Consider the loop Simple in Figure 1**(a)** [CH78]. The notation $\cdots$ in Figure 1**(a)** indicates nondeterministic choice. The corresponding linear transition system $\mathcal{S}_{\mathsf{Simple}}$ is the following:

$$\begin{array}{rl} \overline{x}: & \{j, k\} \\ \theta: & j = 2 \ \wedge \ k = 0 \\ \rho_1: & j' = j + 4 \ \wedge \ k' = k \\ \rho_2: & j' = j + 2 \ \wedge \ k' = k + 1 \end{array}$$

with ρ: $\rho_1 \vee \rho_2$. ■

Example 2.4 (Integer Square-Root). Consider the loop Sqrt in Figure 1**(b)**. It computes the integer square-root z of positive integer x. The following linear transition system $\mathcal{S}_{\mathsf{Sqrt}}$ ignores the branch exiting the loop:

$$\begin{array}{rl} \overline{x}: & \{u, w, x, z\} \\ \theta: & x \geq 1 \ \wedge \ u = 1 \ \wedge \ w = 1 \ \wedge \ z = 0 \\ \rho: & w \leq z \ \wedge \ z' = z + 1 \ \wedge \ u' = u + 2 \ \wedge \ w' = w + u + 2 \end{array}$$

Correctness is stated by the assertion that on exit, z is the integer square-root of x: $z^2 \leq x < (z+1)^2$. With some ingenuity, one may step this assertion back to the loop (by computing weakest preconditions):

$$\Pi : \ (w \leq x \ \rightarrow \ (z+1)^2 \leq x) \ \wedge \ (w > x \ \rightarrow \ x < (z+1)^2) \ .$$

We take Π as the safety assertion for $\mathcal{S}_{\mathsf{Sqrt}}$. ■

The semantics of a transition system are defined in terms of states and computations.

Definition 2.2 (State & Computation). A *state* s of transition system $\mathcal{S}$ is an assignment of values (of the proper type) to variables $\overline{x}$. We write a state s as either a tuple of assignments $\langle x_1 = \mathsf{v}_1, \ \ldots, \ x_n = \mathsf{v}_n \rangle$ or as a conjunction of equations $\hat{s} : x_1 = \mathsf{v}_1 \wedge \cdots \wedge x_n = \mathsf{v}_n$. A computation $\sigma : s_0, s_1, s_2, \ldots$ is an infinite sequence of states such that

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

A state s is reachable by $\mathcal{S}$ if there exists a computation of $\mathcal{S}$ that contains s.

Example 2.5 (Alternating). $\langle x = 1 \rangle$ is a state of transition system $\mathcal{S}_1$. One possible computation of the system is the following:

$$\sigma : \ \langle x = -1 \rangle, \ \langle x = 1 \rangle, \ \langle x = -1 \rangle, \ \langle x = 1 \rangle, \ \ldots \ .$$

Each state of the computation is (obviously) reachable, while state $\langle x = 2 \rangle$ is not reachable. ■

Example 2.6 (Boolean). One possible initial state of Boolean transition system $\mathcal{S}_{\mathsf{B}}$ is

$$s : \ \langle x_0 = \mathsf{true}, \ x_1 = \mathsf{false}, \ x = \mathsf{true}, \ y_0 = \mathsf{false}, \ y_1 = \mathsf{true}, \ y = \mathsf{true}, \ z = \mathsf{true} \rangle \ ,$$

or more succinctly:

$$\hat{s} : \ x_0 \ \wedge \ \neg x_1 \ \wedge \ x \ \wedge \ \neg y_0 \ \wedge \ y_1 \ \wedge \ y \ \wedge \ z \ .$$

A possible computation of the system is the following:

$$\begin{array}{rl} \sigma : & x_0 \ \wedge \ \neg x_1 \ \wedge \ x \ \wedge \ \neg y_0 \ \wedge \ y_1 \ \wedge \ y \ \wedge \ z \ , \\ & \neg x_0 \ \wedge \ x_1 \ \wedge \ x \ \wedge \ y_0 \ \wedge \ \neg y_1 \ \wedge \ y \ \wedge \ z \ , \\ & x_0 \ \wedge \ \neg x_1 \ \wedge \ x \ \wedge \ \neg y_0 \ \wedge \ y_1 \ \wedge \ y \ \wedge \ z \ , \\ & \neg x_0 \ \wedge \ x_1 \ \wedge \ x \ \wedge \ y_0 \ \wedge \ \neg y_1 \ \wedge \ y \ \wedge \ z \ , \\ & \ldots \end{array}$$

Each state of the computation is (obviously) reachable. We will prove that any state in which z is false is unreachable. ■

A *safety assertion* Π of a transition system $\mathcal{S}$ is a first-order formula over the variables $\overline{x}$ of $\mathcal{S}$. It asserts that at most the states s that satisfy Π ($s \models \Pi$) are reachable by $\mathcal{S}$. Invariants and inductive invariants are central to studying safety properties of transitions systems.

Definition 2.3 (Invariant). A formula φ is an *invariant* of $\mathcal{S}$ (or is $\mathcal{S}$-*invariant*) if for every computation $\sigma : s_0, s_1, s_2, \ldots$, for every $i \geq 0$, $s_i \models \varphi$.

Definition 2.4 (Inductive Invariant). A formula φ is $\mathcal{S}$-*inductive* if

- it holds initially: $\forall \overline{x}. \ \theta[\overline{x}] \ \rightarrow \ \varphi[\overline{x}]$ (initiation)
- and it is preserved by ρ: $\forall \overline{x}, \overline{x}'. \ \varphi[\overline{x}] \ \wedge \ \rho[\overline{x}, \overline{x}'] \ \rightarrow \ \varphi[\overline{x}']$ (consecution)

These two requirements are sometimes referred to as *verification conditions*. If φ is $\mathcal{S}$-inductive, then it is $\mathcal{S}$-invariant. When $\mathcal{S}$ is obvious from the context, we omit it from $\mathcal{S}$-inductive and $\mathcal{S}$-invariant.

For convenience, we abbreviate formulae using implication: $\varphi \Rightarrow \psi$ abbreviates $\forall \overline{y}.\ \varphi \rightarrow \psi$, where $\overline{y}$ are all variables of φ and ψ. Then (initiation) is $\theta \Rightarrow \varphi$, and (consecution) is $\varphi \wedge \rho \Rightarrow \varphi'$.

Definition 2.5 (Relatively Inductive Invariant). A formula φ is *$\mathcal{S}$-inductive relative to ψ* if it is $\mathcal{S}$-inductive under the assumption ψ:

- $\theta \ \Rightarrow \ \varphi$ (initiation)
- $\psi \wedge \varphi \wedge \rho \ \Rightarrow \ \varphi'$ (consecution)

Definition 2.6 (Counterexample to Induction). Consider safety assertion Π of transition system $\mathcal{S}$. Π-state s (a state that satisfies Π) is a *counterexample to induction* if it violates (consecution): s has some successor $\neg\Pi$-state.

Example 2.7 (Alternating). The assertion $x \geq -1$ of $\mathcal{S}_1$ is not inductive because (consecution) fails:

$$x \geq -1 \ \wedge \ x' = -x \ \not\Rightarrow \ x' \geq -1 \ .$$

One counterexample to induction is $s : \langle x = 2 \rangle$.

However, the assertion $-1 \leq x \leq 1$ $(-1 \leq x \wedge x \leq 1)$ is inductive: it satisfies (initiation):

$$x = -1 \ \Rightarrow \ -1 \leq x \leq 1 \ ,$$

and (consecution):

$$-1 \leq x \leq 1 \ \wedge \ x' = -x \ \Rightarrow \ -1 \leq x' \leq 1 \ .$$

■

Example 2.8 (Boolean). The assertion z of $\mathcal{S}_{\mathsf{B}}$ is not inductive because (consecution) fails with, for example, the counterexample to induction

$$s : \ x_0 \ \wedge \ x_1 \ \wedge \ \neg x \ \wedge \ y_0 \ \wedge \ y_1 \ \wedge \ \neg y \ \wedge \ z$$

whose successor

$$\neg x_0 \ \wedge \ \neg x_1 \ \wedge \ x \ \wedge \ \neg y_0 \ \wedge \ \neg y_1 \ \wedge \ y \ \wedge \ \neg z$$

violates $\Pi : z$.

In contrast, the formula

$$\neg x_0 \ \vee \ \neg x_1 \ \vee \ \neg x$$

is inductive: it satisfies (initiation),

$$\theta \ \Rightarrow \ \neg x_0 \ \vee \ \neg x_1 \ \vee \ \neg x \ ,$$

because $x_0 \leftrightarrow x_1$ implies $x_0 \vee \neg x_1$; and it satisfies (consecution),

$$(\neg x_0 \ \vee \ \neg x_1 \ \vee \ \neg x) \ \wedge \ \rho \ \Rightarrow \ \neg x_0' \ \vee \ \neg x_1' \ \vee \ \neg x' \ ,$$

because $\neg x_0' \vee \neg x_1' \vee \neg x'$ is equivalent to $x_0 \vee x_1 \vee \neg(x_0 \vee x_1)$ according to the transition relation ρ of $\mathcal{S}_{\mathsf{B}}$. ■

The main problem that we consider is the following: given transition system $\mathcal{S} \colon \langle \overline{x}, \theta, \rho \rangle$ and safety assertion Π, is Π $\mathcal{S}$-invariant? Proving that Π is inductive answers the question affirmatively. But frequently Π is invariant but not inductive. The *inductive assertion method* [Flo67, MP95] suggests finding a formula χ such that $\Pi \wedge \chi$ is inductive; χ is called an *auxiliary* or *strengthening assertion*.

Example 2.9 (Alternating). The assertion $-1 \leq x \leq 1$ is $\mathcal{S}_1$-inductive, while the original safety assertion $\Pi : -1 \leq x$ is not $\mathcal{S}_1$-inductive. Hence, $x \leq 1$ is a strengthening (though not inductive) assertion for $\Pi : -1 \leq x$ that allows us to prove the invariance of Π via induction. Also, each of $-1 \leq x$ and $x \leq 1$ is inductive relative to the other. ■

Example 2.10 (Integer Square-Root). Consider again the transition system $\mathcal{S}_{\mathsf{Sqrt}}$. Its safety assertion

$$\Pi : \ (w \leq x \ \rightarrow \ (z+1)^2 \leq x) \ \wedge \ (w > x \ \rightarrow \ x < (z+1)^2)$$

is not inductive. However, the assertion

$$\chi : \; w = (z+1)^2$$

is inductive; so is $\Pi \wedge \chi$. Hence, χ is a strengthening assertion that allows us to prove the invariance of Π via induction. ■

Example 2.11 (Boolean). $\Pi : z$ is not $\mathcal{S}_\mathsf{B}$ inductive; however, with

$$\chi : \; \begin{bmatrix} (\neg x_0 \;\vee\; \neg x_1 \;\vee\; \neg x) \;\wedge\; (x_0 \;\vee\; x_1 \;\vee\; \neg z) \;\wedge\; x \\ \wedge\; (\neg y_0 \;\vee\; y_1 \;\vee\; \neg z) \;\wedge\; (\neg y_0 \;\vee\; \neg y_1) \;\wedge\; y \end{bmatrix}$$

the assertion $\Pi \wedge \chi$ is $\mathcal{S}_\mathsf{B}$-inductive. Hence, χ is a strengthening assertion that allows us to prove the invariance of Π via induction. ■

If Π is not invariant, then we seek instead a *counterexample trace*.

Definition 2.7 (Counterexample Trace). A *counterexample trace* of system $\mathcal{S}$ and specification Π is a (prefix of a) computation $\sigma : s_0, s_1, s_2, \ldots, s_k$ such that s_k violates Π: $s_k \models \neg\Pi$.

3. Property-Directed Incremental Invariant Generation

The inductive assertion method suggests that to prove that Π is an invariant of $\mathcal{S} \colon \langle \overline{x}, \theta, \rho \rangle$, we should find a strengthening assertion χ such that $\Pi \wedge \chi$ is inductive. How do we find χ?

Suppose that we have an invariant generation procedure P that, given a state s, generates a simple invariant of a fixed form that proves the unreachability of s — if such an invariant exists. For example, for considering Boolean transition systems, we describe a procedure $\mathsf{P}_\mathbb{B}$ that produces an inductive clause that excludes a given state (Section 4); while for considering real-number transition systems, we describe a procedure $\mathsf{P}_\mathbb{R}$ that produces an inductive affine inequality that excludes a given state (Section 5). Additionally, the procedure P accepts a *restriction* formula ψ such as the conjunction of all previously generated invariants. P need only generate a formula that is inductive relative to ψ. In summary, $\mathsf{P}(s, \psi)$ returns a formula that is inductive relative to ψ and that proves that s is unreachable — if the domain contains such a formula.

We describe in this section an incremental strategy for using such an invariant generation procedure P. The ideas are of course classical [MP95], but they motivate new ways of analyzing systems.

3.1. Incremental Invariant Generation

Suppose first that we do not have a desired safety assertion to prove. In this case, our strategy is simply to discover incrementally a sequence $\varphi_1, \varphi_2, \varphi_3, \ldots$ of formulae, each of which is $\mathcal{S}$-inductive relative to those that occur previously. We call such a sequence an *incrementally inductive sequence*. Let χ_i be the conjunction of all φ_j, $j \le i$; hence, $\chi_0 \overset{\text{def}}{=} \mathsf{true}$. Clearly, each χ_i is $\mathcal{S}$-inductive.

Given χ_i, on iteration $i+1$ we seek a new formula φ_{i+1} that is inductive relative to χ_i, so that χ_{i+1} is then also inductive. The challenge is to derive a new piece of information. Ideally, we would discover an assertion φ_{i+1} that satisfies (initiation) and (consecution) and that excludes some state s that χ_i does not exclude:

$$\exists \overline{x}.\; \chi_i[\overline{x}] \wedge \neg\varphi_{i+1}[\overline{x}] \qquad \text{(strengthening)}$$

Given inductive invariant generation procedure P, an approximation of this condition is (1) to choose a state s such that $s \models \chi_i$, and (2) to let $\varphi_{i+1} \overset{\text{def}}{=} \mathsf{P}(s,\ \chi_i)$. P may fail — indeed, s may actually be reachable — in which case, we record the state s and exclude it from consideration in future iterations. There is no criterion for terminating this process. Each χ_i is inductive and yields useful information.

Incremental invariant generation is useful in practice for discovering information about a program in the absence of a specification.

Example 3.1 (Simple). Consider the linear transition system $\mathcal{S}_\mathsf{Simple}$. In Section 5, we describe a procedure $\mathsf{P}_\mathbb{R}$ that discovers inductive invariants in the form of affine inequalities.

- Initially, every state is assume to be reachable. Suppose that state

  $$s_1 : \langle j = -1,\ k = -3\rangle$$

  is selected. $\mathsf{P}_{\mathbb{R}}(s_1,\ \mathsf{true})$ returns the affine inequality

  $$\varphi_1 : \ k \geq 0 \ .$$

- Now we must choose a state that satisfies $\chi_1 : k \geq 0$. Suppose that state

  $$s_2 : \langle j = 2,\ k = 0\rangle$$

  is selected. $\mathsf{P}_{\mathbb{R}}(s_2,\ k \geq 0)$ fails to find an inductive inequality that excludes s_2.
- If the next selected state is

  $$s_3 : \langle j = -1,\ k = 0\rangle \ ,$$

  then $\mathsf{P}_{\mathbb{R}}(s_3,\ k \geq 0)$ might return

  $$\varphi_2 : \ j \geq 0 \ .$$

- If the next selected state — which must satisfy $k \geq 0 \land j \geq 0$ — is

  $$s_4 : \langle j = 0,\ k = 0\rangle$$

  then $\mathsf{P}_{\mathbb{R}}(s_4,\ k \geq 0 \land j \geq 0)$ returns

  $$\varphi_3 : \ j \geq 2k + 2 \ .$$

  In general, any selected state that violates $j \geq 2k + 2$ allows $\mathsf{P}_{\mathbb{R}}$ to discover φ_3.
- There are no more affine invariants to find, although this method will simply continue to generate states that satisfy χ_3 and fail to find excluding invariants.

In this example, each of $k \geq 0$, $j \geq 0$, and $j \geq 2k + 2$ is actually inductive by itself. ■

3.2. Property-Directed Incremental Invariant Generation

Suppose now that we have a desired safety assertion Π to prove. We seek an incrementally inductive sequence $\varphi_1, \varphi_2, \varphi_3, \ldots, \varphi_n$ such that (using the same definition for χ_i as above) $\Pi \land \chi_n$ is $\mathcal{S}$-inductive. Notice that even if each χ_i is inductive only relative to Π, $\Pi \land \chi_n$ is inductive. That is, we may assume Π even as we search for a strengthening assertion.

Given χ_i, on iteration $i + 1$ we seek a new formula φ_{i+1} that is inductive relative to $\Pi \land \chi_i$, so that χ_{i+1} is then also inductive relative to Π. As in the case without Π, the challenge is to derive a new piece of information. Ideally, we would discover an inductive assertion φ_{i+1} that excludes some $(\Pi \land \chi_i)$-state s that has a successor that violates Π:

$$\exists \overline{x}, \overline{x}'.\ \Pi \land \chi_i \land \neg\varphi_{i+1} \land \rho \land \neg\Pi' \qquad (\Pi\text{-strengthening})$$

That is, an ideal procedure would discover an assertion φ_{i+1} that satisfies (initiation), (consecution), and (Π-strengthening).

However, our inductive invariant generation procedure P considers one state at a time. Thus, we seek a counterexample to induction: if (consecution),

$$\Pi \land \chi_i \land \rho \ \Rightarrow \ \Pi'$$

fails, there is a counterexample to induction s such that

$$s \models \exists \overline{x}'.\ \Pi \land \chi_i \land \rho \land \neg\Pi' \ .$$

Let $\varphi_{i+1} \stackrel{\text{def}}{=} \mathsf{P}(s,\ \Pi \land \chi_i)$. If P fails to find a new inductive invariant, we include proving that s is unreachable as a subgoal of proving Π: $\Pi := \Pi \land \neg\hat{s}$ (where $\hat{s}$ is the formula that describes state s).

If $\mathcal{S}$ is a finite-state system, then one of two situations eventually occurs. Either

1. $\Pi \land \chi_n$ is $\mathcal{S}$-inductive at some iteration n, which is indicated by the lack of any counterexample to induction (induction succeeds); or

2. through the addition of subgoals, $\neg\Pi \wedge \theta$ becomes satisfiable, indicating that the original safety assertion does not hold for $\mathcal{S}$.

In the latter case, one can extract a counterexample trace from the subgoals conjoined to the original safety assertion.

If $\mathcal{S}$ is an infinite-state system, then there is no guarantee of termination. However, every state excluded by χ_i cannot be reached without first violating Π. Thus, the intermediate formulae χ_i can be informative, just as bottom-up invariants are informative even when they are not strong enough to prove a desired assertion.

In practice, one must find counterexamples to induction (or simply states s that satisfy χ_i in the case when there is no assertion Π). For analyzing Boolean transition systems (Section 4), we use a propositional satisfiability solver to find states that satisfy χ_i; for analyzing linear transition systems (Section 5), we use a rational linear programming solver; and for analyzing non-linear real-number transition systems (Section 5), we use either a complete decision procedure such as cylindrical algebraic decomposition (CAD) [Tar51, Col75] or an incomplete solver such as numeric constraint solving. Some randomness in the solver will assist in providing a general view of the potential state space, although it is unnecessary for preserving completeness in the finite-state case.

Example 3.2 (Alternating). Consider the linear transition system $\mathcal{S}_1$ in which x alternates between -1 and 1. We use the procedure of Section 5 to find an assertion strengthening the safety assertion $\Pi : x \geq -1$.

- Π satisfies (initiation) but not (consecution):

  $$s_1 : \langle x = 2 \rangle \;\models\; x \geq -1 \;\wedge\; x' = -x \;\wedge\; \neg(x' \geq -1) \;.$$

  s_1 is a counterexample to induction. $\mathsf{P}(s_1,\ \Pi)$ discovers the assertion

  $$\varphi_1 :\; x \leq 1 \;,$$

  which is inductive relative to Π but not on its own.
- $\Pi \wedge \chi_1 : -1 \leq x \leq 1$ is $\mathcal{S}_1$-inductive, proving the $\mathcal{S}_1$-invariance of $\Pi : x \geq -1$.

■

Example 3.3 (Boolean). Consider the Boolean transition system $\mathcal{S}_{\mathsf{B}}$ for which we would like to prove that $\Pi : z$ is invariant. In Section 4, we describe a procedure $\mathsf{P}_{\mathbb{B}}$ that discovers inductive invariants in the form of clauses.

- Π satisfies (initiation) but not (consecution):

  $$\hat{s}_1 : x_0 \wedge x_1 \wedge \neg x \wedge y_0 \wedge y_1 \wedge \neg y \wedge z \;\models\; z \;\wedge\; \rho \;\wedge\; \neg z' \;.$$

  s_1 is a counterexample to induction. $\mathsf{P}_{\mathbb{B}}(s_1,\ \Pi)$ fails to find an inductive clause to exclude s_1, so we set $\Pi := \Pi \wedge \neg\hat{s}_1$.
- Proceeding in a depth-first manner, we seek to prove that s_1 is unreachable. Consider the predecessor of s_1

  $$\hat{s}_2 :\; \neg x_0 \;\wedge\; \neg x_1 \;\wedge\; x \;\wedge\; \neg y_0 \;\wedge\; \neg y_1 \;\wedge\; y \;\wedge\; z$$

  and its predecessor

  $$\hat{s}_3 :\; x_0 \;\wedge\; x_1 \;\wedge\; x \;\wedge\; y_0 \;\wedge\; y_1 \;\wedge\; y \;\wedge\; z \;.$$

  $\mathsf{P}_{\mathbb{B}}(s_2,\ \Pi \wedge \neg\hat{s}_1)$ does not yield an invariant; however $\mathsf{P}_{\mathbb{B}}(s_3,\ \Pi \wedge \neg\hat{s}_1 \wedge \neg\hat{s}_2)$ returns the clause

  $$\varphi_1 :\; \neg x_0 \;\vee\; \neg x_1 \;\vee\; \neg x \;,$$

  which is inductive relative to $\Pi \wedge \neg\hat{s}_1 \wedge \neg\hat{s}_2$.
- Now $\mathsf{P}_{\mathbb{B}}(s_2,\ \Pi \wedge \neg\hat{s}_1 \wedge \chi_1)$ returns clause

  $$\varphi_2 :\; x_0 \;\vee\; x_1 \;\vee\; \neg z \;,$$

  and $\mathsf{P}_{\mathbb{B}}(s_1,\ \Pi \wedge \chi_2)$ returns clause

  $$\varphi_3 :\; x \;.$$

- Subsequent iterations yield clauses

$$\begin{array}{ll} \varphi_4: & \neg y_0 \ \vee \ \neg y_1 \ \vee \ \neg y \\ \varphi_5: & y_0 \ \vee \ y_1 \ \vee \ \neg z \\ \varphi_6: & \neg y_0 \ \vee \ \neg y_1 \\ \varphi_7: & y \end{array}$$

Finally, $\Pi \wedge \chi_7$ is $\mathcal{S}_\mathsf{B}$-inductive, proving the $\mathcal{S}_\mathsf{B}$-invariance of z.

■

3.3. Parallel Analysis

Large systems, such as some of the hardware systems that we study in Section 6, often require many simple inductive assertions to prove a given safety assertion. Many of these assertions are inductive relative to a subset of the previously generated assertions. Hence, one can search for many inductive assertions simultaneously by starting from different counterexamples to induction (or even the same counterexample if aspects of the invariant generation are randomized).

Parallelizing an analysis based on incremental invariant generation is thus straightforward. Each process performs its own analysis and shares the discovered invariants and subgoals with the other processes. In Section 6, we report on experiments that show that relying on some randomness in the process of selecting counterexamples to induction is sufficient to keep the processes from generating too much redundant information, at least with the amount of parallelism available in today's average-size computer clusters. Propositional satisfiability solvers, which we use for our experiments, naturally have some randomness. An alternative to relying on randomness is to use and share blocking clauses [McM02] or a similar device to prevent discovering the same counterexample to induction simultaneously in different processes.

4. Clausal Invariants

In this section, we consider a particular invariant generation procedure $\mathsf{P}_\mathbb{B}(s, \psi)$ for Boolean transition systems. It searches for a *subclause* c of $\neg\hat{s}$ that is *inductive relative to* ψ and whose strict subclauses are not inductive relative to ψ. We call such a clause a *minimal inductive subclause*. A state s can have many minimal inductive subclauses (or none); we are interested in finding just one.

Because a minimal inductive subclause is a subclause of $\neg\hat{s}$, it implies $\neg\hat{s}$ and thus proves that s is unreachable. Additionally, the process of discovering a *minimal* inductive subclause generalizes the argument that s is unreachable to prove that many other related states are unreachable as well.

4.1. Overview

We present an overview of the procedure for finding minimal inductive subclauses; subsequent sections provide the technical details.

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;
- 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 fixpoint theorem [Kna28, Tar55], 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 greatest 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 4.2 describes how to find $\bar{c}$ with a number of satisfiability queries linear in $|c|$.

Example 4.1. Consider the state (first encountered in Example 3.3)

$$\hat{s}_1 : \ x_0 \ \wedge \ x_1 \ \wedge \ \neg x \ \wedge \ y_0 \ \wedge \ y_1 \ \wedge \ \neg y \ \wedge \ z$$

of $\mathcal{S}_\mathsf{B}$ and the corresponding clause

$$c_1 : \ \neg x_0 \ \vee \ \neg x_1 \ \vee \ x \ \vee \ \neg y_0 \ \vee \ \neg y_1 \ \vee \ y \ \vee \ \neg z \ .$$

Checking if

$$\Pi \ \wedge \ c_1 \ \wedge \ \rho \ \Rightarrow \ c_1'$$

reveals counterexample

$$\neg x_0 \ \wedge \ \neg x_1 \ \wedge \ x \ \wedge \ \neg y_0 \ \wedge \ \neg y_1 \ \wedge \ y \ \wedge \ z$$

with corresponding clause

$$d_1 : \ x_0 \ \vee \ x_1 \ \vee \ \neg x \ \vee \ y_0 \ \vee \ y_1 \ \vee \ \neg y \ \vee \ \neg z \ .$$

The best clause that under-approximates $c_1 \wedge d_1$ is $c_1 \sqcap d_1$:

$$c_2 : \ \neg z \ ,$$

which is computed as the intersection of c_1's and d_1's literal sets. However, c_2 does not satisfy (initiation), so it is not an inductive subclause of c_1. ■

Example 4.2. Consider the state (first encountered in Example 3.3)

$$\hat{s}_3 : \ x_0 \ \wedge \ x_1 \ \wedge \ x \ \wedge \ y_0 \ \wedge \ y_1 \ \wedge \ y \ \wedge \ z$$

of $\mathcal{S}_\mathsf{B}$ and the corresponding clause

$$c_1 : \ \neg x_0 \ \vee \ \neg x_1 \ \vee \ \neg x \ \vee \ \neg y_0 \ \vee \ \neg y_1 \ \vee \ \neg y \ \vee \ \neg z \ .$$

Checking if

$$\Pi \ \wedge \ \neg\hat{s}_1 \ \wedge \ \neg\hat{s}_2 \ \wedge \ c_1 \ \wedge \ \rho \ \Rightarrow \ c_1' \tag{1}$$

reveals that c_1 satisfies (consecution) relative to $\Pi \wedge \neg\hat{s}_1 \wedge \neg\hat{s}_2$. It also satisfies (initiation).

The subformula $\neg\hat{s}_1 \wedge \neg\hat{s}_2$ occurs in (1) because s_1 and s_2 were added as subgoals of proving the invariance of Π in Example 3.3. ■

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 the least fixpoint of a monotone *nondecreasing* 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 true otherwise. This minimal subclause d is known as a *prime implicate* [McM02, JS05]. There can be exponentially many such minimal subclauses since the CNF representation of a formula can be exponentially large. But there may not be any prime implicate if $\varphi \not\Rightarrow c$. Section 4.3 discusses an 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 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 implication $\psi \wedge d \wedge \rho \Rightarrow e'$ holds. In other words, it computes $e' : \mathsf{implicate}(\psi \wedge d \wedge \rho, \ c')$, a least over-approximation in $L_{b,c}$ of the strongest postcondition of d.

The operation up can be determinized into a function on $L_{b,c}$ — which 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 least fixpoint $\bar{c}$ is an inductive subclause of c that is small in practice.

Example 4.3. Continuing from Example 4.2, we want to find a small inductive subclause of the inductive clause

$$c_1 : \neg x_0 \lor \neg x_1 \lor \neg x \lor \neg y_0 \lor \neg y_1 \lor \neg y \lor \neg z \ .$$

Computing $\mathsf{implicate}(\theta, c_1)$ yields clause

$$d_1 : \neg x_0 \lor \neg x_1 \ .$$

The clause $\neg y_0 \lor \neg y_1$ is also an implicate of θ. We next compute $\mathsf{up}(L_{d_1,c_1}, d_1)$, yielding clause

$$d_2 : \neg x_0 \lor \neg x_1 \lor \neg x \ .$$

$\mathsf{up}(L_{d_1,c_2}, d_2) = d_2$, so we have reached a fixpoint: d_2 is a small inductive subclause of c_1. ■

The discovered inductive clause is not necessarily a minimal inductive subclause, as different deterministic instantiations of implicate result in different least 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 the least 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 $\mathsf{mic}(c, \psi)$. It returns a minimal subclause of c that is $\mathcal{S}$-inductive relative to ψ, or true if no such clause exists.

Example 4.4. The discovered clause d_2 of Example 4.3 is a minimal inductive subclause. For neither $\neg x_0 \lor \neg x$ nor $\neg x_1 \lor \neg x$ satisfies (initiation), while computing the fixpoint of down on clause $\neg x_0 \lor \neg x_1$ yields false, which also does not satisfy (initiation). ■

Finally, mic is the basis for the invariant generation procedure: for Boolean transition systems, $\mathsf{P}_{\mathbb{B}}(s, \psi)$ is implemented as $\mathsf{mic}(\neg\hat{s}, \psi)$.

Subsequent sections provide the details of the operations introduced in this section.

4.2. Backward Phase

Recall that the monotone nonincreasing function $\mathsf{down}(L_c, d)$ computes the largest subclause $e \sqsubseteq d$ such that the implication $\psi \land e \land \rho \Rightarrow d'$ 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 $\psi \land c \land \rho \Rightarrow c'$ 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 $\psi \land c \land \rho \land \neg c'$ is satisfied by some assignment (s, s') to the unprimed and primed variables. 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 4.1 (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, $e \lor f$; f contains the literals of c_i that are not literals of e. Since (consecution)

```
let rec min p S_0 = function
  | []     → S_0
  | h :: t → if p(S_0 ∪ t)
             then min p S_0 t
             else min p (h :: S_0) t
let minimal p S = min p [] S
```

Fig. 2. Linear search-based minimal

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:

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

$$\psi \wedge \neg e \wedge f \wedge \rho \wedge \neg e' \wedge \neg f' \tag{3}$$

Formula (2) is unsatisfiable because e satisfies (consecution) by assumption. Therefore, formula (3) 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.

4.3. Forward Phase

Recall that the monotone nondecreasing function $\mathsf{up}(L_c, d)$ computes a minimal subclause $e \sqsubseteq c$ such that the implication $\psi \wedge d \wedge \rho \Rightarrow e'$ holds. As explained in Section 4.1, 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 true 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.

Figure 2 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 asymptotically fewer queries to p.

We can do 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 3 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

```
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. 3. Binary search-based minimal

```
@pre p(sup ∪ S0) ∧ ¬p(sup)
@post V ⊆ S0 ∧ p(sup ∪ V)
      ∧ ∀e ∈ V. ¬p(sup ∪ V \ {e})
let rec min p sup S0 =
```

Fig. 4. Annotated prototype of min, where V is the return value

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 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 3 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.

It has been brought to our attention that Junker describes this algorithm in the context of constraint solving [Jun01]. However, we provide an independent proof of correctness and a different complexity analysis.

Theorem 4.1 (Correct). Suppose that S is nonempty, $p(S)$ is true, and $p(\emptyset)$ is false.

1. min p [] S terminates.
2. Let $\bar{S} = \mathsf{min}\ p\ []\ S$. 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 4. Throughout the proof, let $V = \mathsf{min}\ p\ sup\ S_0$ 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 $\mathsf{minimal}$, $sup = \emptyset$; hence, $p(\bar{S}) = p(\emptyset \cup \bar{S}) = \mathsf{true}$ 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 $\mathsf{minimal}$, $\mathsf{sup} = \emptyset$ and $V = \bar{S}$; hence, $\neg p(\bar{S} \setminus \{e\})$ for $e \in \bar{S}$ from invariant (5). □

Theorem 4.2 (Upper Bound). Let $\bar{S} = \mathsf{min}\ p\ []\ S$. Discovering $\bar{S}$ requires making at most $2\left((|\bar{S}| - 1) + |\bar{S}| \lg \frac{|S|}{|\bar{S}|}\right)$ 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 4.3 (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 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,

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

□

```
let rec min f sup S0 =
  if |S0| = 1
  then (sup, S0)
  else let ℓ0, r0 = split S0 in
       let v, C = f(sup ∪ ℓ0) in
       if v
       then min f (sup ∩ C) (ℓ0 ∩ C)
       else let v, C = f(sup ∪ r0) in
            if v
            then min f (sup ∩ C) (r0 ∩ C)
            else let C, ℓ = min f (sup ∪ r0) ℓ0 in
                 let sup = sup ∩ C in
                 let r0 = r0 ∩ C in
                 let C, r = min f (sup ∪ ℓ) r0 in
                 (sup ∩ C, (ℓ ∩ C) ∪ r)
let minimal f S =
  let _, S0 = min f [] S in S0
```

Fig. 5. Binary search-based minimal with additional information

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 5 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.

4.4. Minimality

The procedure mic introduced in Section 4.1 employs a recursive procedure to guarantee minimality of the returned inductive clause. After finding a small inductive clause $\bar{c}$ via the backward and forward analysis, it recurses on each subclause of $\bar{c}$ with one fewer literal.

A worst-case complexity analysis of mic suggests that $O(|c|^3)$ propositional queries are possible to find a minimal inductive clause: each descent step moves one step down L_c, costing $O(1)$ propositional queries and $|c|$ recursive steps; at each recursive step, all but one direct descendant does not have an inductive subclause, costing $O(|c|)$ queries per descendant and hence $O(|c|^2)$ queries overall for a recursive step. However, this cubic cost was not achieved in our experiments.

5. Inequality Invariants

In this section, we discuss a second instance of incremental invariant generation in the context of numerical infinite-state systems. In this domain, our method does not provide any guarantees of termination or completeness. Moreover, compared to the empirical investigation of the clause domain (see Section 6), the empirical work in this domain is preliminary. However, it provides evidence that the incremental approach to invariant generation can be applied more generally than just to finite-state systems.

We describe an invariant generation procedure $\mathsf{P}_{\mathbb{R}}(s, \psi)$ that searches for an affine (or, to a limited extent, polynomial) inequality that is inductive relative to ψ and that excludes the state s. The method follows directly from work on constraint-based generation of affine inequality invariants [CSS03]; the addition is simply to introduce a condition into the constraint system that s should be excluded, corresponding to the (strengthening) or (Π-strengthening) conditions of Section 3. We review the constraint-based method in Section 5.1 and present the adaptation to the incremental setting in Section 5.2. From previous work [CSS03, SSM04], the constraint-based approach can generate one or a small fixed number of inductive inequalities per control location of the system; however, we consider simple loops in our presentation. Even with multiple control locations, however, we still consider just one counterexample to induction at a time.

5.1. Background: Constraint-Based Invariant Generation

The constraint-based generation of inductive affine inequalities invariants [CSS03] follows from *Farkas's Lemma*, which relates a *primal* constraint system S (a conjunction of affine inequalities) over the program variables to a *dual* constraint system over a set of introduced *dual* variables (also called Lagrangian multipliers) [Sch86]. Theorem 5.1 describes the case when S consists of a conjunction of affine inequalities. Then Corollary 5.1 extends the result, with loss of completeness, to the case when some constraints of S are polynomial inequalities. Both are statements of well-known theorems.

Theorem 5.1 (Farkas's Lemma). Consider the following universal constraint system of affine inequalities over real variables $\overline{x} = \{x_1, \ldots, x_m\}$:

$$S : \left[\begin{array}{ccccccccccc} a_{1,0} & + & a_{1,1}x_1 & + & \cdots & + & a_{1,m}x_m & \geq & 0 \\ \vdots & & & & \vdots & & \vdots & & \\ a_{n,0} & + & a_{n,1}x_1 & + & \cdots & + & a_{n,m}x_m & \geq & 0 \end{array}\right]$$

If S is satisfiable, it implies affine inequality $c_0 + c_1 x_1 + \cdots + c_m x_m \geq 0$ if and only if there exist real numbers $\lambda_1, \ldots, \lambda_n \geq 0$ such that

$$c_1 = \sum_{i=1}^{n} \lambda_i a_{i,1} \quad \cdots \quad c_m = \sum_{i=1}^{n} \lambda_i a_{i,m} \quad c_0 \geq \left(\sum_{i=1}^{n} \lambda_i a_{i,0} \right) .$$

Furthermore, S is unsatisfiable if and only if S implies $-1 \geq 0$.

Farkas's Lemma states that the relationship between the primal and dual constraint systems is strict: the universal constraints of the primal system are valid if and only if the dual (existential) constraint system has a solution. Generalizing to polynomials preserves soundness but drops completeness. Stronger generalizations than the following claim are also possible [Cou05, PJ04].

Corollary 5.1 (Polynomial Lemma). Consider the universal constraint system S of polynomial inequalities over real variables $\overline{x} = \{x_1, \ldots, x_m\}$:

$$\begin{array}{ll} A : & \left\{ \begin{array}{cclc} a_{1,0} & + & \sum_{i=1}^{m} a_{1,i} t_i & \geq 0 \\ & \vdots & & \\ a_{n,0} & + & \sum_{i=1}^{m} a_{n,i} t_i & \geq 0 \end{array} \right. \\ \hline C : & \quad \begin{array}{cclc} c_0 & + & \sum_{i=1}^{m} c_i t_i & \geq 0 \end{array} \end{array}$$

where the t_i are monomials over $\overline{x}$. That is, S has the form $A \Rightarrow C$. Construct the dual constraint system as follows. Multiply each row of A by a fresh Lagrangian multiplier λ_i. Then for monomials t_i of even power (e.g., 1, x^2, x^2y^4, etc.), impose the constraint

$$c_i \geq \lambda_1 a_{1,i} + \cdots + \lambda_n a_{n,i} \ ;$$

and for all other terms, impose the constraint

$$c_i = \lambda_1 a_{1,i} + \cdots + \lambda_n a_{n,i} \ .$$

Finally, impose the constraint that all λ_i are nonnegative: $\lambda_i \geq 0$. If the dual constraint system is satisfiable, then the primal constraint system is valid.

Constraint-based linear invariant generation uses Farkas's Lemma to generate affine inequality invariants over linear transition systems. The method begins by proposing an affine inequality *template*

$$c_0 + c_1x_1 + \cdots + c_nx_n \geq 0 \ ,$$

where the c_i are the template variables. It then imposes on the template the conditions (initiation)

$$\theta \ \Rightarrow \ c_0 + c_1x_1 + \cdots + c_nx_n \geq 0$$

and (consecution)

$$\psi \ \wedge \ c_0 + c_1x_1 + \cdots + c_nx_n \geq 0 \ \wedge \ \rho \ \Rightarrow \ c_0 + c_1x_1' + \cdots + c_nx_n' \geq 0 \ ,$$

where ψ is additional information relative to which a solution to the template should be inductive.

Expressing θ, ρ, and ψ in disjunctive normal form reveals a finite set of parameterized linear constraint systems of the form in Theorem 5.1, except for the presence of the parameters c_i. Dualizing the constraint systems according to Farkas's Lemma and conjoining them produces a single large existential conjunctive constraint system over the parameters c_i and the introduced multipliers $\overline{\lambda}$. The dual system is a *bilinear* or a *parameterized linear* system: it is almost linear except for the presence of the parameters c_i. Each solution to this dual constraint system provides values for the c_i corresponding to an inequality that is inductive relative to ψ. Corollary 5.1 and stronger versions [Cou05] provide the mechanism for extending this technique to polynomial inequality invariants and analyzing polynomial transition systems.

Example 5.1 (Simple). For $\mathcal{S}_{\mathsf{Simple}}$, the constraint-based method seeks inductive instantiations of the template

$$p_0 + p_1j + p_2k \geq 0$$

with parameters p_i. The (initiation) condition imposes the following constraint on the template:

$$j = 2 \ \wedge \ k = 0 \ \Rightarrow \ p_0 + p_1j + p_2k \geq 0 \ ,$$

or more simply,

$$p_0 + 2p_1 + 0p_2 \geq 0 \ . \tag{4}$$

Treating each disjunct of ρ separately, the (consecution) condition imposes two constraints:

$$p_0 + p_1j + p_2k \geq 0 \ \wedge \ j' = j + 4 \ \wedge \ k' = k \ \Rightarrow \ p_0 + p_1j' + p_2k' \geq 0 \ ,$$

or more simply

$$p_0 + p_1j + p_2k \geq 0 \ \Rightarrow \ (p_0 + 4p_1) + p_1j + p_2k \geq 0 \ ; \tag{5}$$

and

$$p_0 + p_1j + p_2k \geq 0 \ \Rightarrow \ (p_0 + 2p_1 + p_2) + p_1j + p_2k \geq 0 \ . \tag{6}$$

All together, the (primal) constraint problem is thus

$$\begin{array}{l} p_0 + 2p_1 + 0p_2 \geq 0 \\ \ \wedge \ p_0 + p_1j + p_2k \geq 0 \ \Rightarrow \ (p_0 + 4p_1) + p_1j + p_2k \geq 0 \\ \ \wedge \ p_0 + p_1j + p_2k \geq 0 \ \Rightarrow \ (p_0 + 2p_1 + p_2) + p_1j + p_2k \geq 0 \end{array} \tag{7}$$

According to Theorem 5.1, we then have the following set of dual constraints:

$$p_0 + 2p_1 + 0p_2 \geq 0$$

from (4),

$$p_1 = \lambda_1 p_1 \ \wedge \ p_2 = \lambda_1 p_2 \ \wedge \ p_0 + 4p_1 \geq \lambda_1 p_0$$

from (5), and

$$p_1 = \lambda_2 p_1 \ \wedge \ p_2 = \lambda_2 p_2 \ \wedge \ p_0 + 2p_1 + p_2 \geq \lambda_2 p_0$$

from (6). Conjoining them reveals the dual constraint system over the variables $\{p_0, p_1, p_2, \lambda_1, \lambda_2\}$.

In nontrivial solutions, $\lambda_1 = \lambda_2 = 1$. One solution is $p_0 = 0$, $p_1 = 0$, $p_2 = 1$, corresponding to the inductive invariant $k \geq 0$. ■

5.2. Incremental Invariant Generation

Extending the constraint-based method to incremental invariant generation is straightforward. Recall that $\mathsf{P}_{\mathbb{R}}(s, \psi)$ should attempt to compute an inductive affine or polynomial inequality that proves that s is unreachable. The shape of this inequality is fixed in advance via a template.

For the linear case (the polynomial case is similar), let

$$\underbrace{p_0 + p_1 x_1 + \cdots + p_n x_n}_{T} \geq 0$$

be the template and $\mathcal{C}$ be the dual constraint system generated as described in 5.1. Define the constraint system

$$\mathcal{C}_s : \ \mathcal{C} \ \wedge \ T[s] < 0 \ ,$$

where $T[s]$ is the term that results from substituting the values of the state s for the variables $\overline{x}$. Every solution of $\mathcal{C}_s$ corresponds to an inductive inequality that proves that s is unreachable.

For transition systems with more complicated control structure, the counterexample to induction s corresponds to a particular control location ℓ with associated template T_ℓ. Then $\mathcal{C}_s$ is $\mathcal{C} \wedge T_\ell[s] < 0$; that is, just T_ℓ is explicitly constrained by s.

Example 5.2 (Simple). Consider again the linear transition system $\mathcal{S}_{\mathsf{Simple}}$. Suppose, as in Example 3.1, that the inductive assertion $\chi_2 : k \geq 0 \wedge j \geq 0$ has been discovered and that state

$$s : \ \langle j = 0, \ k = 0 \rangle$$

is chosen. To discover an inductive assertion that proves the unreachability of s, we first fix the template

$$\underbrace{p_0 + p_1 j + p_2 k}_{T} \geq 0 \ .$$

Following the steps of Example 5.1 — except that ψ is used in (consecution) constraints (5) and (6) — yields constraint system $\mathcal{C}$. Finally, we construct the constraint problem

$$\mathcal{C}_s : \ \mathcal{C} \ \wedge \ \underbrace{p_0 + 0 p_1 + 0 p_2}_{T[s]} < 0 \ ,$$

that is, $\mathcal{C} \wedge p_0 < 0$. Two solutions of this constraint system are $j \geq 2$ and $j \geq 2k + 2$. Both are inductive and exclude s. ■

Example 5.3 (Integer Square-Root). Corollary 5.1 allows us to generate some polynomial inequality invariants. Consider again transition system $\mathcal{S}_{\mathsf{Sqrt}}$ and the safety assertion

$$\Pi : \ (w \leq x \ \rightarrow \ (z+1)^2 \leq x) \ \wedge \ (w > x \ \rightarrow \ x < (z+1)^2)$$

To analyze the system, we fix the template

$$\underbrace{\mathsf{Quadratic}(u, w, x, z)}_{T} \geq 0 \ .$$

Quadratic forms the most general parameterized quadratic expression over the given variables; *e.g.*,

$$\mathsf{Quadratic}(x, y) = p_0 + p_1 x + p_2 y + p_3 xy + p_4 x^2 + p_5 y^2 \ .$$

We implemented the analysis in Mathematica [WR05], using Mathematica's numerical solver to solve for counterexamples to induction and the solver of Section 5.3 to solve the constraint systems $\mathcal{C}_s$. Executing the procedure constructs the following sequence of inductive assertions, where each is inductive relative to the conjunction of the previous ones:

$$\begin{array}{lrlr}
\varphi_1: & -x+ux-2xz \geq 0 & \varphi_7: & -1+u \geq 0 \\
\varphi_2: & u \geq 0 & \varphi_8: & -2u-u^2+4w \geq 0 \\
\varphi_3: & u-u^2+4uz-4z^2 \geq 0 & \varphi_9: & -3-u^2+4w \geq 0 \\
\varphi_4: & 3u+u^2-4w \geq 0 & \varphi_{10}: & -5u-u^2+6w \geq 0 \\
\varphi_5: & x-ux+2xz \geq 0 & \varphi_{11}: & -15+22u-11u^2+4uw \geq 0 \\
\varphi_6: & 1+2u+u^2-4w \geq 0 & \varphi_{12}: & -1-2u-u^2+4w \geq 0
\end{array}$$

As usual, $\chi_{12} \stackrel{\text{def}}{=} \varphi_1 \wedge \cdots \wedge \varphi_{12}$.

On the thirteenth iteration, $\Pi \wedge \chi_{12}$ is discovered to be inductive; hence, Π is invariant. Specifically, φ_1 and φ_5 imply $u = 1 + 2z$, while φ_6 and φ_{12} imply $4w = (u + 1)^2$. Thus, $w = (z + 1)^2$, implying Π. ■

5.3. Solving Bilinear Constraint Problems

The constraint systems that arise in Step 2 of the procedures of Section 5.2 are *bilinear constraint problems*: quadratic terms are of the form λp, where λ and p are different variables. While solvable via quantifier elimination [Tar51, Col75], this complete approach does not scale because of the doubly-exponential theoretical cost and corresponding practical performance. Instead, [SSM04, BMS05, Cou05] suggest incomplete heuristic approaches, while [SSM03] describes a complete and relatively efficient method for solving the constraint systems that arise for a special class of transition systems.

We briefly describe one possible incomplete solver that has been found to work in practice. Consider the bilinear constraint system γ. Let each Lagrangian multiplier λ that appears in a bilinear term λp range over a fixed set, say $\{0, 1\}$ (this set $\{0, 1\}$ is sufficient in some cases [SSM03]). Then execute a search. Let γ_i be the current constraint system.

1. If the conjunction of the subset of *linear* constraints of γ_i is unsatisfiable, then return unsatisfiable.
2. Choose a multiplier λ_j that is unassigned and that appears bilinearly in γ_i. If no such λ_j exists, return satisfiable.
3. Try each value v in λ_j's possible solution set:
   (a) Let $\gamma_{i+1} \stackrel{\text{def}}{=} \gamma_i\{\lambda_j \mapsto v\}$.
   (b) Recurse on γ_{i+1}, returning satisfiable if the recursive call returns satisfiable.

Step 1 offers an early detection of unsatisfiability.

For solving the intermediate linear constraint systems, our implementation uses a rational solver [Avi98], thus avoiding a loss of soundness from floating point errors [Cou05].

6. Experiments

In this section, we report on our practical experiences with incremental invariant generation on finite-state systems. Unfortunately, we were not able to run all experiments that one might desire because of limited access to computational resources. However, the evidence nonetheless suggests the value of the incremental approach.

6.1. Implementation

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

6.1.1. SAT Solver

We instrumented `Z-Chaff` version `2004.11.15 Simplified` [MMZ+01] 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.

6.1.2. *Depth-First or Breadth-First Search*

If the invariant generation procedure fails to find an inductive clause excluding a counterexample to induction, the top-level analysis has several options. It can take a depth-first approach in which it focuses on this subgoal before again considering the rest of the given assertion. Alternately, it can continue to consider the assertion and all subgoals simultaneously. We implemented both strategies and indicate which strategy we use to analyze various benchmarks.

6.1.3. *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.

In 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.

In the breadth-first strategy, processes share both inductive clauses and subgoals.

6.1.4. *k-Induction*

We have recently implemented a k-induction [SSS00] strategy. The case $k = 1$ corresponds to normal induction; for larger k, the transition relation is unrolled k times. k-induction provides two advantages: it allows the analysis to search deeper in the state space for counterexamples to induction; and it can allow the invariant generation procedure to find clauses that are k-inductive for some $k > 1$, but not for lower k. However, the resulting satisfiability queries are more difficult to solve.

The strategy we employ is to use $k = 1$ for most of the analysis. However, when a counterexample to induction does not yield an inductive clause, the analysis increments k temporarily; as soon as an inductive clause is again discovered, it decrements k. Globally, a maximum of $k = 5$ was allowed in our experiments.

One difficulty in using k-induction in our context is that the disjunction of two k-inductive clauses is not k-inductive for $k > 1$. Hence, while the forward analysis works with little modification, the backward analysis is no longer complete: it may fail to find a k-inductive clause when $k > 1$. A full search is prohibitively expensive; therefore, we take a heuristic approach: preferably, a literal is dropped in the backward analysis only if it is satisfied in each of the k steps of (consecution); but if no such literal exists, then some satisfied literal is dropped. This heuristic approach is guaranteed to find a 1-inductive clause if it exists, but it may miss a k-inductive clause when $k > 1$.

6.2. Benchmarks

6.2.1. *PicoJava II Set*

We applied our analysis to the PicoJava II microprocessor benchmark set, previously studied in [MA03, McM03, McM05]. 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 [MA03] (it exhausted the available 512MB of memory on problems PJ_{17} and PJ_{18}), and interpolation-based model checking solved 19 [McM03, McM05], each within their allotted times of 1000 seconds on 930MHz machines.

Table 1. 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)

We applied the depth-first strategy without k-induction.

6.2.2. VIS Set

The second set of benchmarks are from the VIS distribution [VIS]. 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) [AS06]. k-induction with strengthening fails on PETERSON and HEAP within 1800 seconds; but BDD-based model checking requires at most a few seconds for each [AS06].

We applied the depth-first strategy without k-induction.

6.2.3. Selected Benchmarks from HWMCC'07

Several benchmarks were difficult for the participating tools in HWMCC'07 [HWM07]. We applied our analysis to some of the hardest ones as determined empirically by the outcome of the competition. Unfortunately, we do not currently have the computational resources to perform exhaustive experiments.

We applied the breadth-first and k-induction strategies, except where noted.

6.3. Results

Table 1 reports results for executing one process on one processor of a 4×1.8GHz computer with 8GB of available memory. For each benchmark, the analysis was run 16 times: 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 are reported. Results are reported only for the nontrivial benchmarks: 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 already inductive. The PicoJava II benchmarks are labeled PJ_x; the others are VIS benchmarks. All 20 of the PicoJava II benchmarks are solved, but three require more than one hour.

Figure 6 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 1. Times for 32 processes are as follows: PJ_6, 8m; PJ_{17}, 70m; PJ_{18}, 9m; and HEAP, 6m. PJ_{17} completes in 50m with 60 processes. All benchmarks complete within one hour with some number of processes.

The plot suggests that time decreases roughly linearly with more processes, but only HEAP trades pro-

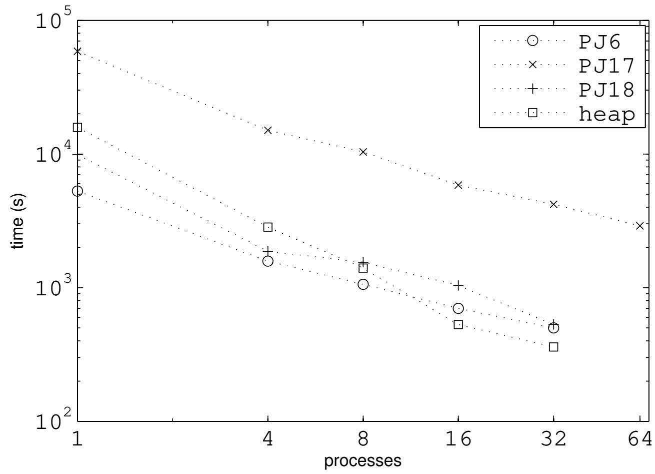

Fig. 6. Time for multiple processes

Table 2. Results on selected HWMCC'07 benchmarks

		1 process			8 processes		
Name	COI	Clauses	Time	Mem/proc (MB)	Clauses	Time	Mem/proc (MB)
INTEL_005	69	150	10s	15	190	2s	15
INTEL_006	182	1150	3m5s	66	1350	35s	45
INTEL_007	608	1720	21m	122	1660	2m40s	80
INTEL_026	349	3060	2h20m	420	4150	33m	320
INTEL_037	3196	384	39m	540	460	8m	380
AMBA_09	52	330	2h20m	1.4K	580	31m	1K
AMBA_10	58	—	—	—	710	1h10m	1.9K
NUSMV.REACTOR^2	76	2620	18m	20	2900	2m10s	15
NUSMV.REACTOR^6	76	3340	30m	22	3140	2m45s	16

cesses for time almost perfectly, possibly because it requires the most clauses. Suboptimal scaling results from generating redundant clauses.

Table 2 reports performance on several empirically difficult benchmarks from HWMCC'07 [HWM07] on an 8-core computer with 8×2.8GHz processors and 16GB of memory. Benchmarks INTEL_005 and INTEL_037 were solved during the competition each by one participant in 12 minutes and 9 minutes, respectively. The other benchmarks were not solved by any participant in the allotted 15 minutes on one processor. We do not report data in one case: the single-process analysis failed to analyze AMBA_10 in a reasonable time. The single-process analysis did not converge on other unsolved benchmarks from HWMCC'07 in a reasonable time, and limited computational resources prevented us from exploring the benchmarks further with the parallel analysis.

Interestingly, the analysis performed well on the NUSMV.REACTOR benchmarks only with depth-first handling of subgoals. Conversely, the AMBA benchmarks required a breadth-first strategy. The k-induction strategy was applied to all benchmarks except the NUMSMV.REACTOR benchmarks. A possible explanation of the poor performance of the depth-first strategy on the AMBA instances is that depth-first search can follow an unnecessarily long path backwards to an informative counterexample to induction that is actually only a few steps from violating the safety assertion. Sometimes, however, as is perhaps the case in the NUSMV.REACTOR benchmarks, certain counterexamples to induction must be encountered that are many steps away from violating the safety assertion.

7. Related Work & Concluding Remarks

7.1. Mathematical Programming-Based Analysis

For constructing witnesses to properties of systems in the form of polynomial functions, mathematical programming was first applied to synthesize *Lyapunov functions* of continuous systems [BY89] (see also [PP02] for a more recent mathematical programming-based approach). Lyapunov functions are a continuous analog of *ranking functions* of programs. A ranking function of a program maps its states into a well-founded set, thus proving that the program always terminates. Construction of affine and polynomial expressions for verification purposes was first studied extensively in the context of ranking function synthesis. Early work in this area looked at the generation of constraint systems over loops with linear assertional guards and linear assignments for which solutions are linear ranking functions [KM75]. More recently, it is observed that duality of linear constraints achieves efficient synthesis of linear ranking functions [CS01, CS02, PR04] and lexicographic linear ranking functions [BMS05]. The approach generalizes (with loss of completeness) to polynomial transition systems through semidefinite programming [Cou05].

Synthesizing invariant properties of systems is more complex because they are fixpoints of the (initiation) and (consecution) conditions. Of course, invariants are often necessary for proving termination, so that termination analysis can be just as complex; however, the synthesis of ranking functions (or Lyapunov functions) alone does not require solving for a fixpoint. As for synthesizing linear ranking functions, duality of linear constraints and nonlinear constraint solving is used to generate affine invariants [CSS03, SSM04]. The technique is specialized to apply to Petri nets, for which the problem is efficiently solvable [SSM03]. The continuous analog of invariants, *barrier certificates*, are synthesized in a similar fashion [PJ04].

7.2. Safety Analysis of Hardware

7.2.1. Qualitative Comparisons

We compare the characteristics of several safety analyses: bounded model checking (BMC) [BCCZ99], interpolation-based model checking (IMC) [McM03, McM05], k-induction (kI) [SSS00, dMRS03, AFF$^+$04, VH06, AS06], predicate abstraction with refinement (CEGAR) [CGJ$^+$03, JKSC05], and our analysis (IIG). 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.

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 IIG operate on the full transition relation. kI strengthens by requiring counterexamples to induction to be ever longer paths. In a finite-state system, there exists a longest loop-free path that ends in an error state. When kI's k is longer than this path, k-induction succeeds. IIG generalizes from counterexamples to induction to inductive clauses to exclude portions of the state space.

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 IIG pose many simple SAT problems in which the transition relation is not unrolled.

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 IIG 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 [dMRS03, AFF$^+$04, VH06, AS06].

Parallelizable. Only IIG 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 IIG and kI can be combined, and IIG can serve as the model checker for CEGAR.

7.2.2. Other Related Work

Blocking clauses are used in SAT-based unbounded model checking [McM02]. Their discovery is refined to produce *prime* blocking clauses, requiring at worst as many SAT calls as literals [JS05]. Our minimal algorithm requires asymptotically fewer SAT calls.

It has been brought to our attention that Junker described an algorithm like minimal in the context of constraint solving [Jun01]. A similar algorithm has also been described for "delta debugging" [Zel99], 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 [dMRS03, AFF$^+$04, VH06, AS06]. Quantifier-elimination [dMRS03], ATPG-based computation of the n-level preimage of s [VH06], and SAT-based preimage computation [AS06] are used to perform the strengthening. Inductive generalization can eliminate exponentially more states than preimage-based approaches.

8. Conclusion

We described a method of generating strong invariants that are sufficient for proving the invariance of given safety assertions. Unlike other methods, our approach is based on computing a sequence of simple assertions, each of which is inductive relative to those appearing before it. Computing each assertion is relatively inexpensive; and several assertions can be computed in parallel. We use counterexamples to induction to guide the invariant generation procedure to produce assertions that are relevant for proving a given safety assertion. We have observed that when our method works in practice, it produces relatively short proofs.

Our ongoing work on hardware analysis focuses on three related problems. First, we continue to refine the clause domain and operations. Second, we are searching for other useful domains for analyzing hardware. Third, we are developing a generalization of this approach to address general LTL and CTL model checking of finite-state systems.

For addressing other temporal properties of infinite-state systems, we must combine the incremental method with synthesizing ranking functions. Earlier work suggests generating supporting invariants simultaneously with the ranking function [BMS05]; however, this method, like other constraint-based approaches, is limited to synthesizing very few supporting invariants at a time. We are pursuing an approach based on using both counterexamples to induction and counterexamples to the existence of a ranking function to direct the generation of supporting invariants.

Another direction for research is to define domains and incremental analyses for addressing aspects of typical infinite-state systems other than numerical data, such as memory and data-structures.

Acknowledgments

The authors wish to thank Prof. A. Aiken, 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 and Prof. Tom Henzinger for the use of their computer clusters.

References

[AFF$^+$04] R. Armoni, L. Fix, R. Fraer, S. Huddleston, N. Piterman, and M. Vardi. SAT-based induction for temporal safety properties. In *BMC*, 2004.

[Aik99] Alex Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[AS06] Mohammad Awedh and Fabio Somenzi. Automatic invariant strengthening to prove properties in bounded model checking. In *DAC*, pages 1073–1076. ACM Press, 2006.

[Avi98] D. Avis. LRS: A revised implementation of the reverse search vertex enumeration algorithm. Technical Report, McGill, 1998.

[AW92] Alex Aiken and Edward Wimmers. Solving systems of set constraints. In *LICS*, pages 329–340, 1992.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS*, pages 193–207, London, UK, 1999. Springer-Verlag.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BM06] Aaron R. Bradley and Zohar Manna. Verification constraint problems with strengthening. In *ICTAC*, volume 3722 of *LNCS*. Springer-Verlag, 2006.

[BM07a] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007.

[BM07b] Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *FMCAD*, 2007.

[BMS05] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *CAV*, volume 3576 of *LNCS*, pages 491–504. Springer Verlag, 2005.

[BY89] S. Boyd and Q. Yang. Structured and simultaneous Lyapunov functions for system stability problems. *Int. J. Control*, 49(6):2215–2240, 1989.

[CC77] Patrick Cousot and Rhadia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71. Springer-Verlag, 1982.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allan Emerson and Prasad Sistla, editors, *CAV*, volume 1855 of *LNCS*, pages 154–169, Chicago, July 2000. Springer-Verlag.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CH78] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among the variables of a program. In *Principles of Programming Languages*, pages 84–96. ACM Press, 1978.

[Col75] G.E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H.Brakhage, editor, *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer-Verlag, 1975.

[Cou05] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, pages 1–24, 2005.

[CS01] Michael Colón and Henny B. Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *LNCS*, pages 67–81. Springer Verlag, April 2001.

[CS02] Michael Colón and Henny B. Sipma. Practical methods for proving program termination. In *CAV*, volume 2404 of *LNCS*, pages 442–454. Springer Verlag, 2002.

[CSS03] Michael Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, volume 2725 of *LNCS*, pages 420–433. Springer-Verlag, July 2003.

[dMRS03] L. de Moura, H. Ruess, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *CAV*, LNCS. Springer-Verlag, 2003.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[HWM07] Hardware model checking competition 2007 (hwmcc'07), 2007.

[JKSC05] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *DAC*, 2005.

[JS05] HoonSang Jin and Fabio Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. In *DAC*, pages 750–753. ACM Press, 2005.

[Jun01] Ulrich Junker. QuickXPlain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI Workshop on Modelling and Solving Problems with Constraints*, 2001.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[KM75] Shmuel M. Katz and Zohar Manna. A closer look at termination. *Acta Informatica*, 5(4):333–352, 1975.

[Kna28] B. Knaster. Un theoreme sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.

[MA03] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.

[McM02] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV*, volume 2404 of *LNCS*, pages 250–264. Springer-Verlag, 2002.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[McM05] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[Min01] Antoine Miné. The octagon abstract domain. In *Analysis, Slicing and Transformation (part of Working Conference on Reverse Engineering)*, IEEE, pages 310–319. IEEE Computer Society, October 2001.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, 2001.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[PJ04] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *HSCC*, volume 2993 of *LNCS*. Springer, 2004.

[PP02] A. Papachristodoulou and Stephen Prajna. On the construction of lyapunov functions using the sum of squares decomposition. In *CDC*, 2002.

[PR04] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[SSM03] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Petri net analysis using invariant generation. In Nachum Derschowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 682–701, Taurmina, Italy, 2003. Springer Verlag.

[SSM04] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constraint-based linear relations analysis. In 11^{th} *Static Analysis Symposium (SAS'2004)*, volume 3148 of *LNCS*, pages 53–68. Springer-Verlag, 2004.

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In Radhia Cousot, editor, *Proc. of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, Paris, France, January 2005. Springer Verlag.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stalmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *LNCS*. Springer-Verlag, 2000.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[VH06] Vishnu C. Vimjam and Michael S. Hsiao. Fast illegal state identification for improving SAT-based induction. In *DAC*, pages 241–246. ACM Press, 2006.

[VIS] *VIS*.

[WR05] Inc. Wolfram Research. Mathematica, version 5.2, 2005.

[WSR02] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[Zel99] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.