Accelerating Interpolants

Hossein Hojjat[1], Radu Iosif[2],
Filip Konečný[2,4], Viktor Kuncak[1], and Philipp Rümmer[3]

[1] Swiss Federal Institute of Technology Lausanne (EPFL)
[2] Verimag, Grenoble, France
[3] Uppsala University, Sweden
[4] Brno University of Technology, Czech Republic

Abstract. We present Counterexample-Guided *Accelerated* Abstraction Refinement (CEGAAR), a new algorithm for verifying infinite-state transition systems. CEGAAR combines *interpolation-based predicate discovery* in counterexample-guided predicate abstraction with *acceleration* technique for computing the transitive closure of loops. CEGAAR applies acceleration to dynamically discovered looping patterns in the unfolding of the transition system, and combines over-approximation with underapproximation. It constructs inductive invariants that rule out an infinite family of spurious counterexamples, alleviating the problem of divergence in predicate abstraction without losing its adaptive nature. We present theoretical and experimental justification for the effectiveness of CEGAAR, showing that inductive interpolants can be computed from classical Craig interpolants and transitive closures of loops. We present an implementation of CEGAAR that verifies integer transition systems. We show that the resulting implementation robustly handles a number of difficult transition systems that cannot be handled using interpolation-based predicate abstraction or acceleration alone.

1 Introduction

This paper contributes to the fundamental problem of precise reachability analysis for infinite-state systems. Predicate abstraction using interpolation has emerged as an effective technique in this domain. The underlying idea is to verify a program by reasoning about its *abstraction* that is easier to analyse, and is defined with respect to a set of predicates [21]. The set of predicates is refined to achieve the precision needed to prove the absence or the presence of errors [2]. A key difficulty in this approach is to automatically find predicates to make the abstraction sufficiently precise [3]. A breakthrough technique is to generate predicates based on *Craig interpolants* [15] derived from the proof of unfeasibility of a spurious trace [22].

While empirically successful on a variety of domains, abstraction refinement using interpolants suffers from the unpredictability of interpolants computed by provers, which can cause the verification process to diverge and never discover a sufficient set of predicates (even in case such predicates exist). The failure of such a refinement approach manifests in a sequence of predicates that rule out longer and longer counterexamples, but still fail to discover inductive invariants.

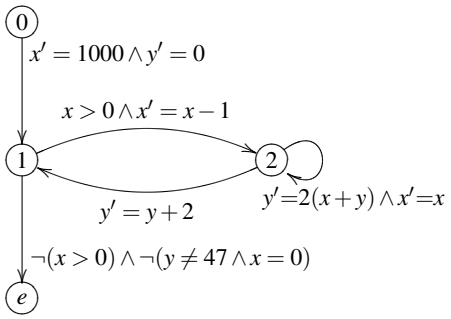
Following another direction, researchers have been making continuous progress on techniques for computing the transitive closure of useful classes of relations on integers [8, 10, 12, 17]. These *acceleration* techniques can compute closed form representation of certain classes of loops using Presburger arithmetic.

A key contribution of this paper is an algorithmic solution to apply these specialized analyses for particular classes of loops to rule out an infinite family of counterexamples during predicate abstraction refinement. An essential ingredient of this approach are interpolants that not only rule out one path, but are also *inductive* with respect to loops along this path. We observe that we can start from any interpolant for a path that goes through a loop in the control-flow graph, and apply a postcondition (or, equivalently a weakest precondition) with respect to the transitive closure of the loop (computed using acceleration) to generalize the interpolant and make it inductive. Unlike previous theoretical proposals [14], our method treats interpolant generation and transitive closure computation as black boxes: we can start from any interpolants and strengthen it using any loop acceleration. We call the resulting technique Counterexample-Guided *Accelerated* Abstraction Refinement, or CEGAAR for short. Our experience indicates that CEGAAR works well in practice.

Motivating Example To illustrate the power of the technique that we propose, consider the example in Figure 1. The example is smaller than the examples we consider in our evaluation (Section 6), but already illustrates the difficulty of applying existing methods.

```
int x,y;
x = 1000; y = 0;
while(x > 0){
  x--;
  while(*) {
    y = 2*(x + y);
  }
  y = y + 2;
}
assert(y != 47 && x == 0);
```

(a)



(b)

Fig. 1. Example Program and its Control Flow Graph with Large Block Encoding

Note that the innermost loop requires a very expressive logic to describe its closed form, so that standard techniques for computing exact transitive closure of loops do not apply. In particular, the acceleration technique does not apply to the innermost loop, and the presence of the innermost loop prevents the application of acceleration to the outer loop. On the other hand, predicate abstraction with interpolation refinement also fails to solve this example. Namely, it enters a very long refinement loop, considering increasingly longer spurious paths with CFG node sequences of the form $\mathbf{0}(\mathbf{12})^i\mathbf{1e}$, for $0 \leq i < 1000$. The crux of the problem is that the refinement eliminates each of these paths one by one, constructing too specific interpolants.

Our combined CEGAAR approach succeeds in proving the assertion of this program by deriving the loop invariant $y\%2 == 0 \wedge x \geq 0$. Namely, once predicate abstraction considers a path where the CFG node **1** repeats (such as **0121e**), it applies acceleration to this path. CEGAAR then uses the accelerated path to construct an inductive interpolant, which eliminates an infinite family of spurious paths. This provides predicate abstraction with a crucial predicate $y\%2 = 0$, which enables further progress, leading to the discovery of the predicate $x \geq 0$. Together, these predicates allow predicate abstraction to construct the invariant that proves program safety. Note that this particular example focuses on proving the absence of errors, but our experience suggests that CEGAAR can, in many cases, find long counterexamples faster than standard predicate abstraction.

Related Work Predicate abstraction has proved is a rich and fruitful direction in automated verification of detailed properties of infinite-state systems [2, 21, 22]. The pioneering work in [4] is, to the best of our knowledge, the first to propose a solution to the divergence problem in predicate abstraction. More recently, sufficient conditions to enforce convergence of refinement in predicate abstraction are given in [3], but it remains difficult to enforce them in practice. A promising direction for ensuring completeness with respect to a language of invariants is parameterizing the syntactic complexity of predicates discovered by an interpolating *split prover* [23]. Because it has the flavor of invariant enumeration, the feasibility of this approach in practice remains to be further understood.

To alleviate relatively weak guarantees of refinement in predicate abstraction in practice, researchers introduced *path invariants* [6] that rule out a family of counterexamples at once using constraint-based analysis. Our CEGAAR approach is similar in the spirit, but uses acceleration [8,10,12,17] instead of constraint-based analysis, and therefore has complementary strengths. Acceleration naturally generates precise *disjunctive invariants*, needed in many practical examples, while constraint-based invariant generation [6] resorts to an ad-hoc unfolding of the path program to generate disjunctive invariants. Acceleration can also infer expressive predicates, in particular modulo constraints, which are relevant for purposes such as proving memory address alignment.

The method that is probably closest to CEGAAR is proposed in [14]. In this work the authors define *inductive interpolants* and prove the existence of effectivelly computable inductive interpolants for a class of affine loops, called *poly-bounded*. The approach is, however, limited to programs with one poly-bounded affine loop, for which initial and error states are specified. We only consider loops that are more restricted than the poly-bounded ones, namely loops for which transitive closures are Presburger definable. On the other hand, our method is more general in that it does not restrict the number of loops occurring in the path program, and benefits from regarding both interpolation and transitive closure computation as black boxes.

The ability to compute closed forms of certain loops is also exploited in algebraic approaches [7]. These approaches can naturally be generalized to perform useful over-approximation [1] and under-approximation. This insight is also helpful in our approach, where we first attempt to perform exact acceleration for the particular subspace of the transition system. If this fails, we resort to over-approximation, and finally

to under-approximation, which, in the worst-case, reduces to standard predicate refinement that can exclude as few as one spurious path.

2 Preliminaries

Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be a set of variables ranging over integer numbers, and $\mathbf{x}'$ be the set $\{x'_1, \ldots, x'_n\}$. A *predicate* is a first-order arithmetic formula P. By $FV(P)$ we denote the set of free variables in P, i.e. variables not bound by a quantifier. By writing $P(\mathbf{x})$ we intend that $FV(P) \subseteq \mathbf{x}$. We write $\bot$ and $\top$ for the boolean constants false and true. A *linear term* t over a set of variables in $\mathbf{x}$ is a linear combination of the form $a_0 + \sum_{i=1}^{n} a_i x_i$, where $a_0, a_1, \ldots, a_n \in \mathbb{Z}$. An *atomic proposition* is a predicate of the form $t \leq 0$, where t is a linear term. *Presburger arithmetic* is the first-order logic over propositions $t \leq 0$; Presburger arithmetic has quantifier elimination and is decidable [29]. For simplicity we consider only formulas in Presburger arithmetic in this paper. A valuation of $\mathbf{x}$ is a function $\nu : \mathbf{x} \rightarrow \mathbb{Z}$. If ν is a valuation of $\mathbf{x}$, we denote by $\nu \models P$ the fact that the formula obtained by replacing each occurrence of x_i with $\nu(x_i)$ is valid. Similarly, an arithmetic formula $R(\mathbf{x}, \mathbf{x}')$ defining a relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$ is evaluated referring to two valuations ν_1, ν_2; the satisfaction relation is denoted $\nu_1, \nu_2 \models R$. The composition of two relations $R_1, R_2 \in \mathbb{Z}^n \times \mathbb{Z}^n$ is denoted by $R_1 \circ R_2 = \{(\mathbf{u}, \mathbf{v}) \in \mathbb{Z}^n \times \mathbb{Z}^n \mid \exists \mathbf{t} \in \mathbb{Z}^n \,.\, (\mathbf{u}, \mathbf{t}) \in R_1 \text{ and } (\mathbf{t}, \mathbf{v}) \in R_2\}$. Let ε be the identity relation $\{(\mathbf{u}, \mathbf{u}) \mid \mathbf{u} \in \mathbb{Z}^n \times \mathbb{Z}^n\}$. We define $R^0 = \varepsilon$ and $R^i = R^{i-1} \circ R$, for any $i > 0$. With these notations, $R^+ = \bigcup_{i=1}^{\infty} R^i$ denotes the *transitive closure* of R, and $R^* = R^+ \cup \varepsilon$ denotes the *reflexive and transitive closure* of R. We sometimes use the same symbols to denote a relation and its defining formula. For a set of n-tuples $S \subseteq \mathbb{Z}^n$ and a relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$, let $post(S, R) = \{\mathbf{v} \in \mathbb{Z}^n \mid \exists \mathbf{u} \in S \,.\, (\mathbf{u}, \mathbf{v}) \in R\}$ denote the *strongest postcondition* of S via R, and $wpre(S, R) = \{\mathbf{u} \in \mathbb{Z}^n \mid \forall \mathbf{v} \,.\, (\mathbf{u}, \mathbf{v}) \in R \rightarrow \mathbf{v} \in S\}$ denote the *weakest precondition* of S with respect to R. We use *post* and *wpre* for sets and relations, as well as for logical formulae defining them.

We represent programs as control flow graphs. A *control flow graph* (CFG) is a tuple $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ where $\mathbf{x} = \{x_1, \ldots, x_n\}$ is a set of variables, Q is a set of *control states*, $\rightarrow$ is a set of edges of the form $q \xrightarrow{R} q'$, labeled with arithmetic formulae defining relations $R(\mathbf{x}, \mathbf{x}')$, and $I, E \subseteq Q$ are sets of *initial* and *error* states, respectively. A *path* in G is a sequence $\theta : q_1 \xrightarrow{R_1} q_2 \xrightarrow{R_2} q_3 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$, where $q_1, q_2, \ldots, q_n \in Q$ and $q_i \xrightarrow{R_i} q_{i+1}$ is an edge in G, for each $i = 1, \ldots, n-1$. We assume without loss of generality that all variables in $\mathbf{x} \cup \mathbf{x}'$ appear free in each relation labeling an edge of G[5]. We denote the relation $R_1 \circ R_2 \circ \ldots \circ R_{n-1}$ by $\rho(\theta)$ and assume that the set of free variables of $\rho(\theta)$ is $\mathbf{x} \cup \mathbf{x}'$. The path θ is said to be a *cycle* if $q_1 = q_n$, and a *trace* if $q_1 \in I$. The path θ is said to be *feasible* if and only if there exist valuations $\nu_1, \ldots, \nu_n : \mathbf{x} \rightarrow \mathbb{Z}$ such that $\nu_i, \nu_{i+1} \models R_i$, for all $i = 1, \ldots, n-1$. A control state is said to be *reachable* in G if it occurs on a feasible trace.

Acceleration The goal of acceleration is, given a relation R in a fragment of integer arithmetic, to compute its reflexive and transitive closure, R^*. In general, defining R^* in

[5] For variables that are not modified by a transition, this can be achieved by introducing an explicit update $x' = x$.

a decidable fragment of integer arithmetic is not possible, even when R is definable in a decidable fragment such as, e.g. Presburger arithmetic. We next present two fragments of arithmetic in which transitive closures of relations are Presburger definable.

Definition 1. *Let* $U(\mathbf{x}) = \{\pm x \pm y \mid x, y \in \mathbf{x}\}$ *be the set of* octagonal terms *over* $\mathbf{x}$*. A formula* $\phi(\mathbf{x})$ *is an* octagonal constraint *if it is equivalent to a finite conjunction of atomic propositions of the form* $u \leq c$*, where* $u \in U(\mathbf{x})$ *and* $c \in \mathbb{Z}$*.*

An *octagonal relation* is a relation defined by an octagonal constraint $R(\mathbf{x}, \mathbf{x}')$. The transitive closure of an octagonal relation is Presburger definable and effectively computable [10, 12].

Definition 2. *A* linear affine relation *is a relation of the form* $\mathcal{R}(\mathbf{x}, \mathbf{x}') \equiv C\mathbf{x} \geq \mathbf{d} \wedge \mathbf{x}' = A\mathbf{x} + \mathbf{b}$*, where* $A \in \mathbb{Z}^{n \times n}$*,* $C \in \mathbb{Z}^{p \times n}$ *are matrices and* $\mathbf{b} \in \mathbb{Z}^n$*,* $\mathbf{d} \in \mathbb{Z}^p$*.* $\mathcal{R}$ *is said to have the* finite monoid property *if and only if the set* $\{A^i \mid i \geq 0\}$ *is finite.*

Notice that linear affine relations are deterministic, unlike the octagonal relations considered in the previous. It is known that the finite monoid condition is decidable [8], and moreover that the transitive closure of a finite monoid affine relation is Presburger definable and effectively computable [8, 17].

Predicate Abstraction Informally, predicate abstraction computes an overapproximation of the transition system generated by a program and verifies whether an error state is reachable in the abstract system. If no error occurs in the abstract system, the algorithm reports that the original system is safe. Otherwise, if a path to an error state (counterexample) has been found in the abstract system, the corresponding concrete path is checked. If this latter path corresponds to a real execution of the system, then a real error has been found. Otherwise, the abstraction is refined in order to exclude the counterexample, and the procedure continues.

Given a CFG $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$, and a (possibly infinite) set of predicates $\mathcal{P}$, an *abstract reachability tree* (ART) for G is a tuple $T = \langle S, \pi, r, e \rangle$ where $S \subseteq Q \times 2^{\mathcal{P} \setminus \{\bot\}}$ is a set of nodes (notice that for no node $\langle q, \Phi \rangle$ in T we may have $\bot \in \Phi$), $\pi : Q \rightarrow 2^{\mathcal{P}}$ is a mapping associating control states with sets of predicates, $i \in I \times \{\top\}$ is the root node, $e \subseteq S \times S$ is a tree-structured edge relation:

- all nodes in S are reachable from the root r
- for all $n, m, p \in S$, $e(n, p) \wedge e(m, p) \Rightarrow n = m$
- $e(\langle q_1, \Phi_1 \rangle, \langle q_2, \Phi_2 \rangle) \Rightarrow q_1 \xrightarrow{R} q_2$ and $\Phi_2 = \{P \in \pi(q_2) \mid post(\bigwedge \Phi_1, R) \rightarrow P\}$

We say that an ART node $\langle q_1, \Phi_1 \rangle$ is *subsumed* by another node $\langle q_2, \Phi_2 \rangle$ if and only if $q_1 = q_2$ and $\bigwedge \Phi_1 \rightarrow \bigwedge \Phi_2$. It is usually considered that no node in an ART is subsumed by another node, from the same ART.

It can be easily checked that each path $\sigma : r = \langle q_1, \Phi_1 \rangle, \langle q_2, \Phi_2 \rangle, \ldots, \langle q_k, \Phi_k \rangle$, starting from the root in T, can be mapped into a trace $\theta : q_1 \xrightarrow{R_1} q_2 \ldots q_{k-1} \xrightarrow{R_{k-1}} q_k$ of G, such that $post(\top, \rho(\theta)) \rightarrow \bigwedge \Phi_k$. We say that θ is a *concretization* of σ, or that σ concretizes to θ. A path in an ART is said to be *spurious* if none of its concretizations is feasible.

3 Interpolation-Based Abstraction Refinement

By *refinement* we understand the process of enriching the predicate mapping π of an ART $T = \langle S, \pi, r, e \rangle$ with new predicates. The goal of refinement is to prevent spurious counterexamples (paths to an error state) from appearing in the ART. To this end, an effective technique used in many predicate abstraction tools is that of *interpolation*.

Given an unsatisfiable conjunction $A \wedge B$, an interpolant I is a formula using the common variables of A and B, such that $A \rightarrow I$ is valid and $I \wedge B$ is unsatisfiable. Intuitively, I is the explanation behind the unsatisfiability of $A \wedge B$. Below we introduce a slightly more general definition of a *trace interpolant*.

Definition 3 ([24]). *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG and*

$$\theta : q_1 \xrightarrow{R_1} q_2 \xrightarrow{R_2} q_3 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$$

be an infeasible trace of G. An interpolant *for θ is a sequence of predicates $\langle I_1, I_2, \ldots, I_n \rangle$ with free variables in $\mathbf{x}$, such that: $I_1 = \top$, $I_n = \bot$, and for all $i = 1, \ldots, n-1$, $post(I_i, R_i) \rightarrow I_{i+1}$.*

Interpolants exist for many theories, including all theories with quantifier elimination, and thus for Presburger arithmetic. Moreover, a trace is infeasible if and only if it has an interpolant (Lemma 10, Appendix A). Including any interpolant of an infeasible trace into the predicate mapping of an ART suffices to eliminate any abstraction of the trace from the ART. We can thus refine the ART and exclude an infeasible trace by including the interpolant that proves the infeasibility of the trace (Lemma 11, Appendix A).

Note that the refinement technique using Definition 3 only guarantees that *one* spurious counterexample is eliminated from the ART with each refinement step. This fact hinders the efficiency of predicate abstraction tools, which must rely on the ability of theorem provers to produce interpolants that are general enough to eliminate more than one spurious counterexample at the time. The following is a stronger notion of an interpolant, which ensures generality with respect to an infinite family of counterexamples.

Definition 4 ([14], Def. 2.4). *Given a CFG G, a* trace scheme *in G is a sequence of the form:*

$$\xi : q_0 \xrightarrow{Q_1} \overset{L_1}{\curvearrowright} q_1 \xrightarrow{Q_2} \ldots \xrightarrow{Q_{n-1}} \overset{L_{n-1}}{\curvearrowright} q_{n-1} \xrightarrow{Q_n} \overset{L_n}{\curvearrowright} q_n \xrightarrow{Q_{n+1}} q_{n+1} \tag{1}$$

where $q_0 \in I$ and:

- *$Q_i = \rho(\theta_i)$, for some non-cyclic paths θ_i of G, from q_{i-1} to q_i*
- *$L_i = \bigvee_{j=1}^{k_i} \rho(\lambda_{ij})$, for some cycles λ_{ij} of G, from q_i to q_i*

Intuitivelly, a trace scheme represents an infinite regular set of traces in G. The trace scheme is said to be *feasible* if and only if at least one trace of G of the form $\theta_1; \lambda_{1i_1} \ldots \lambda_{1i_{j_1}}; \theta_2; \ldots; \theta_n; \lambda_{ni_n} \ldots \lambda_{ni_{j_n}}; \theta_{n+1}$ is feasible.

The trace scheme is said to be *bounded* if $k_i = 1$, for all $i = 1, 2, \ldots, n$. A bounded[6] trace scheme is a regular language of traces, of the form $\sigma_1 \cdot \lambda_1^* \cdot \ldots \cdot \sigma_n \cdot \lambda_n^* \cdot \sigma_{n+1}$, where σ_i are acyclic paths, and λ_i are cycles of G.

[6] This term is used in analogy with the notion of bounded languages [20].

Definition 5 ([14], Def. 2.5). *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG and ξ be an infeasible trace scheme of the form (1). An* interpolant *for ξ is a sequence of predicates $\langle I_0, I_1, I_2, \ldots, I_n, I_{n+1} \rangle$, with free variables in $\mathbf{x}$, such that:*

1. $I_0 = \top$ *and* $I_{n+1} = \bot$
2. $post(I_i, Q_{i+1}) \rightarrow I_{i+1}$, *for all* $i = 0, 1, \ldots, n$
3. $post(I_i, L_i) \rightarrow I_i$, *for all* $i = 1, 2, \ldots, n$

The main difference with Definition 3 is the third requirement, namely that each interpolant predicate (except for the first and the last one) must be *inductive* with respect to the corresponding loop relation. It is easy to see that each of the two sequences:

$$\langle \top,\ post(\top, Q_1 \circ L_1^*),\ \ldots,\ post(\top, Q_1 \circ L_1^* \circ Q_2 \circ \ldots Q_n \circ L_n^*) \rangle \tag{2}$$

$$\langle wpre(\bot, Q_1 \circ L_1^* \circ Q_2 \circ \ldots Q_n \circ L_n^*),\ \ldots,\ wpre(\bot, Q_n \circ L_n^*),\ \bot \rangle \tag{3}$$

are interpolants for ξ, provided that ξ is infeasible (Lemma 2.6 in [14]). Just as for finite trace interpolants, the existence of an inductive interpolant suffices to prove the infeasibility of the entire trace scheme.

Lemma 6. *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG and ξ be an infeasible trace scheme of G of the form (1). If $T = \langle S, \pi, r, e \rangle$ is an ART for G, such that there exists an interpolant $\langle I_i \in \pi(q_i) \rangle_{i=0}^{n+1}$ for ξ, then no path in T concretizes to a trace in ξ.*

4 Counterexample-Guided Accelerated Abstraction Refinement

This section presents the CEGAAR algorithm for predicate abstraction with interpolant-based accelerated abstraction refinement. Since computing the interpolant of a trace scheme is typically more expensive than computing the interpolant of a finite counterexample, we apply acceleration in a demand-driven fashion. The main idea of the algorithm is to accelerate only those counterexamples in which some cycle repeats a certain number of times. For example, if the abstract state exploration has already ruled out the spurious counterexamples $\sigma \cdot \tau$, $\sigma \cdot \lambda \cdot \tau$ and $\sigma \cdot \lambda \cdot \lambda \cdot \tau$, when it sees next the spurious counterexample $\sigma \cdot \lambda \cdot \lambda \cdot \lambda \cdot \tau$, it will accelerate it into $\sigma \cdot \lambda^* \cdot \tau$, and rule out all traces which comply to this scheme. The maximum number of cycles that are allowed to occur in the acyclic part of an error trace, before computing the transitive closure, is called the *delay*, and is a parameter of the algorithm (here the delay was 2). A smaller delay results in a more aggressive acceleration strategy, whereas setting the delay to infinity is equivalent to performing predicate abstraction without acceleration.

The main procedure is CONSTRUCTART which builds an ART for a given CFG, and an abstraction of the set of initial values (Fig. 2). CONSTRUCTART is a worklist algorithm that expands the ART according to a certain exploration strategy (depth-first, breadth-first, etc.) determined by the type of the structure used as a worklist. We assume without loss of generality that the CFG has exactly one initial vertex *Init*. The CONSTRUCTART procedure starts with *Init* and expands the tree according to the definition of the ART (lines 10 and 11). New ART nodes are constructed using NEWARTNODE, which receives a CFG state and a set of predicates as arguments. The algorithm backtracks from expanding the ART when either the current node contains $\bot$ in its set of

```
input CFG G = ⟨x, Q, →, {Init}, E⟩
output ART T = ⟨S, π, Root, e⟩
WorkList ← []
S, π, e ← ∅
Root ← nil
 1: function CONSTRUCTART(Init, initialAbstraction)
 2:     node ← NEWARTNODE(Init, initialAbstraction)
 3:     if Root = nil then
 4:         Root ← node
 5:     end if
 6:     WorkList.add(⟨Init, node⟩)
 7:     while !WorkList.empty() do
 8:         ⟨nextCFGvertex, nextARTnode⟩ ← WorkList.remove()
 9:         for child ← children(nextCFGVertex) do
10:             Let R be such that nextCFGvertex -R-> child in G
11:             Φ = {p ∈ π(child) | POST(⋀ nextARTnode.abstraction, R) ⊢ p}
12:             if ⊥ ∈ Φ or (∃ an ART node ⟨child, Ψ⟩ . ∧Φ ⊢ Ψ) then
13:                 continue
14:             end if
15:             node ← NEWARTNODE(child, Φ)
16:             S ← S ∪ {node}
17:             e ← e ∪ {(nextARTnode, node)}
18:             if child ∈ E and CHECKREFINEERROR(node) then
19:                 report "ERROR"
20:             end if
21:             WorkList.add(⟨child, node⟩)
22:             WorkList.removeAll(nodes from WorkList subsumed by node)
23:         end for
24:     end while
25: end function
```

Fig. 2. The CEGAAR algorithm (a) - High-Level Structure

predicates, or it is subsumed by another node in the ART (line 12). In the algorithm (Fig. 2), we denote logical entailment by $\phi \vdash \psi$ in order to avoid confusion.

The refinement step is performed by the CHECKREFINEERROR function (Fig. 3). This function returns true if and only if a feasible error trace has been detected; otherwise, further predicates are generated to refine the abstraction. First, a minimal infeasible ART path to *node* is determined (line 4). This path is generalized into a trace scheme (line 5). The generalization function FOLD takes *Path* and the delay parameter δ as input and produces a trace scheme which contains *Path*. The trace scheme is obtained by traversing the path and recording the control states encountered in a list. When we encounter a control state which is already in the list, we identified an elementary cycle λ. If the current trace scheme ends with at least δ occurrences of λ, then λ is added as a loop to the trace scheme, provided that its transitive closure can be effectivelly computed. The latter condition can be ensured by verifying that the relation

```
 1: function CHECKREFINEERROR(node)
 2:     traceScheme ← []
 3:     while the ART path Root→···→node is spurious do
 4:         let Path ← ⟨q1,Φ1⟩→...→⟨qn,Φn⟩ be the (unique)
                minimal ART path with pivot ← ⟨q1,Φ1⟩ and ⟨qn,Φn⟩ = node
                such that the CFG path q1→···→qn is infeasible
 5:         newScheme ← FOLD(Path,delay)
 6:         if !ISBOUNDED(newScheme) then
 7:             absScheme ← CONCAT(OVERAPPROX(newScheme),traceScheme)
 8:             if INTERPOLATEREFINE(absScheme,pivot) then
 9:                 return false
10:             else
11:                 newScheme ← UNDERAPPROX(newScheme,Path)
12:             end if
13:         end if
14:         traceScheme ← CONCAT(newScheme,traceScheme)
15:         if INTERPOLATEREFINE(traceScheme,pivot) then
16:             return false
17:         end if
18:         node ← Path.head()
19:     end while
20:     return true
21: end function
```

Fig. 3. The CEGAAR algorithm (b) - Accelerated Refinement

labeling λ is syntactically compliant[7] to either Definition 1 or 2. For space reasons, the pseudo-code of the FOLD functions are given in Appendix B. Once the folded trace scheme is obtained, there are three possibilities:

1. If the trace scheme is not bounded (the test on line 6 passes), we compute a bounded overapproximation of it, in an attempt to prove its infeasibility (line 7). If the test on line 8 succeeds, the original trace scheme is proved to be infeasible and the ART is refined using the interpolants for the overapproximated trace scheme.
2. Else, if the overapproximation was found to be feasible, it could be the case that the abstraction of the scheme introduced a spurious error trace. In this case, we compute a bounded underapproximation of the trace scheme, which contains the initial infeasible path, and replace the current trace scheme with it (line 11). The only requirement we impose on the UNDERAPPROX function is that the returned bounded trace scheme contains *Path*, and is a subset of *newScheme*.
3. Finally, if the trace scheme is bounded (either because the test on line 6 failed, or because the folded path was replaced by a bounded underapproximation on line 11) and also infeasible (the test on line 15 passes) then the ART is refined with the interpolants computed for the scheme. If, on the other hand, the scheme is feasible,

[7] Notice that a relation can be definable by an octagonal constraint even if it is not a conjunction of the form given in Definition 1 – it may contain redundant atomic propositions which are not of this form. Our check is a sufficient, but not necessary condition.

Fig. 4. Underapproximation of unbounded trace schemes. ε stands for the identity relation.

we continue searching for an infeasible trace scheme starting from the head of *Path* upwards (line 18).

Example Let $\theta : q_1 \xrightarrow{P} q_2 \xrightarrow{Q} q_2 \xrightarrow{R} q_1 \xrightarrow{P} q_2 \xrightarrow{R} q_1$ be a path. The result of applying FOLD to this path is the trace scheme ξ shown in the left half of Fig. 4. Notice that this path scheme is not bounded, due to the presence of two loops starting and ending with q_2. A possible bounded underapproximation of ξ, containing the original path θ, is shown in the right half of Fig. 4. □

The iteration stops either when a refinement is possible (lines 9, 16), in which case CHECKREFINEERROR returns false, or when the search reaches the root of the ART and the trace scheme is feasible, in which case CHECKREFINEERROR returns true (line 20) and the main algorithm in Figure 2 reports a true counterexample. Notice that, since we update *node* to the head of *Path* (line 18), the position of *node* is moved upwards in the ART. Since this cannot happen indefinitely, the main loop (lines 3-19) of the CHECKREFINEERROR is bound to terminate.

The INTERPOLATEREFINE function is used to compute the interpolant of the trace scheme, update the predicate mapping π of the ART, and reconstruct the subtree of the ART whose root is the first node on *Path* (this is usually called the *pivot* node). For space reasons, the INTERPOLATEREFINE function is shown in Appendix B.

It can be observed that our procedure is *sound,* in the sense that whenever function CONSTRUCTART terminates with a non-error result, the input program does not contain any reachable error states. Vice versa, if a program contains a reachable error state, CONSTRUCTART is guaranteed to eventually discover a feasible path to this state, since the use of a work list ensures fairness when exploring ARTs.

5 Computing Accelerated Interpolants

This section describes a method of refining an ART by excluding an infinite family of infeasible traces at once. Our method combines interpolation with acceleration in a way which is oblivious of the particular method used to compute interpolants. For instance, it is possible to combine proof-based [27] or constraint-based [31] interpolation with acceleration, whenever computing the precise transitive closure of a loop is possible. In cases when the precise computation fails, we may resort to both over- and under-approximation of the transitive closure. In both cases, the accelerated interpolants are at least as general (and many times more general) than the classical interpolants extracted from a finite counterexample trace.

5.1 Precise Acceleration of Bounded Trace Schemes

We consider first the case of bounded trace schemes of the form (1), where the control states $q_1,\ldots,q_n$ belong to some cycles labeled with relations $L_1,\ldots,L_n$. Under some restrictions on the syntax of the relations labeling the cycles L_i, the reflexive transitive closures L_i^* are effectively computable using acceleration algorithms [8,11,17]. Among the known classes of relations for which acceleration is possible we consider: *octagonal relations* (Definition 1) and *finite monoid affine transformations* (Definition 2). These are all conjunctive linear relations. We consider in the following that all cycle relations L_i belong to one of these classes. Under this restriction, any infeasible bounded trace scheme has an effectivelly computable interpolant of one of the forms (2),(3).

However, there are two problems with applying definitions (2),(3) in order to obtain interpolants of trace schemes. On one hand, relational composition typically requires expensive quantifier eliminations. The standard proof-based interpolation techniques (e.g. [27]) overcome this problem by extracting the interpolants directly from the proof of infeasibility of the trace. Alternatively, constraint-based interpolation [31] reduce the interpolant computation to a Linear Programming problem, which can be solved by efficient algorithms [32]. Both methods apply, however, only to finite traces, and not to infinite sets of traces defined as trace schemes. Another, more important, problem is related to the sizes of the interpolant predicates from (2), (3) compared to the sizes of interpolant predicates obtained by proof-theoretic methods (e.g. [26]), as the following example shows.

Example Let $R(x,y,x',y') : x' = x+1 \wedge y' = y+1$ and $\phi(x,y,\ldots)$, $\psi(x,y,\ldots)$ be some complex Presburger arithmetic formulae. The trace scheme:

$$q_0 \xrightarrow{z=0\wedge z'=z\wedge\phi} \overset{z'=z+2\wedge R}{\underset{\frown}{q_1}} \xrightarrow{z=5\wedge\psi} q_2 \tag{4}$$

is infeasible, because z remains even, so it cannot become equal 5. One simple interpolant for this trace scheme has at program point q_1 the formula $z\%2 = 0$. On the other hand, the strongest interpolant has $(z = 0 \wedge z' = x \wedge \phi) \circ (z' = z+2 \wedge R)^*$ at q_1, which is typically a much larger formula, because of the complex formula ϕ. Note however that ϕ and R do not mention z, so they are irrelevant. □

To construct useful interpolants instead of the strongest or the weakest ones, we therefore proceed as follows. Let ξ be a bounded trace scheme of the form (1). For each control loop $q_i \xrightarrow{R_i} q_i$ of ξ, we define the corresponding *meta-transition* $q_i' \xrightarrow{R_i^*} q_i''$ labeled with the reflexive and transitive closure of R_i. Intuitively, firing the meta-transition has the same effect as iterating the loop an arbitrary number of times. We first replace each loop of ξ by the corresponding meta-transition. The result is the *meta-trace*:

$$\overline{\xi} : q_0 \xrightarrow{Q_1} q_1' \xrightarrow{L_1^*} q_1'' \xrightarrow{Q_2} q_2' \ \ldots\ q_{n-1}'' \xrightarrow{Q_n} q_n' \xrightarrow{L_n^*} q_n'' \xrightarrow{Q_{n+1}} q_{n+1} \tag{5}$$

Since we supposed that ξ is an infeasible trace scheme, the (equivalent) finite meta-trace $\overline{\xi}$ is infeasible as well, and it has an interpolant $I_{\overline{\xi}} = \langle \top, I_1', I_1'', I_2', I_2'', \ldots, I_n', I_n'', \bot \rangle$ in the sense of Definition 3. This interpolant is not an interpolant of the trace scheme ξ, in the sense of Definition 5. In particular, none of I_i', I_i'' is guaranteed to be inductive

with respect to the loop relations L_i. To define compact inductive interpolants based on $I_{\overline{\xi}}$ and the transitive closures L_i^*, we consider the following sequences:

$$I_\xi^{post} = \langle \top, post(I_1', L_1^*), post(I_2', L_2^*), \ldots, post(I_n', L_n^*), \bot \rangle$$
$$I_\xi^{wpre} = \langle \top, wpre(I_1'', L_1^*), wpre(I_2'', L_2^*), \ldots, wpre(I_n'', L_n^*), \bot \rangle$$

The following lemma proves the correctness of this approach.

Lemma 7. *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG and ξ be an infeasible trace scheme of the form (1). Then I_ξ^{post} and I_ξ^{wpre} are interpolants for ξ, and moreover $I_{\xi_i}^{wpre} \rightarrow I_{\xi_i}^{post}$, for all $i = 1, 2, \ldots, n$.*

Notice that computing I_ξ^{post} and I_ξ^{wpre} requires n relational compositions, which is, in principle, just as expensive as computing directly one of the extremal interpolants (2),(3). However, by re-using the meta-trace interpolants, one potentially avoids the worst-case combinatorial explosion in the size of the formulae, which occurs when using (2), (3) directly.

Example Let us consider again the trace scheme (4). The corresponding unfeasible finite trace $\overline{\xi}$ is:

$$q_0 \xrightarrow{z=0 \wedge z'=z \wedge \phi} q_1' \xrightarrow{\exists k \geq 0 \,.\, z'=z+2k \,\wedge\, x'=x+k \,\wedge\, y'=y+k} q_1'' \xrightarrow{z=5 \wedge \psi} q_2$$

A possible interpolant for this trace is $\langle \top, z = 0, \exists k \geq 0 \,.\, z = 2k, \bot \rangle$. An inductive interpolant for the trace scheme, derived from it, is $I_\xi^{post} = \langle \top, post(z = 0, \exists k \geq 0 . z' = z + 2k \wedge x' = x + k \wedge y' = y + k), \bot \rangle = \langle \top,\ z\%2 = 0,\ \bot \rangle$. □

5.2 Bounded Overapproximations of Trace Schemes

Consider a trace scheme (1), not necessarily bounded, where the transitive closures of the relations L_i labeling the loops are not computable by any available acceleration method [8, 11, 17]. One alternative is to find abstractions $L_i^\sharp$ of the loop relations, i.e. relations $L_i^\sharp \leftarrow L_i$, for which transitive closures are computable. If the new abstract trace remains infeasible, it is possible to compute an interpolant for it, which is an interpolant for the original trace scheme. However, replacing the relations L_i with their abstractions $L_i^\sharp$ may turn an infeasible trace scheme into a feasible one, where the traces introduced by abstraction are spurious. In this case, we give up the overapproximation, and turn to the underapproximation technique described in the next section.

The overapproximation method computes an interpolant for a trace scheme ξ of the form (1) under the assumption that the abstract trace scheme:

$$\xi^\sharp : q_0 \xrightarrow{Q_1} \overset{L_1^\sharp}{\curvearrowright} q_1 \xrightarrow{Q_2} \ldots \xrightarrow{Q_{n-1}} \overset{L_{n-1}^\sharp}{\curvearrowright} q_{n-1} \xrightarrow{Q_n} \overset{L_n^\sharp}{\curvearrowright} q_n \xrightarrow{Q_{n+1}} q_{n+1} \tag{6}$$

is infeasible. In this case one can effectivelly compute the interpolants $I_{\xi^\sharp}^{post}$ and $I_{\xi^\sharp}^{wpre}$, since the transitive closures of the abstract relations labeling the loops are computable by acceleration. The following lemma proves that, under certain conditions, computing an interpolant for the abstraction of a trace scheme is sound.

Lemma 8. *Let G be a CFG and ξ be a trace scheme (1) such that the abstract trace scheme $\xi^\sharp$ (6) is infeasible. Then the interpolants $I^{post}_{\xi^\sharp}$ and $I^{wpre}_{\xi^\sharp}$ for $\xi^\sharp$ are also interpolants for ξ.*

To compute abstractions of relations that are guaranteed to have Presburger-definable transitive closures, we can use octagonal relations (Definition 1) and compute the *integer octagonal hull* of a relation L. This is the strongest conjunction $L^\sharp = \bigwedge\{u \leq c \mid u \in U(\mathbf{x} \cup \mathbf{x}'),\ L \rightarrow u \leq c\}$ In practice, if for instance, L is a union of convex polyhedra, one can use Integer Linear Programming [32] to compute $L^\sharp$ efficiently.

5.3 Bounded Underapproximations of Trace Schemes

Let ξ be a trace scheme of the form (1), where each relation L_i labeling a loop is a disjunction $L_{i1} \vee \ldots \vee L_{ik_i}$ of relations for which the transitive closures are effectively computable and Presburger definable. A *bounded underapproximation scheme* of a trace scheme ξ is obtained by replacing each loop $q_i \xrightarrow{L_i} q_i$ in ξ by a bounded trace scheme of the form:

$$\overset{L_{i1}}{\curvearrowright} \quad \overset{L_{i2}}{\curvearrowright} \qquad \overset{L_{ik_i}}{\curvearrowright}$$
$$q_i^1 \xrightarrow{\varepsilon} q_i^2 \xrightarrow{\varepsilon} \ldots\ q_i^{k_i}$$

where ε denotes the identity relation. Let us denote[8] the result of this replacement by $\xi^\flat$. It is manifest that the set of traces $\xi^\flat$ is included in ξ.

Since we assumed that the reflexive and transitive closures L_{ij}^* are effectivelly computable and Presburger definable, the feasibility of $\xi^\flat$ is a decidable problem. If $\xi^\flat$ is found to be feasible, this points to a real error trace in the system. On the other hand, if $\xi^\flat$ is found to be infeasible, let $I_{\xi^\flat} = \langle \top, I_1^1, \ldots, I_1^{k_1}, \ldots, I_n^1, \ldots, I_n^{k_n}, \bot \rangle$ be an interpolant for $\xi^\flat$. A refinement scheme using this interpolant associates the predicates $\{I_i^1, \ldots, I_i^{k_i}\}$ with the control state q_i from the original CFG. As the following lemma shows, this guarantees that any trace that follows the pattern of $\xi^\flat$ is excluded from the ART, ensuring that a refinement of the ART using a suitable underapproximation (that includes a spurious counterexample) is guaranteed to make progress.

Lemma 9. *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E \rangle$ be a CFG, ξ be an infeasible trace scheme of G (1) and $\xi^\flat$ a bounded underapproximation of ξ. If $T = \langle S, \pi, r, e \rangle$ is an ART for G, such that $\{I_i^1, \ldots, I_i^{k_i}\} \subseteq \pi(q_i)$, then no path in T concretizes to a trace in $\xi^\flat$.*

Notice that a refinement scheme based on underapproximation guarantees the exclusion of those traces from the chosen underapproximation trace scheme, and not of all traces from the original trace scheme. Since a trace scheme is typically obtained from a finite counterexample, an underapproximation-based refinement still guarantees that the particular counterexample is excluded from further searches. In other words, using underapproximation is still better than the classical refinement method, since it can potentially exclude an entire family of counterexamples (including the one generating the underapproximation) at once.

[8] The choice of the name depends on the ordering of particular paths $L_{i1}, L_{i2}, \ldots, L_{ik_i}$, however we shall denote any such choice in the same way, in order to keep the notation simple.

6 Experimental Results

We have implemented CEGAAR by building on the predicate abstraction engine Eldarica[9], the FLATA verifier[10] based on acceleration, and the Princess interpolating theorem prover [13,30]. Tables in Figure 5 compares the performance of the Flata, Eldarica, *static acceleration* and CEGAAR on a number of benchmarks.

The benchmarks are all in the Numerical Transition Systems format[11] (NTS). We have considered seven sets of examples, extracted automatically from different sources: (a) C programs with arrays provided as examples of divergence in predicate abstraction [25], (b) verification conditions for programs with arrays, expressed in the SIL logic of [11] and translated to NTS, (c) small C programs with challenging loops, (d) NTS extracted from programs with singly-linked lists by the L2CA tool [9], (e) C programs provided as benchmarks in the NECLA static analysis suite, (f) C programs with asynchronous procedure calls translated into NTS using the approach of [19] (the examples with extension .optim are obtained via an optimized translation method [18]), and (g) models extracted from VHDL models of circuits following the method of [33]. The benchmarks are available from the home page of our tool.

The results on this benchmark set suggest that we have arrived at a fully automated verifier that is robust in verifying automatically generated integer programs with a variety of looping control structure patterns.

An important question we explored is the importance of dynamic application of acceleration, as well as of overapproximation and underapproximation. We therefore also implemented static acceleration [14], a lightweight acceleration technique generalizing large block encoding (LBE) [5] with transitive closures. It simplifies the control flow graph prior to predicate abstraction. In some cases, such as mergesort from the (d) benchmarks and rotation_vc.2 from (b) benchmarks, the acceleration overhead does not pay off. The problem is that static acceleration tries to accelerate every loop in the CFG rather than accelerating the loops occurring on spurious paths leading to error. Acceleration of inessential loops generates large formulas as the result of combining loops and composition of paths during large block encoding.

The CEGAAR algorithm outperforms other approaches in most cases. In the verification of benchmarks totally 11 times the acceleration was exact, 30 times the overapproximation of the loops was successful and in 15 cases over-approximation fails so the tool had to use the under-approximation tactic. This suggests that all techniques that we presented are essential to obtain an effective verifier.

7 Conclusions

We have presented CEGAAR, a new automated verification algorithm for integer programs. The algorithm combines two cutting-edge analysis techniques: interpolation-based abstraction refinement and acceleration of loops. We have implemented CEGAAR and presented experimental results, showing that CEGAAR handles robustly

[9] http://lara.epfl.ch/w/eldarica

[10] http://www-verimag.imag.fr/FLATA.html

[11] http://richmodels.epfl.ch/ntscomp_ntslib

Model	Time [s] F.	E.	S.	D.
(a) Examples from [25]				
anubhav (C)	0.6	1.5	1.8	1.5
copy1 (E)	1.7	8.1	1.2	3.5
cousot (C)	0.5	-	-	4.3
loop1 (C)	0.4	2.1	0.9	2.1
loop (C)	0.4	0.3	0.9	0.3
scan (E)	2.4	-	1.0	2.9
string_concat1 (E)	4.4	-	3.2	5.0
string_concat (E)	4.1	-	2.5	4.2
string_copy (E)	3.7	-	1.5	3.6
substring1 (E)	0.3	1.6	23.9	1.5
substring (E)	1.8	0.6	1.6	0.6
(b) Verification conditions for array programs [11]				
rotation_vc.1 (C)	0.7	2.0	6.3	1.9
rotation_vc.2 (C)	1.3	2.1	202.2	2.1
rotation_vc.3 (C)	1.2	0.3	181.5	0.3
rotation_vc.1 (E)	1.1	1.4	14.9	1.4
split_vc.1 (C)	4.2	2.7	-	2.7
split_vc.2 (C)	2.8	2.1	-	2.1
split_vc.3 (C)	2.9	0.5	-	0.5
split_vc.1 (E)	30.6	2.0	-	2.0

Model	Time [s] F.	E.	S.	D.
(c) Examples from [28]				
boustrophedon (C)	-	-	-	12.2
gopan (C)	0.5	-	-	6.7
halbwachs (C)	-	-	1.6	8.2
rate_limiter (C)	-	7.2	2.7	7.1
(d) Examples from L2CA [9]				
bubblesort (E)	13.1	2.5	3.0	2.5
insdel (E)	0.1	0.2	0.8	0.2
insertsort (E)	1.9	0.7	1.4	0.7
listcounter (C)	0.3	-	0.5	3.4
listcounter (E)	0.3	0.3	0.5	0.3
listreversal (C)	4.8	0.6	1.7	0.6
listreversal (E)	0.6	0.6	4.8	0.6
mergesort (E)	1.1	1.5	237.6	1.5
selectionsort (E)	1.4	1.3	4.4	1.3
(e) NECLA benchmarks				
inf1 (E)	0.1	0.3	0.3	0.3
inf4 (E)	0.8	0.5	0.5	0.5
inf6 (C)	0.1	0.3	0.3	0.3
inf8 (C)	0.3	0.6	0.6	0.6

Model	Time [s] F.	E.	S.	D.
(f) Examples from [19]				
h1 (E)	-	6.3	6.7	6.0
h1.optim (E)	0.7	1.2	2.3	1.3
h1h2 (E)	-	19.8	20.3	18.7
h1h2.optim (E)	1.1	4.2	4.6	4.2
simple (E)	-	-	6.1	6.1
simple.optim (E)	0.7	1.3	2.3	1.3
test0 (C)	-	29.7	28.9	28.3
test0.optim (C)	0.2	5.1	1.6	5.1
test0 (E)	-	5.6	5.9	5.8
test0.optim (E)	0.6	1.3	2.1	1.3
test1.optim (C)	0.7	8.3	9.4	10.1
test1.optim (E)	1.4	6.9	7.6	6.9
test2_1.optim (E)	1.4	4.7	3.8	4.7
test2_2.optim (E)	2.5	4.7	3.8	4.7
test2.optim (C)	6.2	79.5	72.7	65.5
wrpc.manual (C)	0.5	1.2	1.3	1.2
wrpc (E)	-	10.8	11.2	10.7
wrpc.optim (E)	-	3.0	5.4	3.0
(g) VHDL models from [33]				
counter (C)	0.1	1.6	1.6	1.6
register (C)	0.2	1.2	1.2	1.2
synlifo (C)	17.0	7.2	7.3	7.2

Fig. 5. Benchmarks for **F**lata, **E**ldarica without acceleration, Eldarica with acceleration of loops at the CFG level (**S**tatic) and CEGAAR (**D**ynamic acceleration). The letter after the model name distinguishes **C**orrect from models with a reachable **E**rror state. Items with "-" led to a timeout for the respective approach.

a number of examples that cannot be handled by predicate abstraction or acceleration alone. Because many classes of systems translate into integer programs, our advance contributes to automated verification of infinite-state systems in general.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2), February 2011.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN PLDI*, 2001.
3. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS'02*, volume 2280 of *LNCS*, page 158, 2002.
4. S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Form. Methods Syst. Des.*, 15(1):75–92, July 1999.
5. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, 2009.
6. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.
7. R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic bound computation for loops. In *LPAR (Dakar)*, pages 103–118, 2010.
8. B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*, volume PhD Thesis, Vol. 189. Collection des Publications de l'Université de Liège, 1999.
9. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, pages 517–531, 2006.

10. M. Bozga, C. Gîrlea, and R. Iosif. Iterating octagons. In *TACAS '09*, pages 337–351. Springer, 2009.
11. M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, pages 157–172, 2009.
12. M. Bozga, R. Iosif, and F. Konečný. Fast acceleration of ultimately periodic relations. In *CAV*, pages 227–242, 2010.
13. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *IJCAR*, LNCS. Springer, 2010.
14. N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun. Accelerating interpolation-based model-checking. In *TACAS'08*, pages 428–442, 2008.
15. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, September 1957.
16. J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. *JSAT*, 5(1-4):27–56, 2008.
17. A. Finkel and J. Leroux. How to compose presburger-accelerations: Applications to broadcast protocols. In *FST TCS '02*, pages 145–156. Springer, 2002.
18. P. Ganty. Personal communication.
19. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010.
20. S. Ginsburg and E. Spanier. Bounded algol-like languages. *Trans. of the AMS*, 113(2):333–368, 1964.
21. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.
23. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, 2006.
24. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.
25. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
26. D. Kroening, J. Leroux, and P. Rümmer. Interpolating quantifier-free Presburger arithmetic. In *Proceedings, LPAR*, volume 6397 of *LNCS*, pages 489–503. Springer, 2010.
27. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1), 2005.
28. D. Monniaux. Personal Communication.
29. M. Presburger. Über die Vollstandigkeit eines gewissen Systems der Arithmetik. *Comptes rendus du I Congrés des Pays Slaves*, Warsaw 1929.
30. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.
31. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *Proceedings, VMCAI*, volume 4349 of *LNCS*, pages 346–362. Springer, 2007.
32. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.
33. A. Smrcka and T. Vojnar. Verifying parametrised hardware designs via counter automata. In *Haifa Verification Conference*, pages 51–68, 2007.

A Proofs

Lemma 10 ([16]). *A trace* $\theta : q_1 \xrightarrow{R_1} q_2 \xrightarrow{R_2} q_3 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$ *is infeasible if and only if it has an interpolant. In this case, both sequences* $\langle post(\top, R_1 \circ \ldots \circ R_{i-1})\rangle_{i=1}^{n}$ *and*

$\langle wpre(\bot, R_i \circ \ldots R_{n-1})\rangle_{i=1}^{n}$ are interpolants, and for any interpolant $\langle I_i\rangle_{i=1}^{n-1}$ we have $post(\top, R_1 \circ \ldots \circ R_{i-1}) \rightarrow I_i \rightarrow wpre(\bot, R_i \circ \ldots R_{n-1})$ for $i = 1, \ldots, n$.

Lemma 11. *Let $G = \langle \mathbf{x}, Q, \rightarrow, I, E\rangle$ be a CFG and $\theta : q_1 \xrightarrow{R_1} q_2 \ldots q_{n-1} \xrightarrow{R_{n-1}} q_n$ be an infeasible trace of G. If $T = \langle S, \pi, r, e\rangle$ is an ART for G such that there exists an interpolant $\langle I_i \in \pi(q_i)\rangle_{i=1}^{n}$ for θ, then no path in T concretizes to θ.*

Proof. By contradiction, suppose that there exists a path

$$\sigma \; : \; \langle q_1, \Phi_1\rangle, \langle q_2, \Phi_2\rangle, \ldots, \langle q_n, \Phi_n\rangle$$

in T, that concretizes to θ. We show by induction on i, that $I_i \in \Phi_i$, for all $i = 1, \ldots, n$. By the definition of T, $I_1 = \top \in \Phi_1$, always. For the induction step, assume that $I_{i-1} \in \Phi_{i-1}$, for some $i > 1$. By the definition of T we have $\Phi_i = \{P \in \pi(q_i) \mid post(\bigwedge \Phi_{i-1}, R_i) \rightarrow P\}$. Since $post(I_{i-1}, R_i) \rightarrow I_i$, by Definition 3 and $I_{i-1} \in \Phi_{i-1}$, we have $\bigwedge \Phi_{i-1} \rightarrow I_{i-1}$, and by monotonicity of the *post* operator, $post(\bigwedge \Phi_{i-1}, R_i) \rightarrow I_i$. But $I_i \in \pi(q_i)$ which implies $I_i \in \Phi_i$, by the definition of T. Consequently $I_n = \bot \in \Phi_n$, which is in contradiction with the fact that no node in T may contain $\bot$ in its second component. □

Proof of Lemma 6: By contradiction, suppose that there exists a path σ:

$$\langle q_0, \Phi_0\rangle,\ \langle q_{11}, \Phi_{11}\rangle, \ldots, \langle q_{1i_1}, \Phi_{1i_1}\rangle,\ \ldots,\ \langle q_{n1}, \Phi_{n1}\rangle, \ldots, \langle q_{ni_n}, \Phi_{ni_n}\rangle,\ \langle q_{n+1}, \Phi_{n+1}\rangle\rangle$$

in T which concretizes to a trace in ξ. In analogy with the proof of Lemma 11, one shows that:

- $I_0 \in \Phi_0$
- $I_k \in \Phi_{kj}$, for all $k = 1, \ldots, n$ and $j = 1, \ldots, i_k$
- $I_{n+1} \in \Phi_{n+1}$

The third condition of Definition 5 is needed for the proof of the second point above. Since $I_{n+1} = \bot$, this contradicts the fact that no node in T may contain $\bot$ in its second component. □

Proof of Lemma 7: To prove that I_ξ^{post} is an interpolant for ξ, we show the three points of Definition 5. The first point holds by the construction of I_ξ^{post}. For the second point, we have:

$$\begin{array}{rl} post(I_i', L_i^*) \rightarrow I_i'' & \text{, since } I_{\overline{\xi}} \text{ is an interpolant for } \overline{\xi} \\ post(post(I_i', L_i^*), Q_{i+1}) \rightarrow post(I_i'', Q_{i+1}) & \text{, since } post \text{ is monotone} \\ post(I_{\xi_i}^{post}, Q_{i+1}) \rightarrow I_{i+1}' & \text{, since } I_{\overline{\xi}} \text{ is an interpolant for } \overline{\xi} \end{array}$$

We must show next that $I_{i+1}' \rightarrow I_{\xi_{i+1}}^{post}$. For this, we compute:

$$\begin{array}{rl} post(I_{i+1}', L_{i+1}^*) & = \exists \mathbf{z} \,.\, I_{i+1}'(\mathbf{z}) \wedge L_{i+1}^*(\mathbf{z}, \mathbf{x}) \\ & = \exists \mathbf{z} \,.\, I_{i+1}'(\mathbf{z}) \wedge \bigvee_{k=0}^{\infty} L_{i+1}^k(\mathbf{z}, \mathbf{x}) \\ & = \bigvee_{k=0}^{\infty} \exists \mathbf{z} \,.\, I_{i+1}'(\mathbf{z}) \wedge L_{i+1}^k(\mathbf{z}, \mathbf{x}) \\ & = \exists \mathbf{z} \,.\, I_{i+1}'(\mathbf{z}) \wedge \varepsilon \;\vee\; \bigvee_{k=1}^{\infty} \exists \mathbf{z} \,.\, I_{i+1}'(\mathbf{z}) \wedge L_{i+1}^k(\mathbf{z}, \mathbf{x}) \end{array}$$

We have that $\exists \mathbf{z} \,.\, I'_{i+1}(\mathbf{z}) \wedge \varepsilon$ is equivalent to I'_{i+1}, which concludes the second point. For the third point, we compute:

$$\begin{aligned} post(I^{post}_{\xi_i}, L_i) &= \exists \mathbf{z} \,.\, post(I'_i, L^*_i)(\mathbf{z}) \wedge L_i(\mathbf{z}, \mathbf{x}) \\ &= \exists \mathbf{z} \exists \mathbf{t} \,.\, I'_i(\mathbf{t}) \wedge L^*_i(\mathbf{t}, \mathbf{z}) \wedge L_i(\mathbf{z}, \mathbf{x}) \\ &= \exists \mathbf{t} \,.\, I'_i(\mathbf{t}) \wedge L^+_i(\mathbf{t}, \mathbf{x}) \\ &\rightarrow \exists \mathbf{t} \,.\, I'_i(\mathbf{t}) \wedge L^*_i(\mathbf{t}, \mathbf{x}) \\ &= post(I'_i, L^*_i) = I^{post}_{\xi_i} \end{aligned}$$

The proof for the I^{wpre}_{ξ} interpolant is symmetric, using the fact that *post* and *wpre* form a Galois connection. Finally, we have $wpre(I''_i, L^*_i) \rightarrow I'_i \rightarrow post(I'_i, L^*_i)$ which proves the last statement. □

Proof of Lemma 8: We show that $I^{post}_{\xi^\sharp}$ meets the three conditions of Definition 5. The first condition is trivially true, while the proof of the second condition is essentially the same as in the proof of Lemma 7. For the third point, since $L_i \rightarrow L^\sharp_i$, we have:

$$\begin{aligned} post(I^{post}_{\xi^\sharp}, L_i) &= post(post(I'_i, L^{\sharp *}_i), L_i) \\ &\rightarrow post(post(I'_i, L^{\sharp *}_i), L^\sharp_i) \\ &= post(I'_i, L^{\sharp +}_i) \rightarrow I^{post}_{\xi^\sharp} \end{aligned}$$

The proof for $I^{wpre}_{\xi^\sharp}$ is symmetrical. □

Proof of Lemma 9: By contradiction, suppose that there exists a path in T which concretizes to a trace in $\xi^\flat$, and let

$$\underbrace{\langle q_i, \Phi^1_{i1}\rangle, \ \ldots, \ \langle q_i, \Phi^1_{i\ell_{i,1}}\rangle}_{\overset{L_{i1}}{\curvearrowright} q_i}, \ldots, \underbrace{\langle q_i, \Phi^{k_i}_{i1}\rangle, \ \ldots, \ \langle q_i, \Phi^{k_i}_{i\ell_{i,k_i}}\rangle}_{\overset{L_{ik_i}}{\curvearrowright} q_i}$$

be the fragment of the path which corresponds to the unfolding of the sub-trace:

$$\overset{L_{i1}}{\curvearrowright} q_i \xrightarrow{\varepsilon} \overset{L_{i2}}{\curvearrowright} q_i \xrightarrow{\varepsilon} \ldots \overset{L_{ik_i}}{\curvearrowright} q_i$$

One can show, among the lines of the proof of Lemma 7, that $I^j_i \in \Phi^j_{i\ell}$, for all $j = 1, \ldots, k_i$ and $\ell = 1, \ldots, \ell_{i,j}$. In this way, we obtain that the last set Φ contains $\bot$, which contradicts the definition of the ART. □

B Algorithms

The FOLD function (Fig. 6) creates a trace scheme of the form (1) out of the spurious path given as argument. The spurious path is traversed and control states are recorded in a list. When we encounter a control state which is already in the list, we identified an

elementary cycle λ. If the current trace scheme ends with at least δ occurrences of λ (line 9), where $\delta \in \mathbb{N}_\infty$ is the delay parameter, then λ is added as a loop to the trace scheme, provided that its transitive closure can be effectivelly computed. For efficiency reasons, the ISACCELERABLE function is implemented as a sufficient syntactic check of the relation on the loop, namely it is checked whether the relation is syntactically compliant to Definition 1. Notice that a relation can be definable by an octagonal constraint even if it is not a conjunction of the form given in Definition 1 – it may contain redundant atomic propositions which are not of this form.

```
 1: function FOLD(path, delay)
 2:     Scheme, List ← []
 3:     for q ← path do
 4:         if q ∉ List then
 5:             List.add(q)
 6:         else
 7:             Scheme.addPath(List.prefixUntil(q))
 8:             loop ← List.suffixUntil(q)
 9:             if Scheme.endsWith(loop^delay) and ISACCELERABLE(loop) then
10:                 Scheme.remove(loop^delay)
11:                 Scheme.addLoop(loop)
12:             else
13:                 Scheme.addPath(loop).
14:             end if
15:             List ← [q]
16:         end if
17:     end for
18:     return Scheme
19: end function
```

Fig. 6. The Folding Function

.

The INTERPOLATEREFINE (Fig. 7) function returns true if and only if its argument represents an infeasible trace scheme. In this case, new predicates, obtained from the interpolant of the trace scheme, are added to the nodes of the ART. This function uses internally the TRANSITIVECLOSURE procedure (line 2) in order to generate the meta-trace scheme (5). The ACCELERATEINTERPOLANT function (line 7) computes the interpolant for the trace scheme, from the resulting meta-trace scheme. Notice that the refinement algorithm is recursive, as CONSTRUCTART calls CHECKREFINEERROR (line 15), which in turn calls INTERPOLATEREFINE (lines 5,8,12), which calls back CONSTRUCTART (line 12).

```
1:  function INTERPOLATEREFINE(traceScheme, Pivot)
2:      Let metaTrace ← TRANSITIVECLOSURE(traceScheme)
3:      Let interpolant ← INTERPOLATINGPROVERCALL(metaTrace)
4:      if interpolant = ∅ then
5:          return false
6:      end if
7:      Let I ← ACCELERATEINTERPOLANT(interpolant)
8:      for ψ ← I do
9:          Let v be the CFG vertex corresponding to ψ
10:         π ← π[v ← (π(v) ∪ ψ)]
11:     end for
12:     CONSTRUCTART(Pivot, Pivot.abstraction)
13:     return true
14: end function
```

Fig. 7. The Interpolation Function

.