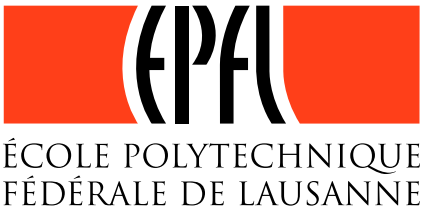
Path Invariants

Dirk Beyer Thomas A. Henzinger
Rupak Majumdar Andrey Rybalchenko



Technical Report No. MTC-REPORT-2006-003
December 22, 2006

Ecole Polytechnique Fédérale de Lausanne
Faculté Informatique & Communications
CH-1015 Lausanne, Switzerland

Path Invariants

Dirk Beyer
SFU

Thomas A. Henzinger
EPFL

Rupak Majumdar
UCLA

Andrey Rybalchenko
EPFL and MPI

Abstract

The success of software verification depends on the ability to find a suitable abstraction of a program automatically. We propose a new method for automated abstraction refinement, which overcomes the inherent limitations of predicate discovery schemes. In such schemes, the cause of a false positive is identified as an infeasible error path, and the abstraction is refined in order to remove that path. By contrast, we view the cause of a false positive —the "spurious counterexample"— as a full-fledged program, whose control-flow graph may contain loops of the original program and represent unbounded computations. The advantages of using such path programs *as counterexamples for abstraction refinement are twofold. First, we can bring the whole machinery of program analysis to bear on path programs: specifically, we use abstract interpretation in the form of constrained-based invariant generation to automatically infer invariants of path programs —so-called* path invariants*. Second, we use path invariants for abstraction refinement in order to remove not one infeasibility at a time, but to remove at once all infeasible error computations that are represented by a path program. Unlike predicate discovery schemes, our method handles loops without unrolling them; it infers abstractions that involve universal quantification and naturally incorporates disjunctive invariants.*

Keywords: Formal Verification, Software Model Checking, Predicate Abstraction, Abstraction Refinement, Invariant Synthesis

1. Introduction

Even the most experienced programmers make mistakes while programming, and they spend much time on testing their programs and fixing bugs. Although mature syntax and type checkers are available today, automatic proof- and bug-finding tools on the semantic level are required to produce robust and reliable code. Program verification has been a central topic of research since the early days of computer science. While it has long been known that *assertions* (program invariants) are the key to proving a program correct [21, 28], the available techniques for automatically proving useful assertions are still rather limited.

We can broadly classify the techniques for proving assertions into two categories. The first class of methods relies on the user to set up a verification framework —i.e., an *abstract interpretation* [14]— within which algorithms, often based on constraint solving, can efficiently search for program invariants. Examples of such verification frameworks include abstract domains (e.g., numerical [14], shapes [42]) and invariant templates (e.g., linear arithmetic [46], uninterpreted functions [3]). With these methods, much care must be spent on choosing, for a given program, a suitable framework which is both sufficiently expressive to limit the number of false alarms and sufficiently inexpensive to compute invariants efficiently.

More recently, an ambitious approach that originated within model checking [8] has been transferred to program verification [2, 27]. This approach, called *counterexample-guided abstraction refinement* (CEGAR), attempts to automatically tune the verification framework to the necessary degree of precision. In CEGAR, a false alarm —called a *counterexample*— is analyzed for information how to refine the abstract interpretation in order to remove the false alarm. This process is iterated until either a proof or a bug is found. The persuasive simplicity of CEGAR has also been its main limitation: a counterexample is an infeasible program path, and to remove that path one adds a predicate on program variables [9, 25] to be tracked by the abstract interpretation. However, a verification framework that consists solely of tracking predicates —i.e., a *predicate abstraction* [23]— is woefully inadequate for many applications. For example, loops are often unrolled iteration by iteration, only to find and remove longer and longer counterexamples. Common loops over arrays cannot be handled at all, as the invariant requires universal quantifiers (rather than quantifier-free predicates).

We overcome these limitations of CEGAR by generalizing the notion of counterexample. For us, a counterexample is not just a single infeasible program path, but a full-fledged program, namely, the smallest syntactic subprogram of the original program which produced the infeasibility. Such a program is called a *path program*. Since a path program may contain loops, it often represents not a single in-

feasibility, but a whole family of infeasibilities —all those obtained from unrolling the loops. Hence, by refining the abstraction in order to remove the counterexample, we remove many (potentially infinitely many) false alarms in one step. However, such a refinement may require more than the addition of a single predicate: in general, it requires the addition of a precise invariant for the path program —the so-called *path invariant*. Thus, instead of relying on heuristics for discovering relevant information about counterexamples, we can bring to bear the entire well-developed machinery for synthesizing program invariants.

Our method scales, because path programs —being small fragments of the original program— pose comparatively simple verification problems. In particular, since path programs contain no branching behavior, the explosive cost of disjunctive reasoning about invariants is avoided. In other words, our approach can be viewed as decomposing a program verification problem into a series of simpler, nondisjunctive problems about fragments of the original program. Each false alarm gives rise to a new counterexample in the form of a path program, and thus, to a new verification subproblem. These new subproblems are generated until either a bug or a proof for the original program is found.

While we are free to apply any program analysis to path programs, we use template-based invariant generation for the combined theories of linear arithmetic and uninterpreted functions [3] to derive invariants of path programs. This allows us to overcome two major limitations of previous CEGAR-based schemes. First, by synthesizing invariants for path programs with loops, we avoid the iterative unwinding of loops suffered by CEGAR tools like SLAM [2] or BLAST [30]. These approaches, by using finite paths as counterexamples, can never guarantee that the next counterexample would not be a simple variation of the current one, where some of the loops are traversed some more times. Path program-based refinement solve this problem.

Second, by synthesizing universally quantified assertions, we can handle a considerably larger class of programs, such as programs whose correctness depends on the contents of arrays. Again, by using finite paths as counterexamples, which look only at finite numbers of array cells, it is fundamentally impossible to make justified universally quantified statements that hold for an unbounded number of array indices. Path programs solve also this problem. We illustrate these two points, and the benefits of nondisjunctive reasoning about path programs, by three motivating examples in Section 2.

Summary. We propose a fundamentally new approach to counterexample-guided abstraction refinement, which does not consider finite program paths, but path programs as counterexamples. Path programs are full-fledged programs, performing possibly unbounded (looping) computations, but with a simple branching structure. From path programs we construct invariants, which in turn are used to refine the analysis of the original program. The resulting invariants eliminate all infeasible error paths that remain within the control-flow structure of the path program, e.g., by arbitrary unwinding of loops. Furthermore, by considering unbounded computations of path programs, unlike previous CEGAR-based methods, we can infer universally quantified invariants.

The invariant generation for path programs becomes the central task within our approach. Note that a path program exhibits only a small portion of the original program, which is controlled by the property of interest. Hence, invariant generation for path programs is easier than for the original program. We can apply existing methods and tools, e.g., abstract interpreters based on widening, or constraint-based invariant generation methods. The use of path programs as counterexamples shifts the focus from heuristics for discovering relevant information, to heuristics for efficiently discovering information (relevance is guaranteed).

Our approach combines the strengths of predicate abstraction and invariant generation. Predicate abstraction performs well for disjunctive reasoning, e.g., case analysis depending on aliasing between pointer variables, or boolean flags that control the program flow. Invariant generation, by contrast, is strong in arithmetic reasoning and capable of quantified reasoning. The method is modular, in that it can be easily integrated into existing CEGAR-based software model checkers. We simply need to replace the predicate discovery module by a call to an invariant synthesizer for path programs.

Related Work. Our work is a synthesis of two approaches to program analysis: counterexample-guided abstraction refinement and invariant synthesis. Our work unifies these approaches by generalizing counterexamples from paths (as they are usually formulated in CEGAR) to program fragments (*path programs*) on which we apply invariant synthesis techniques. As a result, we obtain a program analysis that can automatically generate richer relationships among program variables without paying the high cost of searching through the space of program invariants.

There has been a lot of recent interest in *predicate abstraction* based software model checking [23, 17], where the set of predicates is extended as the analysis proceeds by analyzing spurious counterexamples [8, 2, 26, 7, 25, 40]. The incompleteness of usual implementations of CEGAR-based predicate abstraction is well-known [11, 15], and there have been several attempts to suggest procedures that in the limit gain completeness: through carefully chosen widening operations [1], or through carefully orchestrating the proof search in the underlying decision procedures [31]. In contrast, our technique of path invariant generation is parameterized by the invariant templates used: they are sound and complete modulo the template language, but the required

invariants to prove a program may not exist within the template language.

The second ingredient of our work is *invariant synthesis*. There are several techniques for invariant synthesis, most notably by abstract fixpoint computation on a suitably constructed abstract domain [14, 16, 22, 42], or by a constraint based analysis that instantiates the parameters of an invariant template [3, 10, 32, 44]. While in our concrete instantiation of path invariants, we have chosen this latter algorithm, our framework can equally well be instantiated with an algorithm based on abstract interpretation. Invariants for arithmetic abstract domains have been studied extensively in both styles of analysis: notably by [34, 16, 41] in the abstract interpretation style, and by [5, 10, 13] using constraint-based methods. For quantified invariants involving arrays, [12, 22] give algorithms that computes fixpoints using a carefully constructed array domain. However, the corresponding invariant synthesis problem using constraint solving and template instantiation has not, to the best of our knowledge, been studied before.

The need for *universally quantified* assertions in the analysis of programs manipulating unbounded data structures such as arrays is well-known, and several approaches have been suggested to infer quantified predicates for predicate abstraction [20, 37, 6]. However, while these techniques either require specifying the actual predicates (often with Skolem constants for the quantified variables) [20, 6], or use heuristics to generalize to quantifiers from finite examples [37]. In contrast, we provide a sound and complete invariant generation technique for a class of invariant templates (whose correctness depends on recent work in decision procedures [6] and invariant generation [3]). For templates not within our language, we can still apply our algorithm and generate sound invariants, however, as expected, there is no completeness guarantee.

Treatment of disjunction can be incorporated into the abstract interpretation framework by suitable manipulation of the control-flow graph of the program [39, 43]. Path invariants implement such a manipulation in a property-guided way.

2. Examples

We illustrate our path invariants-based method for automatic refinement on three examples. The formal exposition of the method shall be given in the subsequent sections.

The first example is a program FORWARD, whose correctness argument depends on the interplay between values of counter and data variables during the loop execution. The example shows that path invariants discover relevant predicates that eliminate not only the given counterexample, but also all possible counterexamples that can be obtained by loop unwinding.

The second example is a program INITCHECK that manipulates arrays. The automatic discovery of relevant predicates that contain universal quantification for its correctness proof has been posed as a challenge in previous work on predicate abstraction and discovery [31, 40]. Path invariants discover relevant universally quantified predicates together with predicates over the loop counter.

The third example program PARTITION addresses the difficulty of dealing with global invariants. Since path programs capture only parts of the computations of the original program, the corresponding path invariants may be smaller. Then, they represent parts of the set of reachable states given by "global" invariants, which are captured by some combination of the individual pieces. Thus, path invariants allows one to implement lazy handling of disjunction, which is guided by counterexample traces.

2.1. Example FORWARD: Capture Arbitrary Loop Unwinding

Our first example is the program FORWARD from Figure 1**(a)**, whose correctness argument depends on the interplay between values of counter and data variables during the loop execution. The program executes a loop n times, and in each iteration, depending on some (unmodeled) condition, either increments the variable a by 1 and b by 2, or increments a by 2 and b by 1. At the end of the loop, we want to assert the claim that the sum $a + b$ must be equal to $3n$.

Abstraction Refinement. First, let us briefly describe how current techniques attempt to prove the assertion, and thus set up a background for demonstrating the advantages of path invariants w.r.t. the existing methods.

A standard counterexample-guided abstraction refinement (CEGAR) algorithm implemented in a tool based on predicate abstraction attempts to prove the program FORWARD in the following way. The initial abstraction discards all data relationships (that is, no predicates are tracked), and the initial reachability analysis checks if there is a path in the control-flow graph (CFG) that leads to the assertion being violated. There are such paths in the CFG, and Figure 1**(b)** shows one such abstract counterexample which traverses the `while`-loop once, takes the `then`-branch in the body of the loop, and then fails the assertion after leaving the loop. Notice that while this is a syntactic path in the CFG, the counterexample is *spurious*, that is, cannot be executed by the concrete program.

The second phase of the CEGAR algorithm is to check if the counterexample produced in the reachability phase is genuine or spurious, and if spurious, to find additional predicates that rule out the counterexample. The first step in this analysis is to translate the counterexample into a logical formula called the *path formula* that is satisfiable iff the coun-

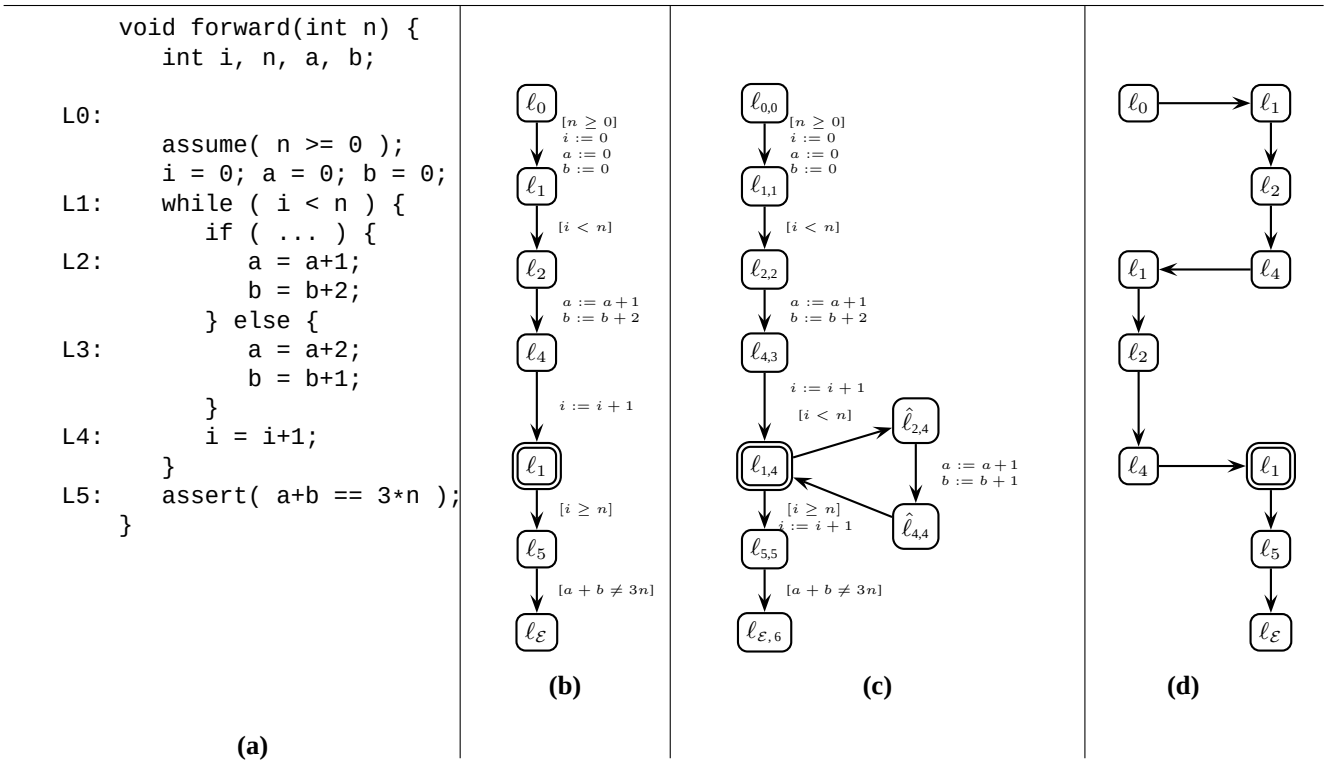


Figure 1. Program FORWARD illustrates discovery of relevant predicates that prevent loop unwinding by path invariants: (a) - the program, (b) - a counterexample, (c) - a path program that corresponds to the counterexample, and (d) - a potential counterexample resulting from loop unrolling (when path invariants are not applied). In path and control-flow graph representation, we use $[\cdot]$ to denote assumptions that represent conditional control statements of the program. As usual, updates are denoted by $:=$. Doubled circles denote locations at exit points of nested blocks of the programs, i.e. exit points of loops.

terexample can be executed in the concrete program [35]. The path formula is the conjunction of constraints derived from the operations along the path when the path is written in static single assignment form, that is, where each assignment to a variable is given a fresh name. The path formula for the counterexample in Figure 1**(b)** is the following conjunction, where each line corresponds to a transition between the control locations:

$$
\begin{array}{ll}
n_0 \geq 0 \wedge & \\
i_1 = 0 \wedge a_1 = 0 \wedge b_1 = 0 \wedge & \ell_0 \rightarrow \ell_1 \\
i_1 < n_0 \wedge & \ell_1 \rightarrow \ell_2 \\
a_2 = a_1 + 1 \wedge b_2 = b_1 + 1 \wedge & \ell_2 \rightarrow \ell_4 \\
i_2 = a_1 + 1 \wedge & \ell_4 \rightarrow \ell_1 \\
i_2 \geq n_0 \wedge & \ell_1 \rightarrow \ell_5 \\
a_2 + b_2 \neq 3n_0 & \ell_5 \rightarrow \ell_{\mathcal{E}}
\end{array}
$$

It is unsatisfiable, since there is no initial valuation of program variables that leads to a program computation along the counterexample. From an unsatisfiable path formula, predicates are extracted that ensure that if the abstraction tracks the predicates, then the current counterexample will be eliminated in subsequent abstract reachability steps. One way to do this is to extract all atomic predicates that appear in a proof of unsatisfiability of the path formula. (In practice, tools implement a more complicated scheme based on interpolants, but that does not change our argument below.) For this counterexample, a possible set of such predicates is

$$\{i = 0, i = 1, a = 0, a = 1, b = 0, b = 2\},$$

which tracks the variables i, a, and b along the path. While this set of predicates eliminates the counterexample, the next round of reachability encounters a longer counterexample obtained by unwinding the loop one more time, for example the counterexample in Figure 1**(d)**. This new counterexample is eliminated by tracking in addition the predicates

$$\{i = 2, a = 2, b = 4\}.$$

In general, in the k-th refinement step, we find the predicates

$$\{i = k, a = k, b = 2k\},$$

and the method does not terminate.

Path Invariants. Our refinement approach is based on *path invariants*, which we use instead of path formulas. We infer path invariants for special *path program*, which construction is guided by the statements that appear in the counterexample. The path program for the counterexample from Figure 1**(b)** is shown in Figure 1**(c)**.

We observe that the path program contains several copies of the control location that are traversed by the counterexample. Its statements are taken from the counterexample, and the control-flow graph captures the counterexample path as well as its arbitrary unwindings. We shall define formally how path programs are constructed in Section 3.

The counterexample passes two times through the control location ℓ_1, which labels the loop entry. So the path program has a loop $\ell_1 \rightarrow \ell_2 \rightarrow \ell_4 \rightarrow \ell_1$ in its CFG at location ℓ_1. Additionally it has copies of the locations that are traversed before exiting the structured block, i.e. before leaving the loop.

To refine the analysis so that the *family* of counterexamples represented by the path program are all refuted at once, we use invariant generation techniques. Since there are loops in the program, we can no longer construct a path formula that is linear in the counterexample length. Instead, we look for *invariant maps*. A *path-invariant map* is a mapping from locations of the path program to formulas such that the following two conditions hold: (Initiation) the initial location of the program is mapped to the predicate $true$, and (Consecution) for each pair of locations ℓ, ℓ' with an edge (ℓ, ρ, ℓ') in the path program, we have that the successor of the predicate at ℓ with respect to the program operation ρ implies the predicate at ℓ'. An invariant map is *safe* if further the error location (i.e., where the assertion fails) is mapped to $false$. Notice that an invariant map of the path program needs not be an invariant map of the original program (when the domain is suitably extended).

In this example we can generate invariants in arithmetic domains, e.g. by applying methods described in [10, 46], and obtain the following invariant map:

$$\begin{aligned}
\eta(\ell_{0,0}) &= \quad true \\
\eta(\ell_{1,1}) &= \quad a+b=3i \\
\eta(\ell_{2,2}) &= \quad i<n \wedge a+b=3i \\
\eta(\ell_{4,3}) &= \quad i<n \wedge a+b=3i+3 \\
\eta(\ell_{1,4}) &= \quad i\leq n \wedge a+b=3i \\
\eta(\hat{\ell}_{2,4}) &= \quad i<n \wedge a+b=3i \\
\eta(\hat{\ell}_{4,4}) &= \quad i<n \wedge a+b=3i+3 \\
\eta(\ell_{5,5}) &= \quad a+b=3n \\
\eta(\ell_{\mathcal{E},6}) &= \quad false.
\end{aligned}$$

The map is safe as $\ell_{\mathcal{E}}$ is mapped to $false$. A subsequent analysis that tracks these formulas at the corresponding locations is guaranteed to eliminate the original counterexample.

Furthermore, any spurious counterexample that is obtained by traversing the path program is eliminated by tracking these formulas. For example, consider a potential unwinding of the given counterexample that traverses the loop twice, i.e., see the path shown in Figure 1**(d)**. When following this path and reaching the control location ℓ_1 for the first time, a program analysis tracking the formulas from the invariant map computes an over-approximation of the reachable states that is at least as strong as the assertion defined by the invariant map at ℓ_1. Since the path-invariant map is inductive and safe, we conclude that the over-approximation computed for the second visit to the location $\ell_{1,4}$ is again as strong as the assertion at $\ell_{1,4}$. This means that the path shown in Figure 1**(d)** cannot appear as a spurious counterexample.

We can use similar reasoning to show that any unwinding of the given counterexample within the CFG of the path program will not produce a counterexample. This means that any path whose sequence of visited control locations is in the language defined by the regular expression $\ell_0\ell_1(\ell_2\ell_4\ell_1)^+\ell_5\ell_{\mathcal{E}}$ can never be reported as a spurious counterexample, once the formulas from the path-invariant map determine the abstraction. The formal justification of this statement, which characterizes the relevance of the predicates obtained from path invariants, relies on the completeness of abstract interpretation [11].

2.2. Example INITCHECK: Universally Quantified Predicates

The previous example showed how path programs can be used to refute a family of counterexamples arising from unrolling a loop. The next example shows how the same technique may be used to infer *quantified* invariants about the program state. Reasoning about many programs that manipulate unbounded data, e.g., stored in container data structures like arrays, requires universally quantified assertions. Usually, these assertions contain universal quantification over indices, positions, or keys, which provide reference to data values stored in the data structure. There exist a fundamental obstacle that prevents the systematic discovery of universally quantified predicates based on (finite) counterexamples. Namely, they can only expose a bounded number of data items that are stored in the data structure. Thus, it is difficult to derive and formally justify universal quantification in the discovered predicates. The next example demonstrates our second technical contribution: in addition to using path invariants in the abstraction refinement phase, we provide an invariant synthesis algorithm that can infer quantified invariant maps for programs that manipulate arrays.

Consider the program INITCHECK in Figure 2**(a)**, which initializes an array to 0 and then checks that all the elements

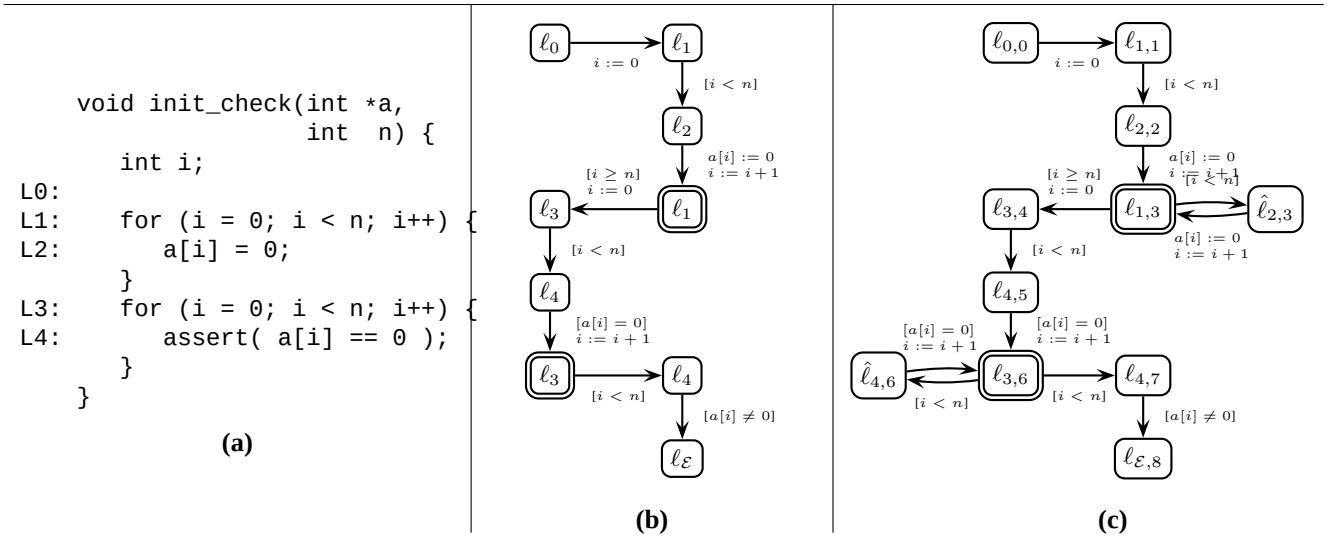


Figure 2. Program INITCHECK illustrates discovery of relevant universally quantified predicates for the challenge example from [31, 40]: (a) - an example program, (b) - a counterexample, (c) - path program that corresponds to the counterexample.

in the array are 0. We wish to prove that all the assertions in the second array pass.

Abstraction Refinement. The path shown in Figure 2**(b)** represents a spurious counterexample that would be found by a verification tool that does not track the array contents precisely. The path contains a statement that corresponds to the assertion violation, which appears after traversing each of the loops once. In particular, from the traversal of the first loop we can conclude that the first element in the array is initialized by zero, $a[0] = 0$. Then, by considering this fact in the second loop, where the equality $a[i] = 0$ is checked for $i = 0$, we conclude that the predicate $a[0] = 0$ is sufficient to eliminate the given counterexample.

However, the fact $a[0] = 0$ eliminates only this particular counterexample. It does not eliminate the next counterexample that traverses each loop twice (and requires tracking the fact $a[1] = 0$). In fact, counterexample-based refinement is likely to generate an infinite family of facts $a[i] = 0$ for $i = 0, 1, \ldots$

We observe that since the number of array elements being initialized and subsequently checked by INITCHECK is determined by the variable n, and hence is arbitrary, no finite number of predicates obtained from finite counterexamples created by on-going loop unwinding will ever suffice to prove the program correct. We need a universally quantified predicate $\forall k : 0 \leq k < n \rightarrow a[k] = 0$ to effectively perform verification of INITCHECK.

Path Invariants. Justification of the universal quantification for the discovered predicates requires consideration of all possible paths that traverse the initialization and checking loops, located at ℓ_1 and ℓ_3, respectively. We use a path program to represent this family of paths. We show the path program for the given counterexample in Figure 2**(c)**.

Given the path program, we can provide systematic justification of universal quantification using path invariants. The technical complication is that we have to infer inductive invariant maps that map certain locations to universally quantified assertions.

An inductive invariant map, say η, for our path program needs to assert that at location ℓ_4 the content of $a[i]$ is zero. Note that the transition to the error location $\ell_{\mathcal{E}}$, which is taken from ℓ_4 if $a[i] = 0$ does not hold, appears within a loop that iteratively increments the value of i. Hence, the assertion assigned by η to the location ℓ_4 must imply $a[i] = 0$ for all values of i reachable at ℓ_4, which lie in the interval from 0 to $n - 1$. We observe that the first loop assigns zero to an array cell $a[i]$ for each value of i that is subsequently checked in the second loop.

We compute the inductive invariant map η that formalizes the above reasons for the non-reachability of the error location in the path program. (See Section 4 for an algorithm computing invariants that contain universal quantification.) The assertions in η restrict the valuation of counter variable i as well as universally quantified statements about the content of the initialized cells in the array a. Assertions for the locations in the first loop only refer to the array content up to the position i, whereas the assertions for the second loop refer to each array cell starting between 1 and $n - 1$. The cell $a[0]$ is not taken into account since the

```
  void partition(int *a, int n) {
     int i, gelen, ltlen;
     int ge[n], lt[n];

L1:     gelen = 0; ltlen = 0;
L2:     for (i = 0; i < n; i++) {
L3:        if (a[i] >= 0) {
L4:           ge[gelen] = a[i];
              gelen++;
           } else {
L5:           lt[ltlen] = a[i];
              ltlen++;
           }
        }
L6:     for (i = 0; i < gelen; i++) {
           assert(ge[i] >= 0);
        }
L7:     for (i = 0; i < ltlen; i++) {
           assert(lt[i] < 0);
        }
  }
```

Figure 3. Program PARTITION **illustrates combining disjunctive reasoning over paths with universally quantified path invariants.**

counterexample implicitly assumes that $a[0] = 0$ holds by passing the assertion first time.

$$
\begin{aligned}
\eta(\ell_{0,0}) &= \mathit{true}\\
\eta(\ell_{1,1}) &= 0 \le i\\
\eta(\ell_{2,2}) &= 0 \le i \wedge i < n\\
\eta(\ell_{1,3}) &= 1 \le i \wedge i \le n \wedge \forall k : 1 \le k < i \rightarrow a[k] = 0\\
\eta(\hat{\ell}_{2,3}) &= 1 \le i \wedge i < n \wedge \forall k : 1 \le k < i \rightarrow a[k] = 0\\
\eta(\ell_{3,4}) &= 0 \le i \wedge \forall k : 1 \le k < n \rightarrow a[k] = 0\\
\eta(\ell_{4,5}) &= 0 \le i \wedge i < n \wedge \forall k : 1 \le k < n \rightarrow a[k] = 0
\end{aligned}
$$

$$
\begin{aligned}
\eta(\ell_{3,6}) &= 1 \le i \wedge i \le n \wedge \forall k : 1 \le k < n \rightarrow a[k] = 0\\
\eta(\hat{\ell}_{4,6}) &= 1 \le i \wedge i < n \wedge \forall k : 1 \le k < n \rightarrow a[k] = 0\\
\eta(\ell_{4,7}) &= 1 \le i \wedge i < n \wedge \forall k : 1 \le k < n \rightarrow a[k] = 0\\
\eta(\ell_{\mathcal{E},8}) &= \mathit{false}
\end{aligned}
$$

By tracking the assertions in the range of the path-invariant map, we are guaranteed that all potential counterexamples that visit a sequence of control locations from the set defined by the regular expression $\ell_0\ell_1(\ell_2\ell_1)^+\ell_3(\ell_4\ell_3)^+\ell_4\ell_{\mathcal{E}}$ are eliminated.

2.3. Example PARTITION: Disjunctive Reasoning

Path invariants find *local* reasons that refute a family of counterexamples. To prove an assertion in the program, though, an analysis may have to iterate through several different path programs, each of which presents a different path to the assertion violation. We now illustrate how a path invariant-based approach within a CEGAR framework can lazily instantiate these different paths, using the path program derived from each counterexample to learn additional facts.

Consider the program PARTITION in Figure 3 that partitions the elements of an input array a into two arrays ge and lt that contain respectively the non-negative and negative elements of a. In order to prove the assertions, we need the loop invariants

$$\forall k : 0 \le k < gelen \rightarrow ge[k] \ge 0 \tag{1}$$

$$\forall k : 0 \le k < lelen \rightarrow lt[k] < 0 \tag{2}$$

at the control location `L3`.

Instead of applying invariant generation on the entire program at once, CEGAR with path invariants will find the two conjuncts of the invariant one at a time. For example, consider first a spurious counterexample that goes through the `then` branch of the conditional in the `for` loop. The corresponding path program looks almost identical to the path program for Example INITCHECK from Figure 2**(c)** (except that instead of a direct write to $ge[i]$, the counterexample contains the operations $\texttt{assume}(a[i] == 0);\ ge[i] = a[i];$). Performing invariant synthesis on this path program leads to a path-invariant map as in Example INITCHECK. In particular, at the location `L3`, we get the invariant from Equation (1).

These invariants, however, are not enough to prove the assertions, and a second counterexample is found. This counterexample goes through the `else` branch of the conditional in the `for` loop. Again, the path program is similar to the path program for Example INITCHECK. This time, the path-invariant map generates the second conjunct of the loop invariant. Together, these assertions are enough to prove the correctness of the program.

The key optimization is that the CEGAR algorithm breaks the search for global program invariants (as is usual in invariant synthesis techniques) into searching for individual components of the invariant, thus restricting the search to a much smaller space.

3. Definitions

Control-Flow Graphs (CFGs). We assume an abstract representation of programs by transition systems [38]. A *program* $P = (X, \mathsf{locs}, \ell_0, \mathcal{T}, \ell_{\mathcal{E}})$ 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 an error location $\ell_{\mathcal{E}} \in \mathsf{locs}$. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in \mathsf{locs}$ are control 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 a 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. For a constraint ρ over $X \cup X'$ and a valuation $(s, s') \in \mathsf{val}.X \times \mathsf{val}.X'$, we write $(s, s') \models \rho$ if the valuation satisfies the constraint ρ. A *computation* of the program P is a sequence $\langle m_0, s_0\rangle\langle m_1, s_1\rangle \ldots \langle m_k, s_k\rangle \in (\mathsf{locs} \times \mathsf{val}.X)^*$, where $m_0 = \ell_0$ is the initial location and for each $i \in \{0, \ldots, k-1\}$, there is a transition $(m_i, \rho, m_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A location ℓ is reachable if $\langle \ell, s\rangle$ appears in some computation for some state s. A program is *unsafe* if $\ell_\mathcal{E}$ is reachable.

A *path* of a program P is a sequence of transitions $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_k)$, where ℓ_0 is the initial location. The path π is *feasible* if there is a computation $\langle \ell_0, s_0\rangle, \ldots, \langle \ell_k, s_k\rangle$ such that for each $i \in \{0, \ldots, k-1\}$, we have $(s_i, s_{i+1}) \models \rho_i$; and *infeasible* otherwise. We write $\mathsf{locs}.\pi$ for the set $\{\ell_0, \ldots, \ell_k\}$ of locations in the path π, and $\tau.\pi$ for the set of transitions $\{(\ell_0, \rho_0, \ell_1), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_k)\}$ in the path. For a program P, we write $\mathsf{Paths}.P$ for the (possibly infinite) set of paths of P.

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 (i.e., the combined theories of linear inequalities and uninterpreted functions) 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 $\eta.\ell \wedge \rho \models \eta.\ell'$. Here, $\eta.\ell'$ is the constraint obtained by substituting variables from x' for the variables from x in $\eta.\ell$.
> **(I2: Safety)** $\eta.\ell_\mathcal{E} = \mathit{false}$.

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 CFG passes through some location in the cutset.

Path Programs and Path Invariants. Consider a program $P = (X, \mathsf{locs}, \ell_0, \mathcal{T}, \ell_\mathcal{E})$ with an error path $\pi = (\ell_0, \rho_0, \ell_1), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_\mathcal{E})$. Let $\mathcal{T}.\pi = \{(\ell_0, \rho_0, \ell_1), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_k)\}$ be the set of transitions in π. Let $\mathsf{Blocks}.\pi = \{B_1, \ldots, B_m\}$ be the set of nested blocks of the control-flow graph of π. Given P and π, we construct the *path program* $P[\pi] = (X, \mathsf{locs}', \ell'_0, \mathcal{T}', \ell'_\mathcal{E})$ over the same variables as follows:

- $\mathsf{locs}' = \{\ell, \hat{\ell} \mid \ell \in \mathsf{locs}\} \times \{0, \ldots, k\}$.

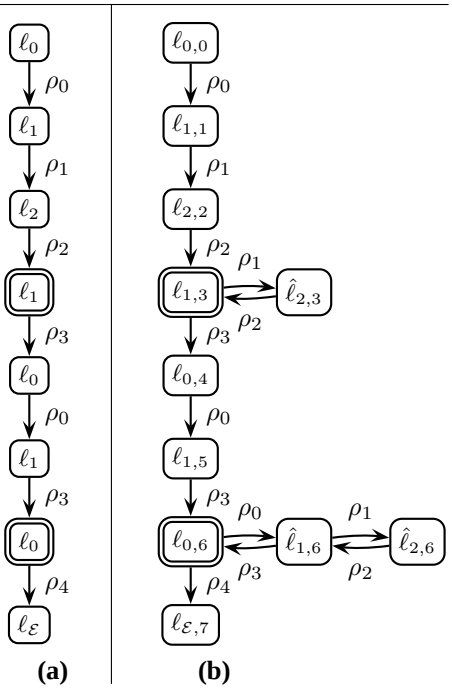


Figure 4. Counterexample and corresponding path program.

- $\ell'_0 = (\ell_0, 0)$.
- For each position $i \in [0..k-1]$ of the path π, the path program contains the transition $((\ell_i, i), \rho_i, (\ell_{i+1}, i+1))$. Moreover, if there is a nested block $B \in \mathsf{Blocks}.\pi$ such that $\ell_i \in B$ and $\ell_{i+1} \notin B$, and B_i is the maximal such block, then the path program contains also the following transitions:

  $((\ell_i, i), X' = X, (\hat{\ell}_i, i))$;
  $((\hat{\ell}, i), \rho, (\hat{\ell}', i))$ for all $(\ell, \rho, \ell') \in \mathcal{T}.\pi$ with $\ell, \ell' \in B_i$;
  $((\hat{\ell}_i, i), X' = X, (\ell_i, i))$.
- $\ell'_\mathcal{E} = (\ell_\mathcal{E}, k)$.

Intuitively, the paths of the path program $P[\pi]$ include the error path π, and in addition all paths that result from π by staying within some nested blocks of π for some additional transitions. Hence the path program $P[\pi]$ may traverse some loops that are traversed by π more often, but it contains no transition that does not occur in π.

Consider, for example, the error path π

0: (ℓ_0, ρ_0, ℓ_1),
1: (ℓ_1, ρ_1, ℓ_2),
2: (ℓ_2, ρ_2, ℓ_1),
3: (ℓ_1, ρ_3, ℓ_0),
4: (ℓ_0, ρ_0, ℓ_1),
5: (ℓ_1, ρ_3, ℓ_0),
6: $(\ell_0, \rho_4, \ell_\mathcal{E})$.

The nested blocks of this path are $B_1 = \{\ell_0, \ell_1, \ell_2\}$ and $B_2 = \{\ell_1, \ell_2\}$, with B_2 being a subblock of B_1. The path program $P[\pi]$ may stay within the inner block B_2 after the third transition, and within the outer block after the fifth transition of π. The complete set of transitions of $P[\pi]$ is:

$$
\begin{array}{l}
((\ell_0,0),\rho_0,(\ell_1,1)),\\
((\ell_1,1),\rho_1,(\ell_2,2)),\\
((\ell_2,2),\rho_2,(\ell_1,3)),\\
((\ell_1,3),\rho_3,(\ell_0,4)),\\
\qquad ((\ell_1,3),X'=X,(\hat{\ell}_1,3)),\\
\qquad ((\hat{\ell}_1,3),\rho_1,(\hat{\ell}_2,3)),\\
\qquad ((\hat{\ell}_2,3),\rho_2,(\hat{\ell}_1,3)),\\
\qquad ((\hat{\ell}_1,3),X'=X,(\ell_1,3)),\\
((\ell_0,4),\rho_0,(\ell_1,5)),\\
((\ell_1,5),\rho_3,(\ell_0,6)),\\
((\ell_0,6),\rho_4,(\ell_{\mathcal{E}},7)),\\
\qquad ((\ell_0,6),X'=X,(\hat{\ell}_0,6)),\\
\qquad ((\hat{\ell}_0,6),\rho_0,(\hat{\ell}_1,6)),\\
\qquad ((\hat{\ell}_1,6),\rho_1,(\hat{\ell}_2,6)),\\
\qquad ((\hat{\ell}_2,6),\rho_2,(\hat{\ell}_1,6)),\\
\qquad ((\hat{\ell}_1,6),\rho_3,(\hat{\ell}_0,6)),\\
\qquad ((\hat{\ell}_0,6),X'=X,(\ell_0,6)).
\end{array}
$$

By viewing $P[\pi]$ instead of π as a counterexample, we will simultaneously handle an unbounded number of error paths, namely, all error paths that extend π and loop within the blocks B_1 and B_2 for an arbitrary number of iterations. We illustrate the example by drawings for the counterexample and the corresponding path program in Figure 4; the transitions with $\rho \equiv X' = X$ are omitted in the figure.

An invariant map for a path program is called a *path invariant*. Since a path program $P[\pi]$ may contain several different locations $(\ell, 1), (\ell, 2), \ldots$ which correspond to the same location ℓ of the original program P, these locations may be mapped to different invariants. This corresponds to unrolling of loops.

4. Algorithms

We now instantiate path invariant based abstraction refinement in a predicate abstraction based CEGAR framework. Then, we present an algorithm to synthesize *universally quantified* invariants over a combination of theories.

4.1. CEGAR with Path Invariants

While path programs are a general technique for program analysis that automatically adjust their precision, we now instantiate the framework in a predicate abstraction-based CEGAR loop, where path-invariant maps are used to suggest additional predicates.

The CEGAR algorithm has conceptually three phases [8, 2, 26]: abstract reachability, counterexample analysis, and abstraction refinement. The *abstract reachability* phase tries to construct a proof of safety for the program, by generating an unwinding of the CFG where each node of the unwinding is annotated with an abstract state. The abstract state is a boolean combination of the current predicates being tracked, and represents an over-approximation of the set of reachable states of the program when it executes the path from the root of the tree to the current node. If a proof is found, that is, if the unwinding does not hit the error location, the algorithm stops. Otherwise the algorithm moves to the *counterexample analysis* phase where it picks a counterexample from the tree (i.e., a path from the root to the error location), and checks if this path is realizable in the concrete program. The counterexample analysis phase first constructs a logical formula from the counterexample that is satisfiable iff the counterexample is concretely executable. If the counterexample is feasible, a bug is found and the algorithm stops. Otherwise, the algorithm proceeds with the *abstraction refinement* phase. However, instead of discovering predicates from the path, as in interpolation-based approaches [25, 18, 40], we construct a path program from the counterexample, and use a constraint-based invariant synthesis algorithm (outlined in the next section) to produce a path-invariant map η. This invariant map is used to refine the predicate abstraction by adding the predicates appearing in $\eta.\ell$ to the location ℓ. After this phase, the overall algorithm again proceeds with the abstract reachability phase. The three steps are repeated until either there is a proof or there is a bug (or, since the problem is undecidable, the loop does not terminate).

The key property of the refinement step using path invariants is that the predicates that appear in the path invariants rule out *all* abstract counterexamples arising from arbitrary unwindings of loops in the path program. This is in contrast to existing implementations of CEGAR where each refinement step is guaranteed to remove only the current abstract counterexample, and hence may get stuck in removing potentially infinitely many counterexamples involving loop unwindings. Let $\mathsf{Reach}.\Pi$ denote the set of all paths in the abstract reachability tree constructed in the abstract reachability phase under abstraction Π.

Theorem 1 *Let P be a program, π a spurious counterexample, and η the invariant map for path program $P[\pi]$. Then for any predicate map Π such that $\eta.\ell \in \Pi.\ell$ for all $\ell \in \pi$, there exists no counterexample in $\mathsf{Reach}.\Pi$ that is also in $\mathsf{Paths}.P[\pi]$.*

As a corollary, if there is a proof of correctness of a program in a fixed template language, then CEGAR with path invariants is guaranteed to produce this proof in a finite number of steps.

4.2. Generation of Path Invariants

The crucial component in our path invariant-based CEGAR algorithm is automatic invariant generation. We can rely on successful existing methods for the generation of invariants via reduction to constraint solving (cf. [10, 13, 32, 44, 45, 46]). Alternatively, we can also exploit the practical applicability of specialized abstract domains developed in the framework of abstract interpretation (cf. [4, 16, 34, 41]).

The above mentioned methods target numerical domains. Practical software verification requires more general predicate domains, including the combination of arithmetic with uninterpreted function symbols and universal quantification [2, 19, 20, 24, 26]. We can exploit recent advances in reasoning about hierarchic combination of theories and a resulting algorithm for invariant generation [6, 33, 47, 3].

Next, we illustrate how we can generate path invariants that contain universal quantification by applying the hierarchical style of reasoning. We consider the path program for the example INITCHECK, which is shown in Figure 2 **(c)**. We observe that it is sufficient to concentrate on the cutpoints in the CFG, and to replace statements between the cutpoints by composed ones. In our example, we shall compute an inductive invariant map η that assigns assertions only to the control locations ℓ_1 and ℓ_3.

Invariant Templates. We follow the template-based approach to the generation of invariants (cf. [5, 10]). We assume that for each control location in the domain of the map η we have a so-called invariant template, which is a parametric assertion over program variables. An example for a simple template is $p_{\mathtt{i}}\mathtt{i} + p_{\mathtt{n}}\mathtt{n} \leq p$, where $p_{\mathtt{i}}$, $p_{\mathtt{n}}$, and p are parameters. This template denotes a set of assertions that can be obtained by giving values to parameters, e.g., $2\mathtt{i} - 3\mathtt{n} \leq 5$. The crux of the template-based approach consists of defining and solving a system of constraints over the template's parameters such that the resulting valuations yield an inductive invariant map.

We assume an auxiliary notation where $t(\mathtt{i},\mathtt{n})$ denotes the linear expression $t_{\mathtt{i}}\mathtt{i} + t_{\mathtt{n}}\mathtt{n} + t$. We use $\mathtt{i}$, $\mathtt{n}$, and $\mathtt{k}$ to denote the program variables and the index variable under universal quantification. Parameter-related variables are p and q. For our example, at the location ℓ_1 we assume the template

$$\varphi = (\forall \mathtt{k} : p^1(\mathtt{i},\mathtt{n}) \leq \mathtt{k} \wedge \mathtt{k} \leq p^2(\mathtt{i},\mathtt{n}) \rightarrow \mathtt{a}[\mathtt{k}] = p^3(\mathtt{i},\mathtt{n})) \ \wedge \\ p^4(\mathtt{i},\mathtt{n}) \leq 0 \ \wedge \ p^5(\mathtt{i},\mathtt{n}) \leq 0 \, .$$

Note that the first conjunct of the template contains universal quantification. The template ψ for the location ℓ_2 is defined in terms of $q^1, \ldots, q^5$ in an analogous way. We write φ' and ψ' to denote the next-step versions of the templates, which are obtained by replacing the program variables $\mathtt{i}$, $\mathtt{n}$, and the array symbol $\mathtt{a}$ by $\mathtt{i}'$, $\mathtt{n}'$, and $\mathtt{a}'$ respectively.

Constraints on Template Parameters. We define a set of constraints over the parameters of the templates φ and ψ that encode the inductiveness and safety conditions. We present one interesting case, which ensures consecution for the program statement that writes into the array. We use the expression $\mathtt{a}\{\mathtt{i} := 0\}$ to denote the array obtained from $\mathtt{a}$ by setting to zero its $\mathtt{i}$-th cell.

$$\varphi \wedge \mathtt{i} < \mathtt{n} \wedge \mathtt{i}' = \mathtt{i}+1 \wedge \mathtt{n}' = \mathtt{n} \wedge \mathtt{a}' = \mathtt{a}\{\mathtt{i} := 0\} \models \varphi' \quad (3)$$

The proof of validity of the implication (3) can be decomposed into three sub-proofs, one for each conjunct in φ'. Again, we consider one particular case, namely when the conjunct of φ' containing universal quantification appears on the right-hand side of the implication. We define auxiliary symbols π and ρ as

$$\pi = \forall \mathtt{k} : p^1(\mathtt{i},\mathtt{n}) \leq \mathtt{k} \wedge \mathtt{k} \leq p^2(\mathtt{i},\mathtt{n}) \rightarrow \mathtt{a}[\mathtt{k}] = p^3(\mathtt{i},\mathtt{n}) \, , \\ \rho = p^4(\mathtt{i},\mathtt{n}) \leq 0 \wedge p^5(\mathtt{i},\mathtt{n}) \leq 0 \wedge \mathtt{i} < \mathtt{n} \, ,$$

and consider the case below.

$$\pi \wedge \rho \wedge \mathtt{i}' = \mathtt{i} + 1 \wedge \mathtt{n}' = \mathtt{n} \wedge \mathtt{a}' = \mathtt{a}\{\mathtt{i} := 0\} \\ \models \forall \mathtt{k} : p^1(\mathtt{i}',\mathtt{n}') \leq \mathtt{k} \wedge \mathtt{k} \leq p^2(\mathtt{i}',\mathtt{n}') \rightarrow \mathtt{a}'[\mathtt{k}] = p^3(\mathtt{i}',\mathtt{n}')$$

Let $\mathtt{k}^*$ be a fresh variable, which we may treat as an auxiliary program variable. Next, we rewrite the universally quantified assertion

$$\pi \wedge \rho \wedge \mathtt{i}' = \mathtt{i} + 1 \wedge \mathtt{n}' = \mathtt{n} \wedge \mathtt{a}' = \mathtt{a}\{\mathtt{i} := 0\} \\ \models p^1(\mathtt{i}',\mathtt{n}') \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}',\mathtt{n}') \rightarrow \mathtt{a}'[\mathtt{k}^*] = p^3(\mathtt{i}',\mathtt{n}') \, ,$$

and eliminate the implication from the right-hand side

$$\pi \wedge \rho \wedge \mathtt{i}' = \mathtt{i} + 1 \wedge \mathtt{n}' = \mathtt{n} \wedge \mathtt{a}' = \mathtt{a}\{\mathtt{i} := 0\} \wedge \\ p^1(\mathtt{i}',\mathtt{n}') \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}',\mathtt{n}') \\ \models \mathtt{a}'[\mathtt{k}^*] = p^3(\mathtt{i}',\mathtt{n}') \, .$$

Primed Program Variables and Array Symbols. Now, we can eliminate primed variables, which requires a case distinction for the elimination of the primed array symbol $\mathtt{a}'$. We need to distinguish two cases. In the first case, the write into the array $\mathtt{a}$ takes place at the same position that is read in the right-hand side, and hence we can directly use the written value. This means that if $\mathtt{i} = \mathtt{k}^*$ then we have $\mathtt{a}'[\mathtt{k}^*] = 0$. In the second case, the value of the cell at position $\mathtt{k}^*$ in the array $\mathtt{a}$ does not change after the update, i.e., we have $\mathtt{a}'[\mathtt{k}^*] = \mathtt{a}[\mathtt{k}^*]$. Thus, the implication (3) is valid if and only if the following two implications are valid:

$$\pi \wedge \rho \wedge p^1(\mathtt{i}+1,\mathtt{n}) \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}+1,\mathtt{n}) \wedge \\ \mathtt{i} = \mathtt{k}^* \models 0 = p^3(\mathtt{i}+1,\mathtt{n}) \quad (4a)$$

$$\pi \wedge \rho \wedge p^1(\mathtt{i}+1,\mathtt{n}) \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}+1,\mathtt{n}) \wedge \\ \mathtt{i} \neq \mathtt{k}^* \models \mathtt{a}[\mathtt{k}^*] = p^3(\mathtt{i}+1,\mathtt{n}). \quad (4b)$$

Universal Quantification. At the next step we replace the conjunct π containing the universally quantified implication

by the quantifier free instances of the implication that are obtained for appropriate valuations of the quantified variable $\mathtt{k}$. We find the appropriate valuations for $\mathtt{k}$ by analyzing the structure of (4a) and (4b).

We collect all occurrences of array read expressions, i.e., of terms $\mathtt{a}[\cdot]$ indexed by positions other than universally quantified index $\mathtt{k}$. We observe that the implication (4a) does not have any such occurrences. Hence, we conclude that the validity of (4a) does not rely on its conjunct π. We can replace the condition (4a) by the condition

$$\begin{array}{l}\rho \wedge p^1(\mathtt{i}+1,\mathtt{n}) \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}+1,\mathtt{n}) \wedge \\ \mathtt{i} = \mathtt{k}^* \models 0 = p^3(\mathtt{i}+1,\mathtt{n}).\end{array} \tag{5}$$

In the implication (4b), the set of occurrences is the singleton set $\{\mathtt{k}^*\}$. Since the array read expression $\mathtt{a}[\mathtt{k}^*]$ appears only on the right-hand side of the implication (4b), its validity depends on π. The conjunct π can yield a constraint on $\mathtt{a}[\mathtt{k}^*]$, via the universal quantification, if $\mathtt{k}^*$ satisfies the premises of the implication in π. That is, we have $\mathtt{a}[\mathtt{k}^*] = p^3(\mathtt{i},\mathtt{n})$ and require the condition

$$\begin{array}{l}\rho \wedge p^1(\mathtt{i}+1,\mathtt{n}) \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}+1,\mathtt{n}) \wedge \\ \mathtt{i} \neq \mathtt{k}^* \models p^1(\mathtt{i},\mathtt{n}) \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i},\mathtt{n}).\end{array} \tag{6}$$

Under assumption that the template parameters satisfy the condition (6), we can drop the implication (4b) and use the following one instead, where $\mathtt{a}[\mathtt{k}^*] = p^3(\mathtt{i},\mathtt{n})$ is an instance of π which we add to the left-hand side of the implication:

$$\begin{array}{l}\mathtt{i} < \mathtt{n} \wedge p^1(\mathtt{i}+1,\mathtt{n}) \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}+1,\mathtt{n}) \wedge \\ \mathtt{i} \neq \mathtt{k}^* \wedge \mathtt{a}[\mathtt{k}^*] = p^3(\mathtt{i},\mathtt{n}) \models \mathtt{a}[\mathtt{k}^*] = p^3(\mathtt{i}+1,\mathtt{n}).\end{array} \tag{7}$$

Array Reads as Function Applications. So far, we reduced the condition (3) containing universal quantification and an array update expression to a conjunction of conditions that require reasoning over arithmetic and array read expressions. We observe that due to the absence of array updates, we can treat array read expressions as applications of uninterpreted function symbols. This means that we only assume the functionality axiom, which states that a read operation from the same array from the same position always produces the same value.

We consider the condition (7). We replace each occurrence of the array read expression $\mathtt{a}[\mathtt{k}^*]$ by a fresh variable and record its origin. Let the first occurrence be replaced by the variable $\mathtt{v}$ and second one by $\mathtt{w}$. Then, we introduce an equality constraint over the fresh variables. The constraint encodes the functionality axiom, as described above, and states that $\mathtt{v}$ and $\mathtt{w}$ are equal if the corresponding array read expressions refer to the same position in the array. In this particular case, the premise is vacuously true, since $\mathtt{v}$ and $\mathtt{w}$ replace the same expression. Finally, we can replace the condition (7) by the following:

$$\begin{array}{l}\rho \wedge p^1(\mathtt{i}+1,\mathtt{n}) \leq \mathtt{k}^* \wedge \mathtt{k}^* \leq p^2(\mathtt{i}+1,\mathtt{n}) \wedge \\ \mathtt{i} \neq \mathtt{k}^* \wedge \mathtt{v} = p^3(\mathtt{i},\mathtt{n}) \wedge \mathtt{v} = \mathtt{w} \models \mathtt{w} = p^3(\mathtt{i}+1,\mathtt{n}).\end{array} \tag{8}$$

Implication Encoding. The sequence of previously described steps transforms the condition (3), which ensures that the template φ yields an assertion satisfying the inductiveness constraint at the control location ℓ_1, to the conjunction of conditions (5), (6), and (8). They contain arithmetic expressions that are linear w.r.t. the program variables $\mathtt{i}$ and $\mathtt{n}$, and auxiliary variables $\mathtt{k}^*$, $\mathtt{v}$, and $\mathtt{w}$. Thus, we can apply the classical approach for encoding the validity of such implications by arithmetic constraints over the template parameters $p_{\mathtt{i}}$, $p_{\mathtt{n}}$, and p. We omit details of this construction, see, e.g., [5, 10, 48] for detailed exposition.

We proceed in the similar way with the remaining conditions on the invariant templates, and translate them into a set of arithmetic constraints. We compute a valuation of template parameters, which yields an inductive invariant map defining a path invariant, by applying specialized constraint solving techniques on the resulting constraint (cf. [10, 44]).

We obtain the following valuation of the template parameters for the control location ℓ_1

$p^1(\mathtt{i},\mathtt{n})$	$p^2(\mathtt{i},\mathtt{n})$	$p^3(\mathtt{i},\mathtt{n})$	$p^4(\mathtt{i},\mathtt{n})$	$p^5(\mathtt{i},\mathtt{n})$
0	$\mathtt{i}-1$	0	$-\mathtt{i}$	$\mathtt{i}-\mathtt{n}$

which yields the invariant

$$(\forall \mathtt{k} : 0 \leq \mathtt{k} \wedge \mathtt{k} \leq \mathtt{i}-1 \rightarrow \mathtt{a}[\mathtt{k}] = 0) \wedge -\mathtt{i} \leq 0 \wedge \mathtt{i}-\mathtt{n} \leq 0\,.$$

Theoretical Foundations. While we have outlined the invariant synthesis algorithm through an example, we can prove that our algorithm is sound and complete for invariant generation for invariant templates from the following language. A template is a conjunction of linear inequalities over the program variables whose coefficients may be parameters that may be conjoined with a universally quantified template

$$\forall \mathtt{k}_1 \ldots \forall \mathtt{k}_n : \bigwedge_i p_i(X) \leq \mathtt{k}_i \wedge \mathtt{k}_i \leq q_i(X) \rightarrow r(X, \mathtt{k}_1, \ldots, \mathtt{k}_n),$$

where $p_i(X)$, $q_i(X)$, and $r(X, \mathtt{k}_1, \ldots, \mathtt{k}_n)$ are linear terms over the program variables X with parameterized coefficients, and the linear expression r can additionally have terms where the variables $\mathtt{k}_i$ appear as indices in array reads (i.e., as terms $\mathtt{a}[k]$ for an array variable $\mathtt{a}$). That is, our algorithm is guaranteed to construct an invariant map for invariant templates in the language of array properties iff an invariant map exists and is expressible in the language. This follows from several observations: first, the decidability of the array property fragment [6], second, the reduction of the invariant synthesis problem for linear arithmetic and uninterpreted functions to linear arithmetic using hierarchic combination of theories [3], and third, the soundness and completeness of invariant generation for linear arithmetic. We omit the technical details. We can still run our invariant synthesis algorithm in case the template does not conform to the above form, but then completeness is no longer guaranteed (results are sound, however).

The complexity of the procedure is influenced by the number of array reads involving quantified variables, since each array read on the r.h.s. involves a case split. In our experiments, we have therefore always tried to find invariants of the tractable form:

$$\forall \mathtt{k} : p(X) \leq \mathtt{k} \wedge \mathtt{k} \leq q(X) \rightarrow \mathtt{a}[\mathtt{k}] = r(X),$$

where $p(X)$, $q(X)$, and $r(X)$ are linear terms over the program variables X with parameterized coefficients, and $\mathtt{a}$ is an array symbol in the program.

5. Experiments

We have implemented the instantiation of our technique using predicate abstraction-based CEGAR and template-based invariant generation as outlined in Section 4. This gives an automatic software verification tool that can reason about universally quantified assertions.

The invariant generation tool is implemented using the Constraint Logic Programming system SICStus Prolog [36], which contains a constraint solver for linear arithmetic [29]. The tool takes as input a path program together with an invariant template map. The template map ranges over cut-points. Invariants for non-cut-point locations are obtained by computing strongest postconditions from cut-points in a standard way. The path programs are constructed from error paths as generated by the Blast model checker.

We have applied our algorithm to examples involving array reasoning, including the examples in Section 2. We present some experimental data collected while applying the tool (on a 1.6 GHz laptop).

Example Forward. In order to compute a path invariant for the path program shown in Figure 1**(c)**, we first try an invariant template map that assigns the template

$$c_i i + c_n n + c_a a + c_b b + c = 0$$

to the control location $\ell_{1,4}$ (which is the only cut-point), where c_i, c_n, c_a, c_b and c are unknown parameters to be instantiated. We choose this template following a simple heuristic that obtains a template by replacing the coefficients of the target assertion by parameters.

Our tool fails to instantiate the above template, and reports the failure in 40 ms. We heuristically refine the template by conjoining an inequality, and obtain the new template

$$\begin{array}{c} c_i i + c_n n + c_a a + c_b b + c = 0 \wedge \\ d_i i + d_n n + d_a a + d_b b + d \leq 0. \end{array}$$

This template is instantiated in 130 ms, and yields the assertion

$$a + b = 3i \wedge a + b \leq 3n.$$

Example InitCheck. The path program shown in Figure 2**(c)** contains an assertion statement that refers to the content of the array a and is accessible inside a loop. Thus, our heuristic proposes the following map containing universally quantified templates for the cut-point locations $\ell_{1,3}$ and $\ell_{3,6}$ (where $c^j(i,n)$ denotes the expression $c_i^j i + c_n^j n + c^j$, and d^j is defined similarly):

$$\begin{array}{c} \forall k : c^1(i,n) \leq k \leq c^2(i,n) \rightarrow a[k] = c^3(i,n) \\ \forall k : d^1(i,n) \leq k \leq d^2(i,n) \rightarrow a[k] = d^3(i,n). \end{array}$$

This template reflects the intuition that the assertion validity depends on the content of a range of array cells.

The tool instantiates the template in 3 s as follows:

$$\begin{array}{c} \forall k : 0 \leq k \leq n-1 \rightarrow a[k] = 0 \\ \forall k : i \leq k \leq n-1 \rightarrow a[k] = 0. \end{array}$$

Note that, compared to the manually created invariant map shown in Section 2, no additional conjuncts are required. We observe that at location $\ell_{3,6}$, the universally quantified index k ranges from i to $n-1$, thus, there is no need for any additional constraints on the value of variable i.

Example Partition. The experimental data for this example is similar to the example InitCheck. Again, no template refinement is required.

6. Discussion

We make two contributions in this paper. First, we propose an abstraction refinement technique that considers program fragments rather than finite paths as counterexamples. Path invariants decouple the problem of synthesizing possibly disjunctive invariants into an efficient search over program paths performed by the CEGAR loop, and a search for program relationships that rule out *all* possible path unwindings of the path program through strong invariant generation techniques. By considering program fragments rather than paths, our refinement algorithms can find expressive assertions on the program state.

Our second contribution is an instantiation of our scheme with path invariant generation based on constraint-based invariant synthesis. In particular, we provide a template-based scheme for synthesizing *universally quantified* invariants (e.g., to reason about array elements) that is sound and complete for our template language.

The initial experiences with the tool are promising. We believe the combination of abstract interpretation based program analysis and abstraction refinement via path invariant generation will provide scalable and precise techniques for proving program assertions. For example, in initial experiments, we could automatically prove a suite of programs (including the ones in Section 2) none of which could be

proved by BLAST, a state-of-the-art software verification tool.

However, path programs are not a panacea for all program verification problems. In particular, we assume simple invariant templates heuristically. It may happen that the program is safe, but the templates guessed by the tool are not strong enough to capture the invariants required for a proof of safety. Also, our technique is geared towards proving safety of systems. Consider a buggy version of Example INITCHECK:

```
for (i = 0; i < 100; i++) {
   a[i] = 1;
}
assert(a[0]==0);
```

In this case, the CEGAR analysis will generate longer and longer counterexample traces, however, the path program is useless since there is no path-invariant map that exhibits the infeasibility of the error path for all unwindings of the loop (there is, in fact, an error trace). We are investigating how our techniques can be combined with techniques that are geared towards falsification.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proc. TACAS*, LNCS 2280, pages 158–172. Springer, 2002.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

[3] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Proc. VMCAI*, LNCS 4349. Springer, 2007.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI*, pages 196–207. ACM, 2003.

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *Proc. CAV*, LNCS 3576, pages 491–504. Springer, 2005.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. VMCAI*, LNCS 3855, pages 427–442. Springer, 2006.

[7] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pages 154–169. Springer, 2000.

[9] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. CAV*, LNCS 2404, pages 265–279. Springer, 2002.

[10] 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.

[11] P. Cousot. Partial completeness of abstract fixpoint checking. In *Proc. SARA*, LNCS 1864, pages 1–15. Springer, 2000.

[12] P. Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice*, LNCS 2772, pages 243–268. Springer, 2003.

[13] P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *Proc. VMCAI*, LNCS 3385. Springer, 2005.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL*, pages 238–252. ACM, 1977.

[15] 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.

[16] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL*, pages 84–96, 1978.

[17] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proc. CAV*, LNCS 1633, pages 160–171. Springer, 1999.

[18] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Proc. TACAS*, LNCS 3920, pages 489–503. Springer, 2006.

[19] 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.

[20] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL'2002*, pages 191–202. ACM, 2002.

[21] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967.

[22] D. Gopan, T. W. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. POPL*, pages 338–350. ACM, 2005.

[23] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.

[24] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proc. PLDI*, pages 376–386. ACM, 2006.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. SPIN*, LNCS 2648, pages 235–239. Springer, 2003.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[29] C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[30] R. Jhala and R. Majumdar. Path slicing. In *Proc. PLDI*, pages 38–47. ACM, 2005.

[31] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proc. TACAS*, LNCS 3920, pages 459–473. Springer, 2006.

[32] D. Kapur. Automatically generating loop invariants using quantifier elimination. In *Proc. Deduction and Applications*, volume 05431. IBFI Schloss Dagstuhl, 2006.

[33] D. Kapur and C. Zarba. A reduction approach to decision procedures. Technical Report TR-CS-2005-44, University of New Mexico, 2005.

[34] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[35] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[36] 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.

[37] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Proc. CAV*, LNCS 3114, pages 135–147. Springer, 2004.

[38] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety*. Springer, 1995.

[39] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proc. ESOP*, LNCS 3444, pages 5–20. Springer, 2005.

[40] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.

[41] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006. to appear.

[42] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 24(3):217–298, 2002.

[43] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Proc. SAS*, LNCS 4134, pages 3–17. Springer, 2006.

[44] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *Proc. SAS*, LNCS 3148, pages 53–68. Springer, 2004.

[45] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. POPL*, pages 318–329. ACM, 2004.

[46] 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.

[47] V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *Proc. CADE*, LNCS 3632, pages 219–234. Springer, 2005.

[48] K. Sohn and A. V. Gelder. Termination detection in logic programs using argument sizes. In *Proc. PODS*, pages 216–226. ACM, 1991.