Invariant Synthesis for Combined Theories*

Dirk Beyer[1], Thomas A. Henzinger[2],
Rupak Majumdar[3], and Andrey Rybalchenko[2,4]

[1] Simon Fraser University, Surrey, B.C., Canada
[2] EPFL, Lausanne, Switzerland
[3] University of California, Los Angeles, U.S.A.
[4] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. We present a constraint-based algorithm for the synthesis of invariants expressed in the combined theory of linear arithmetic and uninterpreted function symbols. Given a set of programmer-specified invariant templates, our algorithm reduces the invariant synthesis problem to a sequence of arithmetic constraint satisfaction queries. Since the combination of linear arithmetic and uninterpreted functions is a widely applied predicate domain for program verification, our algorithm provides a powerful tool to statically and automatically reason about program correctness. The algorithm can also be used for the synthesis of invariants over arrays and set data structures, because satisfiability questions for the theories of sets and arrays can be reduced to the theory of linear arithmetic with uninterpreted functions. We have implemented our algorithm and used it to find invariants for a low-level memory allocator written in C.

1 Introduction

The classical approach to the verification of temporal safety properties of programs requires the construction of *inductive invariants* [9, 16] at each program point, that is, assertions that are true on every program execution reaching that point, and moreover, that are closed under the strongest postcondition operator. Automation of this construction is the main challenge in program verification.

One promising approach for automated invariant computation is *template-based*, where the user specifies a parameterized form of the invariant, and a constraint-based analysis generates relationships on the parameters such that every instantiation of the parameters satisfying the relationships guarantees that the resulting assertions are indeed inductive invariants. This approach has been successfully applied to numerical invariants [5,6,13,19–21], using constraint solving in linear or nonlinear arithmetic. Unlike dataflow analysis techniques, which achieve low running time and convergence at the cost of lost precision (e.g., by widening [7]), the template-based techniques are sound and complete modulo the

* This research was sponsored in part by the grants NSF-CCF-0427202 and NSF-CCF-0546170.

templates used: if there is an inductive invariant expressible using the template, then the methods guarantee to synthesize such an invariant.

Unfortunately, the application of these techniques have been confined so far to numerical domains, where linear-programming based techniques, or decision procedures for the theories of rationals/reals, provide natural constraint solvers. In practice, program verification uses more general predicate domains, for example, combinations of linear arithmetic and equality with uninterpreted functions [1, 8, 10, 11, 18]. Uninterpreted functions are especially useful for modeling memory (for example, dereference operations and field accesses can be modeled as uninterpreted functions).

We present a constraint-based invariant synthesis algorithm for the combined domain of linear arithmetic and uninterpreted functions. Given invariant templates in the language of parameterized linear arithmetic and uninterpreted functions, our algorithm instantiates the parameters such that the resulting assertions are inductive invariants. Moreover, if such an instantiation exists, then the algorithm will find it. The key technical idea of our approach is *hierarchic theory combination* [23], whereby the uninterpreted function terms are compiled away to produce arithmetic constraints. The compilation instantiates "enough" functionality axioms to ensure that functions produce equal outputs for equal inputs. In the worst case, a factorial number of constraint-satisfaction problems in linear arithmetic with parametric coefficients needs to be solved.

Our technique enables us to construct invariants for programs that manipulate pointers. Furthermore, using recent results that reduce theories of data structures such as arrays and sets to the combined theory of linear arithmetic and uninterpreted functions [2, 14], we obtain an invariant-generation technique for templates that involve arrays and set data structures.

We have implemented our algorithm for invariant synthesis and applied it to generate invariants for a simplified low-level memory allocator used in an OS kernel. Our tools infer invariants that contain both arithmetic operations and memory operations (address-of and pointer dereferencing), which are approximated by uninterpreted function symbols. Heuristics for automatically searching through the space of candidate templates are left for future work.

2 Example

We illustrate our approach with the small example shown in Fig. 1. We want to prove the assertion at the end of the while loop. One way to prove an assertion ϕ in a program is to find an *inductive assertion map*, that is, a function η that maps every program location to a set of states such that (I0) the initial location of the program is marked true, (I1) for each edge $\ell \to \ell'$ of the control flow graph marked with operation op, we have $\mathsf{SP}(\eta.\ell, \mathsf{op}) \models_{\mathrm{LI+UIF}} \eta.\ell'$, and (I2) $\eta.\ell_\phi \models_{\mathrm{LI+UIF}} \phi$ for the location ℓ_ϕ of the assertion. Here $\models_{\mathrm{LI+UIF}}$ denotes the implication in the theory used to write invariants and program statements, which is linear arithmetic combined with uninterpreted function symbols in our case.

```
main() {
  int d1, d2;

  d1 = 3;
  d2 = f(4);

  while ( nondet() ) {
    d1 = f(d1+1);
    d2 = f(d2+1);
  }

  assert( d2=f(d1+1) );
}
```

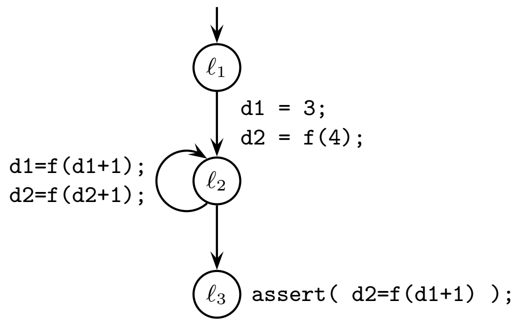

Fig. 1. Example program [10] and its control-flow graph. The invariant to prove is asserted at the location ℓ_3

SP denotes the strongest postcondition operation. For ease of exposition, we shall concentrate on the loop invariant at the location ℓ_2 in the example.

We shall use a template-based technique to infer inductive invariant maps, that is, the user provides a parameterized expression denoting the shape of the invariant, and our inference technique finds instantiations of the parameters that result in an inductive invariant map. For the example, we assume that the template ψ for the loop invariant is

$$\psi : \quad c_{\mathtt{d1}}\mathtt{d1} + c_{\mathtt{d2}}\mathtt{d2} + c_{\mathtt{f}} f(c_{\mathtt{fd1}}\mathtt{d1} + c_{\mathtt{fd2}}\mathtt{d2} + c_{\mathtt{fd}}) = c_{\mathtt{d}},$$

where c_* are parameters to be instantiated. This fixes the form of the invariant to a linear equality between a constant and a linear term where the function f occurs once with a linear argument. We use ψ' to denote the primed version of the template, where the program variables $\mathtt{d1}$ and $\mathtt{d2}$ are replaced by their primed versions $\mathtt{d1}'$ and $\mathtt{d2}'$. The primed variables denote the updated values of variables.

Given a template, our algorithm generates a set of constraints between the parameters that must be satisfied for any parameter instantiation to be an invariant. Condition (I1) requires that the template is true when the loop is entered for the first time

$$\mathtt{d1}' = 3 \wedge \mathtt{d2}' = f(4) \models_{\mathrm{LI+UIF}} \psi', \tag{1}$$

and that it must be preserved under the loop iteration

$$\psi \wedge \mathtt{d1}' = f(\mathtt{d1}+1) \wedge \mathtt{d2}' = f(\mathtt{d2}+1) \models_{\mathrm{LI+UIF}} \psi'. \tag{2}$$

Condition (I2) requires that the invariant implies the desired assertion

$$\psi \models_{\mathrm{LI+UIF}} \mathtt{d2} = f(\mathtt{d1}+1). \tag{3}$$

We now translate each implication into a constraint over the template parameters. This translation is the crucial part of our algorithm, and it computes a

Fresh	Definition
$\mathtt{v}$	$f(c_{\mathtt{fd1}}\mathtt{d1} + c_{\mathtt{fd2}}\mathtt{d2} + c_{\mathtt{fd}})$
$\mathtt{w}$	$f(c_{\mathtt{fd1}}\mathtt{d1}' + c_{\mathtt{fd2}}\mathtt{d2}' + c_{\mathtt{fd}})$
$\mathtt{x}$	$f(\mathtt{d1} + 1)$
$\mathtt{y}$	$f(\mathtt{d2} + 1)$
$\mathtt{z}$	$f(4)$

Table 1. Purified terms and corresponding definitions for the program in Fig. 1

constraint system that has a solution if and only if there exist a valuation of the parameters that yields a desired inductive invariant.

The first step in the translation process is *purification*, which introduces fresh variables for non-arithmetic subterms and stores their definitions. We purify the template and the assertions that describe the program, which produces linear arithmetic assertions together with a set of definitions for the fresh variables. For our template, purification produces the new template

$$c_{\mathtt{d1}}\mathtt{d1} + c_{\mathtt{d2}}\mathtt{d2} + c_{\mathtt{f}}\mathtt{v} = c_{\mathtt{d}},$$

where $\mathtt{v}$ is a new variable whose definition is

$$\mathtt{v} = f(c_{\mathtt{fd1}}\mathtt{d1} + c_{\mathtt{fd2}}\mathtt{d2} + c_{\mathtt{fd}}).$$

We show the definition of fresh variables that are created by purification in Table 1.

Now, each implication (1), (2), and (3) is passed to a function CONSEC, which translates the implications into constraints in linear arithmetic. We informally describe how CONSEC transforms the implication (2). The other cases are similar.

We observe that reasoning about implication (2) requires handling of functionality axioms, that is, the proof of the implication may rely on the fact that the function f produces the same outputs for the same inputs. Since some assertions in (2) are parameterized, we do not know *a priori* which instances of the functionality axioms may appear in the proof. The function CONSEC finds such instances automatically, which are in this case

$$\begin{gathered}\text{if } \mathtt{d1} + 1 = c_{\mathtt{fd1}}\mathtt{d1} + c_{\mathtt{fd2}}\mathtt{d2} + c_{\mathtt{fd}} \text{ then } \mathtt{x} = \mathtt{v},\\ \text{if } \mathtt{d2} + 1 = c_{\mathtt{fd1}}\mathtt{d1}' + c_{\mathtt{fd2}}\mathtt{d2}' + c_{\mathtt{fd}} \text{ then } \mathtt{y} = \mathtt{w}.\end{gathered}$$

Given these axiom instances, we can focus on the following version of the implication (2) that now holds in linear arithmetic:

$$\begin{aligned}&c_{\mathtt{d1}}\mathtt{d1} + c_{\mathtt{d2}}\mathtt{d2} + c_{\mathtt{f}}\mathtt{v} = c_{\mathtt{d}} \ \wedge\ \mathtt{d1}' = \mathtt{x} \ \wedge\ \mathtt{d2}' = \mathtt{y} \ \wedge\ \mathtt{x} = \mathtt{v} \ \wedge\ \mathtt{y} = \mathtt{w}\\ &\models_{\mathrm{LI}}\\ &c_{\mathtt{d1}}\mathtt{d1}' + c_{\mathtt{d2}}\mathtt{d2}' + c_{\mathtt{f}}\mathtt{w} = c_{\mathtt{d}}.\end{aligned} \tag{4}$$

Note that (4) is equivalent to (2) under the assumption that the latter is provable by using the above axiom instances, but (4) does not require any reasoning about uninterpreted function symbols. CONSEC translates (4) by applying

Farkas' lemma of *linear programming*, which states that (4) holds if the consequent of the implication can be obtained from the antecedents by taking a linear combination thereof.

Additionally, CONSEC needs to justify that its choice of the above axiom instances is valid. Let φ denote the (purified) left-hand side of (4) without the conjuncts $\mathtt{x} = \mathtt{v}$ and $\mathtt{y} = \mathtt{w}$ that arise from the axiom instances, that is, $\varphi \equiv c_{\mathtt{d1}}\mathtt{d1} + c_{\mathtt{d2}}\mathtt{d2} + c_{\mathtt{f}}\mathtt{v} = c_{\mathtt{d}} \ \wedge \ \mathtt{d1}' = \mathtt{x} \ \wedge \ \mathtt{d2}' = \mathtt{y}$. To justify the choice of the above axiom instances, CONSEC includes constraints that the premise of the first instance follows from φ, and that the premise of the second instance follows from φ conjoined with $\mathtt{x} = \mathtt{v}$:

$$\begin{aligned} \varphi \ &\models_{\mathrm{LI}} \ \mathtt{d1} + 1 = c_{\mathtt{fd1}}\mathtt{d1} + c_{\mathtt{fd2}}\mathtt{d2} + c_{\mathtt{fd}} \\ \varphi \ \wedge \ \mathtt{x} = \mathtt{v} \ &\models_{\mathrm{LI}} \ \mathtt{d2} + 1 = c_{\mathtt{fd1}}\mathtt{d1}' + c_{\mathtt{fd2}}\mathtt{d2}' + c_{\mathtt{fd}}. \end{aligned} \tag{5}$$

The justification of both facts translates to arithmetic constraints on template coefficients, again by applying Farkas' lemma.

We solve the conjunction of the constraints that encode the validity of implications (4) and (5) together with the constraints for similar implications obtained by translating (1) and (3) into linear arithmetic under particular choice of axiom instances. The resulting parameter instantiation below defines the loop invariant $\mathtt{d2} - f(\mathtt{d1} + 1) = 0$.

$c_{\mathtt{d1}}$	$c_{\mathtt{d2}}$	$c_{\mathtt{f}}$	$c_{\mathtt{fd1}}$	$c_{\mathtt{fd2}}$	$c_{\mathtt{fd}}$	$c_{\mathtt{d}}$
0	1	−1	1	0	1	0

3 Preliminaries

Constraints. Let x be a set of variables, and let a *state* be a valuation of the variables from x. We shall represent sets of states using (quantifier-free) first order formulas with free variables from x.

A *signature* $\Sigma = (F, P)$ for a first order theory consists of a set of function symbols F and a set of predicate symbols P. We assume that the arity of function and predicate symbols are encoded in their names. A constant is a function of arity zero. For a signature $\Sigma = (F, P)$, the set of Σ-terms over x is the smallest set such that (1) each free variable is a Σ-term, (2) each constant symbol $u \in F$ is a Σ-term, and (3) $f(t_1, \ldots, t_n)$ is a Σ-term, given $f \in F$ is a function symbol of arity n, and each t_i is a Σ-term, for $i = 1, \ldots, n$. The set of Σ-atoms is the smallest set such that (1) $s = t$ is a Σ-atom if s and t are Σ-terms, and (2) $p(t_1, \ldots, t_n)$ is a Σ-atom for a predicate symbol $p \in P$ of arity n and each t_i is a Σ-term, for $i = 1, \ldots, n$. The set of Σ-constraints is the smallest set such that each Σ-atom is a Σ-constraint, and $\neg\varphi$ and $\varphi \wedge \psi$ are Σ-constraints whenever φ and ψ are Σ-constraints. Finally, the set of Σ-formulas is the smallest set containing the Σ-atoms that is closed under conjunction, disjunction, and negation. Semantics of formulas is given using Σ-models in the usual way [3].

In this paper, we assume a constraint language of linear arithmetic and uninterpreted functions. That is, in addition to the usual arithmetic operations, we assume that the language has a set of *uninterpreted function symbols* that can be used as primitive operations. Formally, our signature consists of the constant c for each $c \in \mathbb{Q}$, the functions $+$ and $-$, together with a set of uninterpreted function symbols, and the predicate $\leq$.

A Σ-theory is a set of Σ-formulas that is closed under logical consequences. The *satisfiability problem* for a Σ-theory T asks, given a Σ-formula φ, whether some model of the theory T satisfies φ. The theory of *linear arithmetic* (LI) is the theory of the structure of the rationals $\langle \mathbb{Q}, 0, 1, +, \leq \rangle$. The theory of *equality with uninterpreted functions* (UIF) is the theory of equality together with the axiom

$$\forall c_1, \dots, c_n, d_1, \dots, d_n : \bigwedge_{i=1}^{n} c_i = d_i \;\rightarrow\; f(c_1, \dots, c_n) = f(d_1, \dots, d_n),$$

for each uninterpreted function symbol f of arity n. We refer to the right-hand side of the above implication as the *head* of the axiom. Given two terms $f(c_1, \dots, c_n)$ and $f(d_1, \dots, d_n)$ we write $c \approx d$ to denote the premise $\bigwedge_{i=1}^{n} c_i = d_i$ of the corresponding axiom. We write $\models_{\mathrm{LI}}$ and $\models_{\mathrm{LI+UIF}}$ to denote implication in the theory of linear arithmetic and in its combination with uninterpreted function symbols, respectively.

In the following, we work in the combined theory of linear arithmetic and equality with uninterpreted functions, denoted LI+UIF. We reason about LI+UIF using the hierarchic approach [23]. This approach allows one to reduce the reasoning about certain combinations of base and extension theories to the reasoning in the base theory. The reduction is performed by introducing instantiations of the axioms of the extension theory to the base theory. In particular, the combination of linear arithmetic and uninterpreted function symbols admits hierarchic combination [23].

Theorem 1. *[18, 23] The satisfiability problem for LI+UIF is decidable.*

Control Flow Graphs. We assume an abstract representation of programs by transition systems [16]. A *program* $\mathsf{P} = (x, \mathsf{locs}, \ell_0, \mathcal{T}, \mathsf{Good})$ consists of a set x of variables, a set locs of *control locations*, an initial location $\ell_0 \in \mathsf{locs}$, a set $\mathcal{T}$ of *transitions*, and a constraint Good over the variables from x that describes the 'good' states. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ') where $\ell, \ell' \in \mathsf{locs}$ are control flow locations, and ρ is a constraint over free variables from $x \cup x'$, where the variables from x' denote the values of the variables from x in the next state.

A *state* of the program P is a valuation of the variables from x. The set of all states is written $\mathsf{val}.x$. We shall represent sets of states using constraints. A *computation* of the program P is a sequence $\langle m_0, s_0 \rangle \langle m_1, s_1 \rangle \dots \langle m_k, s_k \rangle \in (\mathsf{locs} \times \mathsf{val}.x)^*$ where $m_0 = \ell_0$ is the initial location and for each $i \in \{0, \dots, k-1\}$, there is a transition $(m_i, \rho, m_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models_{\mathrm{LI+UIF}} \rho$. A state s is reachable at ℓ if $\langle \ell, s \rangle$ appears in some computation. A program is *unsafe* if some state $s \notin \mathsf{Good}$ is reachable.

Invariants. An *invariant* at a location $\ell \in \mathsf{locs}$ of P is a set of states containing the states reachable at ℓ. An *invariant map* is a mapping η from locs to LI+UIF constraints such that the following conditions hold:

(I0: Initiation) for the entry location ℓ_0, we have $\eta.\ell_0 = \mathit{true}$.
(I1: Inductiveness) for each $\ell, \ell' \in \mathsf{locs}$ such that $(\ell, \rho, \ell') \in \mathcal{T}$, we have $\eta.\ell \wedge \rho \models_{\mathrm{LI+UIF}} \eta.\ell'$. Here, $\eta.\ell'$ is the constraint obtained by substituting variables from x' for the variables from x in $\eta.\ell$.
(I2: Safety) for each $\ell \in \mathsf{locs}$ we have $\eta.\ell \models_{\mathrm{LI+UIF}} \mathsf{Good}$.

The *invariant synthesis* problem is to construct an invariant map for a given program. For ease of exposition, we assume that an invariant map assigns an invariant to each program location. For efficiency, one can require invariants to be defined only over a program *cutset*, i.e., a set of program locations such that every syntactic cycle in the control flow graph passes through some location in the cutset.

4 Invariant Synthesis for LI+UIF

We now describe our algorithm for invariant synthesis for linear arithmetic and uninterpreted function symbols. Our algorithm follows the constraint-based approach [6, 13, 19–21], which has already provided successful algorithms for the synthesis of linear and non-linear invariants and ranking functions. Our algorithm extends the applicability of invariant generation to the combination of linear arithmetic and uninterpreted function symbols. First, we briefly describe the constraint-based approach, and outline our method of handling uninterpreted function symbols. Then, we provide a formal description of our algorithm.

Constraint-based Invariant Synthesis. The constraint-based approach to invariant generation reduces the computation of an invariant to a constraint solving problem. The approach consists of three steps. First, a *template* assertion that represents an invariant is fixed in an *a priori* chosen language. The parameters in the template are the unknown coefficients that determine the invariant. Second, a set of *constraints* over these parameters is defined which encodes the definition of the invariant. This means that every solution to the constraint system yields an inductive invariant. Third, an *invariant* is obtained by solving the resulting constraint system.

4.1 Invariant Templates

An *invariant template* is an *a priori* fixed parameterized assertion over the program variables. It identifies the unknown parameters, and restricts the "dimensions" of the invariant, e.g., the number of conjuncts and the number of function applications. This means that the form of the template determines the form of the resulting invariant.

We provide the formal definition of the invariant template for a given set of program variables and functions symbols. Let c range over the set of integer constants, v over the set of program variables, f over a fixed set of uninterpreted functions symbols, and α over a fixed set of template parameters. The following grammar defines the set of constraint templates:

$$\begin{array}{ll} \text{Terms} & t ::= v \mid f(e_1, \ldots, e_n) \\ \text{Expressions} & e ::= c \mid c \times t \mid \alpha \times t \mid e_1 + e_2 \mid e_1 - e_2 \\ \text{Constraints} & i ::= e \leq c \mid e_1 = e_2 \\ \text{Templates} & \xi ::= i \mid i \wedge \xi \end{array}$$

An invariant template is a finite conjunction of inequalities. An invariant is *expressible* by the invariant template if there exists a valuation of the template parameters that yields the invariant. Our algorithm computes invariants that are expressible by a given template.

4.2 Algorithm

The invariant synthesis algorithm Inv(LI+UIF) takes as input a program and a template map that assigns an invariant template to each program location. The algorithm computes an invariant map that assigns an invariant to each program location, if there exists an invariant that is expressible by the given invariant template. The algorithm is shown in Fig. 2. It applies an auxiliary function Consec shown in Fig. 3. The function Consec generates constraints between the parameters in the invariant template that ensure the conditions **(I0)**, **(I1)**, and **(I2)**. (We assume that the template map assigns *true* to the initial location ℓ_0, thus **(I0)** is satisfied.) Any satisfying assignment to these constraints gives an instantiation of the invariant template that is an invariant. Next, we describe each step of the algorithm.

Purification. We first purify all (sub-) terms that appear in the invariant templates, and in the program representation. A formula (or constraint) is *purified* if the only atom with an uninterpreted function is of the form $x = f(t_1, \ldots, t_n)$ where x is a variable and $t_1, \ldots, t_n$ are linear terms. A purified formula may be obtained by replacing each subterm of the form $f(e_1, \ldots, e_n)$ by a fresh variable, and recording the corresponding definition. This step creates a map pur that records the correspondence between terms and their purified versions, and a map def that keeps the definitions for fresh variables.

Constraint Generation. We create the constraints by applying the function Consec. The function Consec computes a constraint on the parameters of the templates φ_{pre} and φ_{post} over program variables and primed program variables, respectively, for a transition relation ρ. Let *Params* be the set of parameters that appear in the templates φ_{pre} and φ_{post}. The output of Consec is the constraint over *Params* such that the implication

$$\varphi_{\mathsf{pre}} \wedge \rho \models_{\text{LI+UIF}} \varphi_{\mathsf{post}} \tag{6}$$

is valid for some valuation of *Params* if and only if such a valuation satisfies the constraint.

---

function INV(LI+UIF)
input
  Program $\mathsf{P} = (x, \mathsf{locs}, \ell_0, \mathcal{T}, \mathsf{Good})$
  tmpl: invariant template map
local
  Params: set of parameters that appear in the invariant templates
  pur: purification map that assigns purified LI-terms to LI+UIF-terms
  def: set of definitions for fresh variables created by purification
  $\Phi :=$ constraint over parameters of invariant templates
output
  inv: invariant map from locs to invariants, which is an instantiation of tmpl
begin
  Params := parameters that appear in invariant templates
  $\mathsf{pur}, \mathsf{def} :=$ purification of $\{\mathsf{tmpl}(\ell) \mid \ell \in \mathsf{locs}\} \cup \{\mathsf{Good}\} \cup \mathcal{T}$
  $\Phi := 0 \leq 1$
  foreach $(\ell, \rho, \ell') \in \mathcal{T}$ **do**
    $\Phi := \Phi \wedge$ CONSEC($\mathsf{pur}(\mathsf{tmpl}(\ell))$, $\mathsf{pur}(\rho)$, $\mathsf{pur}(\mathsf{tmpl}(\ell'))$, def)
  done
  foreach $\ell \in \mathsf{locs}$ **do**
    $\Phi := \Phi \wedge$ CONSEC($\mathsf{pur}(\mathsf{tmpl}(\ell))$, $0 \leq 1$, $\mathsf{pur}(\mathsf{Good})$, def)
  done
  if $\exists Params : \Phi$ **then**
    let $\mu : Params \to \mathbb{Q}$ be a satisfying assignment of Φ
    foreach $\ell \in \mathsf{locs}$ **do**
      $\mathsf{inv}(\ell) := \mathsf{tmpl}(\ell)$ where parameters are instantiated by μ
        and fresh variables replaced by definitions in def
    done
    return "Invariant map: inv."
  else
    return "No invariant expressible by template tmpl exists."
end.

Fig. 2. Algorithm INV(LI+UIF) for the synthesis of invariants in linear arithmetic and uninterpreted function symbols. The auxiliary function CONSEC is shown in Fig. 3

CONSEC takes as inputs three linear arithmetic assertions and a set of definitions for fresh variables. The first assertion

$$(P P_{\mathsf{pre}}) \begin{pmatrix} x \\ x_{\mathsf{pre}} \end{pmatrix} \leq p$$

represents the purified version of φ_{pre}, and is given over the program variables x and a vector of the corresponding fresh variables x_{pre}. The second assertion

$$(R R' R_{\mathsf{rel}}) \begin{pmatrix} x \\ x' \\ x_{\mathsf{rel}} \end{pmatrix} \leq r$$

function CONSEC

input

$(P P_{\mathsf{pre}}) \binom{x}{x_{\mathsf{pre}}} \le p$: purified template for pre-location with fresh variables x_{pre}

$(R R' R_{\mathsf{rel}}) \begin{pmatrix} x \\ x' \\ x_{\mathsf{rel}} \end{pmatrix} \le r$: purified transition relation with fresh variables x_{rel}

$(Q Q_{\mathsf{post}}) \begin{pmatrix} x' \\ x_{\mathsf{post}} \end{pmatrix} \le q$: purified template for post-location with fresh variables x_{post}

Def: set of definitions for fresh variables x_{pre}, x_{rel}, and x_{post}

local

Φ: auxiliary constraint over the template parameters P, P_{pre}, p and Q, Q_{post}, q that encodes an implication induced by a particular sequence of axiom instances

fresh : fresh variables defined by Def

output

Ψ: consecution constraint over the template parameters P, P_{pre}, p and Q, Q_{post}, q

begin

$\Psi := 1 \le 0$

$\mathsf{fresh} := x_{\mathsf{pre}} \cup x_{\mathsf{rel}} \cup x_{\mathsf{post}}$

$\mathsf{Inst} := \{c \approx d \to c = d \mid c, d \in \mathsf{fresh}$ and $c = f(c_1, \dots, c_n) \in \mathsf{Def}$ and $d = f(d_1, \dots, d_n) \in \mathsf{Def}\}$

foreach $n \in \{0, \dots, |\mathsf{Inst}|\}$ **do**

$\{c^i \approx d^i \to c^i = d^i\}_{i=1}^n :=$ select sequence of n axiom instances from Inst

$(E_{\mathsf{pre}} E_{\mathsf{rel}} E_{\mathsf{post}}) \begin{pmatrix} x_{\mathsf{pre}} \\ x_{\mathsf{rel}} \\ x_{\mathsf{post}} \end{pmatrix} \le e :=$ inequality representation of $\bigwedge_{i=1}^n c^i = d^i$

$\Phi := \exists \Lambda \in \mathbb{Q}_{\ge 0}^{|q| \times (|p|+|r|+|e|)}$:

$$\Lambda \begin{pmatrix} P & 0 & P_{\mathsf{pre}} & 0 & 0 \\ R & R' & 0 & R_{\mathsf{rel}} & 0 \\ 0 & 0 & E_{\mathsf{pre}} & E_{\mathsf{rel}} & E_{\mathsf{post}} \end{pmatrix} = \begin{pmatrix} 0 & Q & 0 & 0 & Q_{\mathsf{post}} \end{pmatrix} \wedge \Lambda \begin{pmatrix} p \\ r \\ e \end{pmatrix} \le q$$

foreach $k \in \{1, \dots, n\}$ **do**

$(F_{\mathsf{pre}} F_{\mathsf{rel}} F_{\mathsf{post}}) \begin{pmatrix} x_{\mathsf{pre}} \\ x_{\mathsf{rel}} \\ x_{\mathsf{post}} \end{pmatrix} \le f :=$ inequality representation of $\bigwedge_{i=1}^{k-1} c^i = d^i$

$(G G_{\mathsf{pre}} G_{\mathsf{rel}} G_{\mathsf{post}}) \begin{pmatrix} x \\ x_{\mathsf{pre}} \\ x_{\mathsf{rel}} \\ x_{\mathsf{post}} \end{pmatrix} \le g :=$ purified representation of $c^k \approx d^k$

$\Phi := \Phi \wedge \exists \Lambda \in \mathbb{Q}_{\ge 0}^{|g| \times (|p|+|r|+|f|)}$:

$$\Lambda \begin{pmatrix} P & 0 & P_{\mathsf{pre}} & 0 & 0 \\ R & R' & 0 & R_{\mathsf{rel}} & 0 \\ 0 & 0 & F_{\mathsf{pre}} & F_{\mathsf{rel}} & F_{\mathsf{post}} \end{pmatrix} = \begin{pmatrix} G & 0 & G_{\mathsf{pre}} & G_{\mathsf{rel}} & G_{\mathsf{post}} \end{pmatrix} \wedge \Lambda \begin{pmatrix} p \\ r \\ f \end{pmatrix} \le g$$

done

$\Psi := \Psi \vee \Phi$

done

return Ψ

end.

Fig. 3. Function CONSEC computes a constraint over the template parameters which encodes the consecution condition for a given transition relation and invariant templates for the pre- and post-locations

represents the purified version of the transition relation ρ, and is given over the program variables x, their primed versions x', and a vector of the corresponding fresh variables x_{rel}. The third assertion

$$(QQ_{\mathsf{post}}) \begin{pmatrix} x' \\ x_{\mathsf{post}} \end{pmatrix} \leq q$$

is similar to the first one, where the program variables x are substituted by their primed versions x'. The resulting constraint Ψ over the parameters P, P_{pre}, p and Q, Q_{post}, q is satisfiable if and only if implication (6) is valid for some valuation of the parameters.

The constraint computed by the function Consec captures all sequences of instantiations of functionality axioms that may potentially appear in a proof of implication (6). For each such a sequence, which can be empty, we introduce a disjunct that encodes two conditions. The first condition says that the implication holds in the theory of linear arithmetic once all axioms from the sequence are applied. The second condition justifies the application of each axiom in the sequence. We take the disjunction of constraints computed for each sequence, which encodes the choice of an arbitrary sequence.

In algorithm Inv(LI+UIF), we call Consec to capture the constraints **(I1)**, and **(I2)**. First, for each transition we compute the consecution constraint that ensures the closure under the application of the transition relation. Then, we encode the condition that the resulting invariant is sufficiently strong, i.e., it only contains 'good' states.

Correctness. We state the correctness of the algorithm Inv(LI+UIF) in the following theorems.

Theorem 2 (Soundness of Inv(LI+UIF)). *The algorithm* Inv(LI+UIF) *computes an invariant map that is expressible by a given invariant template.*

Proof. We show that the resulting map inv satisfies the consecution condition. The proof that inv also satisfies the initiation and strength conditions is similar.

Let φ_{pre} and φ_{post} be invariant templates instantiated by the algorithm. We show that the implication (6) holds. Let Ψ be the constraint that is computed by applying Consec on the input that corresponds to the transition relation ρ. The valuation of template parameters that defines inv satisfies Ψ. Let Φ be a disjunct of Ψ that is satisfied. We assume that Φ corresponds to the following sequence of instances of the functionality axioms:

$$c^1 \approx d^1 \rightarrow c^1 = d^1, \dots, c^n \approx d^n \rightarrow c^n = d^n.$$

Let $(PP_{\mathsf{pre}}) \left(\begin{smallmatrix} x \\ x_{\mathsf{pre}} \end{smallmatrix} \right) \leq p$, $(RR'R_{\mathsf{rel}}) \begin{pmatrix} x \\ x' \\ x_{\mathsf{rel}} \end{pmatrix} \leq r$, and $(QQ_{\mathsf{post}}) \begin{pmatrix} x' \\ x_{\mathsf{post}} \end{pmatrix} \leq q$ be the purified version of φ_{pre}, ρ, and φ_{post}, respectively.

The first conjunct of Φ ensures that the following implication holds, by Farkas' lemma [22]:

$$(PP_{\mathsf{pre}}) \left(\begin{smallmatrix} x \\ x_{\mathsf{pre}} \end{smallmatrix} \right) \leq p \wedge (RR'R_{\mathsf{rel}}) \begin{pmatrix} x \\ x' \\ x_{\mathsf{rel}} \end{pmatrix} \leq r \wedge \bigwedge_{i=1}^{n} c^i = d^i \models_{\mathrm{LI}} (QQ_{\mathsf{post}}) \begin{pmatrix} x' \\ x_{\mathsf{post}} \end{pmatrix} \leq q.$$

This implication means that the above sequence of instances of functionality axioms is sufficient to prove the implication. The remaining conjuncts of Φ ensure that the axiom instances are applicable, because their premises are satisfied. This follows from the implications below, which are encoded by the remaining conjuncts of Φ. For each $k \in \{1, \ldots, n\}$ we have

$$(PP_{\mathsf{pre}})\left(\begin{smallmatrix} x \\ x_{\mathsf{pre}} \end{smallmatrix}\right) \leq p \;\wedge\; (RR'R_{\mathsf{rel}})\begin{pmatrix} x \\ x' \\ x_{\mathsf{rel}} \end{pmatrix} \leq r \;\wedge\; \bigwedge_{i=1}^{k-1} c^i = d^i \quad \models_{\mathrm{LI}} \quad c^k \approx d^k.$$

Since purification preserves satisfiability, we conclude that the invariant φ_{pre} is closed under the transition relation ρ by the invariant φ_{pre}. $\square$

Theorem 3 (Completeness of Inv(LI+UIF)). *The algorithm* INV(LI+UIF) *computes an invariant map if it is expressible by a given invariant template.*

Proof. Let inv be an invariant map that satisfies the invariant template. We show that the consecution constraint computed by the function CONSEC is satisfiable. The proof that it is also satisfiable in conjunction with initiation and strength constraints is similar.

Let φ_{pre} and φ_{post} be assertions such that for a transition relation ρ the implication (6) holds. By Theorem 5 in [23] we have that the following implication is valid in the theory of linear arithmetic for some sequence of instances of functionality axioms. Furthermore, these instances are only created for the terms that appear in the assertions φ_{pre}, φ_{post}, and ρ. Let

$$c^1 \approx d^1 \to c^1 = d^1, \ldots, c^n \approx d^n \to c^n = d^n$$

be such a sequence. Since purification preserve the satisfiability, we conclude that the conjuncts of Φ encode that (i) the purified version of the assertion φ_{post} is implied by the purified version of $\varphi_{\mathsf{pre}} \wedge \rho$ in conjunction with heads $\bigwedge_{i=1}^{n} c^i = d^i$ of functionality axiom instances from the sequence, and (ii) for each $k \in \{1, \ldots, n\}$ the premise $c^k \approx d^k$ of each axiom instance is implied by $\varphi_{\mathsf{pre}} \wedge \rho$ in conjunction with the axiom heads $\bigwedge_{i=1}^{k-1} c^i = d^i$ applied so far. All implications hold in the theory of linear arithmetic. Hence, the constraint computed by CONSEC is satisfiable. $\square$

We obtain the following corollary of Theorems 2 and 3.

Corollary 1. *The existence of a* LI+UIF*-invariant map that is expressible by a given template is decidable.*

Complexity and Optimizations. Let n be the number of applications of function symbols in the template and in the program description. The algorithm INV(LI+UIF) needs to solve at most $n!$ quantifier elimination problems for rational/real arithmetic constraints of the second degree, where the size of each problem is linear in program description and quadratic in n. The time complexity of each problem is exponential in its size [4].

```
int alloc() {
     assume (kfreelist != 0 && *(kfreelist + 4) == RESERVED);
                      // First page is always reserved.
     prev = kfreelist; curr = *kfreelist; permission = curr + 4;
     while(curr!=0 && *permission == RESERVED) {
       prev = curr; curr = *curr;
       permission = curr + 4;
     }
     L1: assert( *(prev + 4) == RESERVED );
     L2: assert( *prev == curr );
     if (curr!=0) *prev = *curr;
     return curr;
}
```

Fig. 4. A kernel allocator. Our algorithm automatically constructs the loop invariant `*(prev+4)-curr+perm-RESERVED==4 && perm==curr+4`, which implies the first assertion, and the invariant `*prev==curr`, which implies the second assertion

We observe that the construction of the constraint that considers all possible axiom sequences can be done lazily, i.e., we consider new sequences only if the previously discovered ones do not yield a desired invariant map. Such a lazy construction is crucial for practical applicability of INV(LI+UIF), since in many cases only short sequences consisting of at most a pair of axioms suffice.

5 Experiences

We have implemented algorithm INV(LI+UIF) in Sicstus Prolog [15] with linear programming solver [12] and applied it to the verification of low level memory allocators in an operating system. We apply a heuristics that prefers shorter candidate sequences of axiom instances to longer ones while lazily constructing constraints. The invariant templates need to be supplied manually. Solving of non-linear constraints was done by heuristic instantiation of the values for Λ, cf. Fig. 3, and subsequent solving of the resulting linear constraint.

Figure 4 shows a simplified low level memory allocator used in an OS kernel. The variable `kfreelist` is the head of a free list of memory pages. Each memory block contains a pointer to the next free block and also a permission bit that says whether the block can be given to a user process. The permission bit is accessed using address arithmetic by adding 4 bytes to the base address of the memory block. For simplicity, we have removed the type casts from the example code and also ignore overflow issues. We assume that the free list has at least one block and the first block is reserved by the kernel.

The while loop iterates over the free list, looking for the first unreserved free block. This block is returned. The iteration uses two pointers, `curr` pointing to the current block, and `prev` pointing to the previous block. We want to prove the assertion L1, which states that pointer `prev` points to a reserved block, and assertion L2, which states that the next block from pointer `prev` is the block pointed to by `curr` (or null). The invariant requires both linear arithmetic (for the address arithmetic) and uninterpreted functions (for the dereferences).

Assertion `L1`**:** Our invariant synthesis for proving the first assertion required 3.25 s on a 1.7 GHz Linux laptop. The tool tried 105 axiom sequences. Considering sequences of length at most one was sufficient. We used a template that is a conjunction of two equalities[1] (where $\mathit{ref}(\cdot)$ denotes the address-of operator and $\mathit{der}(\cdot)$ is the dereference operator):

$$
\begin{aligned}
&c^1_{\texttt{prev}}\texttt{prev} + c^1_{\texttt{curr}}\texttt{curr} + c^1_{\texttt{perm}}\texttt{perm} + c^1_{\texttt{RESERVED}}\texttt{RESERVED}+\\
&c^1_{\texttt{ref}}\,\mathit{ref}(c^1_{\texttt{refprev}}\texttt{prev} + c^1_{\texttt{refcurr}}\texttt{curr} + c^1_{\texttt{refperm}}\texttt{perm} + c^1_{\texttt{refRESERVED}}\texttt{RESERVED} + c^1_{\texttt{ref}})+\\
&c^1_{\texttt{der}}\,\mathit{der}(c^1_{\texttt{derprev}}\texttt{prev} + c^1_{\texttt{dercurr}}\texttt{curr} + c^1_{\texttt{derperm}}\texttt{perm} + c^1_{\texttt{derRESERVED}}\texttt{RESERVED} + c^1_{\texttt{der}}) = c^1\\
&\wedge\\
&c^2_{\texttt{prev}}\texttt{prev} + c^2_{\texttt{curr}}\texttt{curr} + c^2_{\texttt{perm}}\texttt{perm} + c^2_{\texttt{RESERVED}}\texttt{RESERVED} = c^2.
\end{aligned}
$$

This template leads to the loop invariant

$$-\texttt{curr} + \texttt{perm} - \texttt{RESERVED} + \mathit{der}(\texttt{prev} + 4) = 4 \ \wedge\ -\texttt{curr} + \texttt{perm} = 4.$$

Assertion `L2`**:** For the second assertion we used a template that contains only the first conjunct from the template above, and we obtained the loop invariant

$$-\texttt{curr} + \mathit{der}(\texttt{prev}) = 0.$$

Our implementation computed an invariant that implies the second assertion in 1.28 s, which required enumeration of 44 axiom sequences.

We are working on scaling our algorithm to larger programs. The main complexity arises because invariants can be Boolean combinations of atomic facts.

6 Applications to Data Structures

We now present applications of algorithm Inv(LI+UIF) to the synthesis of invariants in programs that use abstract data structures. The key technical idea is that of a *reduction function*. Let Σ and Ω be signatures with $\Omega \subseteq \Sigma$. Let T be a Σ-theory and R an Ω-theory, such that $R \subseteq T$. We say T *reduces* to R if there is a computable map from Σ-formulas to Ω-formulas such that when applied to a Σ-formula φ, we get an Ω-formula φ^* such that φ and φ^* are T-equivalent, that is, $\models_T \varphi \leftrightarrow \varphi^*$, and φ^* is R-satisfiable iff φ is T-satisfiable.

Given a theory T and a reduction function from T to LI+UIF, we can extend the algorithm Inv(LI+UIF) to generate invariants over T from templates that contain symbols from the theory T in the following way. The intuitive idea is that we first apply the reduction function to reduce templates in the theory T to templates in the theory LI+UIF and then apply the invariant generation algorithm for LI+UIF. The resulting invariant is an invariant also for the theory T.

[1] The implementation supports direct handling of equality and inequality constraints.

Technically, the purification step is identical, while in CONSEC, we apply the reduction function to each definition and then generate the constraints for the resulting formula using the theory LI+UIF. We omit the technical details.

We now show that reduction functions to LI+UIF exist for two interesting theories: the array property fragment and the theory of sets.

Arrays. The theory of arrays has a signature Σ_{array} with the function symbols read and write together with the axiom [17]:

$$\begin{aligned}
&\mathsf{read}(\mathsf{write}(a, i, e), i) = e\,,\\
&i \neq j \;\Rightarrow\; \mathsf{read}(\mathsf{write}(a, i, e), j) = \mathsf{read}(a, j)\,,\\
&(\forall\ i)(\mathsf{read}(a, i) = \mathsf{read}(b, i)) \;\Rightarrow\; a = b\,.
\end{aligned}$$

The variables in the second position of read and write are the *index variables.*

Let I be a set of index variables, which we assume are distinct from the program variables. An *array property* [2] is a universally quantified formula

$$\forall I :\ \varphi(I) \rightarrow \psi(I),$$

where the formula $\varphi(I)$ is a constraint on the index variables and $\psi(I)$ may contain array operations indexed by variables from I. Both $\varphi(I)$ and $\psi(I)$ are syntactically restricted. The index guard $\varphi(I)$ is a Boolean expression over linear arithmetic inequalities over I and the program variables such that each inequality is one of the following:

- a comparison $i \leq j$ between two index variables $i, j \in I$,
- a comparison $i \leq e$ or $e \leq i$ between an index variable $i \in I$ and a linear expression e over program variables.

The value guard $\psi(I)$ is restricted in the following way w.r.t. the usage of the universally quantified index variables I. Every occurrence of such a variable i must be in the index position of a read operation $\mathsf{read}(a, i)$ for some array a. Additionally, no nested read operations that are allowed in $\psi(I)$. The *array property fragment* is the combination of linear arithmetic, uninterpreted function symbols, and array property formulas.

Sets. The theory of sets (with finite cardinality constraints) has a signature Σ_{set} containing the constant symbols $\emptyset$ (empty set) and $\nVdash$ (full set), the binary function symbols $\cup$ (union), $\cap$ (intersection), and $\setminus$ (difference), the unary function symbol $\{\cdot\}$ (singleton), and the binary predicate symbol $\in$ with the standard semantics. In addition it has, for each natural number k, the unary predicate symbols $|\cdot| \geq k$ and $|\cdot| = k$. The element domain is assumed to be finite. The theory T_{set} is the set of all Σ_{set}-sentences that are true in all standard set-structures.

We use the following results on reductions, proved in [2, 14].

Theorem 4 (Reductions to LI+UIF).

1. **[2]** *The set of formulas in the array property fragment reduces to* LI+UIF.
2. **[14]** *The quantifier-free theories of arrays and sets reduce to* LI+UIF.

From the theorem, and the discussion on invariant generation, we get the following corollary.

Corollary 2. *The existence of a T-invariant map that is expressible by a given template is decidable, where T is the theory of arrays, sets, or formulas in the array property fragment.*

7 Conclusion

We presented an algorithm for the synthesis of invariants in the theory of linear arithmetic and uninterpreted function symbols. While expressive, in that many interesting aspects of program behavior can be modeled in (or reduced to) this logic, our technique is ultimately limited by the large space of possible templates that the user must search to provide good templates. In particular, the search space usually becomes too big in the presence of disjunctions in invariant templates. We leave the identification of heuristics for the property-guided construction of invariant templates for future work.

Acknowledgments. We thank Viorica Sofronie-Stokkermans for valuable discussions on hierarchic theory combination.

References

1. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
2. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. VMCAI*, LNCS 3855, pages 427–442. Springer, 2006.
3. C. C. Chang and H. J. Keisler. *Model Theory*. North-Holland, 3rd edition, 1990.
4. G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages*, LNCS 33, pages 134–183. Springer, 1975.
5. M. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV*, LNCS 2725, pages 420–432. Springer, 2003.
6. P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *Proc. VMCAI*, LNCS 3385. Springer, 2005.
7. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proc. PLILP*, LNCS 631, pages 269–295. Springer, 1992.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245. ACM, 2002.
9. R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967.
10. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proc. PLDI*, pages 376–386. ACM, 2006.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

12. C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.
13. D. Kapur. Automatically generating loop invariants using quantifier elimination. In *Proc. Deduction and Applications*, volume 05431. IBFI Schloss Dagstuhl, 2006.
14. D. Kapur and C. Zarba. A reduction approach to decision procedures. Technical Report TR-CS-2005-44, University of New Mexico, 2005.
15. T. I. S. Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, PO Box 1263 SE-164 29 Kista, Sweden, October 2001. Release 3.8.7.
16. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.
17. J. McCarthy. Towards a mathematical science of computation. In *Proc. IFIP Congress*, pages 21–28. North-Holland, 1962.
18. G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
19. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *Proc. SAS*, LNCS 3148, pages 53–68. Springer, 2004.
20. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. POPL*, pages 318–329. ACM, 2004.
21. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. VMCAI*, LNCS 3385, pages 25–41. Springer, 2005.
22. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.
23. V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *Proc. CADE*, LNCS 3632, pages 219–234. Springer, 2005.