SAFETY ANALYSIS OF SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Aaron R. Bradley
May 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Zohar Manna) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Henny B. Sipma)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(David Dill)

Approved for the University Committee on Graduate Studies.

Abstract

As the complexity of and the dependence on engineered systems rises, correctness becomes ever more important. A system is correct with respect to its specification if all of its computations satisfy the specification. When a system's specification is provided in a formal language such as first-order logic, one can attempt to verify or to disprove that the system meets its specification.

Safety specifications are among the most common forms of specification. A safety specification asserts that every state of every computation of the system satisfies the given logical formula. The inductive method is the fundamental technique for analyzing whether a system meets its safety specification. An assertion is inductive if it holds on all initial states of the system and if it is preserved when taking any transition of the system. These two conditions are called verification conditions. If a system's safety specification is inductive over that system, then the system meets its specification. But most safety specifications are not inductive. In these cases, the inductive method suggests finding a strengthening assertion that, conjoined with the specification, is inductive. This process parallels mathematical induction: frequently, the theorem must be strengthened for the inductive argument to succeed. The inductive method is relatively complete for first-order safety specifications.

Two areas of research are crucial to making the inductive method practical. First, decision procedures automate to some extent the task of proving the validity of the first-order verification conditions. Second, invariant generation procedures generate auxiliary inductive assertions, easing the burden on the system developer to discover the strengthening assertion. This thesis presents progress in both areas.

First, this thesis focuses on decision procedures for a class of important non-recursive data structures: arrays and array-like structures. Specifically, decision procedures are developed for deciding satisfiability in fragments of first-order theories of arrays with uninterpreted indices, of arrays with integer indices, and of hashtables. The fragments, called the array (hashtable) property fragments, are larger than the quantifier-free fragments of the respective theories that have been previously studied. Moreover, they are expressive enough to enable the encoding of useful properties about the structures, for example, that two (sub)structures are equal, that all (or a subset of) elements satisfy some property, or that an integer-indexed (sub)array is sorted.

Some of these properties have been studied in isolation in the literature; this thesis unifies and extends this previous work. The decision procedures complement the Nelson-Oppen combination framework so that elements can be interpreted in another theory such as linear arithmetic or recursive data structures. Additionally, undecidability results are proved that suggest that the fragments to which the decision procedures apply are on the edge of decidability: satisfiability in natural extensions to the fragments is undecidable.

The second part of the thesis turns to the problem of discovering strengthening assertions through invariant generation procedures. This work focuses on property-directed incremental invariant generation procedures. Such procedures produce relatively weak inductive assertions at relatively low computational cost; furthermore, the generated assertions are relevant for strengthening the safety specification. Two instances of this methodology are described. The first augments constraint-based generation of affine inequality invariants to be property-directed. In the second instance, a procedure for generating minimal inductive clauses for analysis of finite-state systems is developed and then made to generate the clauses in a property-directed fashion. Experimental evidence suggests that the implementation of the procedure indeed quickly discovers inductive clauses that are useful for strengthening the specification.

Acknowledgment

I am grateful to a number of individuals for their contributions to this work. My parents invested uncountable hours into my education. Andrew has patiently listened to and discussed my ideas. I am grateful to my wife, Sarah, for her support and for many things that I won't list here.

Zohar Manna and Henny Sipma have been generous in their academic and financial support. They have fostered an environment that encourages meaningful research. I hope I have met, at least partially, their expectations. David Dill suggested improvements and different perspectives on several occassions. Tom Henzinger asked motivating questions about early versions of the work on arrays. While they have contributed directly to the research presented in this thesis, all errors and shortcomings are mine.

I thank the members of the STeP group for keeping things interesting. During my Ph.D. studies, the group included Sriram Sankaranarayanan, Henny Sipma, César Sánchez, Matteo Slanina, Calogero Zarba, and Ting Zhang. Bernd Finkbeiner tolerated my undergraduate antics but graduated before I became a Ph.D. student. Outside of the STeP group, Damon Mosk-Aoyama has been a great office mate.

Finally, I would like to thank Dr. and Mrs. Sang Samuel Wang for their generous financial support in the form of a Sang Samuel Wang Stanford Graduate Fellowship. Additional support for this work includes NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102; ARO grant DAAD19-01-1-0723; and NAVY/ONR contract N00014-03-1-0939.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis focuses on the safety analysis of systems. Informally, a system consists of *state* variables and *transitions*. For our purposes, the execution of a system consists of moving at discrete points in time from one state to another via a transition. A *safety property* is an assertion that certain states cannot be reached during the execution of a system. A *safety analysis* attempts to decide whether a system has a given safety property. These concepts are formalized in Section 1.2.

The fundamental technique that underlies all safety analyses is induction over transitions. In this induction principle, the *base case*, also known as *initiation*, asserts that all initial states of the system obey the given safety property. The *inductive step*, also known as *consecution*, asserts that every transition preserves the safety property: if a state satisfies the safety property, then all states than can be reached by taking one transition from the state also satisfy the property. Unfortunately, the inductive step fails on most safety properties. This situation, in which the asserted property does not provide a strong enough hypothesis for the inductive step, is typical when applying induction. When this failure occurs, one seeks a stronger property that is inductive and that also implies the desired property. These concepts are formalized in Section 1.3.

The induction-based analysis presents two challenges. The first challenge is to prove the base case and the inductive step. The second challenge is to strengthen the given property to be inductive.

We address elements of each of these challenges. In Chapter 2, we consider theorem proving in the presence of arrays. An array is a basic and ubiquitous data structure in software that maps a subset of natural numbers to values in some domain. Our contribution is joint work with Henny Sipma and Zohar Manna [BMS06].

In Chapter 3, we consider the procedural construction of strengthened inductive properties. We describe an approach to the synthesis of weak inductive invariants that applies counterexamples to induction to focus the analysis. Our contribution to invariant generation of affine

```
@pre 0 ≤ u, ℓ < |a0|
@post ∀i, j. ℓ ≤ i ≤ j ≤ u → rv[i] ≤ rv[j]
        ∧ |rv| = |a0|
        ∧ ∀i. 0 ≤ i < ℓ → rv[i] = a0[i]
        ∧ ∀i. u < i < |rv| → rv[i] = a0[i]
int[] BubbleSort(int[] a0, int ℓ, int u) {
  int[] a := a0;
  for @ ⊤
    (int m := u; m > ℓ; m := m − 1)
    for @ ⊤
      (int n := ℓ; n < m; n := n + 1)
      if (a[n] > a[n + 1]) {
        int t := a[n];
        a[n] := a[n + 1];
        a[n + 1] := t;
      }
  return a;
}
```

Figure 1.1: BubbleSort with function specification

inequalities is joint work with Zohar Manna [BM06]. Our work on invariant generation of clauses for hardware analysis has not been previously published.

1.1 A Simple Example

To motivate and unify the concepts introduced in this chapter, we verify a simple specified program. BubbleSort, shown in Figure 1.1, should return (if it returns) an array whose elements are sorted in the range $[\ell, u]$, whose elements outside of the range $[\ell, u]$ are as in the input array, and whose multiset of elements are a permutation of the multiset of elements of the input array. Figure 1.1 lists a *function specification* consisting of a function precondition and a function postcondition. The precondition indicates the values of the formal parameters a_0, ℓ, u on which BubbleSort is defined; the postcondition indicates the relation among the output value rv and the formal parameters a_0, ℓ, u upon return.

Let us examine the specification. The function precondition asserts that u and ℓ ought to be within the domain of a_0. The first line of the function postcondition asserts that the output array rv is sorted in the range $[\ell, u]$. The final two lines assert that the rest of rv equals the input a_0. This requirement is sometimes called a *frame condition*. Unlike the assertion that part of the array is sorted, these frame conditions represent common specifications. One often needs to indicate which parts of a data structure a function leaves unchanged.

Unfortunately, the decision procedure of Chapter 2 does not allow us to assert that the output

```
@pre 0 ≤ u, ℓ < |a0|
@post ∀i, j. ℓ ≤ i ≤ j ≤ u → rv[i] ≤ rv[j]
        ∧ |rv| = |a0|
        ∧ ∀i. 0 ≤ i < ℓ → rv[i] = a0[i]
        ∧ ∀i. u < i < |rv| → rv[i] = a0[i]
int[] BubbleSort(int[] a0, int ℓ, int u) {
  int[] a := a0;
  for
      @L1 : [ m ≤ u ∧ |a| = |a0|
              ∧ ∀i, j. m ≤ i ≤ j ≤ u → a[i] ≤ a[j]
              ∧ ∀i, j. ℓ ≤ i ≤ m < j ≤ u → a[i] ≤ a[j]
              ∧ ∀i. 0 ≤ i < ℓ → a[i] = a0[i]
              ∧ ∀i. u < i < |a| → a[i] = a0[i] ]
    (int m := u; m > ℓ; m := m − 1)
    for
        @L2 : [ ℓ < m ≤ u ∧ ℓ ≤ n ≤ m ∧ |a| = |a0|
                ∧ ∀i, j. m ≤ i ≤ j ≤ u → a[i] ≤ a[j]
                ∧ ∀i, j. ℓ ≤ i ≤ m < j ≤ u → a[i] ≤ a[j]
                ∧ ∀i. ℓ ≤ i < n → a[i] ≤ a[n]
                ∧ ∀i. 0 ≤ i < ℓ → a[i] = a0[i]
                ∧ ∀i. u < i < |a| → a[i] = a0[i] ]
      (int n := ℓ; n < i; n := n + 1)
      if (a[n] > a[n + 1]) {
        int t := a[n];
        a[n] := a[n + 1];
        a[n + 1] := t;
      }
  return a;
}
```

Figure 1.2: BubbleSort with inductive annotations

array is a permutation of the input array (for adding the ability to express permutation to the studied fragment makes satisfiability undecidable). Therefore, the second line merely asserts that the length of the output array is the same as the input array. Combined with the frame conditions, only the elements within the range $[\ell, u]$ cannot be accounted for (other than that they are nondecreasing). Specifying properties within decidable fragments frequently requires such compromises in practice.

This specification is a safety property. We prove that the implementation satisfies the specification using the inductive method. Figure 1.2 lists BubbleSort with a strengthened specification. Some of the additional annotations must be added manually, particularly those that assert facts about the manipulated array. However, Chapter 3 discusses techniques that can discover the numerical facts, such as the bounds on the loop variables, procedurally.

$$
\begin{bmatrix}
\ell < m \leq u \ \wedge\ \ell \leq n \leq m \ \wedge\ |a| = |a_0| \\
\wedge\ \forall i, j.\ m \leq i \leq j \leq u \ \rightarrow\ a[i] \leq a[j] \\
\wedge\ \forall i, j.\ \ell \leq i \leq m < j \leq u \ \rightarrow\ a[i] \leq a[j] \\
\wedge\ \forall i.\ \ell \leq i < n \ \rightarrow\ a[i] \leq a[n] \\
\wedge\ \forall i.\ 0 \leq i < \ell \ \rightarrow\ a[i] = a_0[i] \\
\wedge\ \forall i.\ u < i < |a| \ \rightarrow\ a[i] = a_0[i] \\
\wedge\ n < m \ \wedge\ a[n] > a[n+1]
\end{bmatrix}
$$

$$
\Rightarrow
\begin{bmatrix}
\ell < m \leq u \ \wedge\ \ell \leq n+1 \leq m \ \wedge\ |a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle| = |a_0| \\
\wedge\ \forall i, j.\ m \leq i \leq j \leq u \\
\qquad \rightarrow\ a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[i] \leq a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[j] \\
\wedge\ \forall i, j.\ \ell \leq i \leq m < j \leq u \\
\qquad \rightarrow\ a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[i] \leq a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[j] \\
\wedge\ \forall i.\ \ell \leq i < n \\
\qquad \rightarrow\ a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[i] \leq a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[n] \\
\wedge\ \forall i.\ 0 \leq i < \ell \ \rightarrow\ a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[i] = a_0[i] \\
\wedge\ \forall i.\ u < i < |a| \ \rightarrow\ a\langle n \triangleleft a[n+1]\rangle\langle n+1 \triangleleft a[n]\rangle[i] = a_0[i]
\end{bmatrix}
$$

Figure 1.3: One verification condition of BubbleSort

The remaining task is to prove the validity of the verification conditions that the fully annotated implementation induces. For example, the path through the inner loop L_2 in which two elements are swapped induces the verification condition of Figure 1.3. The symbols of this formula lie in both the signatures Σ_{A} and $\Sigma_{\mathbb{Z}}$, so its validity must be determined in the combination theory $T_{\mathsf{A}} \cup T_{\mathbb{Z}}$. Chapter 2 discusses a decision procedure that proves the validity of this formula and the other verification conditions, thus proving that BubbleSort meets its function specification.

1.2 Transition Systems

Software and hardware are modeled in first-order logic via transition systems.

Definition 1.2.1 (Transition System) A *transition system* $\mathcal{S}$: $\langle \overline{x}, \theta, \rho \rangle$ contains three components:

- a set of *program 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 $\overline{x}$. □

Example 1.2.1 The transition system

$$\mathcal{S}_1 : \langle \underbrace{\{x : \mathbb{Z}\}}_{\overline{x}}, \underbrace{x \geq 0}_{\theta}, \underbrace{x' = 0 \ \vee \ x' > x}_{\rho} \rangle$$

describes a system consisting of one real variable x that is initially nonnegative. It evolves as follows: in each step of the system, x is either reset to 0 ($x' = 0$) or increased ($x' > x$). □

Several restricted forms of transition systems will be of interest in this thesis. 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 (see Section 1.3) is also an invariant when considering the variables to range over $\mathbb{Z}$.

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

Definition 1.2.2 (State & Computation) A *state* $\overline{s}$ of transition system $\mathcal{S}$ is an assignment of values (of the proper type) to variables $\overline{x}$. A computation $\sigma : \ \overline{s}_0, \overline{s}_1, \overline{s}_2, \ldots$ is an infinite sequence of states such that

- $\overline{s}_0$ satisfies the initial condition: $\theta[\overline{s}_0]$, or $\overline{s}_0 \models \theta$,
- and for each $i \geq 0$, $\overline{s}_i$ and $\overline{s}_{i+1}$ are related by ρ: $\rho[\overline{s}_i, \overline{s}_{i+1}]$, or $(\overline{s}_i, \overline{s}_{i+1}) \models \rho$.

□

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

Example 1.2.2 $\langle x = 1 \rangle$ is a state of transition system $\mathcal{S}_1$. One possible computation of the system is the following:

$$\sigma : \ \langle x = 1 \rangle, \ \langle x = 2 \rangle, \ \langle x = 0 \rangle, \ \langle x = 11 \rangle, \ \ldots \ .$$

Each state of the computation is (obviously) reachable, while state $\langle x = -1 \rangle$ is not reachable. □

1.3 The Inductive Method

A *safety property* 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 $\overline{s}$ that satisfy Π ($\overline{s} \models \Pi$) are reachable by $\mathcal{S}$. Invariants and inductive invariants are central to studying safety properties of transitions systems.

Definition 1.3.1 (Inductive Invariant) A formula φ is an *invariant* of $\mathcal{S}$ (or is $\mathcal{S}$-*invariant*) if for every computation $\sigma: \overline{s}_0, \overline{s}_1, \overline{s}_2, \ldots$, for every $i \geq 0$, $\overline{s}_i \models \varphi$. 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 entailment: $\varphi \ \Rightarrow\ \psi$ 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 $\varphi \wedge \rho \ \Rightarrow\ \varphi'$. □

Example 1.3.1 The property $\Pi: x \geq 0$ is $\mathcal{S}_1$-inductive. For the following verification conditions are valid (in the theory of integers):

- $x \geq 0 \ \Rightarrow\ x \geq 0$ (initiation)
- $x \geq 0 \ \wedge\ (x' = 0 \ \vee\ x' > x) \ \Rightarrow\ x' \geq 0$ (consecution)

Hence Π is $\mathcal{S}$-invariant. □

The main problem that we consider is the following: Given transition system $\mathcal{S}: \langle \overline{x}, \theta, \rho \rangle$ and specification $\Pi[\overline{x}]$, is Π $\mathcal{S}$-invariant? Proving that Π is inductive answers the question affirmatively. But frequently Π is invariant yet not inductive. The *inductive method* [MP95] suggests finding a formula χ such that $\Pi \wedge \chi$ is inductive; χ is called a *strengthening assertion*. Finding such a χ is the focus of Chapter 3.

Example 1.3.2 Consider the transition system

$$\mathcal{S}_2: \ \langle \{x: \mathbb{Z}\},\ x = -1,\ x' = -x \rangle\ .$$

The property $\Pi: x \geq -1$ is $\mathcal{S}_2$-invariant, for in no state of the only computation of $\mathcal{S}_2$,

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

does x ever have value less than -1. But it is not $\mathcal{S}_2$-inductive:

- $x = -1 \ \Rightarrow\ x \geq -1$ (initiation)
- $x \geq -1 \ \wedge\ (x' = -x) \ \Rightarrow\ x' \geq -1$ (consecution)

The (consecution) condition fails with, for example, a falsifying integer arithmetic interpretation (see Section 1.4) in which $x = 2$.

The strengthening assertion $\chi : x \leq 1$ produces the inductive assertion $\Pi \wedge \chi : \ -1 \leq x \leq 1$:

- $x = -1 \ \Rightarrow \ -1 \leq x \leq 1$ (initiation)
- $-1 \leq x \leq 1 \ \wedge \ (x' = -x) \ \Rightarrow \ -1 \leq x' \leq 1$ (consecution)

Now both verification conditions are valid in integer arithmetic. Hence, Π is $\mathcal{S}_2$-invariant. □

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

Definition 1.3.2 (Counterexample Trace) A *counterexample trace* of system $\mathcal{S}$ and specification Π is a finite sequence of states $\sigma : \ \overline{s}_0, \overline{s}_1, \overline{s}_2, \ldots, \overline{s}_k$ such that

- $\overline{s}_0$ satisfies the initial condition: $\overline{s}_0 \models \theta$,
- for each $i \in [0, k-1]$, $\overline{s}_i$ and $\overline{s}_{i+1}$ are related by ρ: $(\overline{s}_i, \overline{s}_{i+1}) \models \rho$,
- and $\overline{s}_k$ violates Π: $\overline{s}_k \models \neg\Pi$.

□

Example 1.3.3 A counterexample trace to the proposed safety property $\Pi : x \leq 100$ of $\mathcal{S}_1$ is the following:

$$\sigma : \ \langle x = 3 \rangle, \ \langle x = 13 \rangle, \ \langle x = 0 \rangle, \ \langle x = 16 \rangle, \ \langle x = 1369 \rangle \ .$$

Hence, Π is not $\mathcal{S}_1$-invariant. □

1.4 Decision Procedures

Proving the first-order verification conditions of Definition 1.3.1 requires a theorem prover. To automate this task when possible, decision procedures are applied. Chapter 2 discusses decision procedures to reason about arrays and array-like data structures. These data structures are widely used in software and in hardware specifications. It also discusses more theoretical aspects of several theories of arrays to determine bounds on decidability when reasoning about arrays. We introduce the notation and concepts necessary for this study.

A first-order theory T is characterized by its *signature* Σ and a set of *axioms*. The signature Σ is a set of constant, function, and predicate symbols; the function and predicate symbols have fixed arity. A Σ-formula is a first-order formula whose constant, function, and predicate symbols all appear in Σ. A *term* is a variable, constant symbol, or application of a function to a set of

terms. An *atom* is the application of a predicate symbol to a set of terms. A *literal* is an atom or its negation. The axioms of T are a (countable) set of Σ-formulae. A T-*interpretation* $I : \langle D, \alpha \rangle$ is a structure consisting of a domain D and an assignment α that assigns meaning to the symbols of Σ: α assigns each constant symbol c a value $\alpha_I[c] \in D$; each n-ary function symbol f a function $\alpha_I[f] : D^n \to D$; and each n-ary predicate symbol p a predicate $\alpha_I[p] : D^n \to \mathbb{B}$. Moreover, each axiom of T evaluates to true on I according to first-order semantics; in other words, I satisfies each axiom.

One of the main questions about a Σ-formula F is whether it is T-satisfiable: Does there exist a T-interpretation that satisfies F? A *decision procedure* for a theory is an algorithm that decides whether a given Σ-formula is T-satisfiable.

A *fragment* of a theory T is a syntactically restricted set of Σ-formulae. For many theories, the *quantifier-free* fragment is well-studied because satisfiability is often (efficiently) decidable when it is undecidable or of high computational complexity for the full theory. One can view the quantifier-free fragment of a theory T in two ways. If one extends the signature Σ to include a countable set of constant symbols, then a quantifier-free Σ-formula F does not contain variables but just symbols of the extended signature. Alternately, using the original signature Σ, every variable of F is implicitly existentially quantified. Either perspective yields the same result for considering T-satisfiability of F. When we say that a fragment of a theory is *decidable* we mean that there exists a decision procedure for deciding satisfiability of formulae in that fragment.

We often drop T and Σ from T-satisfiable, T-interpretation, Σ-formula, *etc.*, when they are clear from the context.

Common theories include arithmetic without multiplication over integers $T_{\mathbb{Z}}$ and rationals $T_{\mathbb{Q}}$, equality in the presence of uninterpreted functions $T_{=}$, recursive data structures like lists T_{cons}, and arrays T_{A}. Typically, only the quantifier-free fragments are widely used; indeed, satisfiability in the full theories is undecidable except in the case of $T_{\mathbb{Z}}$ and $T_{\mathbb{Q}}$.

Example 1.4.1 The signatures $\Sigma_{\mathbb{Z}}$ and $\Sigma_{\mathbb{Q}}$ of the linear arithmetic theories $T_{\mathbb{Z}}$ and $T_{\mathbb{Q}}$, respectively, include constant symbols 0 and 1 (and, in practice, $2, 3, 4, \ldots$), binary function symbol $+$ and unary function symbol $-$, and binary predicate symbols $<$ and $=$. The formula

$$F : \ \exists x, y.\ 0 < x \ \wedge\ 0 < y \ \wedge\ 2x + 3y < 5$$

is both a $\Sigma_{\mathbb{Z}}$-formula and a $\Sigma_{\mathbb{Q}}$-formula. Moreover, for satisfiability purposes, F is in the quantifier-free fragments of both. While their signatures are similar, the axioms of $T_{\mathbb{Z}}$ and $T_{\mathbb{Q}}$ are not: $T_{\mathbb{Z}}$-interpretations satisfy the same $\Sigma_{\mathbb{Z}}$-formulae as the structure $\mathbb{Z}$, while $T_{\mathbb{Q}}$-interpretations satisfy the same $\Sigma_{\mathbb{Q}}$-formulae as the structures $\mathbb{Q}$ and $\mathbb{R}$. Hence, for example, F is $T_{\mathbb{Z}}$-unsatisfiable but $T_{\mathbb{Q}}$-satisfiable.

The signature $\Sigma_{=}$ of $T_{=}$ includes the binary predicate $=$ and every constant, function, and

predicate symbol. Its axioms force functions and predicates in $T_=$-interpretations to behave as expected under equality; for example, if $x = y$, then $f(x) = f(y)$. The quantifier-free $\Sigma_=$-formula

$$f(f(a)) = a \ \wedge \ f(f(f(a))) = a \ \wedge \ f(a) \neq a$$

is $T_=$-unsatisfiable.

The signature Σ_{cons} of the theory of LISP-like lists T_{cons} includes binary function symbol cons, unary function symbols car and cdr, and unary predicate atom. The Σ_{cons}-formula

$$\neg\mathsf{atom}(x) \ \wedge \ \neg\mathsf{atom}(y) \ \wedge \ \mathsf{car}(x) = \mathsf{car}(y) \ \wedge \ \mathsf{cdr}(x) = \mathsf{cdr}(y) \ \wedge \ x \neq y$$

is T_{cons}-unsatisfiable.

The theory of arrays T_{A} is discussed in detail in Chapter 2. □

In practice, combination theories interest us. For a program does not typically have just one data type but many, including integers, recursive data structures, and arrays. A *combination theory* $T = T_1 \cup T_2$ is defined from two (or more) base theories T_1 and T_2. Its signature $\Sigma = \Sigma_1 \cup \Sigma_2$ is the union of the signatures Σ_1 and Σ_2, and its axioms consist of the union of the axioms of T_1 and T_2. The Nelson-Oppen combination method [NO79] provides a generic method for constructing a decision procedure for the quantifier-free fragment of a combination theory T when decision procedures are available for the quantifier-free fragments of its base theories T_1 and T_2; when the base signatures share only the equality symbol, $\Sigma_1 \cap \Sigma_2 = \{=\}$; and when T_1 and T_2 are stably infinite, a technical requirement that is not relevant to our discussion. For example, the quantifier-free fragment of the combination theory $T_{\mathbb{Z}} \cup T_{\mathsf{A}} \cup T_=$ is decidable via a Nelson-Oppen combination of decision procedures for the quantifier-free fragments of $T_{\mathbb{Z}}$, T_{A}, and $T_=$.

1.5 Further Reading

This introductory chapter presents only the basic notations and concepts necessary for understanding this thesis. I invite the reader interested in a more comprehensive introduction to first-order logic, verification, and decision procedures to read our textbook, *The Calculus of Computation* [BM07]. For example, Chapter 5 of the book discusses the inductive method in great depth. Chapters 2 and 3 cover first-order logic and first-order theories. Chapter 10 discusses the Nelson-Oppen method that allows us to apply our decision procedure for arrays to interesting systems.

Chapter 2

Reasoning About Arrays

Arrays are a basic nonrecursive data type in imperative programming, so reasoning about them is important. McCarthy first axiomatized arrays [McC62], and James King implemented a decision procedure for the quantifier-free fragment for his dissertation [Kin69]. Several authors discuss decision procedures for quantifier-free fragments of various augmented array theories that include predicates useful for reasoning about the sorted aspect of sorting algorithms [Mat81, Jaf81]; and for a restricted quantifier-free fragment of an augmented theory that includes a permutation predicate but excludes arbitrary writes, instead allowing only swaps [SJ80]. Satisfiability in the quantifier-free fragment of the extensional theory, in which two arrays are equal precisely when their corresponding elements are equal, was shown to be decidable [SBDL01]. This chapter is based on our study of arrays [BMS06].

We consider arrays with uninterpreted indices (T_{A}, Section 2.2) and with integer indices ($T_{\mathsf{A}}^{\mathbb{Z}}$, Section 2.3), and with elements interpreted in some theory T. In general, we are interested in element theories T for which satisfiability in the quantifier-free fragments of the combination theories $T_{\mathsf{A}} \cup T$ and $T_{\mathsf{A}}^{\mathbb{Z}} \cup T$ is decidable, perhaps via a Nelson-Oppen combination [NO79]. In each theory T_{A} and $T_{\mathsf{A}}^{\mathbb{Z}}$, we address two questions:

1. Does there exist a fragment for which satisfiability is decidable and in which at least the predicates of [SJ80, Mat81, Jaf81, SBDL01] can be expressed?

2. What is a (tight) upper bound on decidability?

Since the combination with element theories interests us, the questions should be phrased more technically:

1. Does there exist a generic fragment of $T_{\mathsf{A}} \cup T$ ($T_{\mathsf{A}}^{\mathbb{Z}} \cup T$) such that if satisfiability in the quantifier-free fragment of $T_{\mathsf{A}} \cup T$ ($T_{\mathsf{A}}^{\mathbb{Z}} \cup T$) is decidable, then is it also decidable in the larger fragment? Does this generic fragment (with the appropriate element theories) encompass the predicates of [SJ80, Mat81, Jaf81, SBDL01]?

2. Is this fragment an upper bound on decidability? That is, for every "natural" extension to the fragment, does there exist an element theory T such that satisfiability in the quantifier-free fragment of $T_{\mathsf{A}} \cup T$ ($T^{\mathbb{Z}}_{\mathsf{A}} \cup T$) is decidable, but satisfiability in the extended fragment is not decidable?

Addressing the first question, we describe the *array property fragment* of $T_{\mathsf{A}} \cup T$ (Section 2.2) and of $T^{\mathbb{Z}}_{\mathsf{A}} \cup T$ (Section 2.3). This fragment expresses the definitions of the predicates of [SJ80, Mat81, Jaf81, SBDL01] except the permutation predicate. Addressing the second question, we prove negative results for several natural extensions to these array property fragments with integer elements (Section 2.4). For example, adding the permutation predicate results in a fragment for which satisfiability is undecidable. Permutation with equations among unaligned subarrays was shown to be undecidable [Ros86]; however, allowing equations among unaligned subarrays would already result in undecidability in our context because we allow array elements to be fully interpreted in, for example, the theory of integers $T_{\mathbb{Z}}$.

Hashtables are another important data type. They are similar to arrays with uninterpreted indices in that their indices, or keys, are uninterpreted. However, hashtables allow two new interesting operations: first, a key/value pair can be removed; and second, a hashtable's domain — its set of keys that it maps to values — can be read. We formalize reasoning about hashtables in the theory T_{H}. Using the decision procedure of Section 2.2, Section 2.5 describes a decision procedure for the *hashtable property fragment* of T_{H} based on reducing Σ_{H}-formulae to Σ_{A}-formulae in the array property fragment.

A variation on arrays in which elements may be undefined has been considered recently [GNRZ06]; we call such arrays *partial arrays*. As in [SJ80, Mat81, Jaf81, SBDL01], the authors augment the basic theory of arrays with several predicates and provide a decision procedure for the quantifier-free fragment of the resulting theory. The two questions asked above of the standard theories of arrays are relevant to partial arrays as well. We answer these questions and show that the decision procedure for a fragment of partial arrays is a semi-decision procedure for the corresponding fragment of the standard array theories (Section 2.6).

The main technique applied in all the decision procedures of this chapter is to instantiate universally quantified indices over finite sets of index terms appearing in the formula; that is, to convert universal quantification to finite conjunction. Surprisingly, this rather simple basis for the decision procedures carries us essentially to the border of decidability.

2.1 The Theory of Arrays

The theory of arrays T_{A} has signature

$$\Sigma_{\mathsf{A}} : \ \{\cdot[\cdot],\ \cdot\langle\cdot \lhd \cdot\rangle,\ =\} \ ,$$

where

- $a[i]$ is a binary function representing the read of array a at index i;
- $a\langle i \triangleleft v\rangle$ is a ternary function representing the write of value v to index i of array a;
- and $=$ is a binary predicate.

The axioms of T_{A} are the following:

- $\forall x.\ x = x$ (reflexivity)
- $\forall x, y.\ x = y \ \rightarrow\ y = x$ (symmetry)
- $\forall x, y, z.\ x = y \ \wedge\ y = z \ \rightarrow\ x = z$ (transitivity)
- $\forall a, i, j.\ i = j \ \rightarrow\ a[i] = a[j]$ (array congruence)
- $\forall a, v, i, j.\ i = j \ \rightarrow\ a\langle i \triangleleft v\rangle[j] = v$ (read-over-write 1)
- $\forall a, v, i, j.\ i \neq j \ \rightarrow\ a\langle i \triangleleft v\rangle[j] = a[j]$ (read-over-write 2)

Satisfiability in the quantifier-free fragment of T_{A} is easily seen to be decidable based on this axiomatization [Kin69]. Given quantifier-free Σ_{A}-formula F, perform the following recursive steps:

1. If F does not contain any write terms $a\langle i \triangleleft v\rangle$, then
   (a) associate array variables a with fresh function symbol f_a, and replace read terms $a[i]$ with $f_a(i)$;
   (b) and decide the $T_{=}$-satisfiability of the resulting quantifier-free $\Sigma_{=}$-formula [Ack57, Sho78, NO80, DST80].
2. Otherwise, select some read-over-write term $a\langle i \triangleleft v\rangle[j]$ (note that a may itself be a write term) and split on two cases:
   (a) According to (read-over-write 1), replace

   $$F[a\langle i \triangleleft v\rangle[j]] \quad \text{with} \quad F_1 :\ F[v] \ \wedge\ i = j\ ,$$

   and recurse on F_1. If F_1 is found to be T_{A}-satisfiable, return satisfiable.
   (b) According to (read-over-write 2), replace

   $$F[a\langle i \triangleleft v\rangle[j]] \quad \text{with} \quad F_2 :\ F[a[j]] \ \wedge\ i \neq j\ ,$$

   and recurse on F_2. If F_2 is found to be T_{A}-satisfiable, return satisfiable.

If both F_1 and F_2 are found to be T_{A}-unsatisfiable, return unsatisfiable.

Reasoning about arrays with integer indices is often useful. The combination of T_{A} and $T_{\mathbb{Z}}$ yields the theory $T_{\mathsf{A}}^{\mathbb{Z}}$ which is appropriate for reasoning about such arrays. Additionally, it is almost always essential to reason about the elements of arrays. Again, combination theories are appropriate; for example, integer-indexed arrays of list elements can be reasoned about in the combined theory $T_{\mathsf{A}}^{\mathbb{Z}} \cup T_{\mathsf{cons}}$. In all of our results, we keep such combinations in mind.

2.2 Arrays with Uninterpreted Indices

The quantifier-free fragment of T_{A} essentially allows basic reasoning in the presence of arrays. For verification purposes, it allows verifying properties of individual elements but not of entire arrays. However, in practice, reasoning about properties such as equality between arrays is often desired. Using combinations of theories, one would also like to reason about properties such as that all integer elements of an array are positive.

2.2.1 The Array Property Fragment

In this section, we define a decidable fragment of T_{A} that allows some quantification. This fragment is called the *array property fragment* because it allows specifying basic properties of arrays, not just properties of array elements. The principal characteristic of the array property fragment is that array indices can be universally quantified with some restrictions.

Example 2.2.1 In the Σ_{A}-formula

$$a[i] \neq v \ \wedge \ \forall j.\ a\langle i \triangleleft v\rangle[j] = a[j] \ ,$$

the second conjunct asserts that $a\langle i \triangleleft v\rangle$ and a are equal. This formula is T_{A}-unsatisfiable. □

Unfortunately, the use of universal quantification must be restricted to avoid undecidability (see Section 2.4). An *array property* is a Σ_{A}-formula of the form

$$\forall \overline{i}.\ F[\overline{i}] \ \rightarrow \ G[\overline{i}] \ ,$$

where $\overline{i}$ is a list of variables and $F[\overline{i}]$ and $G[\overline{i}]$ are the *index guard* and the *value constraint*, respectively. The index guard $F[\overline{i}]$ is any Σ_{A}-formula that is syntactically constructed according to the following grammar:

$$\begin{array}{rcl} \mathsf{iguard} & \rightarrow & \mathsf{iguard} \wedge \mathsf{iguard} \mid \mathsf{iguard} \vee \mathsf{iguard} \mid \mathsf{atom} \\ \mathsf{atom} & \rightarrow & \mathsf{var} = \mathsf{var} \mid \mathit{evar} \neq \mathsf{var} \mid \mathsf{var} \neq \mathit{evar} \mid \top \\ \mathsf{var} & \rightarrow & \mathit{evar} \mid \mathit{uvar} \end{array}$$

where *uvar* is any universally quantified index variable and *evar* is any constant or free variable.

Additionally, a universally quantified index can occur in a value constraint $G[\vec{i}]$ only in a read $a[i]$, where a is an array term. The read cannot be nested; for example, $a[b[i]]$ is not allowed.

The *array property fragment* of T_{A} then consists of formulae that are Boolean combinations of quantifier-free Σ_{A}-formulae and array properties.

Example 2.2.2 The antecedent of the implication in the Σ_{A}-formula

$$F: \ \forall i.\ i \neq a[k] \ \rightarrow \ a[i] = a[k]$$

is not a legal index guard since $a[k]$ is not a variable (neither a *uvar* nor an *evar*); however, a simple manipulation makes it conform:

$$F': \ v = a[k] \ \wedge \ (\forall i.\ i \neq v \ \rightarrow \ a[i] = a[k])$$

Here, $i \neq v$ is a legal index guard, and $a[i] = a[k]$ is a legal value constraint. F and F' are equisatisfiable.

However, no amount of manipulation can make the following formula conform:

$$G: \ \forall i.\ i \neq a[i] \ \rightarrow \ a[i] = a[k]\ .$$

Thus, G is not in the array property fragment. □

Example 2.2.3 The array property fragment allows expressing equality between arrays, a property referred to as *extensionality*: two arrays are equal precisely when their corresponding elements are equal. For given formula

$$F: \ \cdots \ \wedge \ a = b \ \wedge \ \cdots$$

with array terms a and b, rewrite F as

$$F': \ \cdots \ \wedge \ (\forall i.\ a[i] = b[i]) \ \wedge \ \cdots\ .$$

F and F' are equisatisfiable. Moreover, the index guard in the new subformula is just $\top$, and the value constraint $a[i] = b[i]$ obeys the requirement that i appear only as an index in read terms.

Subsequently, when convenient, we write equality $a = b$ between arrays to abbreviate $\forall i.\ a[i] = b[i]$. □

When considering an element theory T with signature Σ we consider the array property fragment of $T_{\mathsf{A}} \cup T$. Its definition extends naturally from T_{A} because interaction with universally

quantified index variables is already restricted: all quantifier-free constructs arising from Σ are allowed.

Example 2.2.4 The array property fragment allows expressing universal properties of elements of arrays. That is, given some quantifier-free T-formula $F[x]$ with free variable x, the formula

$$\forall i.\ F[a[i]]$$

is in the array property fragment of $T_{\mathsf{A}} \cup T$, as is, for example,

$$\forall i.\ i \neq j \ \rightarrow\ F[a[i]]\ .$$

□

2.2.2 A Decision Procedure

The idea of the decision procedure for the array property fragment is to reduce universal quantification to finite conjunction. That is, it constructs a finite set of index terms such that examining only these positions of the arrays is sufficient.

Example 2.2.5 Consider the formula

$$F:\ a\langle i \lhd v\rangle = a\ \wedge\ a[i] \neq v\ ,$$

which expands to

$$F':\ \forall j.\ a\langle i \lhd v\rangle[j] = a[j]\ \wedge\ a[i] \neq v\ .$$

Intuitively, to determine that F' is T_{A}-unsatisfiable requires examining just index i:

$$F'':\ \bigwedge_{j \in \{i\}} a\langle i \lhd v\rangle[j] = a[j]\ \wedge\ a[i] \neq v\ ,$$

or simply

$$a\langle i \lhd v\rangle[i] = a[i]\ \wedge\ a[i] \neq v\ .$$

Simplifying,

$$v = a[i]\ \wedge\ a[i] \neq v\ ,$$

it is clear that this formula, and thus F, is T_{A}-unsatisfiable. □

Given array property formula F, decide its T_{A}-satisfiability by the following steps.

Step 1

Put F in positive normal form: push negations down to literals.

Step 2

Apply the following rule exhaustively to remove writes:

$$\frac{F[a\langle i \lhd v\rangle]}{F[a'] \ \wedge \ a'[i] = v \ \wedge \ (\forall j.\ j \neq i \ \rightarrow \ a[j] = a'[j])} \text{ for fresh } a' \quad (\mathsf{write})$$

Rules should be read from top to bottom. For example, this rule states that given a formula F containing an occurrence of a write term $a\langle i \lhd v\rangle$, substitute every occurrence of $a\langle i \lhd v\rangle$ with a fresh variable a' and conjoin additional constraints.

This step deconstructs write terms in a straightforward manner, essentially encoding the (read-over-write) axioms into the new formula. After an application of the rule, the resulting formula contains at least one fewer write terms than the given formula.

Step 3

Apply the following rule exhaustively to remove existential quantification:

$$\frac{F[\exists \overline{i}.\ G[\overline{i}]]}{F[G[\overline{j}]]} \text{ for fresh } \overline{j} \quad (\mathsf{exists})$$

Existential quantification can arise during Step 1 if the given formula has a negated array property.

Step 4

Steps 4-6 accomplish the reduction of universal quantification to finite conjunction. The main idea is to select a set of symbolic index terms on which to instantiate all universal quantifiers. The proof of Theorem 2.2.1 argues that the following set is sufficient for correctness.

From the output F_3 of Step 3, construct the *index set* $\mathcal{I}$:

$$\begin{array}{rcll} & & \{\lambda\} & \\ \mathcal{I} & = & \cup & \{t \ : \ \cdot[t] \in F_3 \text{ such that } t \text{ is not a universally quantified variable}\} \\ & & \cup & \{t \ : \ t \text{ occurs as an } \textit{evar} \text{ in the parsing of index guards}\} \end{array}$$

This index set is the finite set of indices that need to be examined. It includes all terms t that occur in some read $a[t]$ anywhere in F (unless it is a universally quantified variable) and all terms t that are compared to a universally quantified variable in some index guard. λ is a fresh constant that represents all other index positions that are not explicitly in $\mathcal{I}$.

Step 5

Apply the following rule exhaustively to remove universal quantification:

$$\frac{H[\forall \bar{i}.\ F[\bar{i}] \ \rightarrow\ G[\bar{i}]]}{H\left[\bigwedge_{\bar{i}\in\mathcal{I}^n} \left(F[\bar{i}] \ \rightarrow\ G[\bar{i}]\right)\right]} \quad (\mathsf{forall})$$

where n is the size of the list of quantified variables $\bar{i}$. This is the key step. It replaces universal quantification with finite conjunction over the index set.

Step 6

From the output F_5 of Step 5, construct

$$F_6 :\ F_5 \ \wedge \bigwedge_{i\ \in\ \mathcal{I}\setminus\{\lambda\}} \lambda \neq i\ .$$

The new conjuncts assert that the variable λ introduced in Step 4 is indeed unique: it does not equal any other index mentioned in F_5.

Step 7

Decide the T_{A}-satisfiability of F_6 using the decision procedure for the quantifier-free fragment.

For deciding the $(T_{\mathsf{A}}\cup T)$-satisfiability of an array property $(\Sigma_{\mathsf{A}}\cup\Sigma)$-formula, use a combination decision procedure for the quantifier-free fragment of $T_{\mathsf{A}} \cup T$ in Step 7. Thus, this procedure is a decision procedure precisely when the quantifier-free fragment of $T_{\mathsf{A}} \cup T$ is decidable.

Theorem 2.2.1 (Sound & Complete) Given Σ_{A}-formula F from the array property fragment, the decision procedure returns satisfiable if F is T_{A}-satisfiable; otherwise, it returns unsatisfiable.

Proof. Inspection proves the equivalence between the input and the output of Steps 1-3. The crux of the proof is that the index set constructed in Step 4 is sufficient for producing a T_{A}-equisatisfiable quantifier-free formula.

That satisfiability of F implies the satisfiability of F_6 is straightforward: Step 5 weakens universal quantification to finite conjunction. Moreover, the new conjuncts of Step 6 do not affect the satisfiability of F_6 since λ is a fresh constant.

The opposite direction is more complicated. Assume that T_{A}-interpretation I is such that $I \models F_6$. We construct a T_{A}-interpretation J such that $J \models F$.

First, define a projection function $\mathsf{proj}_I : \Sigma_{\mathsf{A}}\text{-terms} \to \mathcal{I}$ that maps Σ_{A}-terms to terms of the index set $\mathcal{I}$:

$$\mathsf{proj}_I(t) = \begin{cases} i & \text{if } \alpha_I[t] = \alpha_I[i] \text{ for some } i \in \mathcal{I} \\ \lambda & \text{otherwise} \end{cases}$$

Recall that α_I assigns terms to values within the domain D_I of interpretation I. Extend proj_I to vectors of variables:

$$\mathsf{proj}_I(\overline{i}) = (\mathsf{proj}_I(i_1), \ldots, \mathsf{proj}_I(i_n)) \ .$$

Now, define J to be like I except for its arrays. Under J, let $a[i] = a[\mathsf{proj}_I(i)]$. Technically, we are specifying how α_J assigns values to terms of F and the array read function $\cdot[\cdot]$; however, we can think in terms of arrays.

To prove that $J \models F$, we focus on a particular subformula $\forall \overline{i}.\ F[\overline{i}] \ \rightarrow \ G[\overline{i}]$. Assume that

$$I \ \models \bigwedge_{\overline{i} \in \mathcal{I}^n} \left(F[\overline{i}] \ \rightarrow \ G[\overline{i}] \right) \ ;$$

then also

$$J \ \models \bigwedge_{\overline{i} \in \mathcal{I}^n} \left(F[\overline{i}] \ \rightarrow \ G[\overline{i}] \right) \tag{2.1}$$

by construction of J. We need to prove that

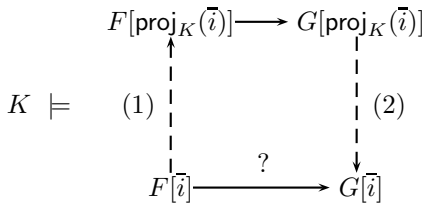
$$J \ \models \ \forall \overline{i}.\ F[\overline{i}] \ \rightarrow \ G[\overline{i}] \ ; \tag{2.2}$$

that is, that

$$J \triangleleft \{\overline{i} \ \mapsto \ \overline{\mathsf{v}}\} \ \models \ F[\overline{i}] \ \rightarrow \ G[\overline{i}]$$

for all $\overline{\mathsf{v}} \in D_J^n$. Let $K = J \triangleleft \{\overline{i} \ \mapsto \ \overline{\mathsf{v}}\}$.

To do so, we prove the two implications represented by dashed arrows in the following diagram:



The top implication holds under K by the assumption (2.1). If both implications (1) and (2) hold under K, then the transitivity of implication implies that the bottom implication holds under K as well.

For (1), we apply structural induction to the index guard $F[\bar{i}]$. Atoms have the form $i = e$, $i \neq e$, $e \neq i$, and $i = j$, for universally quantified variables i and j and existentially quantified variable e. When such a literal is true under K, then so is the corresponding literal $\mathsf{proj}_K(i) = e$, $\mathsf{proj}_K(i) \neq e$, $e \neq \mathsf{proj}_K(i)$, or $\mathsf{proj}_K(i) = \mathsf{proj}_K(j)$, respectively, by definition of proj_K. For example, if $K \models i \neq e$, then $\alpha_K[i]$ is either equal to some $\alpha_K[j]$ for some $j \in \mathcal{I}$ such that $\alpha_K[j] \neq \alpha_K[e]$ or equal to some other value v. In the latter case, $\mathsf{proj}_K(i) = \lambda$; that $\lambda \neq e$ is asserted in F_6 implies that $\mathsf{proj}_K(i) \neq e$.

For the inductive case, consider that neither conjunction nor disjunction evaluates to true when both arguments are false. Thus, (1) holds.

For (2), just note that $\alpha_K[a[i]] = \alpha_K[a[\mathsf{proj}_K(i)]]$ according to the construction of J.

Thus, the bottom implication holds under each variant K of J, so (2.2) holds. □

Example 2.2.6 Consider the array property formula

$$F:\ a\langle \ell \triangleleft v\rangle[k] = b[k]\ \wedge\ b[k] \neq v\ \wedge\ a[k] = v\ \wedge\ (\forall i.\ i \neq \ell\ \rightarrow\ a[i] = b[i])\ .$$

It contains one array property,

$$\forall i.\ i \neq \ell\ \rightarrow\ a[i] = b[i]\ ,$$

in which the index guard is $i \neq \ell$ and the value constraint is $a[i] = b[i]$. It is already in positive normal form. According to Step 2, rewrite F as

$$F_2:\ \begin{array}{c} a'[k] = b[k]\ \wedge\ b[k] \neq v\ \wedge\ a[k] = v\ \wedge\ (\forall i.\ i \neq \ell\ \rightarrow\ a[i] = b[i]) \\ \wedge\ a'[\ell] = v\ \wedge\ (\forall j.\ j \neq \ell\ \rightarrow\ a[j] = a'[j]) \end{array}$$

F_2 does not contain any existential quantifiers. Its index set is

$$\begin{aligned}\mathcal{I} &= \{\lambda\} \cup \{k\} \cup \{\ell\} \\ &= \{\lambda, k, \ell\} \ .\end{aligned}$$

Thus, according to Step 5, replace universal quantification as follows:

$$\begin{aligned}F_5 : \quad & a'[k] = b[k] \ \wedge \ b[k] \neq v \ \wedge \ a[k] = v \ \wedge \bigwedge_{i \in \mathcal{I}} (i \neq \ell \ \rightarrow \ a[i] = b[i]) \\ & \wedge \ a'[\ell] = v \ \wedge \bigwedge_{j \in \mathcal{I}} (j \neq \ell \ \rightarrow \ a[j] = a'[j])\end{aligned}$$

Expanding produces

$$\begin{aligned}F_5' : \quad & a'[k] = b[k] \ \wedge \ b[k] \neq v \ \wedge \ a[k] = v \ \wedge \ (\lambda \neq \ell \ \rightarrow \ a[\lambda] = b[\lambda]) \\ & \wedge \ (k \neq \ell \ \rightarrow \ a[k] = b[k]) \ \wedge \ (\ell \neq \ell \ \rightarrow \ a[\ell] = b[\ell]) \\ & \wedge \ a'[\ell] = v \ \wedge \ (\lambda \neq \ell \ \rightarrow \ a[\lambda] = a'[\lambda]) \\ & \wedge \ (k \neq \ell \ \rightarrow \ a[k] = a'[k]) \ \wedge \ (\ell \neq \ell \ \rightarrow \ a[\ell] = a'[\ell])\end{aligned}$$

Simplifying produces

$$\begin{aligned}F_5'' : \quad & a'[k] = b[k] \ \wedge \ b[k] \neq v \ \wedge \ a[k] = v \ \wedge \ (\lambda \neq \ell \ \rightarrow \ a[\lambda] = b[\lambda]) \\ & \wedge \ (k \neq \ell \ \rightarrow \ a[k] = b[k]) \\ & \wedge \ a'[\ell] = v \ \wedge \ (\lambda \neq \ell \ \rightarrow \ a[\lambda] = a'[\lambda]) \\ & \wedge \ (k \neq \ell \ \rightarrow \ a[k] = a'[k])\end{aligned}$$

Step 6 distinguishes λ from other members of $\mathcal{I}$:

$$\begin{aligned}F_6 : \quad & a'[k] = b[k] \ \wedge \ b[k] \neq v \ \wedge \ a[k] = v \ \wedge \ (\lambda \neq \ell \ \rightarrow \ a[\lambda] = b[\lambda]) \\ & \wedge \ (k \neq \ell \ \rightarrow \ a[k] = b[k]) \\ & \wedge \ a'[\ell] = v \ \wedge \ (\lambda \neq \ell \ \rightarrow \ a[\lambda] = a'[\lambda]) \\ & \wedge \ (k \neq \ell \ \rightarrow \ a[k] = a'[k]) \\ & \wedge \ \lambda \neq k \ \wedge \ \lambda \neq \ell\end{aligned}$$

Simplifying, we have

$$\begin{aligned}F_6' : \quad & a'[k] = b[k] \ \wedge \ b[k] \neq v \ \wedge \ a[k] = v \\ & \wedge \ a[\lambda] = b[\lambda] \ \wedge \ (k \neq \ell \ \rightarrow \ a[k] = b[k]) \\ & \wedge \ a'[\ell] = v \ \wedge \ a[\lambda] = a'[\lambda] \ \wedge \ (k \neq \ell \ \rightarrow \ a[k] = a'[k]) \\ & \wedge \ \lambda \neq k \ \wedge \ \lambda \neq \ell\end{aligned}$$

T_{A}-satisfiability of this quantifier-free Σ_{A}-formula can be decided using the basic decision procedure for the quantifier-free fragment of T_{A}. But let us finish the example. There are two cases to consider. If $k = \ell$, then $a'[\ell] = v$ and $a'[k] = b[k]$ imply $b[k] = v$, yet $b[k] \neq v$. If $k \neq \ell$, then $a[k] = v$ and $a[k] = b[k]$ imply $b[k] = v$, but again $b[k] \neq v$. Hence, F'_6 is T_{A}-unsatisfiable, indicating that F is T_{A}-unsatisfiable. □

Theorem 2.2.2 (Complexity) For subfragments of the array property fragment consisting of bounded-size blocks of quantifiers, T_{A}-satisfiability is NP-complete.

Proof. NP-hardness follows from the NP-hardness of deciding satisfiability in the conjunctive quantifier-free fragment of T_{A}. We recall this well-known result.

Lemma 2.2.1 T_{A}-satisfiability of quantifier-free conjunctive Σ_{A}-formulae is NP-complete.

Proof. That the problem is in NP is simple: for a given formula F, guess a *completion* in which for every instance of a read-over-write $a\langle i \triangleleft v\rangle[j]$, either $i = j$ or $i \neq j$ is conjoined to the formula F. Only a linear number of literals are added. Then check T_{A}-satisfiability of this formula, using the completion to simplify read-over-write occurrences to array reads.

To prove NP-hardness, we reduce from SAT (propositional satisfiability). Given a propositional formula F in CNF, assert

$$v_P \neq v_{\neg P}$$

for each variable P in F; v_P and $v_{\neg P}$ represent the values of P and $\neg P$, respectively. Introduce a fresh constant $\bullet$. Then consider the nth clause of F, say

$$(\neg P \ \vee \ Q \ \vee \ \neg R) \ .$$

In this case, assert

$$a[j_n] \neq \bullet \ \wedge \ a\langle i_P \triangleleft v_{\neg P}\rangle\langle i_Q \triangleleft v_Q\rangle\langle i_R \triangleleft v_{\neg R}\rangle[j_n] = \bullet \ .$$

Intuitively, j_n must be equal to one of the introduced values at i_P, i_Q, or i_R. Therefore, at cell j_n, the corresponding value ($v_{\neg P}$, v_Q, or $v_{\neg R}$) must equal $\bullet$. In this fashion, add an assertion for each clause of F. Conjoin all assertions to form quantifier-free conjunctive Σ_{A}-formula G. G is equisatisfiable to F and of size polynomial in the size of F. Thus, deciding T_{A}-satisfiability of G decides propositional satisfiability of F, so T_{A}-satisfiability is NP-complete. □

That the problem is in NP follows easily from the procedure: instantiating a block of n universal quantifiers quantifying subformula G over index set $\mathcal{I}$ produces $|\mathcal{I}|^n$ new subformulae,

each of length polynomial in the length of G. Hence, the output of Step 6 is of length only a polynomial factor greater than the input to the procedure for fixed n. □

This proof extends naturally to the case in which elements are defined by a theory T: the problem is in NP precisely when satisfiability of the quantifier-free fragment of $T_{\mathsf{A}} \cup T$ is in NP. Element theories that meet this requirement include $T_{\mathbb{Z}}$, $T_{\mathbb{Q}}$, T_{cons}, and $T_{=}$.

We have considered only one-dimensional arrays for ease of presentation. However, the decision procedures can be extended to the multi-dimensional case by considering, for example, arrays of arrays.

2.3 Integer-Indexed Arrays

Software engineers usually think of arrays as integer-indexed segments of memory. Reasoning about indices as integers provides the power of comparison via $\leq$, which enables reasoning about subarrays and properties such as that a (sub)array is sorted or partitioned. In particular, reasoning about subarrays is essential for reasoning about programs that incrementally construct or manipulate arrays.

The theory of integer-indexed arrays $T_{\mathsf{A}}^{\mathbb{Z}}$ is the combination theory $T_{\mathsf{A}} \cup T_{\mathbb{Z}}$: its signature is the union of the signatures of T_{A} and $T_{\mathbb{Z}}$, and its axioms are the axioms of T_{A} and $T_{\mathbb{Z}}$.

2.3.1 The Array Property Fragment

As in Section 2.2, we are interested in the *array property fragment* of $T_{\mathsf{A}}^{\mathbb{Z}}$. An *array property* is again a $\Sigma_{\mathsf{A}}^{\mathbb{Z}}$-formula of the form

$$\forall \overline{i}.\ F[\overline{i}] \ \rightarrow\ G[\overline{i}]\ ,$$

where $\overline{i}$ is a list of integer variables, and $F[\overline{i}]$ and $G[\overline{i}]$ are the *index guard* and the *value constraint*, respectively. The form of an index guard is constrained according to the following grammar:

$$\begin{array}{rcl}
\mathsf{iguard} & \rightarrow & \mathsf{iguard} \wedge \mathsf{iguard} \mid \mathsf{iguard} \vee \mathsf{iguard} \mid \mathsf{atom} \\
\mathsf{atom} & \rightarrow & \mathsf{expr} \leq \mathsf{expr} \mid \mathsf{expr} = \mathsf{expr} \\
\mathsf{expr} & \rightarrow & \mathit{uvar} \mid \mathsf{pexpr} \\
\mathsf{pexpr} & \rightarrow & \mathsf{pexpr}' \\
\mathsf{pexpr}' & \rightarrow & \mathbb{Z} \mid \mathbb{Z} \cdot \mathit{evar} \mid \mathsf{pexpr}' + \mathsf{pexpr}'
\end{array}$$

where *uvar* is any universally quantified integer variable, and *evar* is any existentially quantified or free integer variable.

The form of a *value constraint* is also constrained. Any occurrence of a quantified index

variable i must be as a read into an array, $a[i]$, for array term a. Array reads may not be nested; *e.g.*, $a[b[i]]$ is not allowed.

The *array property fragment* of $T^{\mathbb{Z}}_{\mathsf{A}}$ then consists of formulae that are Boolean combinations of quantifier-free $\Sigma^{\mathbb{Z}}_{\mathsf{A}}$-formulae and array properties.

Example 2.3.1 We list several interesting forms of properties and their definitions in the array property fragment of $T^{\mathbb{Z}}_{\mathsf{A}}$:

- Array equality $a = b$:

  $$\forall i.\ a[i] = b[i]$$

- Bounded array equality $\mathsf{beq}(a, b, \ell, u)$:

  $$\forall i.\ \ell \leq i \leq u \ \rightarrow\ a[i] = b[i]$$

- Universal properties $F[x]$:

  $$\forall i.\ F[a[i]]$$

- Bounded universal properties $F[x]$:

  $$\forall i.\ \ell \leq i \leq u \ \rightarrow\ F[a[i]]$$

- Bounded and unbounded sortedness $\mathsf{sorted}(a, \ell, u)$:

  $$\forall i, j.\ \ell \leq i \leq j \leq u \ \rightarrow\ a[i] \leq a[j]$$

- Partitioned arrays $\mathsf{partitioned}(a, \ell_1, u_1, \ell_2, u_2)$:

  $$\forall i, j,\ \ell_1 \leq i \leq u_1 < \ell_2 \leq j \leq u_2 \ \rightarrow\ a[i] \leq a[j]$$

The last two predicates are necessary for reasoning about sorting algorithms, while the first four forms of properties are useful in general. For example, bounded equality is essential for summarizing the effects of a function on an array — in particular, what parts of the array are unchanged. □

2.3.2 A Decision Procedure

As in Section 2.2, the idea of the decision procedure is to reduce universal quantification to finite conjunction. Given F from the array property fragment of $T_{\mathsf{A}}^{\mathbb{Z}}$, decide its $T_{\mathsf{A}}^{\mathbb{Z}}$-satisfiability as follows:

Step 1

Put F in positive normal form.

Step 2

Apply the following rule exhaustively to remove writes:

$$\frac{F[a\langle i \lhd e\rangle]}{F[a'] \ \wedge \ a'[i] = e \ \wedge \ (\forall j.\ j \neq i \ \rightarrow \ a[j] = a'[j])} \text{ for fresh } a' \quad (\mathsf{write})$$

To meet the syntactic requirements on an index guard, rewrite the third conjunct as

$$\forall j.\ (j \leq i - 1 \ \vee \ i + 1 \leq j) \ \rightarrow \ a[j] = a'[j]\ .$$

Step 3

Apply the following rule exhaustively to remove existential quantification:

$$\frac{F[\exists \overline{i}.\ G[\overline{i}]]}{F[G[\overline{j}]]} \text{ for fresh } \overline{j} \quad (\mathsf{exists})$$

Existential quantification can arise during Step 1 if the given formula has a negated array property.

Step 4

From the output of Step 3, F_3, construct the index set $\mathcal{I}$:

$$\mathcal{I} \ = \ \begin{array}{ll} & \{t \ : \ \cdot[t] \in F_3 \text{ such that } t \text{ is not a universally quantified variable}\} \\ \cup & \{t \ : \ t \text{ occurs as a } \mathsf{pexpr} \text{ in the parsing of index guards}\} \end{array}$$

If $\mathcal{I} = \emptyset$, then let $\mathcal{I} = \{0\}$. The index set contains all relevant symbolic indices that occur in F_3.

Step 5

Apply the following rule exhaustively to remove universal quantification:

$$\frac{H[\forall \overline{i}.\ F[\overline{i}]\ \rightarrow\ G[\overline{i}]]}{H\left[\bigwedge_{\overline{i}\in\mathcal{I}^n} \left(F[\overline{i}]\ \rightarrow\ G[\overline{i}]\right)\right]} \quad (\mathsf{forall})$$

n is the size of the block of universal quantifiers over $\overline{i}$.

Step 6

F_5 is quantifier-free. Decide the $(T_{\mathsf{A}} \cup T_{\mathbb{Z}})$-satisfiability of the resulting formula.

For deciding the $(T_{\mathsf{A}}^{\mathbb{Z}} \cup T)$-satisfiability of an array property $(\Sigma_{\mathsf{A}}^{\mathbb{Z}} \cup \Sigma)$-formula, use a combination decision procedure for the quantifier-free fragment of $T_{\mathsf{A}} \cup T_{\mathbb{Z}} \cup T$ in Step 6. Thus, this procedure is a decision procedure precisely when the quantifier-free fragment of $T_{\mathsf{A}} \cup T_{\mathbb{Z}} \cup T$ is decidable.

Theorem 2.3.1 (Sound & Complete) Given $\Sigma_{\mathsf{A}}^{\mathbb{Z}}$-formula F from the array property fragment, the decision procedure returns satisfiable if F is $T_{\mathsf{A}}^{\mathbb{Z}}$-satisfiable; otherwise, it returns unsatisfiable.

Proof. The proof proceeds using the same strategy as in the proof of Theorem 2.2.1. For the main direction, assume that $T_{\mathsf{A}}^{\mathbb{Z}}$-interpretation I is such that $I \models F_5$. We construct a $T_{\mathsf{A}}^{\mathbb{Z}}$-interpretation J such that $J \models F$. The primary difference from the proof of Theorem 2.2.1 is that the the projection function proj_I is defined so that it maps an index to its nearest neighbor in the index set.

Define a projection function $\mathsf{proj}_I : \Sigma_{\mathsf{A}}\text{-terms} \rightarrow \mathcal{I}$ that maps $\Sigma_{\mathsf{A}}^{\mathbb{Z}}$-terms to terms of the index set $\mathcal{I}$. Let $\mathsf{proj}_I(t) = i \in \mathcal{I}$ such that either

- $\alpha_I[i] \leq \alpha_I[t]\ \wedge\ (\forall j \in \mathcal{I}.\ \alpha_I[j] \leq \alpha_I[t] \rightarrow \alpha_I[j] \leq \alpha_I[i])$
- or $\alpha_I[t] < \alpha_I[i]\ \wedge\ (\forall j \in \mathcal{I}.\ \alpha_I[i] \leq \alpha_I[j])$.

That is, i is the index set term that is t's nearest neighbor under I, with preference for left neighbors. Extend proj_I to vectors of variables:

$$\mathsf{proj}_I(\overline{i}) = (\mathsf{proj}_I(i_1), \ldots, \mathsf{proj}_I(i_n))\ .$$

Using this projection function, the remainder of the proof closely follows the proof of Theorem 2.2.1. Define J to be like I except for its arrays. Under J, let $a[i] = a[\mathsf{proj}_I(i)]$. Now consider a

subformula $\forall \overline{i}.\ F[\overline{i}] \ \rightarrow \ G[\overline{i}]$ and assume that

$$I \ \models \ \bigwedge_{\overline{i} \in \mathcal{I}^n} \left(F[\overline{i}] \ \rightarrow \ G[\overline{i}]\right) \ ; \tag{2.3}$$

immediately, we have

$$J \ \models \ \bigwedge_{\overline{i} \in \mathcal{I}^n} \left(F[\overline{i}] \ \rightarrow \ G[\overline{i}]\right) \ . \tag{2.4}$$

We need to prove that

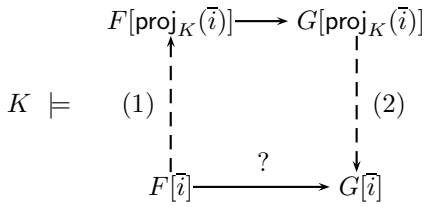
$$J \ \models \ \forall \overline{i}.\ F[\overline{i}] \ \rightarrow \ G[\overline{i}] \ ; \tag{2.5}$$

that is, that

$$J \triangleleft \{\overline{i} \ \mapsto \ \overline{\mathsf{v}}\} \ \models \ F[\overline{i}] \ \rightarrow \ G[\overline{i}]$$

for all $\overline{\mathsf{v}} \in D_J^n$. Let $K = J \triangleleft \{\overline{i} \ \mapsto \ \overline{\mathsf{v}}\}$.

To do so, we prove the two implications represented by dashed arrows in the following diagram:



The top implication holds under K by the assumption (2.4). If both implications (1) and (2) hold under K, then the transitivity of implication implies that the bottom implication holds under K as well.

For (1), we apply structural induction to the index guard $F[\overline{i}]$. Consider an atom of the form $t_1 \leq t_2$; if it holds under K, then also $\mathsf{proj}_K(t_1) \leq \mathsf{proj}_K(t_2)$ holds under K because proj_K is monotonic. Consider an atom of the form $t_1 = t_2$; if it holds under K, then also $\mathsf{proj}_K(t_1) = \mathsf{proj}_K(t_2)$ under K. For the inductive case, consider that neither conjunction nor disjunction evaluates to true when both arguments are false. Thus, (1) holds.

For (2), just note that $\alpha_K[a[i]] = \alpha_K[a[\mathsf{proj}_K(i)]]$ according to the construction of J.

Thus, the bottom implication holds under each variant K of J, so (2.5) holds. □

Example 2.3.2 Consider the following $\Sigma_{\mathsf{A}}^{\mathbb{Z}}$-formula:

$$F: \quad \begin{array}{l} (\forall i.\ \ell \le i \le u \ \rightarrow\ a[i] = b[i]) \\ \ \wedge\ \neg(\forall i.\ \ell \le i \le u+1 \ \rightarrow\ a\langle u+1 \lhd b[u+1]\rangle[i] = b[i]) \end{array}$$

In positive normal form, we have

$$F_1: \quad \begin{array}{l} (\forall i.\ \ell \le i \le u \ \rightarrow\ a[i] = b[i]) \\ \ \wedge\ (\exists i.\ \ell \le i \le u+1 \ \wedge\ a\langle u+1 \lhd b[u+1]\rangle[i] \neq b[i]) \end{array}$$

Step 2 produces

$$F_2: \quad \begin{array}{l} (\forall i.\ \ell \le i \le u \ \rightarrow\ a[i] = b[i]) \\ \ \wedge\ (\exists i.\ \ell \le i \le u+1 \ \wedge\ a'[i] \neq b[i]) \\ \ \wedge\ a'[u+1] = b[u+1] \\ \ \wedge\ (\forall j.\ j \le u+1-1 \ \vee\ u+1+1 \le j \ \rightarrow\ a[j] = a'[j]) \end{array}$$

Step 3 removes the existential quantifier by introducing a fresh constant k:

$$F_3: \quad \begin{array}{l} (\forall i.\ \ell \le i \le u \ \rightarrow\ a[i] = b[i]) \\ \ \wedge\ \ell \le k \le u+1 \ \wedge\ a'[k] \neq b[k] \\ \ \wedge\ a'[u+1] = b[u+1] \\ \ \wedge\ (\forall j.\ j \le u+1-1 \ \vee\ u+1+1 \le j \ \rightarrow\ a[j] = a'[j]) \end{array}$$

Simplifying, we have

$$F_3': \quad \begin{array}{l} (\forall i.\ \ell \le i \le u \ \rightarrow\ a[i] = b[i]) \\ \ \wedge\ \ell \le k \le u+1 \ \wedge\ a'[k] \neq b[k] \\ \ \wedge\ a'[u+1] = b[u+1] \\ \ \wedge\ (\forall j.\ j \le u \ \vee\ u+2 \le j \ \rightarrow\ a[j] = a'[j]) \end{array}$$

The index set is thus

$$\mathcal{I} = \{k, u+1\} \ \cup\ \{\ell, u, u+2\}\ ,$$

which includes the read terms k and $u+1$ and the terms ℓ, u, and $u+2$ that occur as pexprs in the index guards. Then Step 5 rewrites universal quantification to finite conjunction over this

set:

$$F_5: \quad \begin{array}{l} \bigwedge\limits_{i \in \mathcal{I}} (\ell \leq i \leq u \ \rightarrow \ a[i] = b[i]) \\ \wedge \ \ell \leq k \leq u+1 \ \wedge \ a'[k] \neq b[k] \\ \wedge \ a'[u+1] = b[u+1] \\ \wedge \ \bigwedge\limits_{j \in \mathcal{I}} (j \leq u \ \vee \ u+2 \leq j \ \rightarrow \ a[j] = a'[j]) \end{array}$$

Expanding the conjunctions according to the index set $\mathcal{I}$ and simplifying according to trivially true or false antecedents (*e.g.*, $\ell \leq u+1 \leq u$ simplifies to $\bot$, while $u \leq u \ \vee \ u+2 \leq u$ simplifies to $\top$) produces:

$$F_5': \quad \begin{array}{l} (\ell \leq k \leq u \ \rightarrow \ a[k] = b[k]) \ \wedge \ (\ell \leq u \ \rightarrow \ a[\ell] = b[\ell] \ \wedge \ a[u] = b[u]) \\ \wedge \ \ell \leq k \leq u+1 \ \wedge \ a'[k] \neq b[k] \\ \wedge \ a'[u+1] = b[u+1] \\ \wedge \ (k \leq u \ \vee \ u+2 \leq k \ \rightarrow \ a[k] = a'[k]) \\ \wedge \ (\ell \leq u \ \vee \ u+2 \leq \ell \ \rightarrow \ a[\ell] = a'[\ell]) \\ \wedge \ a[u] = a'[u] \ \wedge \ a[u+2] = a'[u+2] \end{array}$$

$(T_{\mathsf{A}} \cup T_{\mathbb{Z}})$-satisfiability of this quantifier-free $(\Sigma_{\mathsf{A}} \cup \Sigma_{\mathbb{Z}})$-formula can be decided using the Nelson-Oppen combination of the basic decision procedure for the quantifier-free fragment of T_{A} with a decision procedure for $T_{\mathbb{Z}}$. But let us finish the example. F_5' is $(T_{\mathsf{A}} \cup T_{\mathbb{Z}})$-unsatisfiable. In particular, note that k is restricted such that $\ell \leq k \leq u+1$; that the first conjunct asserts that for most of this range ($k \in [\ell, u]$), $a[k] = b[k]$; that the third-to-last conjunct asserts that for $k \leq u$, $a[k] = a'[k]$, contradicting $a[k] = a'[k] \neq b[k]$ when $k \leq u$; and that even if $k = u+1$, $a'[k] \neq b[k] = b[u+1] = a'[u+1] = a'[k]$ by the fourth and fifth conjuncts, a contradiction. Hence, F is $T_{\mathsf{A}}^{\mathbb{Z}}$-unsatisfiable. □

2.4 Negative Results

Theorem 2.4.1 Consider the following extensions to the array property fragment of T_{A}:

- Permit an additional quantifier alternation.
- Permit nested reads (*e.g.*, $a[b[i]]$ when i is universally quantified).
- Permit array reads by a universally quantified variable in the index guard.
- Augment the theory with a permutation predicate.

For each resulting fragment, there exists an element theory T such that satisfiability in the array property fragment of $T_{\mathsf{A}} \cup T$ is decidable, yet satisfiability in the resulting fragment of $T_{\mathsf{A}} \cup T$ is undecidable.

Proof. Our main proof technique is the following. Consider the problem of solving Diophantine equations: Does a given polynomial $p(\overline{x})$ have a nonnegative integer root? It has a root iff there exists a finite path from the origin to that root, where each step of the path consists of incrementing a variable x_i by one. For each of the extensions, we show how to encode the existence of such a path into a formula of the extended fragment of $T_{\mathsf{A}} \cup T_{\mathbb{Z}}$. The undecidability of the existence of integer roots then implies the undecidability of satisfiability in the fragment.

Consider a polynomial $p(\overline{x})$, and let $q(\overline{x}) = p(\overline{x})^2$. $q(\overline{x})$ is positive except at its roots (which correspond to the roots of $p(\overline{x})$).

Since the element theory $T_{\mathbb{Z}}$ does not have multiplication, the first task is to reason about $q(\overline{x})$ without multiplication. Suppose that the value of $q(\overline{a})$ is known. To compute q's value at coordinate $\overline{b}$ in which $b_i = a_i + 1$ and $b_j = a_j$ for $j \neq i$, compute its *finite difference* for the ith dimension:

$$\triangle_{x_i} q(\overline{x}) = q(x_1, \ldots, x_i + 1, \ldots, x_n) - q(\overline{x}) \ .$$

Then $q(\overline{b}) = q(\overline{a}) + (\triangle_{x_i} q(\overline{x}))(\overline{a})$. Higher-order finite differences are computed compositionally; for example,

$$\triangle_{x_i x_j} q(\overline{x}) = \triangle_{x_j} \triangle_{x_i} q(\overline{x}) \ .$$

Now construct the following tree from $q(\overline{x})$, starting with $q(\overline{x})$ as the root:

1. If node $r(\overline{x})$ is simply a constant, then it is a leaf.
2. Otherwise, let the children of node $r(\overline{x})$ be its n finite differences $\triangle_{x_i} r(\overline{x})$.

This construction terminates because $q(\overline{x})$ is a polynomial.

Introduce at each non-leaf node $r(\overline{x})$ of the tree a fresh variable $v_{r(\overline{x})}$. For notational convenience, if node $r(\overline{x})$ is a leaf, let $v_{r(\overline{x})}$ be its constant value. Let $\overline{v}$ be the vector of all of these variables.

The following formula simulates the evaluation of $q(\overline{x})$ at the origin $\overline{0}$:

$$T_{q(\overline{0})}[\overline{v}] : \bigwedge_{r(\overline{x})} v_{r(\overline{x})} = r(\overline{0}) \ ,$$

where the conjunction ranges over the non-leaf nodes of the tree. That is, each introduced variable is equated to the value of the polynomial it represents at the origin. The following formula, a

relation between the current values of the tree variables $\overline{v}$ and the next-state values of the tree variables $\overline{v}'$, simulates the incrementing of x_i:

$$T_{x_i+1}[\overline{v},\overline{v}'] : \bigwedge_{r(\overline{x})} v'_{r(\overline{x})} = v_{r(\overline{x})} + v_{\triangle_{x_i} r(\overline{x})} \ ,$$

where $v_{\triangle_{x_i} r(\overline{x})}$ is the variable (or, using our convention, the constant) corresponding to the ith child of node $r(\overline{x})$, and the conjunction ranges over the non-leaf nodes of the tree. The formulae $T_{q(\overline{0})}$ and T_{x_i+1} are $\Sigma_{\mathbb{Z}}$-formulae.

With these tools, we can now address decidability for the extended fragments. Consider first allowing an additional quantifier alternation. Construct the following Σ_{A}-formula:

$$\exists \overline{v}, s.\ \forall i.\ \exists j. \left[\begin{array}{l} T_{q(\overline{0})}[\overline{v}[0]] \\ \quad \wedge\ \left(\overline{v}[j] = \overline{v}[i]\ \vee\ \bigvee_{x_i} T_{x_i+1}[\overline{v}[i], \overline{v}[j]]\right) \\ \quad \wedge\ s[j] = s[i] - v_{q(\overline{x})}[i] \\ \quad \wedge\ s[i] > 0 \end{array} \right] \tag{2.6}$$

This formula simulates a walk from the origin $\overline{x} = \overline{0}$ in which each step consists of either staying in place or moving one positive unit along one dimension x_i. At each step, the current value of the polynomial $q(\overline{x})$ — which is maintained by $v_{q(\overline{x})}$ — is subtracted from s. To satisfy (2.6), there must be some value of $s[0]$ and some sequence of steps such that s never reaches 0. But recall that $v_{q(\overline{x})}$ is always positive except at solutions. Hence, (2.6) is $(T_{\mathsf{A}} \cup T_{\mathbb{Z}})$-satisfiable only when some finite path leads to a solution. Once the solution is reached, then each step consists of remaining in place so that s is no longer decreased.

Consider permitting nested reads. In this case, an array can be used to skolemize the variable j of (2.6):

$$\exists \overline{v}, s, j.\ \forall i. \left[\begin{array}{l} T_{q(\overline{0})}[\overline{v}[0]] \\ \quad \wedge\ \left(\overline{v}[j[i]] = \overline{v}[i]\ \vee\ \bigvee_{x_i} T_{x_i+1}[\overline{v}[i], \overline{v}[j[i]]]\right) \\ \quad \wedge\ s[j[i]] = s[i] - v_{q(\overline{x})}[i] \\ \quad \wedge\ s[i] > 0 \end{array} \right] \tag{2.7}$$

Consider permitting array reads by a universally quantified variable in the index guard. Then the skolem array of (2.7) can be extracted as follows (where it has been renamed to a for convenience):

$$\exists \overline{v}, s, a.\ \forall i, j.\ j = a[i]\ \rightarrow \left[\begin{array}{l} T_{q(\overline{0})}[\overline{v}[0]] \\ \quad \wedge\ \left(\overline{v}[j] = \overline{v}[i]\ \vee\ \bigvee_{x_i} T_{x_i+1}[\overline{v}[i], \overline{v}[j]]\right) \\ \quad \wedge\ s[j] = s[i] - v_{q(\overline{x})}[i] \\ \quad \wedge\ s[i] > 0 \end{array} \right] \tag{2.8}$$

Consider adding a permutation predicate $\mathsf{perm}(a, b)$ which expresses that array b is a permutation of array a. Asserting

$$\mathsf{perm}(a, b) \ \wedge \ \forall i.\ b[i] = a[i] + 1$$

forces a and b to include all integers; asserting further that $a[z] = 0$ allows us to identify a particular position that has value 0. We use this technique to generate a set of identifiers within an array d in the reduction. Two positions represent points in the walk one step away from each other when their identities are one unit apart:

$$\exists \overline{v}, d, e, z, n.\ \forall i, j. \qquad (2.9)$$
$$\left[\begin{array}{l} T_{q(\overline{0})}[\overline{v}[z]] \\ \quad \wedge \left[\begin{array}{l} ((d[j] > 0 \ \wedge \ d[j] = d[i] + 1) \ \vee \ (d[j] < 0 \ \wedge \ d[j] = d[i] - 1)) \\ \quad \rightarrow \ \bigvee_{x_i} T_{x_i+1}[\overline{v}[i], \overline{v}[j]] \end{array}\right] \\ \quad \wedge \ v_{q(\overline{x})}[n] = 0 \\ \quad \wedge \ d[z] = 0 \ \wedge \ \mathsf{perm}(d, e) \ \wedge \ \forall i.\ e[i] = d[i] + 1 \end{array}\right]$$

Two computations are encoded: one following the nonnegative identifiers ($d[j] \geq 0$) and one following the nonpositive identifiers ($d[j] \leq 0$). If either leads to a solution, then at some position n, $v_{q(\overline{x})}[n] = 0$; otherwise, for all n, $v_{q(\overline{x})}[n] > 0$. □

These results carry over to the integer-index case of $T^{\mathbb{Z}}_{\mathsf{A}}$ without modification. Additionally, we have the following for $T^{\mathbb{Z}}_{\mathsf{A}}$.

Theorem 2.4.2 Consider the following extensions to the array property fragment of $T^{\mathbb{Z}}_{\mathsf{A}}$:

- Permit general Presburger arithmetic expressions over universally quantified index variables (even just addition of 1, *e.g.*, $i + 1$) in the index guard or in the value constraint.
- Permit strict inequalities between universally quantified indices.

For each resulting fragment, there exists an element theory T such that satisfiability in the array property fragment of $T^{\mathbb{Z}}_{\mathsf{A}} \cup T$ is decidable, yet satisfiability in the resulting fragment of $T^{\mathbb{Z}}_{\mathsf{A}} \cup T$ is undecidable.

Proof. Addressing the first relaxation, we apply the same technique from the proof of Theorem 2.4.1 and construct the following $(\Sigma^{\mathbb{Z}}_{\mathsf{A}} \cup T_{\mathbb{Z}})$-formula:

$$\exists n, \overline{v}.\ \forall i. \left[\begin{array}{l} T_{q(\overline{0})}[\overline{v}[0]] \\ \quad \wedge \ \bigvee_{x_i} T_{x_i+1}[\overline{v}[i], \overline{v}[i+1]] \\ \quad \wedge \ n \geq 0 \ \wedge \ v_{q(x)}[n] = 0 \end{array}\right] \qquad (2.10)$$

This formula is simpler than those of the previous proof: a path from the origin $\overline{x} = \overline{0}$ to a solution is encoded in a finite subarray from position 0 to position n. At position n, the solution has been reached: $v_{q(\overline{x})}[n] = 0$. If no solution exists, then there cannot be a position $n \geq 0$ at which $v_{q(\overline{x})}[n] = 0$.

For the second relaxation, the following formula associates unique integer identities (via array d) with a subset of positions. Two positions represent points in the walk one step away from each other when their identities are one unit apart. The remaining challenge is to force the existence of a sequence of identities $0, 1, 2, \ldots$; the sequence should terminate with some identity n precisely when a solution exists to $q(\overline{x}) = 0$. To that end, define the *injective* predicate which asserts that the elements of a subarray are unique:

$$\mathsf{inj}(a, \ell, u) : \forall i, j.\ \ell \leq i < j \leq u \ \rightarrow\ a[i] \neq a[j]\ .$$

Then construct the following $(\Sigma_{\mathsf{A}}^{\mathbb{Z}} \cup T_{\mathbb{Z}})$-formula:

$$\exists \overline{v}, d, n.\ \forall i, j.\ \left[\begin{array}{l} T_{q(\overline{0})}[\overline{v}[0]] \\ \quad \wedge\ \big(d[j] = d[i] + 1\ \rightarrow\ \bigvee_{x_i} T_{x_i+1}[\overline{v}[i], \overline{v}[j]]\big) \\ \quad \wedge\ d[0] = 0\ \wedge\ n \geq 0\ \wedge\ d[n] = n\ \wedge\ v_{q(\overline{x})}[n] = 0 \\ \quad \wedge\ \mathsf{sorted}(d, 0, n)\ \wedge\ \mathsf{inj}(d, 0, n) \end{array}\right] \tag{2.11}$$

The last two lines assert that positions 0 through n define a path to a solution.

Using (the definition of) sorted is actually not necessary: $\forall i.\ 0 \leq i \leq n\ \rightarrow\ 0 \leq d[i] \leq n$ achieves the same result. Moreover, the fully injective predicate

$$\mathsf{inj}(a) : \forall i, j.\ i < j\ \rightarrow\ a[i] \neq a[j]$$

can also be used instead of $\mathsf{inj}(a, \ell, u)$. □

The final extension to the array property fragment of $T_{\mathsf{A}}^{\mathbb{Z}}$ has an analog in T_{A}: allow disequalities between universally quantified indices. However, decidability of satisfiability in the resulting fragment remains an open question. Unlike for $T_{\mathsf{A}}^{\mathbb{Z}}$, we cannot define intervals of positions and thus cannot force the existence of a sequence of identities.

Even for integer indices, a permutation predicate is not allowed from Theorem 2.4.1; what about bounded permutation $\mathsf{perm}(a, b, \ell, u)$? Similar to the use of bounded inj, we can define a subarray to have all elements in the range $[0, n]$:

$$\begin{array}{l} a[0] = n\ \wedge\ b[0] = 0\ \wedge\ \mathsf{perm}(a, b, 0, n) \\ \quad \wedge\ \forall i.\ 0 < i \leq n\ \rightarrow\ 0 \leq a[i], b[i] \leq n\ \wedge\ b[i] = a[i] + 1\ . \end{array}$$

The rest of the reduction is similar to (2.11), proving that bounded permutation results in

undecidability as well.

2.5 Hashtables

Hashtables are a common data structure in modern programs. In this section, we describe a theory for hashtables T_{H} and provide a reduction of the hashtable property fragment of Σ_{H}-formulae into the array property fragment of T_{A}.

The signature of T_{H} is the following:

$$\Sigma_{\mathsf{H}} : \{\mathsf{get}(\cdot,\cdot),\ \mathsf{put}(\cdot,\cdot,\cdot),\ \mathsf{remove}(\cdot,\cdot),\ \cdot \in \mathsf{keys}(\cdot),\ =\}\ ,$$

where

- $\mathsf{put}(h,k,v)$ is the hashtable that is modified from h by mapping key k to value v.
- $\mathsf{remove}(h,k)$ is the hashtable that is modified from h by unmapping the key k.
- $\mathsf{get}(h,k)$ is the value mapped by key k, which is undetermined if h does not map k to any value.
- $k \in \mathsf{keys}(h)$ is true iff h maps the key k.

Note that $k \in \mathsf{keys}(h)$ is merely convenient notation for a binary predicate. However, we will exploit this notation in the following useful operations:

- *Key sets* $\mathsf{keys}(h)$ can be combined ($\cup$), intersected ($\cap$), and complemented ($\bar{\cdot}$).
- The predicate $\mathsf{init}(h)$ is true iff h does not map any key.

Each is definable using the basic signature.

The axioms of T_{H} are the following:

- $\forall x.\ x = x$ (reflexivity)
- $\forall x, y.\ x = y\ \rightarrow\ y = x$ (symmetry)
- $\forall x, y, z.\ x = y\ \wedge\ y = z\ \rightarrow\ x = z$ (transitivity)
- $\forall h, j, k.\ j = k\ \rightarrow\ \mathsf{get}(h,j) = \mathsf{get}(h,k)$ (hashtable congruence)
- $\forall h, j, k, v.\ j = k\ \rightarrow\ \mathsf{get}(\mathsf{put}(h,k,v),j) = v$ (read-over-put 1)
- $\forall h, k, v.\ \forall j \in \mathsf{keys}(h).\ j \neq k\ \rightarrow\ \mathsf{get}(\mathsf{put}(h,k,v),j) = \mathsf{get}(h,j)$ (read-over-put 2)
- $\forall h, k.\ \forall j \in \mathsf{keys}(h).\ j \neq k\ \rightarrow\ \mathsf{get}(\mathsf{remove}(h,k),j) = \mathsf{get}(h,j)$ (read-over-remove)

- $\forall h, k, v.\ k \in \mathsf{keys}(\mathsf{put}(h, k, v))$ (keys-put)
- $\forall h, k.\ k \notin \mathsf{keys}(\mathsf{remove}(h, k))$ (keys-remove)

Note the similarity between the first six axioms of T_{H} and those of T_{A}. Key sets complicate the (read-over-put 2) axiom compared to the (read-over-write 2) axiom, while keys sets and key removal require three additional axioms.

2.5.1 The Hashtable Property Fragment

A *hashtable property* has the form

$$\forall \overline{k}.\ F[\overline{k}] \ \rightarrow\ G[\overline{k}]\ ,$$

where $F[\overline{k}]$ is the *key guard*, and $G[\overline{k}]$ is the *value constraint*. Key guards are defined exactly as index guards of the array property fragment of T_{A}: they are positive Boolean combinations of equations between universally quantified keys; and equations and disequalities between universally quantified keys k and other key terms. Value constraints can use universally quantified keys k in hashtable reads $\mathsf{get}(h, k)$ and in key set membership checks. Finally, a hashtable property does not contain any init literals.

Σ_{H}-formulae that are Boolean combinations of quantifier-free Σ_{H}-formulae and hashtable properties comprise the *hashtable property fragment* of T_{H}.

Example 2.5.1 Consider the following hashtable property formula:

$$F:\ \forall k \in \mathsf{keys}(h).\ \mathsf{get}(h, k) \geq 0\ .$$

Its key guard is trivial, while its value constraint is

$$k \in \mathsf{keys}(h)\ \rightarrow\ \mathsf{get}(h, k) \geq 0\ .$$

Suppose that F annotates locations L_1 and L_2 in the following basic path (see Chapter 4 of [BM07] for a discussion of basic paths in program verification):

$@L_1 : F$
`assume` $v \geq 0$;
$\mathsf{put}(h, s, v)$;
$@L_2 : F$

We want to prove that if F holds at L_1, then it holds at L_2. The resulting verification condition

is

$$\forall h, s, v. \left[\begin{array}{l} (\forall k \in \mathsf{keys}(h).\ \mathsf{get}(h,k) \geq 0) \ \wedge\ v \geq 0 \\ \quad \rightarrow\ (\forall k \in \mathsf{keys}(\mathsf{put}(h,s,v)).\ \mathsf{get}(\mathsf{put}(h,s,v),k) \geq 0) \end{array} \right] .$$

The key set $\mathsf{keys}(h)$ provides a mechanism for reasoning about the incremental modification of hashtables. □

Example 2.5.2 To express equality between key sets, $\mathsf{keys}(h_1) = \mathsf{keys}(h_2)$, in T_{H}, write for example

$$\forall k.\ k \in \mathsf{keys}(h_1) \ \leftrightarrow\ k \in \mathsf{keys}(h_2) ,$$

which has a trivial key guard. Then express equality between hashtables, $h_1 = h_2$, with the following formula:

$$\mathsf{keys}(h_1) = \mathsf{keys}(h_2) \ \wedge\ \forall k \in \mathsf{keys}(h_1).\ \mathsf{get}(h_1,k) = \mathsf{get}(h_2,k) .$$

□

2.5.2 A Decision Procedure

Given F from the hashtable property fragment of T_{H}, the decision procedure reduces T_{H}-satisfiability to T_{A}-satisfiability of a formula in the array property fragment. The main idea of the reduction is to represent hashtables h of the given Σ_{H}-formula by two arrays in the Σ_{A}-formula: an array h for the elements and an array keys_h that indicates if a key (an array index) maps to an element. In particular, $\mathsf{keys}_h[k] = \circ$ indicates that k is not mapped by h, while $\mathsf{keys}_h[k] \neq \circ$ ($\mathsf{keys}_h[k] = \bullet$) indicates that h does map k to a value given by $h[k]$.

Step 1

Construct $F \ \wedge\ \bullet \neq \circ$, for fresh constants $\bullet$ and $\circ$.

Step 2

Rewrite F_1 according to the following set of rules:

$$\begin{array}{rcl} F[\mathsf{put}(h,k,v)] & \Longrightarrow & F[h'] \ \wedge\ h' = h\langle k \lhd v\rangle \ \wedge\ \mathsf{keys}_{h'} = \mathsf{keys}_h\langle k \lhd \bullet\rangle \\ F[\mathsf{remove}(h,k)] & \Longrightarrow & F[h'] \ \wedge\ (\forall j \in \mathsf{keys}(h').\ h'[j] = h[j]) \ \wedge\ \mathsf{keys}_{h'} = \mathsf{keys}_h\langle k \lhd \circ\rangle \end{array}$$

for fresh variable h'.

Step 3

Rewrite F_2 according to the following set of rules:

$$\begin{aligned}
F[\mathsf{get}(h,k)] &\implies F[h[k]] \\
F[k \in \mathsf{keys}(h)] &\implies F[\mathsf{keys}_h[k] \neq \circ] \\
F[k \in K_1 \cup K_2] &\implies F[k \in K_1 \vee k \in K_2] \\
F[k \in K_1 \cap K_2] &\implies F[k \in K_1 \wedge k \in K_2] \\
F[k \in \overline{K}] &\implies F[\neg(k \in K)] \\
F[\mathsf{init}(h)] &\implies F[\forall k.\ \neg(k \in \mathsf{keys}(h))]
\end{aligned}$$

where K, K_1, and K_2 are constructed from union, disjunction, and complementation of key set membership atoms. The final four rules define auxiliary operations: the right-side of each is expressible in the hashtable property fragment.

Step 4

Decide the T_{A}-satisfiability of F_3.

Step 2 relies on the defined predicate of equality between arrays, $a = b$, which is defined by $\forall i.\ a[i] = b[i]$.

As in Sections 2.2 and 2.3, hashtable values may be interpreted in some theory T with signature Σ. Then the above procedure is a decision procedure precisely when there is a decision procedure for the array property fragment of $T_{\mathsf{A}} \cup T$.

Theorem 2.5.1 (Sound & Complete) Given Σ_{H}-formula F from the hashtable property fragment, the decision procedure returns satisfiable iff F is T_{H}-satisfiable; otherwise, it returns unsatisfiable.

Proof. Inspection shows that if F is a Σ_{H}-formula from the hashtable property fragment, then F_3 is a Σ_{A}-formula from the array property fragment. First, the rewrite system terminates: the first rule of Step 2 and the first two rules of Step 3 remove instances of Σ_{H}-literals and Σ_{H}-terms without introducing more, while the second rule of Step 2 introduces only one new Σ_{H}-literal that is addressed in Step 3; and the final four rules of Step 3 clearly terminate with a finite number of applications of the first two rules of Step 3. Second, while three rules introduce (universal) quantification, the definition of the hashtable property fragment guarantees that the quantified subformulae occur outside of other quantifiers (except the topmost existential quantifiers). Hence, Step 4 returns satisfiable or unsatisfiable.

We must show that F has a satisfying T_{H}-interpretation precisely when F_3 has a satisfying T_{A}-interpretation. Suppose that T_{H}-interpretation I satisfies F. Construct the T_{A}-interpretation J satisfying F_3 as follows:

- if h maps k to v in I, set $\mathsf{keys}_h[k] = \bullet$ in J and $h[k] = v$;
- otherwise, set $\mathsf{keys}_h[k] = \circ$ and $h[k] = v$ for some arbitrary value v.

This J satisfies the conjuncts added in Step 2; also, $h[k]$ is the same value under J as $\mathsf{get}(h, k)$ under I, and $\mathsf{keys}_h[k] \neq \circ$ under J iff $k \in \mathsf{keys}(h)$ under I, showing the correctness of the first two rules of Step 3 (recall that the remaining rules are auxiliary and easily reduce to the first two).

Similarly, a T_{A}-interpretation J satisfying F_3 induces a T_{H}-interpretation I satisfying F. Construct I as follows:

- if $\mathsf{keys}_h[k] \neq \circ$ and $h[k] = v$ in J, assert $\mathsf{get}(h, k) = v$ and $k \in \mathsf{keys}(h)$ in I;
- otherwise, assert $\neg(h \in \mathsf{keys}(h))$ in I.

Again, the correspondence between the two sides of each rule is clear. □

Example 2.5.3 Consider the following Σ_{H}-formula:

$$G: \ \forall h, s, v. \left[\begin{array}{l} (\forall k \in \mathsf{keys}(h).\ \mathsf{get}(h, k) \geq 0) \ \wedge \ v \geq 0 \\ \quad \rightarrow \ (\forall k \in \mathsf{keys}(\mathsf{put}(h, s, v)).\ \mathsf{get}(\mathsf{put}(h, s, v), k) \geq 0) \end{array} \right]$$

To prove its T_{H}-validity, prove the T_{H}-satisfiability of the following:

$$\begin{array}{ll} F: & (\forall k \in \mathsf{keys}(h).\ \mathsf{get}(h, k) \geq 0) \ \wedge \ v \geq 0 \\ & \wedge \ \neg(\forall k \in \mathsf{keys}(\mathsf{put}(h, s, v)).\ \mathsf{get}(\mathsf{put}(h, s, v), k) \geq 0) \end{array}$$

Step 1 introduces $\bullet$ and $\circ$:

$$F_1: \ F \ \wedge \ \bullet \neq \circ \ .$$

Step 2 reduces hashtable modifications to arrays:

$$\begin{array}{ll} F_2: & (\forall k \in \mathsf{keys}(h).\ \mathsf{get}(h, k) \geq 0) \ \wedge \ v \geq 0 \\ & \wedge \ \neg(\forall k \in \mathsf{keys}(h').\ \mathsf{get}(h', k) \geq 0) \\ & \wedge \ h' = h\langle s \lhd v \rangle \ \wedge \ \mathsf{keys}_{h'} = \mathsf{keys}_h\langle s \lhd \bullet \rangle \\ & \wedge \ \bullet \neq \circ \end{array}$$

Step 3 completes the reduction:

$$\begin{array}{ll} F_3: & (\forall k.\ \mathsf{keys}_h[k] \neq \circ \ \rightarrow \ h[k] \geq 0) \ \wedge \ v \geq 0 \\ & \wedge \ \neg(\forall k.\ \mathsf{keys}_{h'}[k] \neq \circ \ \rightarrow \ h'[k] \geq 0) \\ & \wedge \ h' = h\langle s \lhd v \rangle \ \wedge \ \mathsf{keys}_{h'} = \mathsf{keys}_h\langle s \lhd \bullet \rangle \\ & \wedge \ \bullet \neq \circ \end{array}$$

F_3 is T_{A}-unsatisfiable. In particular, $\mathsf{keys}_{h'} = \mathsf{keys}_h\langle s \lhd \bullet\rangle$ and $h' = h\langle s \lhd v\rangle$, so that for all keys of h (k such that $\mathsf{keys}_h[k] \neq \circ$), $h'[k] = h[k] \geq 0$. For the one new key s of h', we know that $h'[s] = h\langle s \lhd v\rangle[s] = v \geq 0$. Therefore, no key k of h' exists at which $h[k] < 0$, a contradiction. Hence, G is T_{H}-valid.

The decision procedure of Section 2.2 would also prove the T_{A}-unsatisfiability of F_3. □

2.6 Partial Arrays

A variation on arrays in which elements may be undefined has been considered recently [GNRZ06]. We apply the techniques developed for studying standard arrays to study the theory of partial arrays.

The theory of read-only partial arrays $T_{\mathsf{A}}^{\mathsf{RO}}$ has signature

$$\Sigma_{\mathsf{A}}^{\mathsf{RO}} = \{\cdot[\cdot],\ =,\ \bot\}\ .$$

Its signature differs from Σ_{A} in two ways: it does not have a function for constructing a modified array; and it has a new constant symbol $\bot$. Its axioms are just (reflexivity), (symmetry), (transitivity), and (array congruence) of T_{A}. Notice that $\bot$ is undefined. It will play a role later when we define an important fragment of $T_{\mathsf{A}}^{\mathsf{RO}}$.

The theory of partial arrays $T_{\mathsf{A}}^{\bot}$ has signature

$$\Sigma_{\mathsf{A}}^{\bot} = \{\cdot[\cdot],\ \cdot\langle\cdot \lhd \cdot\rangle,\ =,\ \bot\}$$

and includes all the axioms of T_{A}.

2.6.1 The Guarded Fragment

Consider a $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$-formula $F[a[i]]$ with subterm $a[i]$ (possibly with multiple occurrences) in which a and i are free; moreover, i does not appear in any other read term. The *guarded universal quantification* of $F[a[i]]$ is

$$\forall i.\ a[i] \neq \bot\ \rightarrow\ F[a[i]]\ .$$

Then a *guarded* $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$*-formula* is defined recursively as a quantifier-free $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$-formula, the guarded universal quantification of a guarded $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$-formula, or the positive Boolean combination of guarded $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$-formulae. Finally, the *guarded* fragment of $T_{\mathsf{A}}^{\mathsf{RO}}$ consists of Boolean combinations of guarded $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$-formulae.

Example 2.6.1 The seemingly restricted form of guarded universal quantification

$$\forall i.\ a[i] \neq \bot \ \rightarrow\ F[a[i]]$$

is actually not restrictive. For example, write

$$\forall i.\ b[i] \neq \bot \ \wedge\ a[b[i]] \neq \bot \ \rightarrow\ a[b[i]] = i$$

with another quantifier:

$$\forall i.\ b[i] \neq \bot \ \rightarrow\ \forall j.\ a[j] \neq \bot \ \rightarrow\ j = b[i] \ \rightarrow\ a[j] = i\ .$$

The restricted formulation simply makes obvious that each array read by a universally quantified variable is guarded. □

Example 2.6.2 To assert that partial array b is offset from a by one position, write

$$\forall i.\ a[i] \neq \bot \ \rightarrow\ \forall j.\ b[j] \neq \bot \ \rightarrow\ j = i + 1 \ \rightarrow\ a[i] = b[j]\ . \tag{2.12}$$

One might write (2.12) more naturally as

$$\forall i.\ a[i] \neq \bot \ \wedge\ b[i+1] \neq \bot \ \rightarrow\ a[i] = b[i+1]\ .$$

The corresponding Σ_{A}-formula is

$$\forall i.\ a[i] = b[i+1]\ ,$$

which lies outside of the array property fragment of $T^{\mathbb{Z}}_{\mathsf{A}}$. In general, one can express more about partial arrays in the guarded fragment than one can express about standard arrays in the array property fragment. However, notice that in (2.12), a and b need not be defined in the same positions, making the assertion quite weak. □

Example 2.6.3 As a further comparison of standard arrays (T_{A} and $T^{\mathbb{Z}}_{\mathsf{A}}$) and partial arrays ($T^{\perp}_{\mathsf{A}}$), consider asserting the following in each:

$$\mathsf{sorted}(a\langle 0 \lhd 1\rangle\langle 5 \lhd 3\rangle, 0, 5)\ \wedge\ \mathsf{sorted}(a\langle 0 \lhd 4\rangle\langle 5 \lhd 6\rangle, 0, 5) \tag{2.13}$$

Recall that $\mathsf{sorted}(a, \ell, u)$ is defined in the array property fragment of $T^{\mathbb{Z}}_{\mathsf{A}}$ as

$$\forall i, j.\ \ell \leq i \leq j \leq u \ \rightarrow\ a[i] \leq a[j]\ .$$

In the guarded fragment of $T_{\mathsf{A}}^{\perp}$, it is defined as

$$\forall i.\ a[i] \neq \perp \ \rightarrow\ \forall j.\ a[j] \neq \perp \ \rightarrow\ \ell \leq i \leq j \leq u \ \rightarrow\ a[i] \leq a[j]\ .$$

With these definitions, it is clear that (2.13) is $T_{\mathsf{A}}^{\mathbb{Z}}$-unsatisfiable: there is no way of filling in the elements between positions 1 and 4 of a such that both modified arrays are sorted. However, (2.13) is trivially $T_{\mathsf{A}}^{\perp}$-satisfiable: simply fill a with $\perp$. □

Example 2.6.4 As a final example comparing standard arrays (T_{A} and $T_{\mathsf{A}}^{\mathbb{Z}}$) and partial arrays ($T_{\mathsf{A}}^{\perp}$), consider asserting the following in each, where the arrays have integer elements:

$$a[0] = 0 \ \wedge\ a[5] = 4 \ \wedge\ \mathsf{sorted}(a, 0, 5) \ \wedge\ \mathsf{inj}(a, 0, 5)\ . \tag{2.14}$$

In $T_{\mathsf{A}}^{\mathbb{Z}}$, $\mathsf{inj}(a, \ell, u)$ is defined as

$$\forall i, j.\ \ell \leq i < j \leq u \ \rightarrow\ a[i] \neq a[j]\ ;$$

in $T_{\mathsf{A}}^{\perp}$, it is defined as

$$\forall i.\ a[i] \neq \perp \ \rightarrow\ \forall j.\ a[j] \neq \perp \ \rightarrow\ \ell \leq i < j \leq u \ \rightarrow\ a[i] \neq a[j]\ .$$

From Theorem 2.4.2, we know that adding the power to express inj to the array property fragment of $T_{\mathsf{A}}^{\mathbb{Z}}$ results in undecidability; however, it can be expressed in the guarded fragment of $T_{\mathsf{A}}^{\perp}$.

In any case, (2.14) is $(T_{\mathsf{A}}^{\mathbb{Z}} \cup T_{\mathbb{Z}})$-unsatisfiable: there are not enough distinct integers to fill the positions between 1 and 4. However, (2.14) is $(T_{\mathsf{A}}^{\perp} \cup T_{\mathbb{Z}})$-satisfiable: simply fill the interval with $\perp$. □

2.6.2 Decision Procedures

Satisfiability of $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$-formulae from the guarded fragment is decided by the following decision procedure.

Step 1

Put F in positive normal form: push negations down to literals.

Step 2

Apply the following rule exhaustively to remove existential quantification:

$$\frac{F[\exists \overline{i}.\ G[\overline{i}]]}{F[G[\overline{j}]]} \text{ for fresh } \overline{j} \quad (\mathsf{exists})$$

Existential quantification can arise during Step 1 if the given formula has a negated guarded $\Sigma^{\mathsf{RO}}_{\mathsf{A}}$-subformula.

Step 3

Steps 3-4 accomplish the reduction of universal quantification to finite conjunction. The main idea is to select a set of symbolic index terms on which to instantiate all universal quantifiers. The proof of Theorem 2.6.1 argues that the following set is sufficient for correctness.

From the output F_2 of Step 2, construct the *index set* $\mathcal{I}$:

$$\mathcal{I} \ = \ \{t \ : \ \cdot[t] \in F_2 \text{ such that } t \text{ is not a universally quantified variable}\}$$

This index set is the finite set of indices that need to be examined. It includes just the terms t that contain only existentially quantified variables and that occur in some read $a[t]$ somewhere in F.

Step 4

Apply the following rule exhaustively to remove universal quantification:

$$\frac{H[\forall \overline{i}.\ a[i] \neq \bot \ \rightarrow \ F[a[i]]]}{H\left[\bigwedge_{\overline{i} \in \mathcal{I}^n} (a[i] \neq \bot \ \rightarrow \ F[a[i]])\right]} \quad (\mathsf{forall})$$

where n is the size of the list of quantified variables $\overline{i}$. This is the key step. It replaces universal quantification with finite conjunction over the index set.

Step 5

Decide the $T^{\mathsf{RO}}_{\mathsf{A}}$-satisfiability of F_4 using the decision procedure for the quantifier-free fragment.

Unlike in the case of standard arrays, this procedure works just as well on arrays with integer-indexed arrays. While convenient, it also suggests the weakness of this theory.

Theorem 2.6.1 (Sound & Complete) Given $\Sigma_{\mathsf{A}}^{\mathsf{RO}}$-formula F from the guarded fragment, the decision procedure returns satisfiable if F is $T_{\mathsf{A}}^{\mathsf{RO}}$-satisfiable; otherwise, it returns unsatisfiable.

Proof. As in the proof of Theorem 2.2.1, the crux of the proof is that the index set constructed in Step 4 is sufficient for producing a $T_{\mathsf{A}}^{\mathsf{RO}}$-equisatisfiable quantifier-free formula.

That satisfiability of F implies the satisfiability of F_4 is straightforward: Step 4 weakens universal quantification to finite conjunction.

For the opposite direction, assume that $T_{\mathsf{A}}^{\mathsf{RO}}$-interpretation I is such that $I \models F_4$. Then a $T_{\mathsf{A}}^{\mathsf{RO}}$-interpretation J such that $J \models F$ is simple to construct: make it like I, but assign $\perp$ to every array position not referenced by some fixed position in $\mathcal{I}$, which is the set $\mathcal{I}$ under interpretation I.

To prove that $J \models F$, simply note that every instance of a guarded universal quantification

$$\forall i.\ a[i] \neq \perp \ \rightarrow\ F[a[i]]$$

is easily satisfied by J if the corresponding conjunction of F_4 is satisfied by I. For it clearly holds on positions in $\mathcal{I}$; and in other positions, $a[i]$ is $\perp$ under J, falsifying the antecedent. □

Let us now consider the case of arrays with writes, $T_{\mathsf{A}}^{\perp}$. In fact, all we need to do is to insert the following step:

Step 1.5

Apply the following rule exhaustively to remove writes:

$$\frac{F[a\langle i \triangleleft v\rangle]}{F[a'] \ \wedge\ a'[i] = v \ \wedge\ (\forall j.\ j \neq i \ \rightarrow\ a[j] = a'[j])} \text{ for fresh } a' \quad (\mathsf{write})$$

Oddly enough, this step introduces an unguarded formula (which is certainly required to obtain true equality). The following theorem proves that the index set $\mathcal{I}$ of the procedure is still sufficient.

Theorem 2.6.2 (Sound & Complete) Given $\Sigma_{\mathsf{A}}^{\perp}$-formula F from the guarded fragment, the decision procedure returns satisfiable if F is $T_{\mathsf{A}}^{\perp}$-satisfiable; otherwise, it returns unsatisfiable.

Proof. The proof mirrors the proof of Theorem 2.6.1, except in considering the universally quantified formulae introduced for array equality. For such a formula

$$\forall j.\ j \neq i \ \rightarrow\ a[j] = a'[j]\ ,$$

$j \neq i \ \rightarrow \ a[j] = a'[j]$ holds at positions $j \in \mathcal{I}$ under I and J if it holds in the corresponding conjunction of F_4 under I and J. For other positions j, both $a[j]$ and $a'[j]$ are assigned $\perp$ under J and are thus equal. □

2.6.3 Extensions

Can the guarded fragment be extended and still retain decidability? One form of extension is to allow some unguarded universal quantification. Indeed, we saw such a case arise in Step 1.5. Not surprisingly, we can modify the procedure to replace equality between arrays $a = b$ with the unguarded formula $\forall i.\ a[i] = b[i]$; the proof of correctness follows Theorem 2.6.2 closely. Exact bounded equality is similarly possible. Most likely, there are other useful special forms of unguarded formulae that will retain decidability.

However, allowing arbitrary unguarded universal quantification will clearly result in undecidability: allowing subformulae of the form $\forall i.\ a[i] \neq \perp$ enables simple reductions from the negative results of Section 2.4.

Another extension is to allow another (guarded) quantifier alternation via guarded existential quantification:

$$\exists i.\ a[i] \neq \perp \ \wedge \ F[a[i]] \ .$$

This extension is strong, however: it allows us to force certain positions to be defined. We exploit this characteristic in the following reduction which applies the mechanisms developed in the proof of Theorem 2.4.1:

$$\begin{array}{l} \exists \overline{v}, s.\ \bigwedge_{v \in \overline{v}} v[0] \neq \perp \ \wedge \ s[0] \neq \perp \ \wedge \ T_{q(\overline{0})}[\overline{v}[0]] \ \wedge \\ \left[\begin{array}{l} \forall i.\ \left(\bigwedge_{v \in \overline{v}} v[i] \neq \perp \ \wedge \ s[i] \neq \perp\right) \ \rightarrow \\ \qquad \exists j.\ \bigwedge_{v \in \overline{v}} v[j] \neq \perp \ \wedge \ s[j] \neq \perp \ \wedge \left[\begin{array}{l} \left(\overline{v}[j] = \overline{v}[i] \ \vee \ \bigvee_{x_i} T_{x_i+1}[\overline{v}[i], \overline{v}[j]]\right) \\ \quad \wedge \ s[j] = s[i] - v_{q(\overline{x})}[i] \\ \quad \wedge \ s[i] > 0 \end{array} \right] \end{array} \right] \end{array}$$

Hence, another quantifier alternation, even guarded, results in undecidability.

To conclude our discussion of partial arrays, we remark on the relationship between T_{A} and $T_{\mathsf{A}}^{\mathbb{Z}}$ and the guarded fragment of $T_{\mathsf{A}}^{\perp}$. Specifically, the decision procedure for the guarded fragment of $T_{\mathsf{A}}^{\perp}$ is a semi-decision procedure for the corresponding (unguarded) fragments of T_{A} and $T_{\mathsf{A}}^{\mathbb{Z}}$. Consider guarded $\Sigma_{\mathsf{A}}^{\perp}$-formula F and the corresponding unguarded Σ_{A}- or $\Sigma_{\mathsf{A}}^{\mathbb{Z}}$-formula G in which every guarding antecedent $a[i] \neq \perp$ of F has been removed. If T_{A}- or $T_{\mathsf{A}}^{\mathbb{Z}}$-interpretation I satisfies G ($I \models G$), then I satisfies F ($I \models F$). The contrapositive yields a semi-decision procedure for T_{A}- or $T_{\mathsf{A}}^{\mathbb{Z}}$-validity: if F is $T_{\mathsf{A}}^{\perp}$-unsatisfiable ($\neg F$ is $T_{\mathsf{A}}^{\perp}$-valid), then G is T_{A}- or $T_{\mathsf{A}}^{\mathbb{Z}}$-unsatisfiable ($\neg G$ is T_{A}- or $T_{\mathsf{A}}^{\mathbb{Z}}$-valid).

2.7 Conclusion

While there still exist interesting fragments for which decidability remains an open question (specifically, adding the power to express injectivity in T_{A} — but not in $T_{\mathsf{A}}^{\mathbb{Z}}$ — and adding restricted use of unguarded universal quantification in $T_{\mathsf{A}}^{\perp}$), we have largely answered the motivating two questions posed at the beginning of the chapter. The array property fragments of T_{A} and $T_{\mathsf{A}}^{\mathbb{Z}}$ capture many of the predicates previously examined separately, in particular predicates for reasoning about sorting programs and for reasoning about extensionality. Furthermore, they allow expressing many other interesting properties when array elements are interpreted in various theories. Finally, the array property fragment is in some sense natural: simple extensions to the fragment result in undecidability.

As a test of expressiveness and efficiency, we implemented the decision procedures of Sections 2.2 and 2.3 in our verifying compiler πVC and proved that the sorting algorithms BubbleSort, InsertionSort, MergeSort, and QuickSort actually return sorted arrays. Besides requiring the predicates sorted and partitioned, we found that bounded equality and general universal properties were essential for proving the correctness of MergeSort and QuickSort. We expect that these latter properties will be of great value in verifying more general programs that manipulate arrays. Finally, computation time was on the order of seconds.

Chapter 3

Property-Directed Invariant Generation

The inductive method of proving a safety property Π of a transition system $\mathcal{S} : \langle \overline{x}, \theta, \rho \rangle$ requires finding an auxiliary assertion χ such that $\Pi \wedge \chi$ is inductive for the system [MP95]. This chapter describes an invariant generation-based approach to finding auxiliary assertions for proving safety properties of software and hardware systems.

An *invariant generation procedure* discovers auxiliary assertions. An *abstract interpretation* based on forward propagation typically finds one large inductive assertion φ [CC77]. The discovered assertion φ may or may not be a suitable auxiliary invariant for proving the invariance of Π. Incremental invariant generation — for example of affine invariants [CSS03] — produces a sequence of (relatively weak) inductive assertions $\varphi_1, \ldots, \varphi_k$; again, the conjunction of a subset of these assertions may or may not be suitable for proving Π.

In [BM06], we suggested a method for making incremental invariant generation of affine and polynomial inequality invariants property-directed so that each successive φ_i makes progress toward building an auxiliary assertion χ such that $\Pi \wedge \chi$ is inductive. In this chapter, we extend the approach to generating inductive clauses in a directed fashion for analyzing safety properties of finite-state systems. We present both applications in a unified setting.

Section 3.1 presents the property-directed incremental methodology in an abstract setting. Then Sections 3.2 and 3.3 develop instances of this methodology for generating inequality and clausal invariants, respectively. Finally, Section 3.4 discusses related work and further directions for research.

3.1 Property-Directed Incremental Invariant Generation

To prove that Π is an invariant of $\mathcal{S}$: $\langle \overline{x}, \theta, \rho \rangle$, we attempt to find a strengthening assertion χ such that $\Pi \wedge \chi$ is inductive. How do we find χ? This section describes several strategies of incrementally generating χ. In all cases, it is assumed that $\theta \Rightarrow \Pi$; otherwise, a counterexample is easily found.

3.1.1 Incremental Invariant Generation

The first strategy ignores Π while iteratively building a sequence χ_i of ever stronger inductive formulae. Let $\chi_0 \stackrel{\text{def}}{=} \mathsf{true}$. Suppose that we have performed i iterations so far to produce an inductive formula χ_i; that is, χ_i satisfies the (initiation) and (consecution) conditions:

$$\theta \Rightarrow \chi_i \quad \text{and} \quad \chi_i \wedge \rho \Rightarrow \chi_i' \,. \tag{3.1}$$

On iteration $i+1$, we seek a new formula γ that is inductive *relative to* χ_i:

$$\theta \Rightarrow \gamma \quad \text{and} \quad \chi_i \wedge \gamma \wedge \rho \Rightarrow \gamma' \,. \tag{3.2}$$

From (3.1) and (3.2), it follows that $\chi_i \wedge \gamma$ is inductive:

$$\theta \Rightarrow \chi_i \wedge \gamma \quad \text{and} \quad \chi_i \wedge \gamma \wedge \rho \Rightarrow \chi_i' \wedge \gamma' \,.$$

To ensure that γ also provides new information, we require that it satisfy an additional condition called (strengthening):

$$\exists \overline{x}.\ \chi_i \wedge \neg\gamma \,.$$

This condition asserts that γ is not just a logical consequence of χ_i. Instead, $\chi_i \wedge \gamma$ is inductive and is strictly stronger than χ_i: it eliminates more states than χ_i. This iteration ends by updating the inductive information that is known: $\chi_{i+1} := \chi_i \wedge \gamma$. χ_{i+1} is inductive and is strictly stronger than χ_i.

We call this strategy *incremental invariant generation*. Incremental invariant generation is useful in practice for discovering information about a program in the absence of a specification.

3.1.2 Property-Directed Incremental Invariant Generation

The second strategy uses Π to guide the invariant generation and is thus called *property-directed incremental invariant generation*. Again, let $\chi_0 \stackrel{\text{def}}{=} \mathsf{true}$. Suppose that we have performed i

iterations so far to produce a formula χ_i that is inductive relative to Π:

$$\theta \Rightarrow \chi_i \quad \text{and} \quad \Pi \wedge \chi_i \wedge \rho \Rightarrow \chi_i' \,. \tag{3.3}$$

Note that χ_i is not necessarily inductive by itself but is inductive relative to Π. On iteration $i+1$, we seek a new formula γ that is inductive *relative to* $\Pi \wedge \chi_i$:

$$\theta \Rightarrow \gamma \quad \text{and} \quad \Pi \wedge \chi_i \wedge \gamma \wedge \rho \Rightarrow \gamma' \,. \tag{3.4}$$

From (3.3) and (3.4), it follows that $\chi_i \wedge \gamma$ is inductive relative to Π:

$$\theta \Rightarrow \chi_i \wedge \gamma \quad \text{and} \quad \Pi \wedge \chi_i \wedge \gamma \wedge \rho \Rightarrow \chi_i' \wedge \gamma' \,.$$

As in incremental invariant generation, we want to ensure that γ provides new information. We could apply the (strengthening) condition

$$\exists \overline{x}.\ \chi_i \wedge \neg\gamma \,.$$

Alternately, consider when consecution fails:

$$\Pi \wedge \chi_i \wedge \rho \not\Rightarrow \Pi' \,.$$

In this case, there is a *counterexample to induction*: a state s that can lead to a state s' that violates Π. We require that the new invariant γ eliminate such a state s:

$$\exists \overline{x}, \overline{x}'.\ \Pi \wedge \chi_i \wedge \rho \wedge \neg\Pi' \wedge \neg\gamma \,. \tag{3.5}$$

We refer to requirement (3.5) as the (Π-strengthening) verification condition. The invariant generation procedure should thus produce a new inductive formula that satisfies (initiation), (consecution), and (Π-strengthening). This iteration ends by conjoining the new inductive invariant γ to the already known information: $\chi_{i+1} := \chi_i \wedge \gamma$. χ_{i+1} is strictly stronger than χ_i. By construction, states excluded by χ_{i+1} cannot be reached from a θ-state without first passing through some other $\neg\Pi$-state.

The iterations continue until $\Pi \wedge \chi_{i^*}$ is inductive on some iteration i^*. From (3.3), if the following implication holds, then $\Pi \wedge \chi_{i^*}$ is inductive:

$$\Pi \wedge \chi_{i^*} \wedge \rho \Rightarrow \Pi' \,.$$

Notice that although Π is assumed to hold during the construction of each χ_i, the inductiveness of $\Pi \wedge \chi_{i^*}$ proves the invariance of Π as desired.

3.1.3 The Sampling Strategy

Implementing the strengthening conditions directly in a static analysis can be computationally prohibitive. For example, in the application in Section 3.2, quantifier elimination could, in principle, solve the initiation, consecution, and strengthening conditions exactly; however, the analysis would not scale beyond a few variables.

Instead, the analyses of both Sections 3.2 and 3.3 decouple the initiation, consecution, and strengthening conditions. Consider the case of applying (Π-strengthening). First, a state $\overline{s}$ is selected that satisfies

$$\exists \overline{x}'.\ \chi_i[\overline{s}]\ \wedge\ \Pi[\overline{s}]\ \wedge\ \rho[\overline{s},\overline{x}']\ \wedge\ \neg\Pi[\overline{x}']\ , \tag{3.6}$$

so that it can lead in one step to a violation of Π. Then the incremental invariant generation procedure constructs a new inductive assertion γ that satisfies

$$\theta\ \Rightarrow\ \chi_i \wedge \gamma \quad \text{and} \quad \Pi \wedge \chi_i \wedge \gamma \wedge \rho\ \Rightarrow\ \chi_i' \wedge \gamma' \quad \text{and} \quad \neg\gamma[\overline{s}]\ .$$

It is possible that for a given $\overline{s}$, no single assertion γ exists that excludes $\overline{s}$; in this case, a new sample state $\overline{s}$ should be selected. To allow resampling, (3.6) should be solved with a randomized constraint solver that, ideally, selects a satisfying state uniformly at random. In our experience, we have found that randomized constraint solvers with no such guarantee are adequate.

3.2 Inequality Invariants

We now apply the property-directed invariant generation method to the problem of generating polynomial inequality invariants of linear and polynomial transition systems. Section 3.2.1 reviews the *constraint-based* method of inequality invariant generation. Then Section 3.2.2 uses the resulting invariant generator in a property-directed procedure. The primary advantage of using the property-directed method in this application is that it allows using generic constraint solvers. We describe an incomplete but scalable constraint solver in Section 3.2.3.

3.2.1 Constraint-based Invariant Generation

To begin, we review the constraint-solving method based on *Farkas's Lemma* [CSS03] or *Lagrangian relaxation* [Cou05]. Farkas's Lemma 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 (Lagrangian multipliers) [Sch86]. Theorem 3.2.1 describes the case when S consists of a conjunction of affine inequalities. Then Corollary 3.2.1 extends the result, with loss of completeness, to the case when some constraints of S are polynomial inequalities.

Both are statements of classic theorems.

Theorem 3.2.1 (Farkas's Lemma) Consider the following universal constraint system of affine inequalities over real variables $\overline{x} = \{x_1, \ldots, x_m\}$:

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

If S is satisfiable, it entails affine inequality $c_0 + c_1x_1 + \cdots + c_mx_m \geq 0$ iff there exist real numbers $\lambda_1, \ldots, \lambda_n \geq 0$ such that

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

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 possible [Cou05, PJ04]; however, our formulation allows avoiding the loss of soundness from floating point errors (see Section 3.2.3).

Corollary 3.2.1 (Polynomial Lemma) Consider the universal constraint system S of polynomial inequalities over real variables $\overline{x} = \{x_1, \ldots, x_m\}$:

$$A: \left\{\begin{array}{ccccc} a_{1,0} & + & \sum_{i=1}^{m} a_{1,i}t_i & \geq & 0 \\ & & \vdots & & \\ a_{n,0} & + & \sum_{i=1}^{m} a_{n,i}t_i & \geq & 0 \end{array}\right.$$

$$C: \quad c_0 + \sum_{i=1}^{m} c_i t_i \geq 0$$

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 in which variable is of even power (e.g., 1, x^2, x^2y^4, *etc.*, but not xy^3), impose the constraint

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

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

Figure 3.1: Simple

for all other terms, impose the constraint

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

Finally, require 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 [CSS03] uses Farkas's Lemma to generate affine inequality invariants over linear transition systems. The method begins by proposing an affine inequality *template*

$$c_0 + c_1 x_1 + \cdots + c_n x_n \geq 0 \ ,$$

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

$$\theta \ \Rightarrow \ c_0 + c_1 x_1 + \cdots + c_n x_n \geq 0$$

and (consecution)

$$c_0 + c_1 x_1 + \cdots + c_n x_n \geq 0 \ \wedge \ \rho \ \Rightarrow \ c_0 + c_1 x'_1 + \cdots + c_n x'_n \geq 0 \ .$$

Expressing θ and ρ in DNF reveals a finite set of parameterized linear constraint systems of the form in Theorem 3.2.1, 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 an instantiation of the c_i corresponding to an inductive inequality. Corollary 3.2.1 and the stronger versions in [Cou05] provide the mechanism for extending this technique to polynomial inequality invariants and analyzing polynomial transition systems.

Example 3.2.1 (Simple) Consider the loop Simple in Figure 3.1, which first appeared in [CH78]. The $\cdots$ in Figure 3.1 indicates nondeterministic choice. The corresponding linear transition system $\mathcal{S}$ is the following:

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

with $\rho\colon \rho_1 \ \vee \ \rho_2$.

The constraint-based method seeks inductive instantiations of the template

$$p_0 + p_1 j + p_2 k \geq 0$$

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

$$j=2 \ \wedge \ k=0 \ \Rightarrow \ p_0 + p_1 j + p_2 k \geq 0 \ ,$$

or more simply,

$$p_0 + 2p_1 + 0p_2 \geq 0 \ . \tag{3.7}$$

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

$$p_0 + p_1 j + p_2 k \geq 0 \ \wedge \ j' = j+4 \ \wedge \ k' = k \ \Rightarrow \ p_0 + p_1 j' + p_2 k' \geq 0 \ ,$$

or more simply

$$p_0 + p_1 j + p_2 k \geq 0 \ \Rightarrow \ (p_0 + 4p_1) + p_1 j + p_2 k \geq 0 \ ; \tag{3.8}$$

and

$$p_0 + p_1 j + p_2 k \geq 0 \ \Rightarrow \ (p_0 + 2p_1 + p_2) + p_1 j + p_2 k \geq 0 \ . \tag{3.9}$$

All together, the (primal) constraint problem is thus

$$\begin{array}{l} p_0 + 2p_1 + 0p_2 \geq 0 \\ \quad \wedge \ p_0 + p_1 j + p_2 k \geq 0 \ \Rightarrow \ (p_0 + 4p_1) + p_1 j + p_2 k \geq 0 \\ \quad \wedge \ p_0 + p_1 j + p_2 k \geq 0 \ \Rightarrow \ (p_0 + 2p_1 + p_2) + p_1 j + p_2 k \geq 0 \end{array} \tag{3.10}$$

According to Theorem 3.2.1, we then have the following set of dual constraints:

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

from (3.7),

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

from (3.8), 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 (3.9). 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 $k \geq 0$. Section 3.2.3 describes an incomplete constraint solver that is often fast in practice for solving such constraint systems. □

3.2.2 Property-Directed Iterative Invariant Generation

In this section, we apply the constraint-based method to produce a procedure for property-directed incremental invariant generation. Transitioning to incremental invariant generation is not obvious. For example, applying a generic constraint solver to solve the dual constraint problem arising from (3.10) will probably reveal the same solution again and again. The authors of [CSS03] implemented a specialized solver described in [SSM04] to avoid this problem. In contrast, the method of [Cou05] is essentially restricted to verification constraints in which a single solution solves the problem (*e.g.*, to prove termination by finding a ranking function; see Section 3.4).

We describe two iterative procedures based on the sampling strategy of approximating additional (strengthening) or (Π-strengthening) conditions.

Incremental Invariant Generation

As in Section 3.1, suppose that we have generated so far the inductive invariant χ_i. We seek a new invariant γ that is inductive relative to χ_i and, according to the (strengthening) condition, such that γ excludes at least one state allowed by χ_i. Ideally, we would impose the conditions (initiation), (consecution), and (strengthening) simultaneously. However, the resulting constraint system would have both universally and existentially quantified variables and would thus not be amenable to analysis via Theorem 3.2.1 or Corollary 3.2.1 (although the constraint system would be solvable in principle at high computational cost [Tar51, Col75]). The following procedure

applies the sampling strategy of Section 3.1.3 to decouple the (strengthening) condition from the others.

Let

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

be the template. Perform the following steps:

1. Solve the existential constraint system χ_i. The solution is a state $\overline{s}$.

2. Impose the conditions (initiation) and (consecution) on the template. For (initiation), impose

   $$\theta \;\Rightarrow\; p_0 + p_1 x_1 + \cdots + p_n x_n \geq 0 \;;$$

   for (consecution), use the known χ_i:

   $$\chi_i \;\wedge\; p_0 + p_1 x_1 + \cdots + p_n x_n \geq 0 \;\wedge\; \rho \;\Rightarrow\; p_0 + p_1 x'_1 + \cdots + p_n x'_n \geq 0 \;.$$

   Apply Theorem 3.2.1 or Corollary 3.2.1 to generate the dual constraint system ψ. Finally, solve the existential constraint system

   $$T\{\overline{x} \mapsto \overline{s}\} < 0 \;\wedge\; \psi \;, \tag{3.11}$$

   where $T\{\overline{x} \mapsto \overline{s}\}$ represents the substitution of the state $\overline{s}$ for the variables $\overline{x}$. A solution that assigns $\overline{q}$ to $\overline{p}$ represents the inequality

   $$q_0 + q_1 x_1 + \cdots + q_n x_n \geq 0$$

   that is inductive relative to χ_i and that excludes the state $\overline{s}$. If a solution is not found, return to Step 1 to choose another state $\overline{s}$.

3. Optimize the discovered solution $\overline{q}$. Solve the following optimization problem:

   minimize p_0
   subject to
   $$(T \geq 0 \;\wedge\; \psi)\{p_1 \mapsto q_1,\; \ldots,\; p_n \mapsto q_n\}$$

   At most q_0 from Step 2 is returned as the minimum of p_0. Set q_0 to the new solution.

4. Let $\chi_{i+1} \stackrel{\text{def}}{=} \chi_i \;\wedge\; T\{\overline{p} \mapsto \overline{q}\} \geq 0$.

Step 1 and condition (3.11) addresses the (strengthening) condition, while Steps 2 and 3 address

the (initiation) and (consecution) conditions.

The constraint problem of Step 1 can be solved in numerous ways, including using decision procedures or numerical constraint solvers for solving linear and semi-algebraic constraint problems. It is best if these solvers can be randomized so that a wide selection of sample points is possible. The constraint problem of Step 2 can be solved using a solver for bilinear constraint problems (see Section 3.2.3).

Note that the strict inequality of Step 2 is easily handled by transforming the feasibility problem into an optimization problem:

maximize ϵ
subject to
$T\{\overline{x} \mapsto s\} \leq -\epsilon \ \wedge \ \psi$

The original system is feasible if and only if the maximum is positive.

Example 3.2.2 Consider again the linear transition system of Example 3.2.1. Let $\chi_0 \stackrel{\text{def}}{=} \mathsf{true}$. On the first iteration, let

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

be the template. Step 1 returns a point satisfying χ_0, namely any point. Suppose that it is $s_1 : (j : -1, \ k : -3)$. According to Step 2, impose (initiation) and (consecution) on $T \geq 0$ and let ψ be the dual constraint system. Because χ_0 is true, the dual constraint system of Example 3.2.1 is ψ:

$$(p_0 + 2p_1 + 0p_2 \geq 0) \ \wedge \ \begin{pmatrix} p_1 = \lambda_1 p_1 \\ \wedge \ p_2 = \lambda_1 p_2 \\ \wedge \ p_0 + 4p_1 \geq \lambda_1 p_0 \end{pmatrix} \ \wedge \ \begin{pmatrix} p_1 = \lambda_2 p_1 \\ \wedge \ p_2 = \lambda_2 p_2 \\ \wedge \ p_0 + 2p_1 + p_2 \geq \lambda_2 p_0 \end{pmatrix} \quad (3.12)$$

Then solve

$$T\{j \mapsto -1, \ k \mapsto -3\} < 0 \ \wedge \ \psi \ , \quad \textit{i.e.}, \ p_0 - p_1 - 3p_2 < 0 \ \wedge \ \psi \ ,$$

which is a constraint system over $\{p_0, p_1, p_2, \lambda_1, \lambda_2\}$. One possible solution corresponds to the inductive formula $\gamma : k \geq 0$. Indeed, $k \geq 0$ excludes $s_1 : (j : -1, \ k : -3)$ and is inductive. Step 3

attempts to improve upon $k \geq 0$ by tightening the constant coefficient:

minimize p_0
subject to
$p_0 + k \geq 0 \ \wedge \ \psi\{p_1 \mapsto 0, \ p_2 \mapsto 1\}$

However, $k \geq 0$ is already as strong as possible. Hence, $\chi_1 \stackrel{\text{def}}{=} \chi_0 \ \wedge \ k \geq 0$. The assertion $\chi_1 : k \geq 0$ excludes the state $s_1 : (j : -1, \ k : -3)$ and any other state in which $k < 0$. Thus, no future iteration can again discover $k \geq 0$ (or any weaker assertion).

On the next iteration, Step 1 could find, for example, state $s_2 : (j : -1, \ k : 5)$, which satisfies $\chi_1 : k \geq 0$. Steps 2 and 3 would then discover inductive assertion $j \geq 0$ to eliminate this point, resulting in $\chi_2 : k \geq 0 \ \wedge \ j \geq 0$. A third or fourth iteration will probably yield the invariant $j \geq 2k + 2$: any sample point in which $j < 2k + 2$ (and $k \geq 0 \ \wedge \ j \geq 0$), such as $(j : 5, \ k : 3)$, reveals this invariant. Of the remaining points in the (j, k) plain, approximately half violate $j \geq 2k + 2$. Then

$$\chi_3 : \ k \geq 0 \ \wedge \ j \geq 0 \ \wedge \ j \geq 2k + 2 \ ,$$

which is the strongest possible conjunctive affine invariant of the system. □

Property-Directed Incremental Invariant Generation

Let

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

be the template. Perform the following steps:

1. Solve the existential constraint system corresponding to the imposed strengthening condition:

   - χ_i for (strengthening),
   - $\Pi \wedge \chi_i \wedge \rho \wedge \neg\Pi'$ for (Π-strengthening).

   If the system for (Π-strengthening) does not have a solution, then declare that Π is $\mathcal{S}$-invariant. Otherwise, the solution is a state $\overline{s}$.

Step 2-4 are like in *incremental invariant generation* with one difference. If a solution is not found in Step 2 and a (Π-strengthening) condition was imposed in Step 1, possibly impose the (strengthening) condition in the next attempt. For there may not currently exist an inductive inequality that can exclude a state that leads to a violation of Π.

```
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
```

Figure 3.2: Sqrt

Example 3.2.3 (Integer Square-Root) The loop Sqrt in Figure 3.2 computes the integer square-root z of a positive integer x. On exit, the following relation should hold between z and x:

$$z^2 \leq x < (z+1)^2 \ .$$

Taking preconditions reveals that the following should be invariant at the top of the loop:

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

However, Π is not $\mathcal{S}$-inductive. We apply the iterative procedure using condition (Π-strengthening) for Step 1 to generate a χ such that $\chi \ \Rightarrow \ \Pi$.

On each iteration, we use 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.*,

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

Because T is a polynomial template, we use Corollary 3.2.1 instead of Theorem 3.2.1.

One execution of the procedure constructs the following sequence of inductive assertions, where each is inductive relative to the conjunction of the previous ones:

$$\begin{array}{rrrr}
\gamma_1 : & -x + ux - 2xz \geq 0 & \gamma_7 : & -1 + u \geq 0 \\
\gamma_2 : & u \geq 0 & \gamma_8 : & -2u - u^2 + 4w \geq 0 \\
\gamma_3 : & u - u^2 + 4uz - 4z^2 \geq 0 & \gamma_9 : & -3 - u^2 + 4w \geq 0 \\
\gamma_4 : & 3u + u^2 - 4w \geq 0 & \gamma_{10} : & -5u - u^2 + 6w \geq 0 \\
\gamma_5 : & x - ux + 2xz \geq 0 & \gamma_{11} : & -15 + 22u - 11u^2 + 4uw \geq 0 \\
\gamma_6 : & 1 + 2u + u^2 - 4w \geq 0 & \gamma_{12} : & -1 - 2u - u^2 + 4w \geq 0
\end{array}$$

Thus, $\chi_{12} \stackrel{\text{def}}{=} \gamma_1 \wedge \cdots \wedge \gamma_{12}$.

On the thirteenth iteration, it is discovered that no state satisfies $\chi_{12} \wedge \neg\Pi$, proving that $\Pi \wedge \chi_{12}$ is inductive and that Π is invariant. Specifically, γ_1 and γ_5 entail $u = 1 + 2z$, while γ_6 and γ_{12} entail $4w = (u+1)^2$. Thus, $w = (z+1)^2$, entailing Π. □

3.2.3 Solving Bilinear Constraint Problems

The constraint systems that arise in Step 2 of the procedures of Section 3.2.2 are *bilinear constraint problems*: quadratic terms are of the form λp, where λ and p are different variables. Specifically, the variable λ is an introduced Lagrangian multiplier, while the variable p is a parameter (an unknown coefficient in the template). For example, the constraint problem (3.12) is bilinear. While solvable via quantifier elimination [Tar51, Col75], this complete approach does not scale. Instead, [SSM04, BMS05a, 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 ψ. 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 when the disabled case is ignored [SSM03]). Then execute a search. Let ψ_i be the current constraint system.

1. If the conjunction of the subset of *linear* constraints of ψ_i is unsatisfiable, then return unsatisfiable.
2. Choose a multiplier λ_j that is unassigned and that appears bilinearly in ψ_i. If no such λ_j exists, return satisfiable.
3. Try each value v in λ_j's possible solution set:
   (a) Let $\psi_{i+1} \stackrel{\text{def}}{=} \psi_i\{\lambda_j \mapsto v\}$.
   (b) Recurse on ψ_{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 the possible loss of soundness from floating point errors [Cou05].

3.3 Clausal Invariants

Scaling verification to large circuits requires some form of abstraction relative to the asserted property. Many safety analyses of systems operate by abstracting a system's transition relation

relative to the property [GS97, CGJ+03, HJMS02, MA03, McM03, McM05]. In contrast, we describe a safety analysis of finite-state systems that operates on a system's full transition relation. The analysis incrementally strengthens the asserted property until it is inductive. Each iteration produces one inductive clause. The construction of each inductive clause is guided by a state that violates the inductiveness of the property. Therefore, the generated invariants are relevant for proving the property. Because the analysis is property-directed, it scales to proving properties that are local to submodules of relatively large circuits.

The analysis is just another instance of incremental invariant generation. However, let us consider the analysis from the point of view of standard model checking [CE82]. Suppose that an explicit-state backward model checker discovers a state s that leads to a violation of safety property Π. A purely explicit-state algorithm would update Π to be $\Pi \wedge \neg s$ and continue on this new property. If the property holds, then an inductive strengthening of Π is eventually discovered. Now consider computing the preimage of s to the fixpoint formula F: that is, F represents precisely those states that can reach s. Then update Π to be $\Pi \wedge \neg F$. For efficiency, one can under-approximate F. This computation converges to the same inductive assertion as in the first algorithm.

Now consider our alternate approach. State s and its negation $\neg s$ are described by a conjunction of literals and a clause, respectively, say

$$s : x_1 \wedge \neg x_2 \wedge x_3 \wedge \cdots$$
$$\neg s : \neg x_1 \vee x_2 \vee \neg x_3 \vee \cdots$$

Any subclause c of $\neg s$ that is inductive for the system is necessarily stronger than $\neg F$ of the preimage computation described above. Why? For c to be inductive, the represented set of states cannot include any state that can lead to s. Moreover, c can be much stronger than $\neg F$ because reasoning based on an inductive argument allows us to eliminate states other than just those that can lead to s. Finally, c can be much more efficient to compute; in practice, preimage computations can be memory intensive, whereas a clause is small. A *minimal inductive subclause* (MIC) of $\neg s$ is an inductive subclause of $\neg s$ that does not contain any strict subclauses that are also inductive. When a (minimal) inductive subclause c of $\neg s$ exists, update Π to be $\Pi \wedge c$; otherwise, just update Π to be $\Pi \wedge \neg s$.

The success of our approach depends on two points. First, we need a fast way of finding a minimal inductive subclause when one exists. We address this challenge in Section 3.3.1. Second, small minimal inductive clauses must exist in practice. The empirical evidence of Section 3.3.4 suggests that they do. Our parallel implementation of our safety analysis solves all 20 PicoJava II benchmarks proposed by Ken McMillan [MA03, McM03, McM05], whereas proof-based abstraction [MA03] and interpolation-based model checking [McM03, McM05] solve 18 and 19, respectively. Additionally, our implementation succeeds on several benchmarks from the VIS

distribution [VIS] that are difficult for analyses based on k-induction [AS06]. In Section 3.4, we discuss related techniques and the potential for hybrid approaches.

First, we introduce some terminology and notation. A *literal* is a positive x or negative $\neg x$ occurrence of a program variable $x \in \overline{x}$. A *clause* c is a disjunction of literals. Notationally, we write a clause as a disjunction of literals $c : \ell_1 \vee \cdots \vee \ell_k$. The size $|c|$ of a clause c is the number of literals it contains (the size of the empty clause, equivalent to false, is 0). A *subclause* $d \sqsubseteq c$ is the disjunction of a subset of c's literals. Using this terminology and that of Section 3.1, each iteration of invariant generation produces a clause γ that is inductive relative to $\Pi \wedge \chi_i$. Hence, χ_i is a conjunction of i clauses.

3.3.1 Generating Minimal Inductive Subclauses

Consider the following problem. Given a Boolean transition system $\mathcal{S} : \langle \overline{x}, \theta, \rho \rangle$ and a clause c_0, find a *minimal inductive subclause* $c \sqsubseteq c_0$ if one exists.

Definition 3.3.1 (Minimal Inductive Clause) A *minimal inductive clause* c is $\mathcal{S}$-inductive (possibly relative to some formula ψ, such as $\Pi \wedge \chi_i$), and it does not contain any strict subclause $d \sqsubset c$ that is also $\mathcal{S}$-inductive (again, possibly relative to ψ). □

Shortly, we describe an algorithm for discovering a minimal inductive subclause (if one exists) of a given clause c_0. The idea of the algorithm is to traverse the lattice L_{c_0} defined by the subclauses of c_0, where

- the elements are the subclauses of c_0;
- comparison $\sqsubseteq$ is the weak subset relation between clauses' literal sets;
- and join $\sqcup$ and meet $\sqcap$ are the union and the intersection, respectively, between the literal sets of clauses.

Motivation: The Obvious Algorithm

Let us first consider the obvious approach to finding a minimal inductive subclause: execute a backward abstract interpretation to discover the unique largest inductive subclause, which is the greatest fixpoint d in L_{c_0} of $\mathcal{S}$. The algorithm is then applied recursively to each direct descendant of d in the lattice. It terminates with clause e when the greatest fixpoint computed for each of e's direct descendants does not cover the initial condition of $\mathcal{S}$; e is a minimal inductive subclause of c_0. There may be many minimal inductive subclauses of c_0.

How does one compute the largest inductive subclause of c_0? From abstract interpretation theory [CC77], one computes the greatest fixpoint in the lattice L_{c_0} by iteratively computing and under-approximating preimages. The direct approach under-approximates the preimage wp. For

clause c, one seeks the unique largest subclause $d \sqsubseteq c$ such that $d \Rightarrow \mathsf{wp}(c,\ \mathcal{S})$. One computes d by considering each literal ℓ of c in turn and testing whether $\ell \Rightarrow \mathsf{wp}(c,\ \mathcal{S})$; ℓ is in d iff the implication holds.

This approach has two problems. First, computing the greatest fixpoint is computationally expensive. In lattice L_{c_0}, it requires making $O(|c_0|)$ propositional satisfiability queries for each of $O(|c_0|)$ rounds of under-approximation; hence, $O(|c_0|^2)$ queries in total. Our solution exploits the clause domain and requires solving only $O(|c_0|)$ queries.

Second, descending the lattice L_{c_0} by finding successively stronger fixpoints can be slow. But once a fixpoint d is discovered, one can execute a forward abstract interpretation in the lattice L_d to discover, in practice, a relatively small fixpoint. By standard abstract interpretation theory, one computes an over-approximation to a least fixpoint by iteratively computing and over-approximating postimages. However, executing the forward abstract iteration naively also requires solving $O(|c_0|^2)$ queries. Our solution exploits a special property of monotonic queries to require solving only $O(m \log |c_0|)$ queries, where m is the size of the discovered clause.

Note that the forward abstract interpretation should only be executed within a lattice rooted by an inductive subclause of c_0; otherwise, it is likely that the discovered fixpoint is simply true even if a better fixpoint exists. Why? First, there may be exponentially many best over-approximations to a computed postimage because we are working with clauses. (Recall that the conjunctive normal form (CNF) representation of a propositional formula may be exponentially larger than the formula itself.) Second, each iteration of the forward abstract interpretation is blind to the clauses that are available to it in the future when it makes its choice among the best over-approximating clauses. Many choices may lead to paths in the lattice that do not contain inductive clauses. Therefore, the backward abstract interpretation is necessary.

From this description of the obvious approach, we derive the following high-level view of the problem and our proposed solution. First, the overall algorithm for finding a minimal inductive subclause of c_0 executes backward abstract interpretations to find successively smaller fixpoints (which are inductive subclauses of c_0). As an important optimization, it interleaves forward abstract interpretations to "jump" far down the lattice. Second, implementing the backward and forward abstract interpretations in a straightforward manner is computationally infeasible. Therefore, the remainder of our work is to identify fast methods of computing greatest fixpoints and over-approximations to least fixpoints.

Finding a Minimal Inductive Subclause

Figure 3.3 presents the algorithm MIC for finding a minimal inductive subclause: $\textsc{mic}(\psi, c_0)$ returns a minimal subclause $c \sqsubseteq c_0$ that is inductive relative to ψ if such a clause exists, and true otherwise. It formalizes the proposed algorithm of the previous section. First, it computes the greatest fixpoint c in the lattice L_{c_0} with DOWN. Second, it computes a (small, in practice)

```
let rec MIC(ψ, c_0) =
  {Find largest inductive subclause of c_0.}
  let c = DOWN(ψ, c_0) in
  if c = true
  then return true {no inductive subclause of c_0}
  else {Find "small" inductive subclause of c.}
       let d = UP(ψ, c) in
       {Recursively try each (|d| − 1)-subclause.}
       foreach ℓ ∈ d do
         let e = MIC(ψ, d \ {ℓ}) in
         if e ≠ true then return e
       done;
       {Nothing smaller was found: return d as minimal.}
       return d
```

Figure 3.3: Finding a minimal subclause of c_0 that is $\mathcal{S}$-inductive relative to ψ

```
let rec DOWN(ψ, c_0) =
  {Check that initiation holds.}
  if ¬(θ ⇒ c_0)
  then return true
  else {Check that consecution holds.}
       if ψ ∧ c_0 ∧ ρ ⇒ c_0'
       then return c_0
       else {Select a (¬c_0)-predecessor.}
            let s = ∃x̄'. ψ ∧ c_0 ∧ ρ ∧ ¬c_0' in
            return DOWN(ψ, (c_0 ⊓ ¬s))
```

Figure 3.4: Finding the largest subclause of c_0 that is $\mathcal{S}$-inductive relative to ψ

fixpoint d in the lattice L_c with DOWN. Finally, it recurses on each of the direct descendants of d to attempt to find an even smaller clause than d.

In more detail, the algorithm DOWN (Figure 3.4) moves down L_{c_0} to find the largest inductive subclause c of c_0. If no such clauses exists, then MIC returns the (weakest possible) inductive invariant true. Otherwise, the discovered clause c is the top element of a sublattice L_c of L_{c_0}. The algorithm UP (Figure 3.5) performs an abstract interpretation in L_c, climbing L_c and returning an inductive element $d \sqsubseteq c$. Since the root c of the lattice is an inductive clause, UP returns c at worst. As we are interested in a minimal inductive clause, MIC then recurses on each immediate subclause of d, returning a clause if one is found. Otherwise, it returns d.

Let us consider $\text{DOWN}(\psi, c_0)$, which finds the greatest fixpoint in L_{c_0} according to the semantics given by the formula ψ and the transition relation ρ. If a subclause of c_0 that is inductive relative to ψ exists, the largest one is unique: it is the union of every inductive subclause of c_0. The downward traversal of the lattice L_{c_0} progresses according to ψ and the transition relation

ρ. If it is currently at element d of the lattice, then one of two possibilities exists: either d is inductive relative to ψ so that DOWN has found the largest inductive subclause, or it is not. In the latter case, there exists a state s that satisfies $\psi \wedge d$ and that can transition to a state that does not satisfy d. Formally, s satisfies the formula

$$\exists \overline{x}'.\ \psi \wedge d \wedge \rho \wedge \neg d' \ .$$

Any inductive strengthening of d must exclude s. Viewing s as a conjunction of literals, the weakest clause that excludes s is $\neg s$. Therefore, the strongest clause in the sublattice L_d of d that also excludes s is $d \sqcap \neg s$.

Proposition 3.3.1 (Largest Inductive Subclause) If DOWN(ψ, c_0) is not true, then it is the largest subclause of c_0 that is $\mathcal{S}$-inductive relative to ψ. This clause is unique. Moreover, it is discovered in at most $O(|c_0|)$ SAT calls.

Proof. Let the output clause be c. From the structure of DOWN, it is apparent that c is $\mathcal{S}$-inductive relative to ψ. We prove that every clause $d \sqsubseteq c_0$ that is $\mathcal{S}$-inductive relative to ψ is a subclause of the output clause c, proving that c is both the largest possible clause and unique: for if another "largest" inductive subclause exists, then it must be a subset of c, a contradiction.

Suppose that $d \not\sqsubseteq c$ yet $d \sqsubseteq c_0$ and d is $\mathcal{S}$-inductive relative to ψ. Without loss of generality, suppose that $d \not\sqsubseteq c_0 \sqcap \neg s$ — that is, the loss of some literal of d occurs at this level of recursion. Decompose c_0 as $d \vee e$ and consider selecting state s: $\exists \overline{x}'.\ \psi \wedge (d \vee e) \wedge \rho \wedge \neg d' \wedge \neg e'$. Distributing over disjunction, we find two possibilities: one of

$$\exists \overline{x}'.\ \psi \wedge d \wedge \rho \wedge \neg d' \wedge \neg e' \tag{3.13}$$

$$\exists \overline{x}'.\ \psi \wedge e \wedge \neg d \wedge \rho \wedge \neg d' \wedge \neg e' \tag{3.14}$$

is satisfiable. But (3.13) is unsatisfiable because d is $\mathcal{S}$-inductive relative to ψ. Therefore, the discovered state s satisfies $e \wedge \neg d$; and $\neg s$ (and hence $c_0 \sqcap \neg s$) contains d, contradicting our assumption.

Regarding the complexity claim, it is easy to see that each level of recursion must either terminate with an inductive subclause or remove at least one literal. □

An effective optimization in MIC and DOWN is to cache the clauses c examined by DOWN. Caching these sets avoids applying DOWN to a clause d when it has already been applied to superclause c ($d \sqsubseteq c$). For if c does not have an inductive subclause, then neither does d.

Let us turn to UP (Figure 3.5). First, recall that UP is unnecessary for the correctness of MIC; however, it is essential to a practical implementation of MIC (improving performance by orders of magnitude in our experiments). MIC with only DOWN progresses down the lattice of subclauses

```
{Precondition: c_0 is S-inductive relative to ψ.}
let UP(ψ, c_0) =
  {Initiation.}
  c := CONSEQUENCE(θ, c_0);
  {Consecution.}
  loop forever do
    {Compute successor clause. In practice, compute successors of new literals.}
    let d = UNPRIME(CONSEQUENCE(ψ ∧ c ∧ ρ, c'_0)) in
    if d ⊑ c
    then return c
    else c := c ⊔ d
  done
```

Figure 3.5: An abstract interpretation on c_0

from one inductive clause to the next smaller one. Adding UP accelerates the search by jumping down the lattice to small inductive subclauses.

UP(ψ, c_0) performs a forward abstract interpretation [CC77] in the lattice L_{c_0} defined by subclauses of c_0. Typical of an abstract interpretation, UP first computes a smallest element c of the lattice that approximates θ: $\theta \Rightarrow c$. It then iteratively applies an abstract strongest postcondition operator until a fixpoint is reached. Our abstract postcondition operator is

$$\text{UNPRIME}(\text{CONSEQUENCE}(\psi \wedge c \wedge \rho, c'_0)) \ .$$

To compute it, a smallest subclause of c'_0 is chosen to over-approximate the logical consequence of $\psi \wedge c \wedge \rho$. Because c_0 is inductive relative to ψ, this element is at most c'_0. The returned clause is then unprimed. The only technical aspect of this abstract interpretation is computing *minimal consequences* efficiently in the abstract postcondition operator.

Definition 3.3.2 (Minimal Consequence) A clause c is a *minimal consequence* of α if it is a logical consequence of α ($\alpha \Rightarrow c$), and there does not exist a strict subclause $d \sqsubset c$ that is also a consequence of α. □

CONSEQUENCE(α, c) should return a minimal consequence $d \sqsubseteq c$ of α. Clausal *consequences* can be computed similarly to *blocking clauses* [McM02], and *minimal* consequences can be computed similarly to *prime* blocking clauses [JS05] (or *prime implicates*; see Section 3.4 for further discussion). One method of enforcing primality of the resulting clause requires solving a number of propositional satisfiability queries linear in the number of literals of the original clause [JS05]. However, we discovered an algorithm that requires a number of calls linear in the number of literals in the discovered prime clause and only logarithmic in the number of literals in the original clause.

```
let rec CONS(α, support, c_0) =
  {Invariant: At least one literal in c_0 is required.}
  if |c_0| = 1
  then return c_0
  else let ℓ_0, r_0 = SPLIT(c_0) in
       {Is either (support ⊔ ℓ_0) or (support ⊔ r_0) a consequence of α?}
       if α ⇒ (support ⊔ ℓ_0)
       then return CONS(α, support, ℓ_0)
       else if α ⇒ (support ⊔ r_0)
            then return CONS(α, support, r_0)
            else {No, so we need literals from both ℓ_0 and r_0.}
                 let ℓ = CONS(α, r_0 ⊔ support, ℓ_0) in
                 let r = CONS(α, ℓ ⊔ support, r_0) in
                 return (ℓ ⊔ r)

let CONSEQUENCE(α, c) =
  if α ⇒ false
  then return false
  else return CONS(α, true, c)
```

Figure 3.6: Finding a minimal consequence of α in c

In principle, computing a minimal consequence of a formula α is straightforward because entailment is monotonic for disjunction: if c is a consequence of α, then so is $c \vee d$. Therefore, each literal ℓ of the given clause c_0 can be examined in turn. If the clause without ℓ is still a consequence, then drop ℓ; otherwise, keep it. This algorithm is essentially the *lifting* strategy of [JS05]. In practice, however, a minimal consequence is usually small relative to the given clause c_0, yet this method requires solving a number of satisfiability queries linear in $|c_0|$. We prefer an algorithm that requires solving a number of queries linear in the size of the returned minimal consequence.

CONSEQUENCE(α, c_0) is derived from binary search: it performs a binary search until literals from both halves of the divided clause are required. It then performs a binary search on each branch, using the literals from the other branch as support. Figure 3.6 details this algorithm.

Proposition 3.3.2 (Minimal Consequence) If CONSEQUENCE(α, c_0) returns c, then c is a minimal consequence of α. Moreover, it required solving at most $2\left((|c|-1) + |c| \lg \frac{|c_0|}{|c|}\right)$ propositional satisfiability queries.

In other words, it runs linearly in the size of the discovered minimal clause and only logarithmically in the size of the given clause. When the discovered clause is nearly the same size as the given clause, the algorithm runs linearly in the size of the given clause.

Let us return to the forward abstract interpretation. A simple optimization lifts the complexity result to computing the inductive clause itself. In particular, each iteration should compute

just the new literals that are necessary beyond the current ones: if c is the current clause and d is c_0 without any literal of c (so $c_0 = c \vee d$), then compute

$$\textsc{unprime}(\textsc{consequence}(\psi \ \wedge \ c \ \wedge \ \rho \ \wedge \ \neg c', \ d')) \ .$$

When each iteration of the abstract interpretation adds few literals relative to $|c_0|$, each literal of the inductive clause costs a number of queries logarithmic in $|c_0|$. Thus, the final inductive clause costs a number of queries logarithmic in $|c_0|$ and linear in its size.

As the CONSEQUENCE function has not been previously studied to our knowledge, we provide a proof of correctness in Section 3.3.3 and a complexity analysis that shows that it is nearly optimal in the number of satisfiability queries that it solves.

Proposition 3.3.3 (Minimal Inductive Subclause) If $\textsc{mic}(\psi, c)$ is not true, then it is a minimal subclause of c that is $\mathcal{S}$-inductive relative to ψ; otherwise, c does not have any subclauses that are $\mathcal{S}$-inductive relative to ψ.

Proof. Consider one level of recursion. From Proposition 3.3.1, c_0 contains an inductive subclause iff $c = \textsc{down}(\psi, c_0)$ is not true. In this case, $d = \textsc{up}(\psi, c)$ is not true: UP simply implements an abstract interpretation on the lattice L_c, so that the correctness of CONSEQUENCE (see Theorem 3.3.2) implies its correctness. Hence, $\textsc{mic}(\psi, c)$ returns an inductive subclause of c iff one exists.

Combining this argument with the loop of MIC proves that d is minimal. For if $\textsc{mic}(\psi, d \setminus \{\ell\})$ is true for $\ell \in d$, then d must be minimal. □

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 observed in our experiments.

3.3.2 A Complete Analysis

The MIC algorithm is the basis for our safety analysis. If Π is an invariant of $\mathcal{S}$, then a finite number of (relatively) inductive clauses are discovered to comprise an assertion χ such that $\Pi \wedge \chi$ is $\mathcal{S}$-inductive. Otherwise, the algorithm finds a counterexample trace. We call the algorithm *finite-state inductive strengthening* (FSIS).

FSIS is essentially a refinement of an explicit-state backward model checker [CE82]. It maintains a growing formula χ that is inductive relative to Π. When the explicit-state backward analysis finds a state s that can lead to a violation of Π, MIC is applied to $\neg s$ to produce a minimal inductive subclause of $\neg s$ that excludes s, the states that can reach s, and many other states as well. This clause is conjoined to χ. If $\neg s$ does not have an inductive subclause, then

```
let FSIS(S : ⟨x̄, θ, ρ⟩, Π) =
  χ := true;
  while Π ∧ χ ∧ ρ ⇏ Π′ do
    let s = ∃x̄′. Π ∧ χ ∧ ρ ∧ ¬Π′ in
    if ∃x̄. θ ∧ s
    then return false
    else let γ = MIC(Π ∧ χ, ¬s) in
         if γ = true
         then Π := Π ∧ ¬s
         else χ := χ ∧ γ
  done;
  return true
```

Figure 3.7: Finite-state inductive strengthening of Π

$\neg s$ is conjoined to Π: proving that s is unreachable is now a subgoal of proving the invariance of Π. Figure 3.7 presents the algorithm.

Theorem 3.3.1 (Complete) FSIS$(\mathcal{S}, \Pi)$ finds a formula χ such that $\Pi \wedge \chi$ is $\mathcal{S}$-inductive and returns true if and only if Π is $\mathcal{S}$-invariant; otherwise, it returns false.

Completeness is theoretically trivial given that explicit-state model checking is already complete. Section 3.3.4 presents preliminary experimental evidence of the practical value of FSIS.

3.3.3 The CONSEQUENCE Algorithm

In this section, we continue our study of the CONSEQUENCE algorithm of Section 3.3.1. We consider a more abstract problem: Given a finite set O and a predicate $p : 2^O \to \{\mathsf{true}, \mathsf{false}\}$ such that $p(O)$ and

$$p(S_0) \Rightarrow p(S_1) \quad \text{when } S_0 \subseteq S_1 \subseteq O \ ,$$

find a set $M \subseteq O$ such that $p(M)$ and, for $N \subset M$, $\neg p(N)$. p is a *monotonic* predicate on the set O. M is a *minimal satisfying subset* of O with respect to the predicate p. S may contain multiple minimal satisfying subsets. Figure 3.8 presents the algorithm for this general case.

In the case of CONSEQUENCE(α, c), the literals of the clause c comprise O, and the predicate p decides if a given clause is entailed by α. Calling p corresponds to solving one propositional satisfiability query.

Theorem 3.3.2 (Correct) Suppose that O is nonempty, $p(O)$, and $\neg p(\emptyset)$.

1. MIN$(O,\ \emptyset,\ p)$ terminates.

```
let rec MIN(p, sup, S) =
  if |S| = 1
  then return S
  else let ℓ0, r0 = SPLIT(S) in
       if p(sup ∪ ℓ0)
       then return MIN(p, sup, ℓ0)
       else if p(sup ∪ r0)
            then return MIN(p, sup, r0)
            else let ℓ = MIN(p, r0 ∪ sup, ℓ0) in
                 let r = MIN(p, ℓ ∪ sup, r0) in
                 return (ℓ ∪ r)

let MINIMAL(p, O) = return MIN(p, ∅, O)
```

Figure 3.8: Finding a minimal satisfying subset

2. Let $M = \text{MIN}(O,\ \emptyset,\ p)$. Then $p(M)$, and for each $e \in M$, $\neg p(M \setminus \{e\})$.

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

For the second claim, we prove first that $p(M)$. We then prove that for each $e \in M$, $\neg p(M \setminus \{e\})$.

For the first part of the second claim, we prove that the following are invariants of MIN, letting V be the return value:

1. the support $sup \cup S$ satisfies p: $p(sup \cup S)$

2. the union of sup and the return value V satisfies p: $p(sup \cup V)$

Invariants (1) and (2) are proved simultaneously. For the base case of (1), $p(sup \cup S)$, note that $p(\emptyset \cup O) = p(O)$, 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 $sup \cup r_0 \cup \ell_0 = sup \cup S$ for the third recursive call; and that $p(sup \cup \ell \cup r_0) = p(sup \cup r_0 \cup \ell)$, which is true by invariant (2).

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

In the first call to MIN in MINIMAL, $sup = \emptyset$; hence, $p(M) = p(\emptyset \cup M) = \text{true}$ by invariant (2).

To prove that M is minimal (*i.e.*, that for each $e \in M$, $\neg p(M \setminus \{e\})$), consider the invariants

3. the return value is a subset of S: $V \subseteq S$

4. the support sup does not satisfy p: $\neg p(sup)$

5. the union of *sup* and the return value is minimal: $\neg p(sup \cup V \setminus \{e\})$ for $e \in V$

For invariant (3), note for the base case that the first return of MIN returns $V = S$ 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 $V = \ell \cup r \subseteq \ell_0 \cup r_0 = S$.

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*; that at the third recursive call, $\neg p(sup \cup r_0)$; and that at the fourth recursive call, it is known that $\neg p(sup \cup \ell_0)$ 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 is necessary. The next two returns hold by the inductive hypothesis. For the final return, we know by the inductive hypothesis that $\neg p(sup \cup \ell \cup r \setminus \{e\})$ for $e \in r$; hence, all of r is necessary. Additionally, from the inductive hypothesis, $\neg p(sup \cup r_0 \cup \ell \setminus \{e\})$ for $e \in \ell$, and $\neg p(sup \cup r_0 \cup \ell \setminus \{e\})$ implies that $\neg p(sup \cup r \cup \ell \setminus \{e\})$ by monotonicity of p and because $r \subseteq r_0$ by invariant (3); hence, all of ℓ is necessary.

In the first call to MIN at MINIMAL, SUP $= \emptyset$ and $V = M$; hence, $\neg p(M \setminus \{e\})$ for $e \in M$ from invariant (5). □

Theorem 3.3.3 (Upper Bound) Let $M = \text{MIN}(O,\ \emptyset,\ p)$. Discovering M required at most

$$2\left((|M| - 1) + |M| \lg \frac{|O|}{|M|}\right)$$

queries to p.

Proof. Suppose that $|M| = 2^k$ and $|O| = n2^k$ for some $k, n > 0$. Each element of M induces one divergence at some level in the recursion. At worst, these divergences occur evenly distributed at the beginning, inducing $|M|$ separate binary searches over sets of size $\frac{|O|}{|M|}$. Hence, $|M| - 1$ calls to MIN diverge, while $|M| \lg \frac{|O|}{|M|}$ calls behave like in a binary search. Noting that each call results in at most 2 queries to p, we have the claimed upper bound in this special case, which is also an upper bound for the general case. □

For studying the lower bound on complexity of the problem, consider the following more rigidly defined problem. For a parameter m, consider sets O with precisely one minimal satisfying subset of size at most m. For such a set O, determine its minimal satisfying subset.

Theorem 3.3.4 (Lower Bound) Any algorithm for determining the minimal satisfying subset of O (of size at most m) requires making $\Omega\left(m \lg\left(\frac{n-m}{m}\right)\right)$ queries to p when $m \ll n$, and $\Omega(n)$ queries when m is approximately n.

Proof. As the subset can be any subset of O of size at most m, any algorithm must be able to distinguish among

$$\sum_{i=1}^{m} C(n,i) = \sum_{i=1}^{m} \frac{n!}{i!(n-i)!}$$

possible results using only queries to p. Thus, the height of a decision tree must have height at least

$$\lg\left(\sum_{i=1}^{m} \frac{n!}{i!(n-i)!}\right) .$$

When m is approximately n, the sum is approximately 2^n so that $\Omega(n)$ queries to p are required. When m is small compared to n, only the final term of the sum is significant, and, using Stirling's approximation,

$$\begin{aligned} \lg \tfrac{n!}{m!(n-m)!} &\geq \lg n! - \lg m! - \lg(n-m)! \\ &\quad - o(\lg m + \lg(n-m)) \\ &= \Omega\left(n \lg\left(\tfrac{n}{n-m}\right) + m \lg\left(\tfrac{n-m}{m}\right)\right) . \end{aligned}$$

The first term is insignificant when $m \ll n$. □

Hence, the algorithm is in some sense optimal. However, a clause may have an exponential number (in the size of the clause) of minimal consequences. In this situation, the lower bound analysis does not apply.

3.3.4 Experiments

Implementation

We implemented FSIS in O'Caml and used `Z-Chaff` version "2004.11.15 Simplified" [MMZ+01], making minor modifications to use it incrementally.

Three optimizations are important. First, conversion to CNF is minimized through caching CNF formulae within the SAT solver. Second, the implementation computes a cone of influence [BCCZ99] and considers for clauses only variables arising from latches. Third, for large systems, the implementation grows the set of considered variables: when insufficient, the set is grown by considering all relevant variables for one iteration.

The analysis is easily parallelized. Each process executes its own instance of FSIS and relies on the randomness of the SAT solver to avoid considering the same states as other processes. It reports each new discovered inductive clause to a central server, which then returns the new inductive clauses reported since the previous communication.

Table 3.1: Single-process analysis of benchmarks

#	Time (s)	Clauses	SAT	COI	Time (s) #2
2	48	11	450	306	40
3	31	4	120	306	50
5	84	194	17407	88	114
6	2329	442	34188	318	2518
7	19	64	4379	67	16
8	29	72	4841	90	26
9	4	24	1022	46	6
10	6	7	281	54	7
13	235	20	810	352	235
15	1270	195	15211	353	728
16	929	209	14803	290	1265
17	57096	1718	501920	211	76324
18	3460	436	62123	143	12833
19	285	99	5765	52	334

Table 3.2: Multi-process analysis of hard benchmarks

	2		4		8	
#	Time	Clauses	Time	Clauses	Time	Clauses
6	1278	500	945	586	502	598
15	474	126	774	207	253	160
16	562	234	665	316	297	348
17	34860	1716	14419	1593	8304	1872
18	5369	1052	2111	734	1431	1093

Results

We studied two sets of benchmarks. The PicoJava II microprocessor benchmark set, previously studied in [MA03, McM03, McM05], has the characteristic that the circuits are relatively large, yet only small sections of the circuits are relevant for proving the properties. Standard BDD-based model checking [BCM$^+$92] fails, so abstracting the system or its state-space with respect to the property is important. Of the 20 benchmarks, which are safety properties of the instruction fetching and caching modules of the PicoJava II chip, proof-based abstraction solved 18 [MA03], and interpolation-based model checking solved 19 [McM03, McM05]. Table 3.1 reports the performance results for a single process. The **Clauses**, **SAT**, and **COI** columns list the number of generated clauses, the number of SAT calls, and the number of active variables in the cone of influence, respectively. Results can vary widely across runs, so we report times for a second run in column **Time (s) #2**. Properties for benchmarks 0, 1, 4, 11, 12, and 14 are already inductive,

Table 3.3: VIS benchmarks

Name	Time	Clauses	Name	Time	Clauses
PETERSON	2	4	HEAP	8605	2551
PPC60X_2.1	71	3	PPC60X_2.2	69	3
PPC60X_2.5	61	1	PPC60X_2.9	381	69

so results are not reported in the table. For the large benchmarks 2, 3, 6, 12, 15, and 16, we use the growing-variable-set heuristic. Table 3.2 reports the performance results for analyzing the hardest benchmarks with multiple processes.

The second set of benchmarks is from the VIS distribution [VIS]. We applied the analysis to several valid properties of models that are difficult for standard k-induction (though easy for standard BDD-based model checking) [AS06]. Table 3.3 reports the results. For comparison, 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]. PETERSON, HEAP, and the PPC60X_2 models have 16, 29, and 94 latches, respectively.

Our timing results were obtained using 3 GHz processors.

3.4 Related Work

3.4.1 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 [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. [KM75] discusses the generation of constraint systems over loops with linear assertional guards and linear assignments for which solutions are linear ranking functions. In [CS01, CS02], it is observed that duality of linear constraints achieves efficient synthesis. [PR04] proves that this duality-based method is complete for single-path loops. [BMS05a] presents a complete method for the general case and shows how lexicographic linear ranking functions can also be computed efficiently in practice. In [Cou05], the approach is generalized by using semidefinite programming to approximate the polynomial case.

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. [CSS03] proposes using Farkas's Lemma and nonlinear constraint solving to generate affine invariants. A solver for this method is described in [SSM04]. [SSM03] specializes the technique to Petri nets, for which the problem is efficiently solvable. [BMS05c] proposes an idea for addressing the case in which system variables must be treated as integers, for which duality is unknown. The continuous analog of invariants, *barrier certificates*, was introduced in [PJ04]. They, too, apply a method based on constraint solving.

Invariants are often necessary for proving termination: while the possible state-space may not map into a well-founded domain, the reachable state-space may. One direction for future research is to extend the directed approach to termination. While [BMS05a] presents a method for constructing up to a fixed number of *supporting invariants*, a directed incremental approach would allow the number to be unbounded.

3.4.2 Safety Analysis of Hardware

Qualitative Comparisons

We consider the characteristics of several related safety analyses: bounded model checking (BMC) [BCCZ99], interpolation-based model checking (IMC) [McM03, McM05], k-induction (kI) [SSS00, dMRS03, AFF$^+$04, VH06, AS06], predicate abstraction with refinement (CEGAR) [CGJ$^+$03], and our analysis (FSIS). These analyses are similar in important ways: they are fundamentally based on computing an inductive set that excludes all error states; they consider the property to prove during the computation; and they use a SAT solver as the main computational resource. Additionally, unlike standard model checking [CE82, BCM$^+$90, McM02], they use both the initial condition and the desired property to construct implicitly or explicitly an abstraction of some kind.

We consider now their differences, defining several dimensions for comparison:

A. *Abstraction*: What is the mechanism for abstraction? Does it abstract the transition relation (TR)? Or does it directly abstract the state space through induction (SS)?

B. *Use of SAT solver*: Does it pose a few difficult SAT problems (FD)? Or does it pose many simple SAT problems (MS)?

C. *Intermediate results*: Are intermediate results transferable to other analyses (IT)? Or must it run to convergence to provide useful information (IU)?

D. *Parallelizable*: Is it naturally and easily parallelized (P)? Or is it naturally serial (S)?

Table 3.4 summarizes the results. An analysis may not fit precisely at one end of each dimension; however, this classification provides a first approximation.

Table 3.4: Summary of comparison

Analysis	**A**	**B**	**C**	**D**
BMC	SS	FD	IU	S
IMC	TR	FD	IT	S
kI	SS	FD	IU	S
CEGAR	TR	MS	IT	S
FSIS	SS	MS	IT	P

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

BMC, IMC, and kI pose relatively few difficult SAT problems in which the transition relation is unrolled many times. In contrast, CEGAR and FSIS pose many simple SAT problems in which the transition relation is not unrolled.

Each major iteration of IMC and CEGAR produces an inductive set that is informative even when it is not strong enough to prove the property. Each successive iteration of FSIS produces a stronger formula that excludes states that cannot be reached without previously violating the property. Intermediate iterations of BMC and kI are not really useful. Exceptions include forms of strengthening, which we discuss in greater depth below [dMRS03, AFF$^+$04, VH06, AS06].

Finally, only FSIS is natural to parallelize. The difficulty of subproblems grows with successive iterations in BMC, IMC, and kI so that parallelization across iterations is not useful. The SAT solver itself would have to be parallelized. In CEGAR, each iteration is dependent on the predicates just learned, though parallelization could occur at a finer level.

Related Work

SAT-based unbounded model checking for safety properties also generates clauses; at convergence, their conjunction is inductive [McM02]. However, our approach differs fundamentally. First, the semantics of clauses differ: in SAT-based model checking, each clause excludes a set of states that lead to a violation; in our approach, each clause represents an inductive set of states. Hence, the algorithms to compute the clauses differ markedly: in particular, our method computes several approximations to fixpoints to generate a single clause, whereas SAT-based

model checking computes clausal representations of one-step preimages. Second, the different semantics cause the behavior of the two approaches to differ: SAT-based model checking generates many clauses, each at low computational expense; our approach produces fewer clauses at (relatively) higher expense. Finally, SAT-based unbounded model checking is an implementation of CTL model checking with a clausal representation. It computes the fixpoint exactly, whereas our method abstracts the state space relative to the initial condition and the property.

Not surprisingly, however, the two methods share a concept: *blocking clauses* are used in SAT-based unbounded model checking [McM02]. Their discovery is refined in [JS05] to produce *prime* blocking clauses, requiring at worst as many SAT calls as literals. The CONSEQUENCE function of Section 3.3.1 requires asymptotically fewer SAT calls. An algorithm similar to the CONSEQUENCE algorithm is described in [Zel99]. However, that algorithm can handle only sets in which there exists precisely one minimal satisfying subset.

Strengthening based on under-approximating the states that can reach a violating state s has been 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 backward model checking [AS06] have been used to perform the strengthening. Our inductive basis for strengthening eliminates significantly more states in practice and at least as many in theory. The first clause discovered by DOWN already excludes all states that can reach s; in practice, a much smaller clause is usually found. This stronger clause eliminates many states from which s is not reachable.

An important direction for future research is to combine our incremental induction-based strengthening with k-induction. This hybrid analysis would vary k in each parallel process according to computational demands and to the success in finding MICs.

3.5 Conclusion

Incremental invariant generation procedures like the two described in this chapter characteristically synthesize relatively weak invariants at low computational cost. Typically, they share the same problem as *monolithic* procedures like polyhedra-based abstract interpretation [CH78] and symbolic model checking [BCM$^+$90]: they produce too much information (at unnecessary computational cost). We describe a general technique for directing incremental invariant generation according to a given property and discuss two instances of the technique. The evidence of Section 3.3.4 suggests that incremental invariant generation can indeed be targeted to a property so that relatively few invariants are actually generated.

Our work on safety analysis of hardware suggests an important direction for further research. The analysis is motivated by a classically deductive approach to verification [MP95]. Can classically deductive techniques motivate fast analyses for other temporal properties as well?

Bibliography

[Ack57] W. Ackermann. Solvable cases of the decision problem. *The Journal of Symbolic Logic*, 22(1):68–72, March 1957.

[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.

[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.

[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+90] Jerry R. Burch, Edmund M. Clarke, Ken L. McMillan, David L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, pages 428–439. IEEE Computer Society Press, June 1990.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. *Inf. Comput.*, 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.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007.

[BMS05a] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *Proc.* 17^{th} *Intl. Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 491–504. Springer Verlag, July 2005.

[BMS05b] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 3580 of *LNCS*, pages 1349–1361. Springer Verlag, 2005.

[BMS05c] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *CONCUR*, 2005.

[BMS05d] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Radhia Cousot, editor, *Proc. of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, Paris, France, January 2005. Springer Verlag.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation:* 7^{th} *International Conference, (VMCAI)*, volume 3855 of *LNCS*, pages 427–442, Charleston, SC, January 2006. Springer Verlag.

[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 4^{th} *ACM Symp. Princ. of Prog. Lang.*, 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, Workshop*, pages 52–71, London, UK, 1982. 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 5^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 84–97, January 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 *Proc. Verification, Model*

Checking, and Abstract Interpretation: 5^{th} *International Conference (VMCAI)*, pages 1–24, 2005.

[CS01] Michael Colón and Henny B. Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (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 *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification*, 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 *Computer Aided Verification*, 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.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[GNRZ06] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Deciding extensions of the theory of arrays by integrating decision procedures and instantiation strategies. In *JELIA*. Springer-Verlag, 2006.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proc.* 9^{th} *Intl. Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In 29^{th} *ACM Symp. Princ. of Prog. Lang. (POPL)*, pages 58–70. ACM Press, 2002.

[Jaf81] Joxan Jaffar. Presburger arithmetic with array segments. *Inf. Processing Letters*, 12(2), 1981.

[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.

[Kin69] James King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, September 1969.

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

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

[Mat81] Prabhaker Mateti. A decision procedure for the correctness of a class of programs. *J. ACM*, 28(2), 1981.

[McC62] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress 62*, 1962.

[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.

[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, New York, 1995.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Sys.*, 1(2):245–257, October 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal for the Association for Computing Machinery*, 27(2):356–364, April 1980.

[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.

[Ros86] L. E. Rosier. A note on Presburger arithmetic with array segments, permutation and equality. *Inf. Process. Lett.*, 22(1):33–35, 1986.

[SBDL01] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science (LICS)*, 2001.

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

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[SJ80] Norihisa Suzuki and David Jefferson. Verification decidability of Presburger array programs. *J. ACM*, 27(1), 1980.

[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.

[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.* Univ. of California Press, Berkeley, 2nd edition, 1951.

[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. `http://visi.colorado.edu/~vis`.

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