Instantiation-Based Invariant Discovery*

Temesghen Kahsai and Yeting Ge and Cesare Tinelli

The University of Iowa

Abstract. We present a general scheme for automated instantiation-based invariant discovery. Given a transition system, the scheme produces k-inductive invariants from templates representing decidable predicates over the system's data types. The proposed scheme relies on efficient reasoning engines such as SAT and SMT solvers, and capitalizes on their ability to quickly generate counter-models of non-invariant conjectures. We discuss in detail two practical specializations of the general scheme in which templates represent partial orders. Our experimental results show that both specializations are able to quickly produce invariants from a variety of synchronous systems which prove quite useful in proving safety properties for these systems.

1 Introduction

The automated verification of hardware or software systems benefits greatly from the specification of invariants, state properties that hold over all iterations of a program loop or over all reachable states of a transition system. Since invariants are notoriously difficult or time-consuming to specify manually, a lot of research in verification over the years has been dedicated to their automatic generation.

In much of previous work, invariants are synthesized from a system's description (formal specification or source code), using sophisticated algorithms guided by the semantics of the description language. In this paper, we propose a complementary approach based on a somewhat brute-force invariant *discovery* scheme which has proven quite effective in our experimental evaluation. The approach looks for possible invariants by sifting through a large set of automatically generated formulas. These formulas are all instances of the same template, the parameter of the scheme, representing a decidable relation over one of the system's data types.

Our approach relies on efficient reasoning engines such as SAT and SMT solvers, and capitalizes on their ability to quickly generate counter-models. For the invariant discovery scheme to be practical, they key point is to encode large sets of candidate invariants compactly and process them efficiently. One case when this is possible is when the chosen template represents a partial order, that is, a reflexive, transitive and antisymmetric relation. This paper investigate two specializations of the scheme, one for general partial order sets (posets) and one for binary posets.

Our primary intended use for the discovered invariants is to assist the automatic verification of safety properties. To illustrate the effectiveness of our approach, we developed a new tool based on it. While our invariant discovery scheme can be applied

* This work was partially supported by AFOSR grant #AF9550-09-1-0517.

to any transition system, our tool applies to programs in the synchronous data flow language Lustre [8] and generates invariants over their Boolean and integer variables. We have carried extensive experiments with a large set of Lustre programs and annotated with safety properties. Our experimental results indicate that our techniques are quite effective in practice. As we discuss later, the generated invariants considerably increase the number of provable safety properties; moreover, they do not slow down the processing of safety properties already provable without those invariants.

Related work. Automatic invariant generation has been intensively investigated since the 1970s, producing a large body of literature. Manna and Pnueli [10] provide a compendium of this research and an extensive set of references. They present a number of methods for generating invariants to prove safety properties, which have been later extended by others (e.g., [13, 2]). These methods could be classified as either *top-down* or *bottom-up*. Top-down invariant generation begins with a property to be verified for a particular system. When attempts to prove the property fail, various heuristics are applied to strengthen it. Bottom-up methods look at the system and use it to deduce properties of it. Until recently, invariants generated with these methods tended to be simple properties and not very useful. The invariant discovery scheme described in this paper could be classified as a bottom-up method. Its major distinction with respect to previous approaches is its ability to produce more complex invariants efficiently.

Counterexample guided refinement is a popular technique in model checking that has also been used for invariant generation [15, 3, 11]. Thalmaier *et al.* propose an induction-guided refinement process to approximate reachability analysis by providing inductive invariants to a SAT-based property checker [14]. Such analysis is based on BDD techniques. Another line of research on invariant generation builds on predicate abstraction techniques [6, 11]. De Moura *et al.* describe invariant strengthening techniques based on quantifier elimination. That work is one of the first to use modern SMT solvers as reasoning engines for the verification of safety properties. There is recent interest in using SMT-solvers for generating inductive loop invariants. Srivastava and Gulwani describe a technique combining templates and predicate abstraction [12].

The work by Hunt *et al.* [9] is more closely related to ours, and was in fact its main inspiration. They propose a SAT-based method to prove safety properties of circuits that uses induction to identify equivalent sub-circuits inexpensively before attempting to prove the given property. This equivalence information either implies the property directly or can be used to decrease the amount of state space traversal by the main model checking procedure. Compared with that work, our approach is more general, with respect to both the transition systems it applies to and the relations it discovers between sub-circuits.

Synopsis. In the next sub-section we give a brief description of the notions and notations that will be used throughout the paper. Section 2 presents a general scheme for invariant discovery using k-induction. Section 3 describes two specializations of the general scheme. Experimental results are reported in Section 4. Section 5 concludes with a discussion of further research.

Formal Preliminaries We denote finite tuples (or vectors) by letters in bold font. If $\boldsymbol{t}$ is an n-tuple, $\boldsymbol{t}(i)$ is the i-th element of $\boldsymbol{t}$ for $i = 1, \ldots, n$.

For generality, we consider here an arbitrary logic $\mathcal{L}$ (with classical semantics) extending propositional logic. We employ $\mathcal{L}$'s notion of variable, term, formula, free variable, model, and formula satisfiability in a model. Relevant examples of $\mathcal{L}$ are propositional logic or any of the logics used in SMT: linear arithmetic, linear arithmetic with uninterpreted function symbols, and so on. If Γ is a set of formulas in $\mathcal{L}$, a model $\mathcal{M}$ *satisfies* Γ if it satisfies every formula in it; Γ is *$\mathcal{L}$-(un)satisfiable* in $\mathcal{L}$ if some (no) model of $\mathcal{L}$ satisfies it. We define an entailment relation $\models_{\mathcal{L}}$ in $\mathcal{L}$ as usual: for any set $\Gamma \cup \{F\}$ of formulas in $\mathcal{L}$, we have that $\Gamma \models_{\mathcal{L}} F$ iff every model of $\mathcal{L}$ that satisfies Γ satisfies F as well. Two formulas F and G are *$\mathcal{L}$-equivalent* if $F \models_{\mathcal{L}} G$ and $G \models_{\mathcal{L}} F$.

If F is a formula with free variables $x_1, \ldots, x_m$, and $t_1, \ldots, t_m$ are any terms in the logic, we use $F[t_1, \ldots, t_m]$ to denote the formula obtained from F by simultaneously replacing each occurrence of x_i in F by t_i, for all $i = 1, \ldots, m$. Abusing the notation, we will write $F[x_1, \ldots, x_m]$ also to denote that F has free variables $x_1, \ldots, x_m$, and sometimes just $F[\_, \ldots, \_]$ when the name of the free variables is unimportant.

Let Q be a set of *states*, a *state space*. A *transition system $\mathcal{S}$ over Q* is a pair $(\mathcal{S}_{\mathrm{I}}, S_{\mathrm{T}})$ where $\mathcal{S}_{\mathrm{I}} \subseteq Q$ is the set of $\mathcal{S}$'s *initial states*, and $S_{\mathrm{T}} \subseteq Q \times Q$ is $\mathcal{S}$'s *transition relation*. A state $q \in Q$ is 0-*reachable* if $q \in \mathcal{S}_{\mathrm{I}}$; it is k-*reachable* with $k > 0$ if it is $(k-1)$-reachable or $(s, q) \in S_{\mathrm{T}}$ for some $(k-1)$-reachable state s. A state is *($\mathcal{S}$-)reachable* if it is k-reachable for some $k \geq 0$. We assume some encoding of the state space Q in terms of n-tuples of ground terms in $\mathcal{L}$, for some fixed n.[1] Then, we say that (the encoding of) a state $\boldsymbol{q}$ *satisfies* a formula $F[\boldsymbol{x}]$, where $\boldsymbol{x}$ is an n-tuple of distinct variables, if $F[\boldsymbol{x}]$ is satisfied by every model of $\mathcal{L}$ interpreting $\boldsymbol{x}$ as $\boldsymbol{q}$. This terminology extends to formulas over several n-tuples of free variables in the obvious way.

Let $\mathcal{S} = (\mathcal{S}_{\mathrm{I}}, S_{\mathrm{T}})$ be a transitions system. A *(state) property* is any formula $P[\boldsymbol{x}]$ over an n-tuple $\boldsymbol{x}$ of variables. It is *invariant (for $\mathcal{S}$)* if it is satisfied by all $\mathcal{S}$-reachable states. An *$\mathcal{L}$-encoding* of $\mathcal{S}$ is a pair $(I[\boldsymbol{x}], T[\boldsymbol{x}, \boldsymbol{y}])$ of formulas of $\mathcal{L}$ respectively over the n-tuples of variables $\boldsymbol{x}$ and $\boldsymbol{x}, \boldsymbol{y}$, where

- $I[\boldsymbol{x}]$ is a formula satisfied exactly by the initial states of $\mathcal{S}$;
- $T[\boldsymbol{x}, \boldsymbol{y}]$ is a formula satisfied by two reachable states $\boldsymbol{q}, \boldsymbol{q}'$ iff $(\boldsymbol{q}, \boldsymbol{q}') \in \mathcal{S}_{\mathrm{T}}$.

For any formula F over a single state and formula G over two states, we will write F_i and G_{i+1} as an abbreviation of $G[\boldsymbol{x}_i]$ and $G[\boldsymbol{x}_i, \boldsymbol{x}_{i+1}]$, respectively, where $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i+1}$ are n-tuples of distinct variables.

Definition 1. *A state property $P[\boldsymbol{x}]$ is* k-inductive (wrt T) *for some $k \geq 0$ if*

$$I_0 \wedge T_1 \wedge \cdots \wedge T_k \models_{\mathcal{L}} P_0 \wedge \cdots \wedge P_k \tag{1}$$

$$T_1 \wedge \cdots \wedge T_{k+1} \wedge P_0 \wedge \cdots \wedge P_k \models_{\mathcal{L}} P_{k+1} \tag{2}$$

A property is inductive in the usual sense if it is 0-inductive. Every property that is k-inductive for some k is invariant (but not vice versa). An invariant $P[\boldsymbol{x}]$ is *trivial* if $T_1 \models_{\mathcal{L}} P_1$. Note that this includes all properties $P[\boldsymbol{x}]$ that are *valid* in $\mathcal{L}$.

[1] Depending on $\mathcal{L}$, states may be encoded for instance as n-tuples of Boolean constants or as n-tuples of integer constants, and so on.

Require: a template formula $R[\_, \_]$ and a term set U
Ensure: P is invariant
$i := 0$
$C := \bigwedge\{R[s,t] \mid s,t \in U\}$
---------- Phase 1 ----------
repeat
  $i := i + 1$; *refined* := *FALSE*
  repeat
    $a := \mathsf{SAT}(I_0 \wedge T_1 \wedge \cdots \wedge T_{i-1} \wedge \neg C_{i-1})$
    if $a = (\boldsymbol{q}_0, \ldots, \boldsymbol{q}_{i-1})$ **then**
      $C := \mathsf{filter}(C, \boldsymbol{q}_{i-1})$; *refined* := *TRUE*
  until $a = \mathsf{unsat}$
until $\neg$*refined*
---------- Phase 2 ----------
$k := i - 1$
repeat
  $a := \mathsf{SAT}(T_1 \wedge \cdots \wedge T_k \wedge C_0 \wedge \cdots \wedge C_{k-1} \wedge \neg C_k)$
  if $a = (\boldsymbol{q}_0, \ldots, \boldsymbol{q}_k)$ **then**
    $C := \mathsf{filter}(C, \boldsymbol{q}_k)$
until $a = \mathsf{unsat}$
$P := C$
---------- Phase 3 ----------
repeat
  $a := \mathsf{SAT}(T_1 \wedge \neg C_1)$
  if $a = (\boldsymbol{q}_0, \boldsymbol{q}_1)$ **then**
    $C := \mathsf{filter}(C, \boldsymbol{q}_1)$
until $a = \mathsf{unsat}$
$P := P \setminus C$

Fig. 1. Pseudo-code for the general invariant discovery scheme. The function SAT implements the $\mathcal{L}$-solver. It takes a formula F over n states and returns either unsat or a sequence of n states that satisfies F. The function filter takes a conjunctive property P and a state $\boldsymbol{q}$ and returns the property obtained from P by removing all conjuncts that are falsified by $\boldsymbol{q}$. In the last statement, $P \setminus C$ denotes the conjunction of the conjuncts of P that do not occur in C.

2 A general scheme for invariant discovery

Given an $\mathcal{L}$-encoding $S = (I[\boldsymbol{x}], T[\boldsymbol{x}, \boldsymbol{y}])$ of a system $\mathcal{S} = (\mathcal{S}_{\mathrm{I}}, \mathcal{S}_{\mathrm{T}})$, we are interested in discovering invariants for $\mathcal{S}$ automatically. We describe here a general scheme for doing so. The scheme is parameterized by a *template formula* $R[\_, \_]$ and produces invariants for $\mathcal{S}$ that are conjunction of instances $R[s,t]$ of R where s, t are in principle arbitrary terms over a single state.[2] The scheme relies on the existence of an $\mathcal{L}$*-solver*, a decision procedure for $\mathcal{L}$-satisfiability, which for each $\mathcal{L}$-satisfiable formula $F[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$ is also able to return a state list $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$ that satisfies $F[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$.[3] The scheme also relies on a procedure that can generate from S a non-empty *instanti-*

[2] The restriction to binary templates is used here only to simplify the exposition.
[3] Modern SAT or SMT solvers are of course examples of $\mathcal{L}$-solvers for specific $\mathcal{L}$'s.

ation set U of terms over $\boldsymbol{x}$ to be used to generate the instance of R. In this setting, a naive approach would be to check every possible instance $R[s,t]$ individually for invariance. This would be highly impractical since the number of instances of R is quadratic in the size of the instantiation set U. In our approach, we check the satisfiability of all instances at the same time and rely on the model generation ability of the $\mathcal{L}$-solver to weed out several non-invariant instances at once.

The general scheme consists of a simple two-phase procedure, with an optional third phase. Given the formula $R[\_,\_]$ and the term set U, the first phase starts with the optimistic conjecture that the property

$$C[\boldsymbol{x}] = \bigwedge_{s,t \in U} R[s,t]$$

is invariant. Then, it uses the $\mathcal{L}$-solver to weaken that conjecture by eliminating from it as many conjuncts $R[s,t]$ as possible—specifically, all conjuncts falsified by a k-reachable state, for some heuristically determined k. The resulting formula C is passed to the second phase, which attempts to prove C k-inductive by establishing the entailment (2) in Definition 1. Counterexamples to (2), i.e., models that falsify the entailment, are used to weaken C further by eliminating additional conjuncts until (2) holds. The final formula—the empty conjunction in the worst case—is guaranteed to be invariant. That formula can be further processed in the optional third phase by removing from it any conjunct that is a trivial invariant. The rationale for the last phase is that trivial invariants are never needed, for being directly implied by the formula encoding the transition relation, and including them could put extra burden on the $\mathcal{L}$-solver.

The pseudo-code for the procedure sketched above is provided in Figure 1. The termination condition for Phase 1 is a heuristic one: the search for the value k stops when C is falsified by no k-reachable states. Furthermore, every conjunct of C that does not pass the test in Phase 2 is conservatively assumed not to be invariant (even if it may be k'-inductive for some $k' > k$) and removed. It is not difficult to show that both phases are terminating. The final C is invariant because, by construction, it is k-inductive for the final k.

The practical feasibility of this invariant discovery scheme depends on the possibility of representing the conjecture C compactly, i.e., by an equivalent formula using less than $O(n^2)$ space with n being the size of the instantiation set U, and refining it efficiently, i.e., in less than $O(n^2)$ time . This may not be the case in general for arbitrary template formulas $R[\_,\_]$. Hence, we focus on a class of templates for which in practice, if not in theory, these space and time costs are sub-quadratic in n: *$\mathcal{L}$-formulas denoting a partial order*. Common useful examples of partial orders include implication over the Booleans, the usual orderings over numeric domains, set inclusion over finite sets, as well as equality over any domain.

3 Partial order templates

In this section, we describe two specializations of the general invariant discovering scheme provided in Figure 1. Both specializations rely on the properties of partial orders in order to represent the conjunctive conjecture C compactly and process it efficiently. We start with one that works for any domain $\mathbb{D}$ and partial order $\preceq \subseteq \mathbb{D} \times \mathbb{D}$

provided that both the identity relation $\approx$ (i.e., equality) over $\mathbb{D}$ and the partial order $\preceq$ are expressible in a logic $\mathcal{L}$ with a decidable satisfiability problem. For example, this is the case when $\mathcal{L}$ is rational (resp., linear integer) arithmetic and $\preceq$ is $\leq$ or $\geq$ over the rational numbers (resp., the integers). Then, we discuss a further specialization for binary domains. For simplicity, in both cases we assume that $\approx$ and $\preceq$ are built-in symbols of $\mathcal{L}$. As a consequence, the template $R[\_, \_]$ will be just $\_ \preceq \_$.

Let U be again the given instantiation set, and let M be a sequence $(q_1, \ldots, q_m)$ of $m \geq 0$ states from Q. To each $t \in U$ we associate an m-vector $\boldsymbol{v}_t$ where, for $i = 1, \ldots, m$, $\boldsymbol{v}_t(i)$ is the *value* of t in state q_i, i.e., the element of $\mathbb{D}$ that t evaluates to in q_i. The state sequence M induces an equivalence relation $\equiv_M$ over the terms in U where $s \equiv_M t$ iff $\boldsymbol{v}_s = \boldsymbol{v}_t$.

Definition 2. *Let M be a state sequence. Suppose $\equiv_M$ has m equivalence classes and let $r_1, \ldots, r_m$ be their respective representatives. Let the point-wise extension of $\preceq$ to m-vectors over $\mathbb{D}$ be denoted by $\preceq$ as well.*[4] *The* strongest conjecture C_M consistent with M *is the smallest conjunction of $\approx$- and $\preceq$-atoms that satisfies the following.*

1. *For each $i = 1, \ldots, m$ and $t \in U \setminus \{r_i\}$, C_M contains $t \approx r_i$ if $t \equiv_M r_i$.*
2. *For each distinct $i, j = 1, \ldots, m$, C_M contains the atom $r_i \preceq r_j$ if $\boldsymbol{v}_{r_i} \preceq \boldsymbol{v}_{r_j}$.*

We can specialize the procedure described in Figure 1 by using the formula C_M above instead of C where M is a sequence of states produced by the $\mathcal{L}$-solver. We describe this specialization in the following. We consider just Phase 1 since the other phases are analogous.

Specializing the general scheme (Phase 1) For each iteration of the repeat loop in Phase 1 let M be the sequence of all the states generated until then (those passed to filter in Figure 1). Initially, M is the empty sequence, which means that $\equiv_M$ is $U \times U$ and so C_M has the form $t_2 \approx t_1 \wedge \cdots \wedge t_m \approx t_1$ with $\{t_1, \ldots, t_m\} = U$. Calls to filter now amount to computing the formula C_M for the most recent M. This specialization maintains the following (meta-)invariants on M: for all $s, t \in U$, (i) $s \equiv_M t$ iff none the models generated by the $\mathcal{L}$-solver so far falsifies the formula $s \approx t$, i.e., contradicts the conjecture that $s \approx t$ is invariant; (ii) $\boldsymbol{v}_s \preceq \boldsymbol{v}_t$ iff at least one of the models so far falsifies the formula $t \preceq s$ but none falsify $s \preceq t$; in other words, the evidence so far disproves the conjecture that $s \approx t$ is invariant but not that $s \preceq t$ is.

Relying on the two properties above it possible to show that, at each step of Phase 1, the formula C_M is $\mathcal{L}$-equivalent to the formula C in Figure 1. The formula C_M is more compact than C because it replaces the quadratically many $\preceq$-atoms between distinct $\equiv_M$-equivalent terms by linearly-many equality atoms between these terms and their equivalence class representative (e.g., $\{t_2 \approx t_1, t_3 \approx t_1\}$ in place of $\{t_1 \preceq t_2, t_1 \preceq t_3, t_2 \preceq t_3, t_2 \preceq t_1, t_3 \preceq t_1, t_3 \preceq t_2\}$).

An even more compact version of C_M is possible by exploiting the transitivity of $\preceq$. In concrete, this can be done by computing a minimal, or close to minimal, base for the poset $(\mathcal{V}_M, \preceq)$ where $\mathcal{V}_M = \{\boldsymbol{v}_{r_1}, \ldots, \boldsymbol{v}_{r_m}\}$ ($\boldsymbol{v}_{r_1}, \ldots, \boldsymbol{v}_{r_m}$ are again the representatives of $\equiv_M$'s classes). A *base* for the poset is a binary relation B on $\mathcal{V}_M$ whose

[4] So $(u_1, \ldots, u_n) \preceq (v_1, \ldots, v_n)$ iff $u_i \preceq v_i$ for $i = 1, \ldots, n$.

Require: $(\mathcal{V}_M, \preceq)$ is a poset,
Ensure: $\mathcal{C}$ is set of chains over $\mathcal{V}_M$, $\sigma : \mathcal{V}_M \to 2^{\mathcal{V}_M}$, and $\boldsymbol{v} \preceq \boldsymbol{v}'$ for all $\boldsymbol{v}' \in \sigma(\boldsymbol{v})$
  $\mathcal{C} := \emptyset$; $\sigma := \emptyset$
  for $\boldsymbol{v} \in \mathcal{V}_M$ **do**
    $\sigma := \sigma \cup \{\boldsymbol{v} \mapsto \emptyset\}$
    for $c \in \mathcal{C}$ **do**
      $i := \mathsf{greatestBelow}(\boldsymbol{v}, c)$
      $j := \mathsf{leastAbove}(\boldsymbol{v}, c)$
      if $j = 1$ **then**
        insert $\boldsymbol{v}$ at the beginning of c
      else
        if $i = j - 1$ **then**
          insert $\boldsymbol{v}$ at position j in c
        else
          if $i =$ the length of c **then**
            append $\boldsymbol{v}$ at the end of c
          else
            if $0 < i$ **then**
              add $\boldsymbol{v}$ to $\sigma(c(i))$
            if $0 < j$ **then**
              add $c(j)$ to $\sigma(\boldsymbol{v})$
    if $\boldsymbol{v}$ was not inserted into any chain **then**
      $\mathcal{C} := \mathcal{C} \cup \{[\boldsymbol{v}]\}$

Fig. 2. A partial order sorting procedure. The call $\mathsf{greatestBelow}(\boldsymbol{v}, c)$ returns the position in the chain c of its greatest element smaller than $\boldsymbol{v}$, if any; otherwise, it returns 0. The call $\mathsf{leastAbove}(\boldsymbol{v}, c)$ returns the position of the least element of c larger than $\boldsymbol{v}$, if any; otherwise, it returns 0. The notation $c(i)$ stands for the i-th element of c.

transitive closure B^+ coincides with $\preceq$ over $\mathcal{V}_M$.[5] A base is *minimal* if no strict subset of B is a base for the poset. Then, given a base B, Requirement 2 in the definition of C_M (Definition 2) can be relaxed to having C_M contain $r_i \preceq r_j$ only if $(\boldsymbol{v}_{r_i}, \boldsymbol{v}_{r_j}) \in B$.

Partial order sorting One way to compute a base B for the poset $(\mathcal{V}_M, \preceq)$ is to use a procedure for partial order sorting. We describe here a procedure that, while probably not as efficient in general as those in the most recent literature (see, e.g., [4]), is much simpler to describe and implement, and is explicitly geared towards posets with many incomparable elements such as those generated by our invariant discovery scheme.

A *chain* over $\mathcal{V}_M$ is a list $[\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_p]$ of members of $\mathcal{V}_M$ such as $\boldsymbol{v}_1 \preceq \boldsymbol{v}_2 \preceq \ldots \preceq \boldsymbol{v}_p$. Our sorting procedure takes the set $\mathcal{V}_M$ as input, and computes a set $\mathcal{C}$ of chains over $\mathcal{V}_M$ as well as a mapping σ from $\mathcal{V}_M$ to $2^{\mathcal{V}_M}$ such that $\boldsymbol{v} \preceq \boldsymbol{v}'$ for all $\boldsymbol{v}' \in \sigma(\boldsymbol{v})$. In essence, $\mathcal{C}$ is a selection of chains in the partial order, and for each element $\boldsymbol{v}$ in a chain, $\sigma(v)$ collects all the immediate successors of $\boldsymbol{v}$ in chains of $\mathcal{C}$ that do not contain v. The base B for the poset $(\mathcal{V}_M, \preceq)$ is obtained by collecting all pairs $(\boldsymbol{v}, \boldsymbol{v}')$ such that $\boldsymbol{v}' \in \sigma(\boldsymbol{v})$ or $\boldsymbol{v}$ and $\boldsymbol{v}'$ occur consecutively in a chain of $\mathcal{C}$.

[5] That is, for all distinct $\boldsymbol{v}, \boldsymbol{v}' \in \mathcal{V}_M$, $\boldsymbol{v} \preceq \boldsymbol{v}'$ iff $(\boldsymbol{v}, \boldsymbol{v}') \in B^+$.

The procedure, shown in Figure 2, works as follows. For each $\boldsymbol{v} \in \mathcal{V}_M$ and for each existing chain $c \in \mathcal{C}$, it inserts $\boldsymbol{v}$ into c if possible. That is the case if, with respect to $\preceq$, $\boldsymbol{v}$ is smaller than the first value of c, greater than the last, or in between two consecutive elements of c. Otherwise, if c contains elements smaller than $\boldsymbol{v}$, it adds $\boldsymbol{v}$ to the set $\sigma(\boldsymbol{v}_i)$ where $\boldsymbol{v}_i$ is the greatest of these elements; also, if c contains elements greater than $\boldsymbol{v}$, it adds $\boldsymbol{v}_j$ to the set $\sigma(\boldsymbol{v})$ where $\boldsymbol{v}_j$ is the least of these elements. If the procedure is unable to add $\boldsymbol{v}$ to any existing chain, it puts $\boldsymbol{v}$ in its own chain and adds that to $\mathcal{C}$.

Example 1. We briefly illustrate the partial order sorting procedure where $\mathbb{D}$ is the domain of the integers and $\preceq$ is the usual $\leq$ relation. Consider a sequence M with two states. Let $s, t, q, r, p \in U$ be terms, and let the associated poset $(\mathcal{V}_M, \preceq)$ be $(\{\boldsymbol{v}_s, \boldsymbol{v}_t, \boldsymbol{v}_q, \boldsymbol{v}_r, \boldsymbol{v}_p\}, \leq)$ where

$$\boldsymbol{v}_s = (6, 5), \quad \boldsymbol{v}_t = (5, 2), \quad \boldsymbol{v}_q = (5, 3), \quad \boldsymbol{v}_r = (10, 2), \quad \boldsymbol{v}_p = (2, 4)\ .$$

Initially, the chain $\mathcal{C}$ and the mapping σ are empty. The following table shows the value of $\mathcal{C}$ and σ after each main iteration of the sorting procedure.

	$\mathcal{C}$	σ
1	$[\boldsymbol{v}_s]$	$\boldsymbol{v}_s \mapsto \emptyset$
2	$[\boldsymbol{v}_t, \boldsymbol{v}_s]$	$\boldsymbol{v}_s \mapsto \emptyset$, $\boldsymbol{v}_t \mapsto \emptyset$
3	$[\boldsymbol{v}_t, \boldsymbol{v}_q, \boldsymbol{v}_s]$	$\boldsymbol{v}_s \mapsto \emptyset$, $\boldsymbol{v}_t \mapsto \emptyset$, $\boldsymbol{v}_q \mapsto \emptyset$, $\boldsymbol{v}_r \mapsto \emptyset$
4	$[\boldsymbol{v}_t, \boldsymbol{v}_q, \boldsymbol{v}_s]$, $[\boldsymbol{v}_r]$	$\boldsymbol{v}_s \mapsto \emptyset$, $\boldsymbol{v}_t \mapsto \{\boldsymbol{v}_r\}$, $\boldsymbol{v}_q \mapsto \emptyset$, $\boldsymbol{v}_r \mapsto \emptyset$
5	$[\boldsymbol{v}_t, \boldsymbol{v}_q, \boldsymbol{v}_s]$, $[\boldsymbol{v}_r]$, $[\boldsymbol{v}_p]$	$\boldsymbol{v}_s \mapsto \emptyset$, $\boldsymbol{v}_t \mapsto \{\boldsymbol{v}_r\}$, $\boldsymbol{v}_q \mapsto \emptyset$, $\boldsymbol{v}_r \mapsto \emptyset$, $\boldsymbol{v}_p \mapsto \{\boldsymbol{v}_s\}$

□

Analysis of the sorting procedure Our sorting procedure is trivially terminating because the input $\mathcal{V}_M$ is finite and the set $\mathcal{C}$ and map σ are initially empty. It is correct in the sense that the set B determined by $\mathcal{C}$ and σ is a base of $(\mathcal{V}_M, \preceq)$. It is not optimal because it may produce non-disjoint chains, giving rise to non-minimal bases; but it seemed to work fairly well during the experimental evaluation we describe in Section 4.

A coarse-grained worst-case complexity analysis shows that the procedure has time complexity $O(nwh)$, where w is the *width* of the poset $(\mathcal{V}_M, \preceq)$, the cardinality of the largest anti-chain in it, h is the *height* of the poset, the length of its longest chain, and n is the cardinality of $\mathcal{V}_M$.[6] This analysis assumes that comparing two elements of $\mathcal{V}_M$ for $\preceq$ takes constant time and that we store chains into arrays, which allows the functions greatestBelow and leastAbove in Figure 2 to be implemented by binary search. The former assumption does not generally hold because $\preceq$ is a point-wise ordering over vectors. One can make it only with a careful implementation based on the fact that the elements of $\mathcal{V}_M$ are built incrementally at each round of the invariant discovery procedure: vectors of length $k+1$ are obtained by adding a new component to vectors of length k. Since $(u_1, \ldots, u_{k+1}) \preceq (v_1, \ldots, v_{k+1})$ iff $(u_1, \ldots, u_k) \preceq (v_1, \ldots, v_k)$ and $u_{k+1} \preceq v_{k+1}$, by caching in a hash table the results of vector comparisons at round k, vector comparisons at round $k+1$ can be reduced to two constant time operations.[7]

[6] Note that $h \leq n - w + 1$, $h = n$ when $w = 1$, and $h = 1$ when $w = n$.

[7] The hash table will have quadratic size only in the worst case when a linear number of vectors are all pairwise comparable.

A recent and efficient partial sorting algorithm by Daskalakis *et al.* based on merge sort [4] has complexity $O(w^2 n \log \frac{n}{w})$, where again n is the cardinality of the poset and w its width. This complexity and that of our procedure do not easily compare in general. But we note that the posets we work with tend to have a small height, because most value vectors are incomparable. Now, with an upper bound on a poset's height, the poset's width grows proportionally with its cardinality. This makes our procedure quadratic in n and the one by Daskalakis *et al.* more than cubic.

3.1 Binary domains

When the domain $\mathbb{D}$ has cardinality 2, for example in the Boolean case, there is a better way to compute a base B for the poset $(\mathcal{V}_M, \preceq)$. Instead of a partial order sorting procedure, we can use one that represents B more directly as a directed acyclic graph (dag) G_M whose nodes are the equivalence classes of $\equiv_M$, and whose edges represent (selected) pairs in $\preceq$. More precisely, the set of edges is such that for all distinct equivalence classes S and T of $\equiv_M$ with respective representatives s and t, S and T are connected in G_M iff $\boldsymbol{v}_s \preceq \boldsymbol{v}_t$. The graph for the initial, empty state sequence is simply the graph with no edges and a single node, the whole instantiation set U.

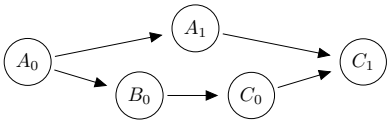
Graph generation We developed a procedure to compute the graph G_M for state sequences M, relying on the fact that each M is built incrementally, by appending a new state $\boldsymbol{q}$ to a previous sequence L. Given a sequence L and its graph G_L, and a new state $\boldsymbol{q}$, the procedure computes the graph G_M for the sequence M obtained by appending $\boldsymbol{q}$ to L. We do not describe the procedure in detail here for space constraints. Instead, we give a general intuition on how it works.

Assume for concreteness that $\mathbb{D} = \{0, 1\}$ and $0 \preceq 1$, and let X be an arbitrary node of the old graph G_L. For $i = 0, 1$, let X_i be the set consisting of all the terms in X that evaluate to i in the new state $\boldsymbol{q}$. The set X_i becomes a node of the new graph G_M iff $X_i \neq \emptyset$. In other words, G_M gets a node identical to X if all the terms of X have the same value in $\boldsymbol{q}$, and gets two new nodes, partitioning X, otherwise. Whenever both X_0 and X_1 are added to G_M, the edge $X_0 \longrightarrow X_1$ is also added. Edges between old nodes in G_L are inherited by the corresponding new nodes consistently with the ordering induced by M. In general, every edge $X_i \longrightarrow Y_j$ of G_M (where X and Y are nodes of G_L) comes from a path of length at most 2 from X to Y in G_L; moreover, $i \leq j$. The effect of the procedure is best illustrated with an example.

Example 2. Let $M = (L, \boldsymbol{q})$ and suppose G_L is the following dag.

Suppose B_1 is empty, and A_0, A_1, B_0, C_0, C_1 are all non-empty. The procedure starts by creating G_M with the following nodes and edges.

Then it adds edges derived from G_L, returning the following dag as G_M.



The edge $A_0 \longrightarrow B_0$ comes from $A \longrightarrow B$. Similarly, $B_0 \longrightarrow C_0$ comes from $B \longrightarrow C$. In contrast, $A_1 \longrightarrow C_1$ comes from the path $A \longrightarrow B \longrightarrow C$, because of the absence of B_1. □

The procedure works in three phases. In the first phase, it scans G_L's node set to generate the nodes of G_M and the edges between nodes X_0 and X_1. It also builds a map from each node of G_L to its corresponding node(s) in G_M. In the second phase, it traverses the dag G_L bottom up (from leaves to roots), to determine for each of its nodes which nodes of G_M should inherit the node's incoming edges, and how. A marking mechanism is used to visit each node of G_L only once. In the third phase, it scans G_L's edge set to generate the corresponding edges in G_M.

Analysis of the graph generation procedure From the above high-level description of the procedure it is perhaps already clear that its time complexity is linear in the number of nodes and edges of G_L. The linearity, however, comes at the cost of sub-optimality. Since in the second phase each node of G_L is visited only once, it is possible for G_M to end up containing *redundant edges*, edges connecting directly two nodes also connected by a longer path. Redundant edges lead to non-minimal bases for the associated poset because the inequations they generate are implied by other inequations in the base.

For example, the edge $A \longrightarrow C$ is redundant if $A \longrightarrow B$ and $B \longrightarrow C$ are also in G_M. By the transitivity of $\preceq$, the inequation $r_A \preceq r_B$ between the A's and C's representatives is then superfluous. In our implementation, discussed next, redundant edges are removed in an optional post-processing step on the final dag.

4 Experimental Evaluation

To evaluate experimentally the specialized invariant discovery procedures described in the previous section we implemented two instances of the general invariant discovery scheme: one for the domain of linear integer arithmetic, with $\leq$ as the partial order,[8] and one for the Boolean domain, with implication as the partial order. The instances are implemented in a new tool, called KIND-INV [9], built with components of the KIND model checker [7]. Kind was developed to check safety properties of Lustre programs. Lustre [8] is a synchronous data-flow language with infinite streams of values of three basic types: bool, int, and real. It is typically used to model circuits at a high level or control software in embedded devices.

[8] This instance works with rational numbers as well, but we ignore that here for simplicity.

[9] System and experimental data can be found at http://clc.cs.uiowa.edu/Kind/.

KIND is a k-induction-based model checker for programs in an idealized version of Lustre that uses (mathematical) integers in place of machine integer values, and rational numbers in place of floating values. The underlying logic of KIND, and of KIND-INV, is a quantifier-free logic that includes both propositional logic and linear arithmetic. We'll refer to it as $\mathcal{IL}$ (for Idealized Lustre logic) here. Lustre programs can be readily encoded in $\mathcal{IL}$ as transition systems of the sort we use here (see [7] for more details). The SMT solvers CVC3 [1] and Yices [5] are used, in alternative, as satisfiability solvers for this logic. A Lustre program can be structured as a set of modules called *nodes* which can be understood as macros. KIND-INV currently takes a single-node Lustre program as input. A multi-node program can be treated by expanding it in advance to a behaviorally equivalent single-node one. The invariants discovered by KIND-INV are then added to the Lustre input program as "assertions." Contrary to other languages, such as C, assertions in Lustre are expressions of type bool that are *assumed* to be true at each execution step of the program.

KIND-INV accepts two options for generating invariants: **`bool`** and **`int`**. The first option produces invariants of the form $s \rightarrow t$ or $s = t$ where s and t are Lustre Boolean terms. The second produces invariants of the form $s \leq t$ or $s = t$ where s and t are integer terms. The instantiation set U currently consists of heuristically selected terms from the input Lustre program plus some distinguished constant terms such as true and false. Note that bool terms may contain int terms, as in $(\mathsf{x} + \mathsf{y} > 0)$ or done, and vice versa, as in $\mathsf{x} + (\mathsf{if\ y} > 0\ \mathsf{then\ y\ else\ 1})$.

KIND-INV provides three binary options affecting invariant generation. The first two work only with the **`bool`** invariant option, the last one with both options:

`No_Ands :` When this flag is turned on, KIND-INV will not consider candidate terms of the form $s \wedge t$. The rationale behind this flag is that, conjunctive terms lead to many trivial invariants, for instance, those of the form $(s \wedge t) \rightarrow s$. Having too many of these unnecessary invariants can be burdensome for the SMT-solver, limiting the effectiveness of the non-trivial invariants in the generated assertion.

`No_Redundant_Edges :` When this flag is on, KIND-INV will remove redundant edges from the final dag storing the computed poset (see Section 3.1).

`No_Trivial_Invariants :` This flags governs whether the third phase of the invariant discovery procedure is performed or not. Its rationale is that the third phase is expensive and may not be worthwhile.

Evaluation setup To evaluate KIND-INV, we used a benchmark set derived from the one used in [7], which consists of a variety of benchmarks from several sources. Each benchmark in the original set is a Lustre program together with a single property to check, expressed as a Lustre bool term. Our derived set discards some duplicate benchmarks—included in the original set by mistake—and converts each program to a single-node one using the *pollux* tool from the Lustre 4 distribution.

Let us call a benchmark *valid* if its safety property holds for the associated program, and *invalid* otherwise. KIND is able to prove 438 of the 941 benchmarks in our set invalid by returning a (independently verified) counter-example trace for the program. KIND reports 309 of the remaining benchmarks as valid, and diverges on the remaining 194 benchmarks, even with very large timeout values. We conjecture that those 194

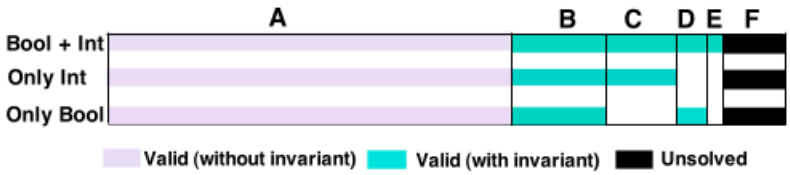


Fig. 3. Distribution of solved and unsolved benchmarks for three classes of invariants. Green bars indicate the percentage of benchmarks solvable only with invariants. All bars are drawn to scale.

unsolved benchmarks are all valid but contain a property that is either k-inductive for an extremely large k or, more plausibly, not k-inductive for any k.

For the experiments described here the benchmark set consists of the valid and the unsolved benchmarks, 503 in total. Our main goal was to evaluate how effective the invariants generated by KIND-INV are at improving KIND's *precision*, measured as the percentage of solved benchmarks. The experiments were run on a small dedicated cluster of identical machines with a 3.0 GHz Intel Pentium 4 cpu, 1GB of memory and Redhat Enterprise Linux 4.0. Version 1.0.9 of the Yices solver was used both for KIND-INV and KIND.

In a first step, we ran KIND-INV on the benchmark set twice, once for the bool and once with the **`int`** invariant generation option. For each of the benchmarks where KIND-INV did not time out, we obtained a set of invariants, and added them to the benchmark as a single conjunctive assertion. The added assertion was the constant `true` when KIND-INV timed out or ended up discarding all conjectures from the initial set. To make sure that the added assertions were indeed invariants, we verified each of them independently by formulating it as a safety property and asking KIND to prove it.[10] In a second step, we ran KIND in *inductive mode* on each benchmark, with and without the assertion that collects the discovered invariants. In that mode, KIND attempts to prove the benchmark's property by k-induction, using any assertion in the program to strengthen the k-induction hypothesis with the invariant in the assertion. The timeout for KIND-INV was set to 300 seconds and that for KIND to 120 seconds.

We did an extensive evaluation over our benchmarks with various configurations. By and large, all configurations are comparable in terms of the precision achieved by KIND when using their generated invariants. The only significant differences are with respect to invariant generation speed. A statistical analysis of the results obtained with the various configurations, not reported here, indicated that the following configuration is superior to the others: **`No_Ands = on`**, **`No_Redundant_Edges = on`**, and **`No_Trivial_Invariants = off`**. Hence, we report our results just for that configuration.

Precision results The size of the generated invariants, measured as their number of conjuncts, varies from 0 to 1150, with a median value of 133. With the **`bool`** option, KIND-INV times out in 19 cases (out of 503), and terminates normally but with an

[10] Since KIND-INV and KIND used the same SMT solver it is possible that we missed incorrect assertions because of a bug in the solver, but we believe this to be unlikely.

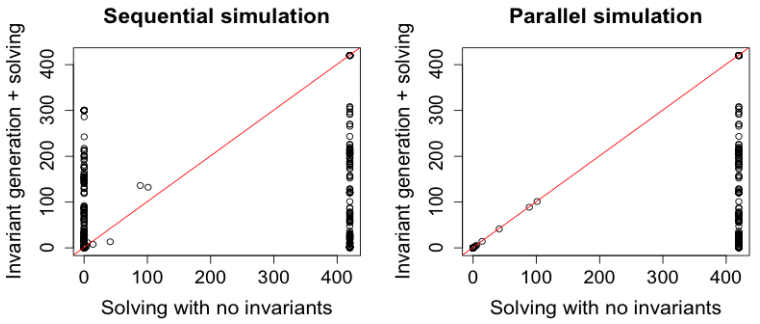


Fig. 4. Solving times without invariants versus int invariant generation times plus solving times with int invariants. In the parallel simulation, solving without invariants is attempted during invariant generation. Invariants are then used once available, and only if still needed.

empty invariant in 2 cases. With the **`int`** option, it times out in 62 cases and terminates with an empty invariant in 12 cases.

Using only bool *invariants*, i.e., invariants generated by KIND-INV with the **`bool`** option, KIND is able to prove 40% of the 194 previously unsolved benchmarks; using int *invariants*, invariants generated with the **`int`** option, it proves 53% of the unsolved benchmarks; using both bool and int invariants, it proves 63% of the unsolved benchmarks. In the three cases above, Kind's precision over all 513 benchmarks grows from 61% (without invariants) to 77%, 82%, and 85%, respectively. For all the newly solvable benchmarks the properties goes from (most likely) not k-inductive for any k to k-inductive with some $k \leq 16$. The set of new benchmarks solved with bool invariants and that solved with int invariants have a large overlap, which we find somewhat surprising. Less surprising is that using bool and int invariants together allows KIND to solve all the benchmarks solvable with either type alone, and more.

The addition of invariants preserves the set of benchmarks proved valid by KIND without them. Furthermore, it often shortcuts the k-induction process. In fact, without invariants, 14.5% of the previously valid benchmark have a safety property that is k-inductive for some $k > 1$; that percentage goes down respectively to 6.7%, 8.7% and 3.8%, with only bool, only int and both bool and int invariants.

Figure 4 summarizes graphically the various effects achieved with bool and int invariants, alone and in combination. For each of these three cases, column A represents benchmarks solvable by KIND without invariants; columns B to E represent benchmarks solvable with the generated invariants; column F represents benchmarks that remain unsolved, either because KIND-INV was not able to generate an invariant for them or because the generated invariant is not helpful. Columns C and D represent the benchmarks solved only with int and only with bool invariants, respectively. Column E represents the benchmarks solved only with both bool and int invariants together.

Runtime results Adding invariants to previously solvable benchmarks systematically makes them slightly faster to solve. The total time to solve them decrease from 305.7 to 246.5 seconds. Individual solving times in the presence of invariants are very small;

on average just 0.95s for all solvable benchmarks. In addition to the substantial increase in precision, this provides further evidence that our invariant discovery procedure produces high quality invariants. Invariant generation has of course its own, non-insignificant cost. Over the whole benchmark set, KIND-INV runtimes vary from less than a second to hundreds of seconds, to timing out at 300s. However, their median value is fairly small: 22.4s for int invariants and just 6.3s for bool ones. For the great majority of benchmarks (84%) bool invariant generation takes less than a minute per benchmark.

Evaluating invariant generation costs against the increase in precision is a difficult task because it also depends on the relative importance of precision versus prompt response. A supporting argument is that invariant generation and k-induction model checking can be done in parallel—with invariants fed to the k-induction loop as soon as they are generated—mitigating this way the cost of invariant generation. Developing a parallel model checker integrating KIND and KIND-INV was beyond the scope of this work. An approximate analysis, however, can be provided with a rough conceptual simulation of such a concurrent system.

Since the synchronization overhead in the parallel model checker would be arguably very small, we can ignore it here for simplicity. Then we can imagine the parallel checker's runtimes to be, for each benchmark, the minimum between the following two values: (i) the time KIND takes to prove the property without invariants and (ii) the sum of the times KIND-INV takes to output an invariant and KIND takes to prove the property using that invariant. The scatter plots in Figure 4 illustrate this comparison with int invariants—the results are similar for bool invariants. The first plot compares for each benchmark the runtime of KIND with no invariants and a 420s timeout[11] against the runtime of a hypothetical sequential checker that uses KIND-INV with a timeout of 120s, to add an invariant to the program, and then calls KIND with a timeout of 300s. The considerable invariant generation time penalty paid by the sequential checker (illustrated by all the points above the diagonal lines in the first plot) essentially disappears with the parallel checker, as shown in the second plot.

5 Conclusion and Future Work

We presented a novel scheme for discovering invariants in transition systems. The scheme is parametrized by a formula template representing a decidable relation over the system's datatypes, and by a set of terms used to instantiate the template. Its main features are that it checks all template instances for invariance at the same time and makes heavy use of a satisfiability solver for the logic in which the system and the instances are encoded. We described two specializations of the scheme to templates representing partial orders where we can exploit the properties of posets to achieve space and time efficiencies. Initial experimental results are very encouraging in terms of the speed of invariant generation and the effectiveness of the generated invariants in automating the verification of safety properties.

In the implementation discussed in the previous section, invariant generation is done off-line. We are developing a parallel model checking architecture and implementation

[11] Increasing the timeout from 300s to 420s does not change the set of solved benchmarks.

in which k-induction and invariant generation are done concurrently, with invariants fed to the k-induction loop as soon as they are produced.

Our invariant discovery scheme lumps together, in a single invariant produced at the end, instances of the template that may be k-inductive for different values of k. We believe that the effectiveness of the parallel model checking architecture would increase if invariant instances were identified and output progressively—with k-inductive instances produced before $(k+1)$-inductive ones. We are working on a new version of the scheme based on this idea.

We are also investigating techniques for compositional reasoning with synchronous systems based on the invariant discovery method presented in this paper. The main idea is to generate invariants separately for each module of a multi-module system, and then use them to aid the verification of properties of the entire system.

References

1. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *CAV'07*, volume 4590 of *LNCS*, 2007.
2. S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Form. Methods Syst. Des.*, 15(1):75–92, 1999.
3. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD '02*, pages 19–32. Springer-Verlag, 2002.
4. C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin. Sorting and selection in posets. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 392–401, 2009.
5. B. Dutertre and L. de Moura. The YICES SMT solver. Technical report, SRI International, 2006.
6. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI '09*, pages 120–135, Berlin, Heidelberg, 2009.
7. G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with SMT-based techniques. In *FMCAD '08*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.
8. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
9. W. Hunt, S. Johnson, P. Bjesse, and K. Claessen. *SAT-Based Verification without State Space Traversal*, volume 1954, pages 409–426. Springer Berlin / Heidelberg, 2000.
10. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
11. S. Pandav, K. Slind, and G. Gopalakrishnan. Counterexample guided invariant discovery for parameterized cache coherence protocol verification. In *CHARME 2005*, pages 317–331, 2005. LNCS 2144.
12. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. *SIGPLAN Not.*, 44:223–234, 2009.
13. J. X. Su, D. L. Dill, and C. W. Barrett. Automatic generation of invariants in processor verification. In *In FMCAD '96, volume 1166 of LNCS*, pages 377–388, 1996.
14. M. Thalmaier, M. D. Nguyen, M. Wedler, D. Stoffel, J. Bormann, and W. Kunz. Analyzing k-step induction to compute invariants for SAT-based property checking. In *DAC '10*, pages 176–181. ACM, 2010.
15. A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In *TACAS '01*, pages 113–127. Springer-Verlag, 2001.