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 method for automated abstraction refinement which overcomes some limitations of current predicate discovery schemes. In current schemes, the cause of a false alarm 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 alarm —the *spurious counterexample*— as a full-fledged program, namely, a fragment of the original program whose control-flow graph may contain loops and represent unbounded computations. There are two advantages to using such *path programs* as counterexamples for abstraction refinement. First, we can bring the whole machinery of program analysis to bear on path programs, which are typically small compared to the original program. Specifically, we use constraint-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 at once all (possibly infinitely many) infeasible error computations that are represented by a path program. Unlike previous predicate discovery schemes, our method handles loops without unrolling them; it infers abstractions that involve universal quantification and naturally incorporates disjunctive reasoning.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

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 [20, 27], the available techniques for automatically finding useful assertions are still rather limited.

We can broadly classify the techniques for deriving provable assertions into two categories. The first class of methods relies on the user to set up a verification framework —i.e., an *abstract interpretation* [13]— 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 [4, 15], shapes [39]) and invariant templates (e.g., linear arithmetic [43], 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, 26]. 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 [2, 25] —i.e., a *predicate abstraction* [22]— to be tracked by the abstract interpretation. However, a verification framework that consists solely of tracking predicates based on individual infeasible program paths 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) whose finite instantiations are added by each successive refinement step.

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 produces the infeasibility. Such a program is called a *path program*. Since a path program may contain loops, it often represents not a single infeasibility, 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 simple, quantifier-free predicate expressing a relationship between program variables: 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.

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 more likely to scale 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). In other words, path programs decompose a program verification problem into a series of simpler problems about fragments of the original program.

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, uninterpreted functions, and universal quantification over arrays [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] and BLAST [26]. 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 solves 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.

Our approach combines the strengths of predicate abstraction and invariant generation. Predicate abstraction performs well for case analysis-based reasoning, e.g., reasoning that depends on aliasing between pointer variables, or on boolean flags that control the program flow. Invariant generation, by contrast, is strong in arithmetic reasoning and capable of quantified reasoning. Our refinement 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 between program variables without paying the high cost of searching through the space of program invariants for the original program.

There has been much recent interest in *predicate abstraction*-based software model checking [16,22], where the set of predicates is extended as the analysis proceeds by analyzing spurious counterexamples [2, 7, 25, 26, 28, 37]. The incompleteness of traditional implementations of CEGAR-based predicate abstraction is well-known [10, 14], and there have been several attempts to suggest procedures that, in the limit, gain completeness: through carefully choosing widening operations [1], or through carefully orchestrating the proof search in the underlying decision procedures [29]. In contrast, our technique is parameterized by the invariant generation to apply on path programs. There exist invariant generators that are sound and complete modulo the template language, but the invariants required to prove a program may not exist within the template language.

There are several techniques for invariant synthesis, most notably by abstract fixpoint computation on a suitably constructed abstract domain [13, 39], or by a constraint-based analysis that instantiates the parameters of an invariant template [30, 41]. While in our concrete instantiation of path invariants, we have chosen the 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: in the abstract interpretation style [15, 32, 38], and using constraint-based methods [5,9,12]. For quantified invariants involving arrays, there are algorithms that compute fixpoints using a carefully constructed array domain [11,21]. The main drawback of abstract interpretation methods is a high rate of false alarms (due to a lack of precision for efficient analyses), and the main obstacle to applying constraint-based methods is their high computational cost. Constraint-based algorithms often do not scale well to large programs, and therefore most of their applications have been limited to tricky but small programs. Path invariants automatically produce small subproblems, making the application of these techniques feasible by restricting attention to small programs. The overall CEGAR loop combines these subproblems into a proof of correctness of the entire program.

The need for *universally quantified* assertions in the analysis of programs that manipulate unbounded data structures such as arrays is well-known, and several approaches have been suggested to use quantified assertions for predicate abstraction. However, these techniques either require the user to specify the assertions (often with Skolem constants for the quantified variables) [19], or use heuristics to derive quantifiers by generalization from finite examples [34]. In contrast, we apply an invariant-generation technique that is sound and complete for a class of invariant templates [3]. The language of our invariants is the combined theory of linear arithmetic and uninterpreted functions, extended with a universally quantified array fragment [6]. For templates outside the given template language, we can still apply our algorithm and generate sound invariants, but 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 [36, 40]. We can use path invariants to implement such a manipulation in a property-guided way; see Section 5.

2. Examples

We illustrate the use of path invariants 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 identify relevant predicates that eliminate not only a given counterexample path π , but also all counterexample paths that can be obtained from π by unwinding loops.

The second example is a program INITCHECK, which manipulates arrays. Its correctness proof requires loop invariants that contain universal quantifiers, and the automatic discovery of such invariants has been posed as a challenge in previous work on predicate abstraction and discovery [29,37]. Path invariants identify rel-

```
          void forward(int n) {
             int i, n, a, b;

A:           assume( n >= 0 );
             i = 0; a = 0; b = 0;
B:           while( i < n ) {
                if( ... ) {
C:                 a = a+1;
                   b = b+2;
                } else {
D:                 a = a+2;
                   b = b+1;
                }
E:              i = i+1;
             }
F:           assert( a+b == 3*n );
          }
```

(a)

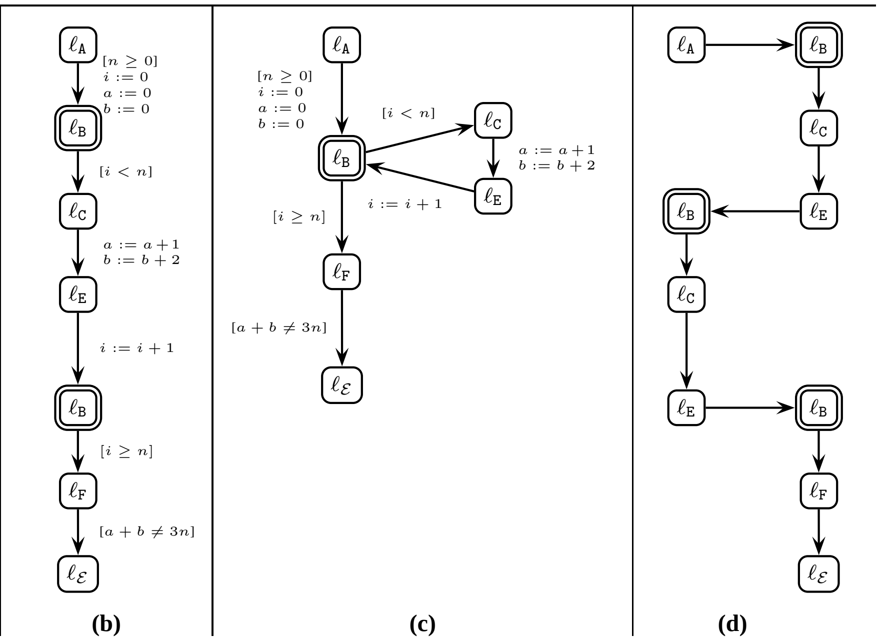


Figure 1. Program FORWARD illustrates the discovery of relevant predicates that prevent loop unwinding: **(a)** program; **(b)** counterexample path; **(c)** CFG of the path program that is extracted from the counterexample path; and **(d)** potential new counterexample path resulting from loop unrolling when path invariants are not tracked. In the path and CFG representation, we use $[\cdot]$ to denote assumptions that represent conditional control statements of the program. As usual, updates are denoted by $:=$. Double circles denote locations at entry points of nested blocks of a program, i.e., entry points of loops.

evant universally quantified formulas 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 some of the computations of the original program, the corresponding path invariants may be smaller, and represent only parts of the set of reachable program states. The corresponding global invariants, which cover all reachable states, can be derived from a combination of several path invariants. Thus, path invariants allow us to implement a lazy construction of program invariants, which is guided by counterexamples.

2.1 Example FORWARD: Capture Arbitrary Loop Unwinding

Our first example is 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 using path invariants over existing methods. A standard counterexample-guided abstraction refinement (CEGAR) algorithm implemented in a tool like BLAST 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 reachability analysis (first phase of CEGAR) checks if there is a path in the control-flow graph (CFG) that leads to an error location, where the assertion is violated. There are such paths in the CFG, and Figure 1**(b)** shows one such counterexample path, which traverses the `while`-loop once, takes the `then`-branch in the body of the loop, and then violates the assertion after leaving the loop. Notice that while this is a syntactic path in the CFG, the counterexample path is *spurious*, that is, it cannot be executed by the program.

The second phase of CEGAR is to check if the produced counterexample path is genuine or spurious, and if spurious, to proceed with abstraction refinement, i.e., to find additional predicates that rule out the path. The counterexample path is translated into a logical formula called the *path formula*, which is satisfiable iff the counterexample path can be executed in the program [33]. 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 control locations:

$$\begin{array}{ll}
n_0 \geq 0 \wedge i_1 = 0 \wedge a_1 = 0 \wedge b_1 = 0 \wedge & \ell_A \to \ell_B \\
i_1 < n_0 \wedge & \ell_B \to \ell_C \\
a_2 = a_1 + 1 \wedge b_2 = b_1 + 2 \wedge & \ell_C \to \ell_E \\
i_2 = i_1 + 1 \wedge & \ell_E \to \ell_B \\
i_2 \geq n_0 \wedge & \ell_B \to \ell_F \\
a_2 + b_2 \neq 3n_0 & \ell_F \to \ell_\mathcal{E}
\end{array}$$

The formula is unsatisfiable, because there is no initial valuation of program variables that leads to a program execution along the counterexample path.

In the third phase of CEGAR, predicates are extracted from the unsatisfiable path formula, and added to the predicate abstraction. This refined abstraction ensures that the new predicates are tracked during subsequent reachability analyses, and that therefore the current counterexample path will not reoccur. One way to discover predicates 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 [25, 37], but this does not change our argument below.) For our counterexample path, 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 given counterexample path, the next round of reachability analysis encounters a longer counterexample path which is obtained by unwinding the loop one more time, namely, the path shown in Figure 1**(d)**. This new counterexample path is eliminated by tracking in addition the predicates in the set

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

In general, in the k-th refinement round, we find the set of predicates $\{i = k, a = k, b = 2k\}$, and the method does not terminate.

Path Invariants. Our new refinement approach is based on identifying path invariants. Path invariants are not inferred from path formulas, but from special *path programs*, whose construction is guided by the statements that appear along a counterexample path. The path program for the counterexample path in Figure 1**(b)** is shown in Figure 1**(c)**. We observe that the path program contains only control locations that are traversed by the counterexample path. Its statements are taken from the counterexample path, and its CFG captures the counterexample path as well as its unwindings. We shall define formally how path programs are constructed in Section 3. The counterexample path passes two times through the control location $\ell_{\mathtt{B}}$, which labels the loop entry. So the path program has a loop $\ell_{\mathtt{B}} \to \ell_{\mathtt{C}} \to \ell_{\mathtt{E}} \to \ell_{\mathtt{B}}$ in its CFG at location $\ell_{\mathtt{B}}$.

To refine the analysis so that the *family* of counterexample paths represented by the path program are all refuted at once, we apply invariant-generation techniques. Since there are loops in the program, we can no longer construct a path formula that is linear in the length of the counterexample. Instead, we look for *invariant maps*. A *path-invariant* map is a mapping from the locations of a path program to formulas such that the following two conditions hold: (initiation) the initial location of the path program is mapped to the formula $true$, and (inductiveness) for each pair of locations ℓ and ℓ' with an edge (ℓ, ρ, ℓ') in the path program, the successor of the formula at ℓ with respect to the program operation ρ implies the formula at ℓ'. The path-invariant map is *safe* if the error location (i.e., the location that violates the assertion that is to be proved) is mapped to the formula $false$. Notice that an invariant map of a path program need not be an invariant map of the original program when the set of locations is extended, because it may violate the inductiveness requirement.

In our example we can generate invariants in arithmetic domains, e.g., by applying methods described in [9, 43], and obtain the following path-invariant map:

$$\begin{aligned}
\eta.\ell_{\mathtt{A}} &= \quad true \\
\eta.\ell_{\mathtt{B}} &= \quad 3i = a + b \wedge a + b \leq 3n \\
\eta.\ell_{\mathtt{C}} &= \quad 3i = a + b \wedge a + b \leq 3n \\
\eta.\ell_{\mathtt{E}} &= \quad 3i + 3 = a + b \wedge a + b - 3 \leq 3n \\
\eta.\ell_{\mathtt{F}} &= \quad a + b = 3n \\
\eta.\ell_{\mathcal{E}} &= \quad false
\end{aligned}$$

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

Furthermore, any spurious counterexample path that is obtained by traversing the path program is eliminated by tracking these formulas. For example, consider a potential unwinding of the original counterexample path, which traverses the loop twice, as shown in Figure 1**(d)**. When following this path and reaching the control location $\ell_{\mathtt{B}}$ for the first time, a program analysis tracking the formulas from the path-invariant map computes an overapproximation of the reachable states at $\ell_{\mathtt{B}}$ that is at least as strong as the formula defined by the map at $\ell_{\mathtt{B}}$. Since the path-invariant map is inductive and safe, we conclude that the overapproximation computed for the second visit to the location $\ell_{\mathtt{B}}$ is again as strong as the formula assigned to $\ell_{\mathtt{B}}$. 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 original counterexample within the CFG of the path program will not produce a new (spurious) counterexample. This means that any path whose sequence of visited control locations is in the language defined by the regular expression $\ell_{\mathtt{A}}\ell_{\mathtt{B}}(\ell_{\mathtt{C}}\ell_{\mathtt{E}}\ell_{\mathtt{B}})^*\ell_{\mathtt{F}}\ell_{\mathcal{E}}$ 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 formulas obtained from path invariants, relies on the completeness of abstract interpretation [10].

2.2 Example InitCheck: Universally Quantified Predicates

The previous example showed how path programs can be used to refute a family of counterexample paths 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., data stored in container data structures like arrays, requires universally quantified formulas. Usually, these formulas 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 invariants based on (finite) counterexample paths. Namely, such paths can expose only a bounded number of data items that are stored in the data structure. Thus, it is difficult to derive and formally justify universal quantification over discovered predicates. However, from path programs we can infer quantified invariants by simultaneously considering all unrollings of a loop. The next example demonstrates how an invariant-synthesis algorithm for inferring quantified invariants [3] can be applied to path programs.

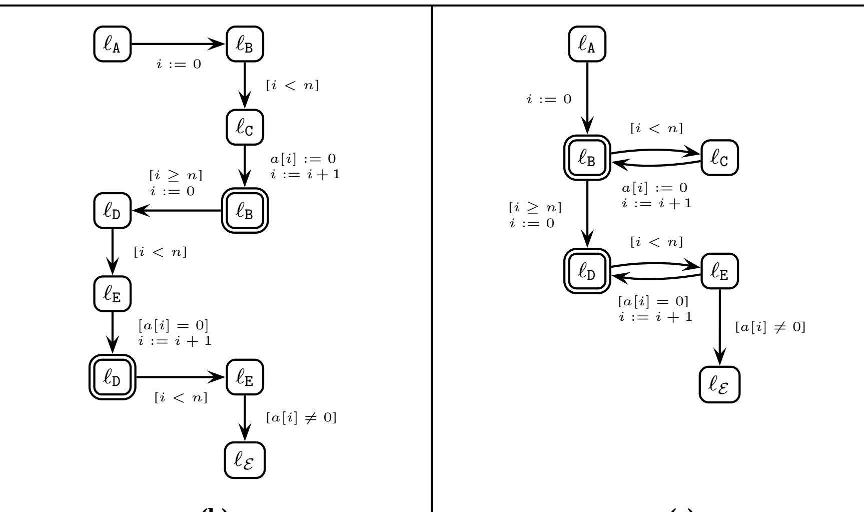
Consider the program InitCheck from Figure 2**(a)**, which initializes an array to 0, and then checks that all elements in the array are 0. We wish to prove that all assertions hold.

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 loop once. From the first part of the counterexample path (traversal of the first loop) we can conclude that the first element in the array is initialized to 0, and discover the predicate $a[0] = 0$. Then, by considering this fact in the second part in the counterexample path, 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, tracking the predicate $a[0] = 0$ eliminates only this particular counterexample path. It does not eliminate the longer counterexample path which traverses each loop twice; this would require tracking the predicate $a[1] = 0$. In fact, counterexample-based predicate-abstraction refinement is likely to generate an infinite family of predicates $a[i] = 0$, one for each $i \geq 0$. 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 counterexample paths created by loop unwinding will suffice to prove the program correct. We need the universally quantified formula $\forall k : 0 \leq k < n \to a[k] = 0$ to verify InitCheck.

```
     void init_check(int *a, int  n) {
        int i;
A:
B:      for( i = 0; i < n; i++ ) {
C:         a[i] = 0;
        }
D:      for( i = 0; i < n; i++ ) {
E:         assert( a[i] == 0 );
        }
     }
```

(a)



(b) **(c)**

Figure 2. Program InitCheck illustrates the discovery of universally quantified invariants for the challenge example from [29, 37]: **(a)** program; **(b)** counterexample path; **(c)** CFG of the path program that is extracted from the counterexample path.

Path Invariants. Justification of the universal quantification requires consideration of all possible paths that traverse the initialization and checking loops located at ℓ_B and ℓ_D, respectively. We use a path program to represent this family of paths. The path program extracted from the original counterexample path is shown in Figure 2**(c)**. Using this path program, we can provide a systematic justification of universal quantification by deriving path invariants. The technical complication is that we have to infer invariant maps that map certain locations to universally quantified formulas.

An inductive invariant map, say η, for our path program needs to assert that at location ℓ_E the contents of $a[i]$ is 0. Note that the transition to the error location $\ell_{\mathcal{E}}$, which is taken from ℓ_E if $a[i] \neq 0$ holds, appears within a loop that iteratively increments the value of i. Hence, the formula assigned by η to the location ℓ_E must imply $a[i] = 0$ for all values of i that are reachable from ℓ_E, that is, all values of i in the interval from 0 to $n-1$. We observe that the first loop assigns 0 to an array cell $a[i]$ for each value of i that is subsequently checked in the second loop.

We compute the path-invariant map η that formalizes the above reasons for the non-reachability of the error location in the path program. (See [3] for a discussion of algorithms for computing invariants that contain universal quantification.) The formulas in η restrict the value of the counter variable i and contain universally quantified statements about the contents of the initialized cells of the array a. The formulas for the locations in the first loop refer only to the array contents up to the position i, whereas the formulas for the second loop refer to each array element between 0 and $n-1$:

$$\begin{aligned}
\eta.\ell_A &= \quad \mathit{true} \\
\eta.\ell_B &= \quad \forall k : 0 \leq k < i \rightarrow a[k] = 0 \\
\eta.\ell_C &= \quad \forall k : 0 \leq k < i \rightarrow a[k] = 0 \\
\eta.\ell_D &= \quad \forall k : 0 \leq k < n \rightarrow a[k] = 0 \\
\eta.\ell_E &= \quad \forall k : i \leq k < n \rightarrow a[k] = 0 \\
\eta.\ell_{\mathcal{E}} &= \quad \mathit{false}.
\end{aligned}$$

By tracking the four formulas in the range of the path-invariant map, we are guaranteed that all potential counterexample paths that visit a sequence of control locations from the set defined by the regular expression $\ell_A \ell_B (\ell_C \ell_B)^* \ell_D (\ell_E \ell_D)^* \ell_E \ell_{\mathcal{E}}$ are eliminated.

2.3 Example Partition: Incremental Construction

Path invariants identify *local* reasons that refute a family of counterexample paths. 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 family of paths to a violation of the assertion. We now illustrate how path invariants can be used within a CEGAR framework to incrementally construct global invariant maps, using path programs derived from different counterexample paths to learn additional information.

Consider the program Partition in Figure 3, which partitions the elements of an input array a into two arrays ge and lt, which contain, respectively, the elements of a greater or equal to 0, and less than 0. In order to prove the assertions, we need a loop invariant at location B which is the conjunction of

$$\forall k : 0 \leq k < \mathit{gelen} \rightarrow \mathit{ge}[k] \geq 0 \qquad (1)$$

$$\forall k : 0 \leq k < \mathit{ltlen} \rightarrow \mathit{lt}[k] < 0 \qquad (2)$$

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

These invariants, however, are not enough to prove the assertions, and a second counterexample path is found. This path traverses the `else`-branch of the conditional in the `for`-loop. Again, the path program is similar to the path program from Example InitCheck. This time, the path-invariant map generates the second conjunct of the loop invariant, i.e., Equation (2). Together, the conjuncts suffice to prove the assertions. Thus, the CEGAR algorithm breaks the search for global program invariants (as performed by invariant synthesis techniques) into several searches for individual components of the invariant, thus restricting the searches to smaller spaces.

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

      A:      gelen = 0; ltlen = 0;
      B:      for( i = 0; i < n; i++ ) {
                 if( a[i] >= 0 ) {
      C:            ge[gelen] = a[i];
                    gelen++;
                 } else {
      D:            lt[ltlen] = a[i];
                    ltlen++;
                 }
              }
      E:      for( i = 0; i < gelen; i++ ) {
      F:         assert( ge[i] >= 0 );
              }
      G:      for( i = 0; i < ltlen; i++ ) {
      H:         assert( lt[i] < 0 );
              }
        }
```

Figure 3. Program PARTITION illustrates how reasoning over several path programs can be combined.

3. Path Programs and Invariants

Programs. We assume an abstract representation of programs by transition systems [35]. A *program* $P = (X, L, \ell_0, \mathcal{T}, \ell_\mathcal{E})$ consists of a set X of variables, a set L of control locations, an initial location $\ell_0 \in L$, a set $\mathcal{T}$ of transitions, and an error location $\ell_\mathcal{E} \in L$. Each transition $\tau \in \mathcal{T}$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in L$ are control locations, and ρ is a constraint over free variables from $X \cup X'$. The variables from X denote values at control location ℓ, and the variables from X' denote the values of the variables from X at control location ℓ'. We assume that the error location $\ell_\mathcal{E}$ does not have any outgoing transitions. The sets of locations and transitions naturally define a directed graph, called the *control-flow graph* (CFG) of the program. (Note that we put the transition constraints at the edges of the graph.)

A *state* of the program P is a valuation of the variables from X. The set of all states is denoted $\mathsf{val}.X$. 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 $\langle \ell_0, s_0\rangle, \langle \ell_1, s_1\rangle, \dots, \langle \ell_k, s_k\rangle \in (L \times \mathsf{val}.X)^*$, where ℓ_0 is the initial location and for each $i \in \{0, \dots, k-1\}$, there is a transition $(\ell_i, \rho, \ell_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A state s is *reachable* at location ℓ if $\langle \ell, s\rangle$ appears in some computation. The program is *safe* if the error location $\ell_\mathcal{E}$ appears in no computation. A *path* of the program P is a sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \dots, (\ell_{k-1}, \rho_{k-1}, \ell_k)$ of transitions, where ℓ_0 is the initial location. The path π is *feasible* if there is a computation $\langle \ell_0, s_0\rangle, \dots, \langle \ell_k, s_k\rangle$ such that for each $i \in \{0, \dots, k-1\}$, we have $(s_i, s_{i+1}) \models \rho_i$. If the path π ends at the error location, i.e., $l_k = \ell_\mathcal{E}$, then we call π an *error path* (or *counterexample path*). Feasible counterexample paths are referred to as *genuine*; infeasible ones, as *spurious*.

Invariants. An *invariant* of P at a location $\ell \in L$ is a set of states containing the states reachable at ℓ. An *invariant map* is a function η from L to formulas over program variables from X such that the following conditions hold:

Initiation: For the initial location ℓ_0, we have $\eta.\ell_0 = \mathit{true}$.
Inductiveness: For each $\ell, \ell' \in L$ such that $(\ell, \rho, \ell') \in \mathcal{T}$, the formula $\eta.\ell \wedge \rho$ implies $(\eta.\ell')'$. Here, $(\eta.\ell')'$ is the formula obtained by substituting variables from X' for the variables from X in $\eta.\ell'$.
Safety: For the error location $\ell_\mathcal{E}$, we have $\eta.\ell_\mathcal{E} = \mathit{false}$.

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. We consider a program $P = (X, L, \ell_0, \mathcal{T}, \ell_\mathcal{E})$ together with an error path $\pi = (\ell_0, \rho_0, \ell_1), \dots, (\ell_{k-1}, \rho_{k-1}, \ell_\mathcal{E})$. Given P and π, we construct the *path program* $P^\pi = (X^\pi, L^\pi, \ell_0^\pi, \mathcal{T}^\pi, \ell_\mathcal{E}^\pi)$ such that

- the set of variables remains the same: $X^\pi = X$;
- the program locations are exactly those that are visited by the path: $L^\pi = \{\ell_1, \dots, \ell_{k-1}, \ell_\mathcal{E}\}$;
- the initial and error locations remain the same: $\ell_0^\pi = \ell_0$ and $\ell_\mathcal{E}^\pi = \ell_\mathcal{E}$;
- the set of transitions is restricted to the transitions that are taken along the path: $\mathcal{T}^\pi = \{(\ell_0, \rho_0, \ell_1), \dots, (\ell_{k-1}, \rho_{k-1}, \ell_\mathcal{E})\}$.

Intuitively, the paths of the path program P^π include the error path π, and in addition all paths that result from π by unrolling it following the control-flow graph of P. Hence the path program P^π may traverse some loops that are traversed by π more often, but it contains no transitions that do not occur in π.

An invariant map η^π for a path program P^π is called a *path-invariant map*.

Computation of Path Invariants. There are several methods to generate path invariants for a path program, e.g., via abstract interpretation with specialized domains (cf. [4, 15, 21, 32, 38, 39]), and via reduction to constraint solving (cf. [3, 5, 9, 12, 30, 41, 42, 43]). In our implementation of path invariants, we have used *template-based invariant generation* [5, 9]. By exploiting recent advances in reasoning about the hierarchical combination of theories [6, 31, 44], our algorithm [3] is able to generate invariants over the combined theory of arithmetic and uninterpreted function symbols with universal quantification. This combined theory is known to be expressive enough for a wide variety of software verification problems [2, 18, 19, 24, 26], including reasoning about arrays [6].

In template-based invariant synthesis, 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 constraint over program variables. An example for a simple template constraint over the program variables i and n is

$$p_i \cdot i + p_n \cdot n \leq p,$$

where p_i, p_n, and p are unknown parameters whose values need to be determined. This template denotes a set of formulas that can be obtained by giving values to the parameters, e.g., $2i - 3n \leq 5$. The crux of the template-based approach consists in defining and solving a system of constraints over the template's parameters such that the resulting values yield an inductive invariant map.

The constraints over the parameters of the template encode the initiation, inductiveness, and safety conditions of invariant maps. For example, for each transition (ℓ, ρ, ℓ') of a program, we generate the constraint that states that the invariant template at ℓ conjoined with the transition relation ρ implies the invariant template at ℓ'. This constrains the values of the parameters of the templates at ℓ and ℓ' to values that ensure the inductiveness of the generated invariants. In addition, we add constraints that the template at the initial location ℓ_0 is true, and the template at the error location $\ell_\mathcal{E}$

implies $false$. A solution to the constraints then provides values for the template parameters. For the combined theory of linear arithmetic and uninterpreted function symbols, all constraints fall into a class that can be solved using the constraint solving techniques described in [9].

In our implementation, we use the algorithm from our previous work [3] to infer *universally quantified* invariants for array programs, in addition to linear arithmetic and uninterpreted function symbols. We construct a suitable template by analyzing a given path program. If the program contains an assertion that is iteratively checked within a loop, then we add a universally quantified implication to the template. The right-hand-side of the implication contains a generalization of the assertion. The position at which the assertion reads from the array becomes a fresh, universally quantified variable. For the left-hand-side of the implication we choose a conjunction of linear inequalities over the program variables and the fresh variable. For example, given the program InitCheck and the path program from Figure 2**(c)**, we construct the templates φ and ψ for the locations $\ell_{\mathtt{B}}$ and $\ell_{\mathtt{D}}$, respectively, as

$$\begin{aligned}\varphi &= \quad (\forall k : p^1(i,n) \le k \le p^2(i,n) \rightarrow a[k] = p^3(i,n)),\\ \psi &= \quad (\forall k : q^1(i,n) \le k \le q^2(i,n) \rightarrow a[k] = q^3(i,n)),\end{aligned}$$

where

$$\begin{aligned}p^r(i,n) &= \quad p^r_i \cdot i + p^r_n \cdot n + p^r \quad \text{for each } r \in \{1,\dots,3\},\\ q^r(i,n) &= \quad q^r_i \cdot i + q^r_n \cdot n + q^r \quad \text{for each } r \in \{1,\dots,3\}.\end{aligned}$$

Our implementation of the quantified invariant generator finds an instantiation of the parameters that yields the path-invariant map shown in Section 2.2.

4. CEGAR with Path Invariants

We apply path invariants in a predicate abstraction-based CEGAR loop, where path invariants are used to suggest formulas to refine the predicate abstraction. However, our technique to consider *path programs* (instead of paths) as counterexamples can be used in any CEGAR-based program analysis.

A *predicate abstraction* Π is a function that maps each control location to a set of formulas over the program variables, namely, those formulas whose truth values are tracked by the program analysis. Conceptually, the CEGAR algorithm has three phases [2, 8]: abstract reachability, counterexample analysis, and abstraction refinement. The *abstract-reachability* phase tries to construct a safety proof for the program by unwinding the CFG into a labeled tree, where each node of the unwinding is annotated with an abstract state. The *abstract state* at a node with location ℓ is a boolean combination of the formulas in $\Pi.\ell$, which represents an overapproximation of the set of reachable states of the program when it executes the path from the root of the tree to the current node. The root is labeled with the abstract state $true$. If an abstract state at a node v implies the abstract state at another node with the same location, then the unwinding stops at the node v. This unwinding process produces an *abstract reachability tree* (ART); see [26] for a formal definition.

If the abstract reachability tree does not contain the error location, then a safety proof is found, and the algorithm terminates. Otherwise, the algorithm moves to the *counterexample-analysis* phase. In this phase, a *counterexample path* (i.e., a path from the root to the error location) is chosen from the ART, and the algorithm checks if this counterexample is genuine (i.e., if the error path is feasible). For implementing this check, a logical formula is constructed from the counterexample path which is satisfiable iff the path is feasible. If the formula is satisfiable, then a bug is found and the algorithm stops. Otherwise, the algorithm proceeds with the *abstraction-refinement* phase. Now, instead of discovering new predicates for the predicate abstraction from the spurious counterexample path, using for example interpolation-based approaches [17, 25, 37], we construct the path program P^π from the counterexample path π. Then we use an invariant-synthesis algorithm to produce a path-invariant map η^π for P^π [3]. This invariant map is used to refine the predicate abstraction: we add all atomic predicates that appear in $\eta.\ell$ to the predicate abstraction $\Pi.\ell$ at each location ℓ, in this process treating universally quantified formulas as atomic predicates. Note that if some unwinding of the error path π is feasible, then no invariant map η^π can be found.

```
     void disj() {
       int x, y;

A:     x = 0; y = 50;
B:     while( x < 100 ) {
         if ( x < 50 ) {
C:         x = x+1;
         } else {
D:         x = x+1;
           y = y+1;
         }
       }
E:     assert( y >= 100 );
F:     assert( y <= 100 );
     }
```

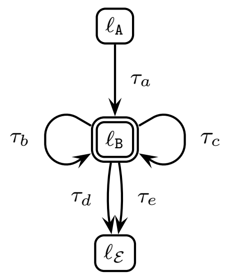


Figure 4. Program Disj, which requires reasoning with disjunctive invariants (cf. [23]), and its CFG. We omit intermediate (non-cutpoint) locations. The transition τ_a initializes the variables before entering the loop. The transitions τ_b and τ_c traverse the loop by taking the positive and negative branch, respectively. The transitions τ_d and τ_e exit the loop and violate the first and second assertion, respectively.

After abstraction refinement, the algorithm proceeds with another abstract-reachability phase, this time tracking more predicates (including universally quantified formulas). The three phases are repeated until either a proof or a bug is found (or, since the problem is undecidable, the loop does not terminate).

The key property of the refinement step using path invariants is that formulas that appear in path-invariant maps rule out *all* counterexample paths that arise from arbitrary unwindings of loops in the path program. This is in contrast to the existing implementations of CEGAR, where each refinement step only guarantees to remove a single counterexample path, and hence, may get stuck in removing infinitely many counterexample paths that result from unrolling a loop. In the following theorem, let $\mathsf{Reach}.\Pi$ denote the set of all rooted paths in the abstract reachability tree that is constructed for a predicate abstraction Π.

Theorem 1 (Refinement Progress). *Consider an error path π of the program P, and an invariant map η^π for the corresponding path program P^π. For every predicate abstraction Π such that for each $\ell \in L^\pi$ we have $\eta^\pi.\ell \subseteq \Pi.\ell$, the set $\mathsf{Reach}.\Pi$ of paths in the abstract reachability tree does not contain any paths that are constructed only from transitions in P^π.*

5. Disjunctive Reasoning with Path Invariants

In this section, we show that path invariants can facilitate disjunctive reasoning when dealing with program invariants. We illustrate the need for disjunctive reasoning using the example program Disj from Figure 4. When attempting to verify this program, our invariant-generation algorithm fails to compute an invariant map for the path program extracted from a counterexample path that traverses both branches of the loop and violates the second asser-

```
A:  while(...) {
      ...
      if(...) {
        ...
B:      break;
      } else {
C:      ...
D:      while(...) {
          ...
        }
E:      ...
      }
F:    ...
    }
G:  ...
```

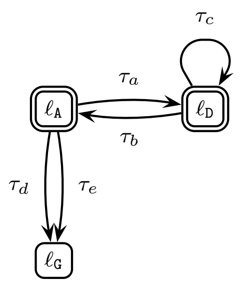


Figure 5. Control-flow graph that contains two nested blocks (we omit intermediate, non-cutpoint locations). The inner block corresponds to the `while`-loop at location $\ell_{\mathtt{D}}$. Its entry/exit point is the location $\ell_{\mathtt{D}}$. The outer block corresponds to the `while`-loop at location $\ell_{\mathtt{A}}$. Its entry point is the location $\ell_{\mathtt{A}}$, and its exit points are $\ell_{\mathtt{A}}$ and $\ell_{\mathtt{B}}$.

tion. Any attempts to increase the number of conjuncts in the template fail. This is because proving the validity of the assertions requires an invariant map η that assigns a disjunctive linear invariant $(x \leq 50 \wedge y = 50) \vee (50 < x \leq 100 \wedge x = y)$ to location $\ell_{\mathtt{B}}$. Such a disjunctive invariant is not captured by the conjunctive templates used in [3]. We propose *disjunctive path programs*, an extension of path programs, to support disjunctive reasoning.

Disjunctive Path Programs. We assume that control-flow graphs are structured into blocks. Each *block* is a strongly connected component of the CFG. Each block is induced by a control-flow statement that allows code to be executed repeatedly; examples of such statements include `while`-loop and `for`-loop statements. We exclude `goto`-statements from consideration to simplify the exposition. Thus, each block B has a single entry location, denoted by $\mathsf{Entry}.B$, which is also an exit location. The block B contains all nodes and edges of the CFG that are strongly connected to $\mathsf{Entry}.B$. A block may have additional exit locations, which correspond to `break`-statements. Nested loop statements induce nested blocks, i.e., strongly connected components that are not maximal. We illustrate blocks, nesting, and entry/exit locations in Figure 5. The control-flow graph shows transition τ_a, which connects the locations $\ell_{\mathtt{A}}$ and $\ell_{\mathtt{D}}$ by following the negative branch of the `if`-statement and traversing the non-cutpoint location $\ell_{\mathtt{C}}$. The body of the inner loop corresponds to τ_c, which is a self-loop at the location $\ell_{\mathtt{D}}$. The transition τ_b connects the inner loop with $\ell_{\mathtt{A}}$. The transition τ_d exits the outer `while`-loop by violating its condition. The transition τ_e exits the outer loop by taking the `break`-statement.

Before giving a formal definition of disjunctive path programs (DPPs), we informally describe how they differ from (plain) path programs, as introduced in Section 3. In DPPs, each control location visited by a path appears several times, indexed by its position in the path. Thus, the invariant map for DPPs can assign different formulas to the same location at different positions. Furthermore, DPPs contain copies of transitions that are repeatedly reachable from a given location (i.e., lie within a common block) at each position of the location. See Figures 6 and 7 for examples of DPPs that are constructed according to the definition below.

We consider a program $P = (X, L, \ell_0, \mathcal{T}, \ell_{\mathcal{E}})$ together with an error path $\pi = (\ell_0, \rho_0, \ell_1), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_{\mathcal{E}})$. We write $L.\pi$ for the set $\{\ell_0, \ldots, \ell_{k-1}, \ell_{\mathcal{E}}\}$ of locations visited by the path π, and $\mathcal{T}.\pi$ for the set $\{(\ell_0, \rho_0, \ell_1), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_{\mathcal{E}})\}$ of transitions in π. Let $\mathsf{Blocks}.\pi$ be the set of blocks of the control-flow graph of the program P which contain the set of locations $L.\pi$. Given P and π, we construct the *disjunctive path program* $P^{\vee\pi} = (X^{\vee\pi}, L^{\vee\pi}, {\ell_0}^{\vee\pi}, \mathcal{T}^{\vee\pi}, {\ell_{\mathcal{E}}}^{\vee\pi})$ such that

- the set of variables remains the same: $X^{\vee\pi} = X$;
- the program locations contain two copies for each location to model loop unwindings, and are paired with labels for the positions in the path: $L^{\vee\pi} = \{\ell, \hat{\ell} \mid \ell \in L.\pi\} \times \{0, \ldots, k\}$;
- the initial location is the first location in the path π, labeled with 0, that is, ${\ell_0}^{\vee\pi} = (\ell_0, 0)$;
- for each position $i \in [0..k-1]$ of the path π, the disjunctive path program contains the transition $((\ell_i, i), \rho_i, (\ell_{i+1}, i+1))$; moreover, if $\pi.i$ is a back-edge in the CFG, i.e., if there is a block $B \in \mathsf{Blocks}.\pi$ such that $\ell_i \in B$ and $\ell_{i+1} = \mathsf{Entry}.B$, then the DPP contains also the following transitions:
  - $((\ell_{i+1}, i+1), \rho_{X'=X}, (\hat{\ell}_{i+1}, i+1))$,
  - $((\hat{\ell}_j, i+1), \rho_j, (\hat{\ell}_{j+1}, i+1))$ for each $(\ell_j, \rho_j, \ell_{j+1}) \in \mathcal{T}.\pi$ such that $j \leq i$ and both ℓ_j and ℓ_{j+1} are in B,
  - $((\hat{\ell}_{i+1}, i+1), \rho_{X'=X}, (\ell_{i+1}, i+1))$,

  where $\rho_{X'=X}$ denotes the constraint that preserves the valuation of all variables, i.e., $\rho_{X'=X} = \bigwedge_{x \in X} x' = x$;
- the error location is ${\ell_{\mathcal{E}}}^{\vee\pi} = (\ell_{\mathcal{E}}, k)$.

Intuitively, the paths of the disjunctive path program $P^{\vee\pi}$ 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 unwindings of loops. Hence, the path program $P^{\vee\pi}$ may iterate some loops that are traversed by π more often, but it contains no transition that does not occur in π. Consider, e.g., an error path π of the program from Figure 5 that first traverses the `while`-loop without going into the inner `while`-loop, then traverses the inner `while`-loop once, and finally leaves the outer `while`-loop by executing the `break`-statement (we write $\ell_{\mathcal{E}}$ for $\ell_{\mathtt{G}}$):

$$\ell_{\mathtt{A}} \to \ell_{\mathtt{C}} \to \ell_{\mathtt{D}} \to \ell_{\mathtt{E}} \to \ell_{\mathtt{F}} \to$$
$$\ell_{\mathtt{A}} \to \ell_{\mathtt{C}} \to \ell_{\mathtt{D}} \to \ell_{\mathtt{D}} \to \ell_{\mathtt{E}} \to \ell_{\mathtt{F}} \to$$
$$\ell_{\mathtt{A}} \to \ell_{\mathtt{B}} \to \ell_{\mathcal{E}}$$

This counterexample path and the corresponding disjunctive path program are shown in Figure 6. The nested blocks of this path are $B_1 = \{\ell_{\mathtt{A}}, \ldots, \ell_{\mathtt{F}}\}$ and $B_2 = \{\ell_{\mathtt{D}}\}$, with B_2 being nested in B_1. The complete set of transitions of the DPP $P^{\vee\pi}$ is

$$\begin{array}{l}
((\ell_{\mathtt{A}}, 0), \rho_a, (\ell_{\mathtt{D}}, 1)),\\
((\ell_{\mathtt{D}}, 1), \rho_b, (\ell_{\mathtt{A}}, 2)),\\
\\
\qquad ((\ell_{\mathtt{A}}, 2), \rho_{X'=X}, (\hat{\ell}_{\mathtt{A}}, 2)),\\
\qquad ((\hat{\ell}_{\mathtt{A}}, 2), \rho_a, (\hat{\ell}_{\mathtt{D}}, 2)),\\
\qquad ((\hat{\ell}_{\mathtt{D}}, 2), \rho_b, (\hat{\ell}_{\mathtt{A}}, 2)),\\
\qquad ((\hat{\ell}_{\mathtt{A}}, 2), \rho_{X'=X}, (\ell_{\mathtt{A}}, 2)),\\
\\
((\ell_{\mathtt{A}}, 2), \rho_a, (\ell_{\mathtt{D}}, 3)),\\
((\ell_{\mathtt{D}}, 3), \rho_c, (\ell_{\mathtt{D}}, 4)),\\
\\
\qquad ((\ell_{\mathtt{D}}, 4), \rho_{X'=X}, (\hat{\ell}_{\mathtt{D}}, 4)),\\
\qquad ((\hat{\ell}_{\mathtt{D}}, 4), \rho_c, (\hat{\ell}_{\mathtt{D}}, 4)),\\
\qquad ((\hat{\ell}_{\mathtt{D}}, 4), \rho_{X'=X}, (\ell_{\mathtt{D}}, 4)),\\
\\
((\ell_{\mathtt{D}}, 4), \rho_b, (\ell_{\mathtt{A}}, 5)),\\
\\
\qquad ((\ell_{\mathtt{A}}, 5), \rho_{X'=X}, (\hat{\ell}_{\mathtt{A}}, 5)),\\
\qquad ((\hat{\ell}_{\mathtt{A}}, 5), \rho_a, (\hat{\ell}_{\mathtt{D}}, 5)),\\
\qquad ((\hat{\ell}_{\mathtt{D}}, 5), \rho_c, (\hat{\ell}_{\mathtt{D}}, 5)),\\
\qquad ((\hat{\ell}_{\mathtt{D}}, 5), \rho_b, (\hat{\ell}_{\mathtt{A}}, 5)),\\
\qquad ((\hat{\ell}_{\mathtt{A}}, 5), \rho_{X'=X}, (\ell_{\mathtt{A}}, 5)),\\
\\
((\ell_{\mathtt{A}}, 5), \rho_e, (\ell_{\mathcal{E}}, 6)).
\end{array}$$

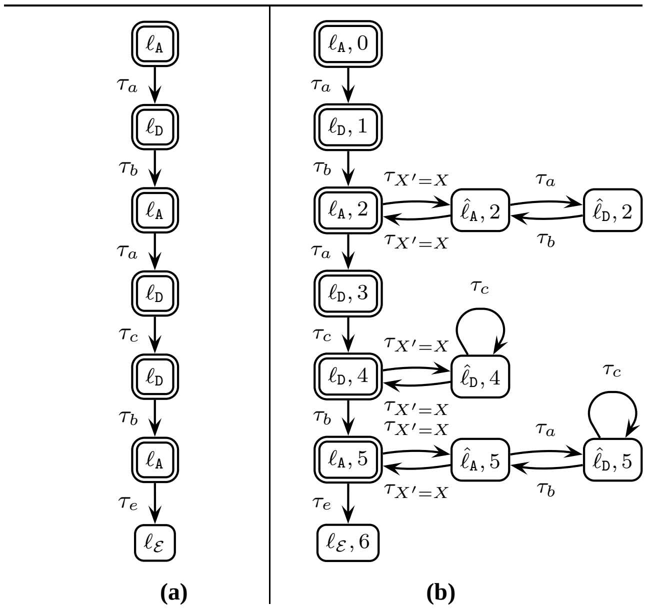


Figure 6. Counterexample and CFG of the corresponding DPP.

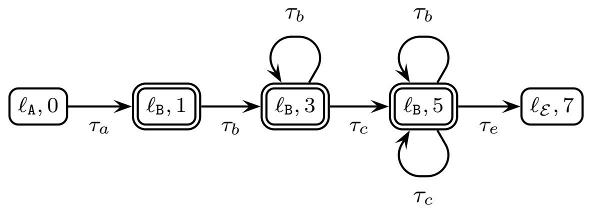


Figure 7. Disjunctive path program for the example from Figure 4 and a path that traverses the locations $\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{C}}, \ell_{\mathtt{B}}, \ell_{\mathtt{D}}, \ell_{\mathtt{B}}, \ell_{\mathtt{E}}, \ell_{\mathtt{F}}, \ell_{\mathcal{E}}$.

By viewing $P^{\vee\pi}$ instead of π as a counterexample, we can simultaneously handle an unbounded number of error paths (similarly to a path program for π), namely, all error paths that extend π by loop unwindings of the blocks B_1 and B_2 for an arbitrary number of iterations. In contrast to a path program for π (whose control-flow graph is almost identical to the one shown in Figure 5 with an exception that the transition τ_d is removed), the DPP $P^{\vee\pi}$ contains several copies of the locations $\ell_{\mathtt{A}}$ and $\ell_{\mathtt{D}}$, as well as additional loops at these locations. They induce a finer partition of the set of reachable states of $P^{\vee\pi}$, which is reflected by a disjunctive path invariant.

Disjunctive Path Invariants. An invariant map $\eta^{\vee\pi}$ for a disjunctive path program $P^{\vee\pi}$ may contain several formulas for a control location $\ell \in L$ of the program P. Let L_ℓ and $\hat{L}_\ell$ be the set of $P^{\vee\pi}$-locations that correspond to a P-location ℓ at different positions in the control-flow graph and their additional copies:

$$L_\ell = \{(\ell, i_1), \dots, (\ell, i_m)\}, \text{ and } \hat{L}_\ell \quad = \{(\hat{\ell}, j_1), \dots, (\hat{\ell}, j_n)\}.$$

We can use the disjunction of formulas in the map $\eta^{\vee\pi}$ over all copies of ℓ, which is

$$\bigvee_{(\ell,i)\in L_\ell} \eta^{\vee\pi}(\ell, i) \ \vee \bigvee_{(\hat{\ell},j)\in \hat{L}_\ell} \eta^{\vee\pi}(\hat{\ell}, j),$$

to refine the abstraction at location ℓ as proposed in Section 4.

	Locations traversed by the path	Time
FORWARD	$\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{C}}, \ell_{\mathtt{E}}, \ell_{\mathtt{B}}, \ell_{\mathtt{F}}, \ell_{\mathcal{E}}$	0.05 s
	$\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{D}}, \ell_{\mathtt{E}}, \ell_{\mathtt{B}}, \ell_{\mathtt{F}}, \ell_{\mathcal{E}}$	0.12 s
INITCHECK	$\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{C}}, \ell_{\mathtt{B}}, \ell_{\mathtt{D}}, \ell_{\mathtt{E}}, \ell_{\mathtt{D}}, \ell_{\mathtt{E}}, \ell_{\mathcal{E}}$	0.27 s
PARTITION	$\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{C}}, \ell_{\mathtt{B}}, \ell_{\mathtt{E}}, \ell_{\mathtt{F}}, \ell_{\mathtt{E}}, \ell_{\mathtt{F}}, \ell_{\mathcal{E}}$	1.2 s
	$\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{D}}, \ell_{\mathtt{B}}, \ell_{\mathtt{E}}, \ell_{\mathtt{G}}, \ell_{\mathtt{H}}, \ell_{\mathtt{G}}, \ell_{\mathtt{H}}, \ell_{\mathcal{E}}$	2.4 s
DISJ	$\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{C}}, \ell_{\mathtt{B}}, \ell_{\mathtt{D}}, \ell_{\mathtt{B}}, \ell_{\mathtt{E}}, \ell_{\mathcal{E}}$	0.43 s
	$\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{C}}, \ell_{\mathtt{B}}, \ell_{\mathtt{D}}, \ell_{\mathtt{B}}, \ell_{\mathtt{E}}, \ell_{\mathtt{F}}, \ell_{\mathcal{E}}$	2.8 s

Table 1. Experiments with computing path invariants. "Time" gives the time taken by the invariant generator (on a 1.7 GHz laptop running SICSTUS Prolog). We computed a disjunctive path invariant for the last error path.

For example, we recall the program DISJ from Figure 4. Given the error path π that traverses the locations $\ell_{\mathtt{A}}, \ell_{\mathtt{B}}, \ell_{\mathtt{C}}, \ell_{\mathtt{B}}, \ell_{\mathtt{D}}, \ell_{\mathtt{B}}, \ell_{\mathtt{E}}, \ell_{\mathtt{F}}, \ell_{\mathcal{E}}$, we obtain the DPP $P^{\vee\pi}$ shown in Figure 7. We compute an invariant map $\eta^{\vee\pi}$ such that

$$\begin{aligned} \eta^{\vee\pi}(\ell_{\mathtt{B}}, 1) &= \quad x = 0 \wedge y = 50 \\ \eta^{\vee\pi}(\ell_{\mathtt{B}}, 3) &= \quad y \leq 50 \wedge 0 \leq 1 \\ \eta^{\vee\pi}(\ell_{\mathtt{B}}, 5) &= \quad x \leq 100 \wedge y \leq x \end{aligned}$$

together with initiation and safety conditions $\eta^{\vee\pi}(\ell_{\mathtt{A}}, 0) = \mathit{true}$ and $\eta^{\vee\pi}(\ell_{\mathcal{E}}, 7) = \mathit{false}$. This map instantiates an invariant template that assigns two conjuncts to each location. Any attempt to instantiate a simpler template with a single conjunct per location failed. (Although, it would suffice to have a single conjunct at the location $(\ell_{\mathtt{B}}, 3)$, namely, $y \leq 50$.) Thus, for the location $\ell_{\mathtt{B}}$ of P we obtain the disjunctive assertion

$$(x = 0 \wedge y = 50) \vee (y \leq 50) \vee (x \leq 100 \wedge y \leq x),$$

which we use to compute a predicate abstraction of the program P that is precise enough to rule out all spurious counterexamples that violate the second assertion.

6. Conclusion

We proposed a 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. However, path programs are usually small fragments of the original program, thus permitting more efficient analyses. We automatically generate invariants for path programs, which deliver property-dependent information used to refine the analysis of the original program. The path invariants eliminate all infeasible error paths that remain within the control-flow structure of the path program, i.e., which result from an arbitrary unwinding of the loops within the path program. Furthermore, by considering unbounded computations of path programs, unlike previous predicate abstraction-based CEGAR methods, we can infer universally quantified predicates. This is necessary for reasoning about unbounded data structures such as arrays.

We have applied our algorithm, as outlined in Section 4, to small examples involving arithmetic and array reasoning, including the examples from Sections 2 and 5. The initial experiments with our prototype implementation are promising; see Table 1. We show the computation times for the synthesis of the most interesting path invariants (i.e., those with loops). We used the obtained predicates in the CEGAR loop, which on the listed examples required only a few refinement steps for each program. We note that none of these examples could be proved by BLAST [26], which analyzes longer and longer counterexample paths without terminating.

Acknowledgments. The first author is supported in part by the grant SFU/PRG, 06-3. The second author is supported in part by the Swiss National Science Foundation. The third author is sponsored in part by the NSF grants CCF-0427202 and CCF-0546170. The fourth author is supported in part by Microsoft Research through the European Fellowship Programme.

References

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

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pp. 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, pp. 378–394. 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*, pp. 196–207. ACM, 2003.

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

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. VMCAI*, LNCS 3855, pp. 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.*, 30:388–402, 2004.

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

[9] M. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. CAV*, LNCS 2725, pp. 420–432. Springer, 2003.

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

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

[12] P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *Proc. VMCAI*, LNCS 3385, pp. 1–24. Springer, 2005.

[13] 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*, pp. 238–252. ACM, 1977.

[14] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proc. PLILP*, LNCS 631, pp. 269–295. Springer, 1992.

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

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

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

[18] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pp. 234–245. ACM, 2002.

[19] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. POPL*, pp. 191–202. ACM, 2002.

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

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

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

[23] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Proc. POPL*, pp. 277–289. ACM, 2007.

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

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

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

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

[28] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-SOFT: Software verification platform. In *Proc. CAV*, LNCS 3576, pp. 301–306. Springer, 2005.

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

[30] D. Kapur. Automatically generating loop invariants using quantifier elimination. Technical Report 05431 (*Deduction and Applications*), IBFI Schloss Dagstuhl, 2006.

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

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

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

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

[35] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

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

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

[38] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Comp.*, 19:31–100, 2006.

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

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

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

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

[43] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. VMCAI*, LNCS 3385, pp. 25–41. Springer, 2005.

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