Program Analysis as Constraint Solving

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Saurabh Srivastava *
University of Maryland, College Park
saurabhs@cs.umd.edu

Ramarathnam Venkatesan
Microsoft Research, Redmond
venkie@microsoft.com

Abstract

A constraint-based approach to invariant generation in programs translates a program into constraints that are solved using off-the-shelf constraint solvers to yield desired program invariants.

In this paper we show how the constraint-based approach can be used to model a wide spectrum of program analyses in an expressive domain containing disjunctions and conjunctions of linear inequalities. In particular, we show how to model the problem of context-sensitive interprocedural program verification. We also present the first constraint-based approach to weakest precondition and strongest postcondition inference. The constraints we generate are boolean combinations of quadratic inequalities over integer variables. We reduce these constraints to SAT formulae using bit-vector modeling and use off-the-shelf SAT solvers to solve them.

Furthermore, we present interesting applications of the above analyses, namely bounds analysis and generation of most-general counter-examples for both safety and termination properties. We also present encouraging preliminary experimental results demonstrating the feasibility of our technique on a variety of challenging examples.

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; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithms, Theory, Verification

Keywords Program Verification, Weakest Precondition, Strongest Postcondition, Most-general Counterexamples, Bounds Analysis, Non-termination Analysis, Constraint Solving

1. Introduction

Discovering inductive program invariants is critical for both proving program correctness and finding bugs. Traditionally, iterative fixed-point computation based techniques like data-flow analyses [25], abstract interpretation [11] or model checking [13] have been used for discovering these invariants. An alternative is to use a *constraint-based invariant generation* [8, 10, 7, 32] approach that translates (the second-order constraints represented by) a program into (first-order quantifier-free) satisfiability constraints that can be solved using off-the-shelf solvers. The last decade has witnessed a revolution in SAT/SMT based methods enabling solving of industrial sized satisfiability instances. This presents a real opportunity to leverage these advances for solving hard program analysis problems.

Constraint-based techniques offer two other advantages over fixed-point computation based techniques. First, they are goal-directed and hence have the potential to be more efficient. Secondly, they do not require the use of widening heuristics that are used by fixed-point based techniques leading to loss of precision that is often hard to control.

In this paper, we present constraint-based techniques for three classical program analysis problems, namely *program verification*, *weakest precondition* generation and *strongest postcondition* generation over the abstraction of linear arithmetic. Using this core framework of analyses we further show interesting applications to bounds analysis and finding most-general counterexamples to safety and termination properties. A distinguishing feature of our preliminary tool is that it can uniformly handle a large variety of challenging examples that otherwise require many different specialized techniques for analysis. The key contributions of this paper lie in the uniform constraint-based approach to core program analyses (Sections 2–5) and their novel applications (Section 6).

The goal of *program verification* is to discover invariants that are strong enough to verify given assertions in a program. Current constraint-based techniques are limited to discovering conjunctive invariants in an intraprocedural setting. We present a constraint-based technique that can generate linear arithmetic invariants with bounded *boolean structure* (Section 2), which also allows us to extend our approach to a *context-sensitive interprocedural setting* (Section 3). A key idea of our approach is a scheme for reducing second-order constraints to SAT constraints and this can be regarded as an independent contribution to solving a special class of second-order formulas. Another key idea concerns an appropriate choice of cut-set, which has until now been overlooked in other constraint-based techniques. Our tool can verify assertions (safety properties) in benchmark programs (used by alternative state-of-the-art techniques) that require disjunctive invariants and sophisticated procedure summaries. We also show how constraint-based invariant generation can be applied to verifying termination properties as well as the harder problem of *bounds analysis* (Section 6.1).

The goal of *strongest postcondition generation* is to infer precise invariants in a given program so as to precisely characterize the set of reachable states of the program. Current constraint-based invariant generation techniques work well only in a program verification setting, when the problem enforces the constraint that the invariant should be strong enough to verify the assertions. However, in absence of assertions in programs, there is no guarantee about the precision of invariants. We describe a constraint-based technique that can be used to discover some form of strongest invariants (Section 5). In the area of fixed-point computation based

* This author performed the work reported here during a summer internship at Microsoft Research.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

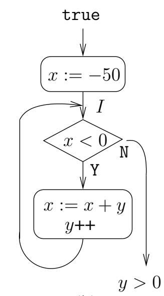
techniques, the problem of generating precise invariants has led to development of several widening heuristics that are tailored to specific classes of programs [40, 18, 16, 17]. Our tool can uniformly discover precise invariants for all such programs.

The goal of *weakest precondition generation* is to infer the weakest precondition that ensures validity of all assertions in a given program. We present a constraint-based technique for discovering some form of weakest preconditions (Section 4). Our tool can generate weakest preconditions of safety as well as termination properties for a wide variety of programs that cannot be analyzed uniformly by any other technique.

We also describe an interesting application of weakest precondition generation, namely generating *most-general counterexamples* for both safety (Section 6.2) and termination (Section 6.3) properties. The appeal of generating most-general counterexamples (as opposed to generating any counterexample) lies in characterizing all counterexamples in a succinct specification that provides better intuition to the programmer. For example, if a program has a bug when $(n \geq 200 + y) \wedge (9 > y > 0)$, then this information is more useful than simply generating any particular counterexample, say $n = 356 \wedge y = 7$ (Figure 10). We have also successfully applied our tool to generate weakest counterexamples for termination of some programs (taken from recent work [22]).

```
PV1 (int y) {
   x := -50;
   while (x < 0) {
      x := x + y;
      y++;
   }
   assert(y > 0)
}
```

(a)



(b)

$$\begin{array}{rcl} \multicolumn{3}{l}{\forall_{x,y}\phi(I):} \\ \texttt{true} & \Rightarrow & I[-50/x] \\ I \wedge x < 0 & \Rightarrow & I[(y{+}1)/y, (x{+}y)/x] \\ I \wedge x \geq 0 & \Rightarrow & y > 0 \end{array}$$

(c)

Figure 1. (a) A program verification example (b) The corresponding control flow graph (c) Constraint generated from the program over the unknown loop invariant I at loop header. Our tool generates a disjunctive solution $(x < 0 \vee y > 0)$ for the invariant I.

2. Program Verification

Given a program with some assertions, the program verification problem is to verify whether or not the assertions are valid. The challenge in program verification is to discover the appropriate invariants at different program points, especially inductive loop invariants that can be used to prove the validity of the given assertions. (The issue of discovering counterexamples, in case the assertions are not valid, is addressed in Section 6.2.)

Program model In this paper, we consider programs that have linear assignments, i.e., assignments of the form $x := e$, or non-deterministic assignments $x :=?$. We also allow for assume and assert statements of the form $\texttt{assume}(p)$ and $\texttt{assert}(p)$, where p is some boolean combination of linear inequalities $e \geq 0$. Here x denotes some program variable that takes integral values, and e denotes some linear arithmetic expression. Since we allow for `assume` statements, without loss of generality, we assume that all conditionals in the program are non-deterministic.

2.1 Background: Conversion of programs to constraints

The problem of program verification can be reduced to the problem of finding solutions to a second-order constraint. The second-order unknowns in this constraint are the unknown program invariants that are inductive and strong enough to prove the desired assertions. In this section we describe the conversion of programs to constraints.

We first illustrate the process of constraint generation for an example program. Consider the program in Figure 1(a) with its control flow graph in Figure 1(b). The program's precondition is `true` and postcondition is $y > 0$. To prove the postcondition, we need to find a loop invariant I at the loop header. There are three paths in the program that constrain I. The first corresponds to the entry case; the path from `true` to I. The second corresponds to the inductive case; the path that starts and ends at I and goes around the loop. The third corresponds to the exit case; the path from I to $y > 0$. Figure 1(c) shows the corresponding formal constraints.

We now show how to generate such constraints in a more general setting of any arbitrary procedure. The first step is to choose a cut-set. A *cut-set* is a set of program locations (called *cut-points*) such that each cycle in the control flow graph passes through some program location in the cut-set. One simple way to choose a cut-set is to include all targets of back-edges in any depth first traversal of the control-flow graph. (In case of structured programs, where all loops are natural loops, this corresponds to choosing the header node of each loop.) However, as we will discuss in Section 2.3, some other choices of cut-set may be more desirable from an efficiency/precision viewpoint. For notational convenience, we assume that the cut-set always includes the program entry location $\pi_{\texttt{entry}}$ and exit location $\pi_{\texttt{exit}}$.

We then associate each cut-point π with a *relation* I_π over program variables that are live at π. The relations $I_{\pi_{\texttt{entry}}}$ and $I_{\pi_{\texttt{exit}}}$ at program's entry and exit locations respectively are set to `true`, while the relations at all other cut-points are unknown relations that we seek to discover. Two cut-points are *adjacent* if there is a path in the control flow graph from one to the other that does not pass through any other cut-point. We establish constraints between the relations at adjacent cut-points π_1 and π_2 as follows. Let $\texttt{Paths}(\pi_1, \pi_2)$ denote the set of paths between π_1 and π_2 that do not pass through any other cut-point. We use the notation $\texttt{VC}(\pi_1, \pi_2)$ to denote the constraint that the relations I_{π_1} and I_{π_2} at adjacent cut-points π_1 and π_2 respectively are consistent with respect to each other:

$$\texttt{VC}(\pi_1, \pi_2) \quad = \quad \forall X \left(\bigwedge_{\tau \in \texttt{Paths}(\pi_1, \pi_2)} (I_{\pi_1} \Rightarrow \omega(\tau, I_{\pi_2})) \right)$$

Above, X denotes the set of program and fresh variables that occur in I_{π_1} and $\omega(\tau, I_{\pi_2})$. The notation $\omega(\tau, I)$ denotes the weakest precondition of path τ (which is a sequence of program instructions) with respect to I and is as defined below:

$$\begin{array}{rclrcl} \omega(\texttt{skip}, I) & = & I & \omega(\texttt{assume}\, p, I) & = & p \Rightarrow I \\ \omega(x := e, I) & = & I[e/x] & \omega(\texttt{assert}\, p, I) & = & p \wedge I \\ \omega(x :=?, I) & = & I[r/x] & \omega(S_1; S_2, I) & = & \omega(S_1, \omega(S_2, I)) \end{array}$$

where r is some fresh variable and the notation $[e/x]$ denotes substitution of x by e and may not be eagerly carried out across unknown relations.

Let π_1, π_2 range over pairs of adjacent cut-points. Then any solution to the unknown relations I_π in the following (verification) constraint (which may also have substitutions), yields a valid proof of correctness.

$$\bigwedge_{\pi_1,\pi_2} \texttt{VC}(\pi_1,\pi_2) \tag{1}$$

Observe that this constraint is universally quantified over the program variables and is a function of $\vec{I}$, the vector of relations I_π at all cut-points (including $I_{\pi_{entry}}, I_{\pi_{exit}}$). We therefore write it as the verification constraint $\forall X.\phi(\vec{I})$. For program verification $I_{\pi_{entry}}$ and $I_{\pi_{exit}}$ are set to `true`. Going back to the example, the second-order constraints corresponding to the program in Figure 1(a) are shown in Figure 1(c) and correspond to the entry, inductive and exit constraints for the loop.

2.2 Constraint solving

In this section we show how to solve the second-order constraint from Eq. 1 that represents the verification condition of unknown relations at cut-points. One way to solve these constraints for discovering the unknown invariants I_π is to use fixed-point based techniques like abstract interpretation. Another (significantly manual) approach is to require the programmer to provide the invariants at the cut-points, which can then be verified using a theorem prover. Instead, we take the approach of reducing the second-order constraint into a boolean formula such that a satisfying assignment to the formula maps to a satisfying assignment for the second-order constraint. Throughout this section, we will illustrate our reduction over the constraints from Figure 1(c).

Our constraint-solving approach involves three main steps. First, we assume some invariant templates (possibly disjunctive) and reduce the second-order constraints to first-order constraints over the unknown parameters of the templates. We then make use of Farkas' lemma [38] to translate the first-order constraints (with universal quantification) into an existentially quantified multi-linear quadratic constraint. These constraints are then translated into a SAT formula using bit-vector modeling (instead of solving them using specialized mathematical solvers [8, 10]). These three steps are detailed below.

Step 1 First, we convert second-order unknowns to first-order unknowns. Instead of searching for a solution to unknown relations (which are second-order entities) from an arbitrary domain, we restrict the search to a template that is some boolean combination of linear inequalities among program variables. For example, an unknown relation can have the template $(\sum_i a_i x_i \geq 0 \wedge \sum_i b_i x_i \geq 0) \vee (\sum_i c_i x_i \geq 0 \wedge \sum_i d_i x_i \geq 0)$, where a_i, b_i, c_i, d_i are all unknown integer constants and x_i are the program variables. The template can either be provided by the user (for example, by specifying the maximum number of conjuncts and disjuncts in DNF representation of any unknown relation), or we can have an iterative scheme in which we progressively increase the size of the template until a solution is found. Given such templates, we replace the unknown relations in the constraint in Eq. 1 by the templates and then apply any pending substitutions to obtain a first-order logic formula with unknowns that range over integers.

For the example in Figure 1(a), a relevant invariant template is $a_1x + a_2y + a_3 \geq 0 \vee a_4x + a_5y + a_6 \geq 0$, where the a_i's are (integer) unknowns to be discovered. If the chosen domain for the template is not expressive enough then the constraints will be unsatisfiable. On the other hand if there is redundancy then redundant templates can always be instantiated with `true` or `false` as required. This step of the reduction translates the verification constraint in Figure 1(c) with second-order unknowns I to first-order unknowns a_i's. For example, the first constraint in Figure 1(c) after Step 1 is $\texttt{true} \Rightarrow (-50a_1 + a_2y + a_3 \geq 0) \vee (-50a_4 + a_5y + a_6 \geq 0)$.

Step 2 Next, we translate first-order universal quantification to first-order existential quantification using Farkas' lemma (at the cost of doing away with some integral reasoning). Farkas' lemma implies that a conjunction of linear inequalities $e_i \geq 0$ (with integral coefficients) is unsatisfiable over rationals iff some non-negative (integral) linear combination of e_i yields a negative quantity, i.e.,

$$\forall X \left(\neg(\bigwedge_i e_i \geq 0)\right) \iff$$
$$\exists \lambda > 0, \lambda_i \geq 0 \left[\forall X \left(\sum_i \lambda_i e_i \equiv -\lambda\right)\right]$$

The reverse direction of the above lemma is easy to see since it is not possible for a non-negative linear combination of non-negative expressions e_i to yield a negative quantity. The forward direction also holds since the only way to reason about linear inequalities over rationals is to add them, multiply them by a non-negative quantity or add a non-negative quantity.

The universal quantification on the right hand side of the above equivalence is over a polynomial equality, and hence can be gotten rid of by equating the coefficients of the program variables X on both sides of the polynomial equality.

We can convert any universally quantified linear arithmetic formula $\forall X(\phi)$ into an existentially quantified formula using Farkas' lemma as follows. We convert ϕ in conjunctive normal form $\bigwedge_i \phi_i$, where each conjunct ϕ_i is a disjunctions of inequalities $\bigvee_j e_i^j \geq 0$. Observe that $\forall X(\phi) = \bigwedge_i \forall X(\phi_i)$ and that ϕ_i can be rewritten as $\neg \bigwedge_j (-e_i^j - 1 \geq 0)$. Hence, Farkas' lemma, as stated above, can be applied to each $\forall X(\phi_i)$.

We illustrate the application of this step over the first constraint from Figure 1(c) that we obtained after Step 1. After Step 1 we have $\texttt{true} \Rightarrow e_1 \geq 0 \vee e_2 \geq 0$ (where $e_1 \equiv -50a_1 + a_2y + a_3 \geq 0$ and $e_2 \equiv -50a_4 + a_5y + a_6 \geq 0$ as obtained earlier). After expanding the implication we get a constraint that is already in CNF form and therefore the corresponding unsatisfiability constraint is $\neg((-e_1 - 1 \geq 0) \wedge (-e_2 - 1 \geq 0))$. Farkas' lemma can now be applied to yield $\exists \lambda_1, \lambda_2 \geq 0, \lambda > 0 (\forall_{x,y} \lambda_1(-e_1 - 1) + \lambda_2(-e_2 - 1) \equiv -\lambda)$. Now we can collect the coefficients for x, y to get a first-order existential constraint. Notice that λ_1 (respectively λ_2) is multiplied with the coefficients inside e_1 (respectively e_2) and therefore this is a multi-linear quadratic constraint over integer variables. Equating the coefficients of y and the constant term we get the constraints: $(50a_1\lambda_1 - a_3\lambda_1 - \lambda_1) + (50a_4\lambda_2 - a_6\lambda_2 - \lambda_2) = -\lambda$ and $a_2\lambda_1 + a_5\lambda_2 = 0$.

Application of Farkas' lemma leads to a loss of completeness since we do away with some integral reasoning. For example, Farkas' lemma cannot help us prove unsatisfiability of $3x \geq 1 \wedge 2x \leq 1$, where x ranges over integers. However, we have not found this loss of completeness to be a hindrance in any of our benchmark examples.

Step 3 Next, we convert the first-order existentially quantified (or quantifier-free) formula obtained from Step 2 to a SAT formula. The formula that we obtain from Step 2 is a conjunction of (multi-linear quadratic polynomials) over integer variables. We convert such a formula into a SAT formula by modeling integer variables as bit-vectors and encoding integer operations like arithmetic, multiplication, and comparison as boolean operations over bit-vectors.

Our approach to constraint solving is sound in the sense that any satisfying solution to the SAT formula yields a valid proof of correctness. However, it is not complete, i.e., there might exist a valid proof of correctness but the SAT formula might not be satisfiable. This is not unexpected since program verification in general is an undecidable problem, and no algorithm can be expected to be both sound and complete. However, our constraint solving approach is complete under two assumptions (i) the unknown invariants are instances of given templates, (ii) checking consistency of invariants at adjacent cut-points does not require integral reasoning. We have found that both these assumptions are easily met for our benchmark examples. The real challenge instead lies in finding the satisfiability assignment for the SAT formula, for which the recent engineering advances in SAT solvers seem to stand up to the task.

2.3 Choice of cut-set

The choice of a cut-set affects the precision and efficiency of our algorithm (or, in fact, of any other constraint-based technique). The choice of a cut-set has been overlooked in constraint-based approaches. [4] recently proposed a technique for performing fixed-point computation on top of constraint-based technique to regain some precision, which we claim was inherently lost in the first place because of a non-optimal choice of cut-set. In this section, we describe a novel strategy for choosing a cut-set that strikes a good balance between precision and efficiency.

From definition of a cut-set, it follows that we need to include some program locations from each loop into the cut-set. One simple strategy is to include all header nodes (or targets of back-edges) as cut-points. Such a choice of cut-set necessitates searching/solving for unknown relations over disjunctive relations when the proof of correctness involves a disjunctive loop invariant. It is interesting to note that for several programs that require disjunctive loop invariants, there is another choice for cut-set that requires searching for unknown relations over only conjunctive domains. Furthermore, even the number of conjuncts required are less compared to those required when the header nodes are chosen to be cut-points. This choice for cut-set corresponds to choosing one cut-point on each path inside the loop. In presence of multiple sequential conditionals inside a loop, this requires expanding the control-flow inside the loop into disjoint paths and choosing a cut-point anywhere on each disjoint path. In fact, this choice for cut-set leads to the greatest precision in the following sense.

THEOREM 1. *Let C be a cut-set that includes a program location on each acyclic path inside a loop (after expansion of control flow inside the loop into disjoint paths). Suppose that the search space for unknown relations is restricted to templates that have a specified boolean structure. If there exists a solution for unknown relations corresponding to any cut-set, then there also exists a solution for unknown relations corresponding to cut-set C.*

The proof of Theorem 1 is given in the full version of this paper [21]. Furthermore, there are several examples that show that the reverse direction in Theorem 1 is not true (i.e., there exists a solution to the unknown relations corresponding to cut-set C, but there is no solution to unknown relations corresponding to some other choice of cut-set). This is illustrated by the example in Figure 2 (discussed below).

Examples Consider the example shown in Figure 2. Let π_i denote the program point that *immediately precedes* the statement at line i in the program. The simplest choice of cut-set corresponds to choosing the loop header (program location π_2). The inductive invariant that is required at the loop header, and is discovered by our tool, is the disjunction $(0 \leq x \leq 51 \wedge x = y) \vee (x \geq 51 \wedge y \geq 0 \wedge x + y = 102)$. If we instead choose the cut-set to be $\{\pi_4, \pi_6\}$ (based on the strategy described in Theorem 1), then the inductive invariant map is conjunctive. This is significant because conjunctive invariants are easier to discover. Our tool discovers the inductive invariant map $\{\pi_4 \mapsto (y \geq 0 \wedge x \leq 50 \wedge x = y), \pi_6 \mapsto (y \geq 0 \wedge x \geq 50 \wedge x + y = 102)\}$ in such a case.

```
    PV2() {
1     x := 0; y := 0;
2     while (true) {
3       if (x ≤ 50)
4         y++;
5       else
6         y--;
7       if (y < 0)
8         break;
9       x++;
10    }
11    assert(x = 102)
    }
```

Figure 2. Another program verification example (taken from [16]) that requires a disjunctive invariant at the loop header. However, a non-standard choice of cutset (as suggested in Theorem 1) leads to conjunctive invariants.

However, the choice of cut-set mentioned in Theorem 1 does not always obviate the need for disjunctive invariants. The example in Figure 1(a) has no conditionals inside the loop, and yet any (linear) inductive invariant required to prove the assertion is disjunctive (e.g., $(x < 0) \vee (y > 0)$, which is what our tool discovers). Heuristic proposals [34, 4] for handling disjunction will fail to discover invariants for such programs.

3. Interprocedural Analysis

The ω computation described in previous section is applicable only in an intraprocedural setting. In this section, we show how to extend our constraint-based method to perform a precise (i.e., context-sensitive) interprocedural analysis.

Precise interprocedural analysis is challenging because the behavior of the procedures needs to be analyzed in a potentially unbounded number of calling contexts. Procedure inlining is one way to do precise interprocedural analysis. However, there are two problems with this approach. First, procedure inlining may not be possible at all in presence of recursive procedures. Second, even if there are no recursive procedures, procedure inlining may result in an exponential blowup of the program.

A more standard way to do precise interprocedural analysis is to compute procedure summaries, which are relations between procedure inputs and outputs. These summaries are usually structured as sets of pre/postcondition pairs (A_i, B_i), where A_i is some relation over procedure inputs and B_i is some relation over procedure inputs and outputs. The pre/postcondition pair (A_i, B_i) denotes that whenever the procedure is invoked in a calling context that satisfies constraint A_i, the procedure ensures that the outputs will satisfy the constraint B_i. However, there is no automatic recipe to efficiently construct or even represent these procedure summaries, and abstraction specific techniques may be required. Data structures and algorithms for representing and computing procedure summaries have been described over the abstraction of linear constants [33], and linear equalities [29]. Recently, some heuristics have been described for the abstraction of linear inequalities [39].

In this section, we show that a constraint-based approach is particularly suited to discovering such useful pre/postcondition (A_i, B_i) pairs. The key idea is to observe that the desired behavior of most procedures can be captured by a small number of such (unknown) pre/postcondition pairs. We then replace the procedure calls by these unknown behaviors and assert that the procedure, in fact, has such behaviors as in assume-guarantee style reasoning [23]. For ease of presentation and without loss of generality, let

```
IP1() {
   x := 5;  y := 3;
   result := Add(x, y);
   assert(result = 8);
}
Add(int i, j) {
   if i ≤ 0
      ret := j;
   else
      b := i − 1;
      c := j + 1;
      ret := Add(b, c);
   return ret;
}
```
(a)

```
IP2() {
   result :=M(19)+M(119);
   assert(result = 200);
}
M(int n) {
   if(n > 100)
     return n − 10;
   else
     return M(M(n + 11));
}
```
(b)

Figure 3. Interprocedural analysis examples. (a) is taken from [39, 30]. (b) is the famous McCarthy 91 function [28, 27, 26], which requires multiple pre/postcondition pairs.

us assume that a procedure does not read/modify any global variables; instead all global variables that are read by the procedure are passed in as inputs, and all global variables that are modified by the procedure are returned as outputs.

Suppose we conjecture that there are q interesting pre/postcondition pairs for procedure $P(\boldsymbol{x})\{S; \texttt{return}\, \boldsymbol{y};\}$ with the vector of formal arguments $\boldsymbol{x}$ and vector of return values $\boldsymbol{y}$. In practice, the value of q can be iteratively increased until invariants are found that make the constraint system satisfiable. Then, we can summarize the behavior of procedure P using q tuples (A_i, B_i) for $1 \leq i \leq q$, where A_i is some relation over procedure inputs $\boldsymbol{x}$, while B_i is some relation over procedure inputs and outputs $\boldsymbol{x}$ and $\boldsymbol{y}$. We assert that this is indeed the case by generating constraints for each i as below and asserting their conjunction:

$$\texttt{assume}(A_i);\ S;\ \texttt{assert}(B_i); \tag{2}$$

We compile away procedure calls $\boldsymbol{v} := P(\boldsymbol{u})$ on any simple path by replacing them with the following code fragment:

$$\boldsymbol{v} :=?; \texttt{assume}\left(\bigwedge_i (A_i[\boldsymbol{u}/\boldsymbol{x}] \Rightarrow B_i[\boldsymbol{u}/\boldsymbol{x}, \boldsymbol{v}/\boldsymbol{y}])\right); \tag{3}$$

Observe that in our approach, there is no need, in theory, to have q different pre/postcondition pairs. In fact, the summary of a procedure can also be represented as some formula $\phi(\boldsymbol{x}, \boldsymbol{y})$ (with arbitrary Boolean structure) that represents relation between procedure inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$. In such a case, we assert that ϕ indeed is the summary of procedure P by generating constraint for $\{S; \texttt{assert}(\phi(\boldsymbol{x}, \boldsymbol{y}));\}$, and we compile away a procedure call $\boldsymbol{v} := P(\boldsymbol{u})$ by replacing it by the code fragment $\boldsymbol{v} :=\ ?; \texttt{assume}(\phi[\boldsymbol{u}/\boldsymbol{x}, \boldsymbol{v}/\boldsymbol{y}])$. However, our approach of maintaining multiple symbolic pre/postcondition pairs (which is also inspired by the data structures used by the traditional fixed-point computation algorithms) is more efficient since it enforces more structure on the assume-guarantee proof and leads to lesser unknown quantities and simpler constraints.

Examples Consider the example shown in Figure 3(a). Our tool verifies the assertion by generating the pre/post pair $(i \geq 0, \texttt{ret} = i+j)$ for procedure Add. This example illustrates that only relevant pairs are computed for each procedure. In addition to serving as the base case of the recursion the true branch of the condition inside Add has the concrete effect formalized by the pre/post pair $(i < 0, \texttt{ret} = j)$. However, this behavior is not needed to prove any assertion in the program and is therefore suppressed.

The procedure $\texttt{M(int}\ n)$ in Figure 3(b) is the widely known McCarthy91 function whose most accurate description is given by the pre/post pairs $(n > 100, \texttt{ret} = n - 10)$ and $(n \leq 100, \texttt{ret} = 91)$. The function has often been used as a benchmark test for automated program verification. The goal directed nature of the verification problem allows our tool to derive $(101 \leq n \leq 119, \texttt{ret} = n - 10)$ and $(n \leq 100, \texttt{ret} = 91)$ as the pairs that prove the program assertion. As such, it discovers only as much as is required for the proof.

4. Weakest Precondition

Given a program with some assertions, the problem of weakest precondition generation is to infer the weakest precondition $I_{\pi_{\texttt{entry}}}$ that ensures that whenever the program is run in a state that satisfies $I_{\pi_{\texttt{entry}}}$, the assertions in the program hold. In Section 6 we show that a solution to this problem can be a useful tool for a wide range of applications.

In this section, we present a constraint-based approach to inferring weakest preconditions under a given template. Since a precise solution to this problem is undecidable, we work with a relaxed notion of weakest precondition. For a given template structure (as defined in step 2.1 in Section 2), we say that A is a weakest precondition if A is a precondition that fits the template and involves constants whose absolute value is at most c (where c is some given constant such that the solutions of interest are those that involve constants whose absolute value is at most c) and there does not exist a weaker precondition than A with similar properties.

The first step in a constraint-based approach to weakest precondition generation is to treat the precondition $I_{\pi_{\texttt{entry}}}$ as an unknown relation in Eq. 1, unlike in program verification where we set $I_{\pi_{\texttt{entry}}}$ to be true. However, this small change merely encodes that any consistent assignment to $I_{\pi_{\texttt{entry}}}$ is a valid precondition, not necessarily the weakest one. In fact, when we run our tool with this small change for any example, it returns false as a solution for $I_{\pi_{\texttt{entry}}}$. Note that false is always a valid precondition, but not necessarily the weakest one.

One simple approach to finding the weakest precondition may be to search for a precondition that is weaker then the current solution (which can be easily enforced by adding another constraint to Eq. 1), and to iterate until none exists. However, this approach can have a very slow progress. When we analyzed Figure 4(a) (discussed below) using this approach, our tool iteratively produced $i \geq j+127, i \geq j+126, \ldots, i \geq j$ under a modeling that used 8-bit two's-complement integers. In general this naïve iterative technique will be infeasible. We need to augment the constraint system to encode the notion of a weakest relation.

We can encode that $I_{\pi_{\texttt{entry}}}$ is a weakest precondition as follows. The verification constraint in Eq. 1 can be regarded as function of two arguments $I_{\pi_{\texttt{entry}}}$ and I_{r}, where I_{r} denote the relations at all cut-points except at the program entry location, and can thus be written as $\forall X.\phi(I_{\pi_{\texttt{entry}}}, I_{\mathrm{r}})$. Now, for any other relation I' that is strictly weaker than $I_{\pi_{\texttt{entry}}}$, it should not be the case that I' is a valid precondition. This can be stated as the following constraint.

$$\begin{aligned} &\forall X.\phi(I_{\pi_{\texttt{entry}}}, I_{\mathrm{r}}) \quad \wedge \\ &\qquad \forall I', I'_{\mathrm{r}} \left(\texttt{weaker}(I', I_{\pi_{\texttt{entry}}}) \Rightarrow \neg \forall X.\phi(I', I'_{\mathrm{r}})\right) \end{aligned}$$

where $\texttt{weaker}(I', I_{\pi_{\texttt{entry}}}) \stackrel{\text{def}}{=} (\forall X.(I_{\pi_{\texttt{entry}}} \Rightarrow I') \wedge \exists X.(I' \wedge \neg I_{\pi_{\texttt{entry}}}))$.

The trick of using Farkas' lemma to get rid of universal quantification (Step 2.2 in Section 2) cannot be applied here because there is existential quantification nested inside universal quantification. In this section we describe some iterative techniques for generating weakest preconditions. We present two different novel approaches in Sections 4.1 and 4.2.

```
WP1(int i, j) {
    x := y := 0;
    while (x ≤ 100) {
        x := x + i;
        y := y + j;
    }
    assert(x ≥ y)
}
```

(a)

```
Merge(int m1, m2, m3) {
    assert(m1 ≥ 0 ∧ m2 ≥ 0)
    k := i := 0;
    while (i < m1) {
        assert(0 ≤ k < m3)
        A[k++] = B[i++];
    }
    i := 0;
    while (i < m2) {
        assert(0 ≤ k < m3)
        A[k++] = C[i++];
    }
}
```

(b)

Figure 4. Weakest Precondition Examples.

```
   WPreFromPre(Input: Precondition I)
1  D_{i,j} := 0;
2  foreach 1 ≤ i,j ≤ n:
3      low := 0; high := MaxN;
4      while (high − low > 1/MaxD)
5          mid := (high + low)/2;
6          D_{i,j} := mid;
7          if ∃ a precondition I' s.t. I(D) ⇒ I'
8              then low := mid;
9              else high := mid;
10     D_{i,j} := low;
11 Output a precondition I' s.t. I(D) ⇒ I'.
```

Figure 5. A binary-search based iterative algorithm for computing a weakest precondition starting from any non-false precondition.

Examples For the procedure in Figure 4(a), our tool generates two different conjunctive preconditions (which individually ensure the validity of the given assertion): (i) $(i \geq j)$, which ensures that when the loop terminates then $x \geq y$, (ii) $(i \leq 0)$, which ensures that the loop never terminates making the assertion unreachable and therefore trivially true.

Figure 4(b) shows an array merge procedure that is called to merge two arrays B, C of sizes m_1, m_2 respectively into a third one A of size m_3. The procedure is correct if no invalid array access are made (stated as the assertions inside the loops) when it is run in an environment where the input arrays B and C are proper (i.e. $m_1, m_2 \geq 0$, which is specified as an assertion at the procedure entry). For the `Merge` procedure in Figure 4(b), our tool generates two different conjunctive preconditions $m_3 \geq m_1 + m_2 \wedge m_1 \geq 0 \wedge m_2 \geq 0$ and $m_1 = 0 \wedge m_2 = 0$.

4.1 Binary search strategy

First, note that without loss of generality we can assume that the weakest precondition to be discovered is a conjunctive invariant. This is because we can obtain the disjunctive weakest precondition as disjunctions of disjoint weakest conjunctive solutions. [1]

THEOREM 2. *Let* $I \equiv \bigwedge_{i=1}^{n} e_i \geq 0$ *be some non-false precondition. For any* $n \times (n+1)$ *matrix* D *of non-negative constants, let* $I(D)$ *denote the formula* $\bigwedge_{i=1}^{n} \left(D_{i,n+1} + \sum_{j=1}^{n} D_{i,j} e_i \geq 0 \right)$. *Let* I' *be some weakest precondition (in our template structure) s.t.* $I \Rightarrow I'$. *Then,*

A1. There exists a non-negative matrix D' *such that* $I' = I(D')$.

A2. For any matrix D'' *that is strictly larger than* D' *(i.e.,* $D''_{i,j} \geq D'_{i,j}$ *for all* i, j *and* $D''_{i,j} > D'_{i,j}$ *for some* i, j*),* $I(D'')$ *is not a precondition (in our template structure).*

A3. For any (non-negative) matrix D''' *that is smaller than* D' *(i.e.,* $D'''_{i,j} \geq D'_{i,j}$ *for all* i, j*),* $I(D''')$ *is a precondition.*

PROOF: A1 follows from Farkas' lemma. A2 follows from the fact that $I(D'')$ is strictly weaker than $I(D')$ and $I(D')$ is a weakest precondition. A3 follows from the fact that $I(D''')$ is stronger than $I(D')$.

□

[1] The significance of generating a weakest conjunctive solution that is disjoint with other weakest conjunctive solutions already generated lies in the fact that the number of weakest conjunctive solutions may potentially be unbounded. However, the number of weakest disjoint conjunctive solutions is finite.

Theorem 2 suggests a binary search based algorithm (described in Figure 5) for finding a weakest precondition. The parameters `MaxN` and `MaxD` denote an upper bound on the values of the numerator and denominator of any rational entry of the matrix D' referred to in Theorem 2(A1). Since the absolute values of all coefficients in I and I' are bounded above by c, `MaxD` and `MaxN` are bounded above by $N^{N/2} \times c^N$, where $N = n^2$. [2]

Observe that the preconditions in line 7 and line 11 can be generated by simply adding the additional constraint $I(D) \Rightarrow I_{\pi_{\text{entry}}}$ to the verification condition for the procedure, and then solving for the resulting constraint using the technique discussed in Section 2. Also note that the matrix D computed at the end is not exactly the matrix D' referred to in Theorem 2(A1) but is close enough in the sense that any precondition weaker than $I(D)$ is a weakest one.

The algorithm in Figure 5 involves making a maximum of $n^2 \times \log(\texttt{MaxN} \times \texttt{MaxD})$ queries to the constraint solver. Hence, it is useful to start with a non-false precondition with the least value of n (where n denotes the number of conjunctions of linear inequalities in the input precondition I). Such a precondition can be found by iteratively increasing the number of conjuncts in the template for the precondition until one is found.

In the next Section, we describe another algorithm for generating a weakest precondition, which we found to be more efficient for our benchmark examples.

4.2 Locally pointwise-weakest strategy

For simplicity of presentation, we assume that each non-trivial maximally strongly connected component in the control flow graph has exactly one cut-point (an assumption that can also be ensured by simple transformations [21]). However, the results in this section can be extended to the general setting without this assumption.

The algorithm for generating a weakest precondition is described in Figure 6. Line 8 initializes I_π to a *pointwise-weakest* relation (defined below) for each cut-point π in reverse topological order of the control dependences between different cut-points. (Note that since we assume that each maximal SCC does not have more than one cut-point, there are no cyclic control dependences

[2] This is because the entries in matrix D are solutions to a system of linear equations each of whose coefficients are bounded in absolute value by c. These linear equations are obtained by equating the coefficients of corresponding variables in the n equivalences represented by $I' = I(D')$. The solution to each unknown in a system of linear equations can be described by ratio of two determinants whose entries are coefficients of the linear equations. Since there can be at most n^2 linearly independent equations among n^2 unknowns, each entry in matrix D can be expressed as ratio of two determinants, each of size at most $N \times N$ where $N = n^2$, and all of whose entries are bounded in absolute value by c.

```
  WPre(Input: Neighborhood structure N)
1 foreach cutpoint π in reverse topological order:
2     I := false;
3     while ∃ a relation I' at π s.t.
4       (a)  ⋀_{π' ∈ Successors(π)} VC(π, π')[I_π ← I']
5       (b)  I ⇒ I' but I' ⇏ I
6       (c)  ⋀_{I'' ∈ N(I')} ( ⋁_{π' ∈ Successors(π)} ¬VC(π, π') ) [I_π ← I'']
7     do {I := I'};
8     I_π := I;
9 Output I_{π_entry};
```

Figure 6. Another iterative algorithm for computing a weakest precondition based on an input neighborhood structure N.

between different cut-points.) We define a relation I at a cut-point to be pointwise-weakest if it is a weakest relation that is consistent with respect to the relations at its neighboring (successor) cut-points. It is easy to see that the pointwise-weakest relation thus generated at the program entry location will be a weakest precondition.

The while loop in Line 3 generates a pointwise-weakest relation at a cut-point π by generating a *locally pointwise-weakest* relation (as defined below) with respect to the input neighborhood structure N in each iteration and repeating the process to obtain a weaker locally pointwise-weakest relation until one exists. (This process is conceptually similar to iterating over local minimas to obtain a global minima.) We say that a relation I is a locally pointwise-weakest with respect to a neighborhood N if it is a weakest relation among its neighbors that is consistent with respect to the relation at its neighboring (successor) cut-points. A locally pointwise-weakest relation can be generated by simply solving the constraints on Lines 5-6 using the technique discussed in Section 2. Observe that the constraints $I' \not\Rightarrow I$ and $\neg\mathtt{VC}(\pi, \pi')$ are already existentially quantified, and hence do not require the application of Farkas' lemma to remove universal quantification. The only difference is that we now obtain quadratic inequalities as opposed to quadratic equalities obtained at the end of Step 2 (on Page 3) of our constraint-solving methodology. However, the bit-vector modeling in Step 3 works equally well for quadratic equalities as well as inequalities. Also note that the neighborhood structure N should be such that it should be possible to enumerate all elements of $\mathtt{N}(I')$ for any invariant template I'.

The performance of our algorithm crucially depends on the choice of the input neighborhood structure, which affects the number of iterations of the loop in Line 3. A denser neighborhood structure may result in lesser number of iterations of the while loop (i.e., a lesser number of queries to the constraint solver), but a larger sized query as a result of the condition in Line 7. We describe below a neighborhood structure that we found to be quite efficient for our purposes; in fact, it required upto 3 iterations for most of our benchmark examples. However, (unlike the binary search strategy described in previous section), we have not been able to prove a formal bound on the worst-case number of queries to the constraint solver that our choice of neighborhood structure can yield because of repeated iterations of the while loop.

4.2.1 Neighborhood Structure

In this section, we describe the neighborhood structure N that we used in our experiments. The set of relations that are in the neighborhood N of a conjunctive relation (in which, without loss of generality, we assume that all inequalities are independent of each

```
Swap(int x) {
  while (*)
    if (x = 1)
      x := 2;
    else if (x = 2)
      x := 1;
  assert(x ≤ 8);
}
```
(a)

```
SP2() {
  d := t := s := 0;
  while(1)
    if (*)
      t++; s := 0;
    else if (*)
      if (s < 5)
        d++; s++;
}
```
(b)

Figure 7. (a) Weakest precondition example that has two locally pointwise-weakest relations at program entry. (b) Strongest Postcondition example taken from [16, 17].

other) are as described below.

$$\begin{aligned} \mathtt{N}(\bigwedge_{i=1}^{n} e_i \geq 0) \quad = \quad & \{e_j + 1 \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid 1 \leq j \leq n\} \ \cup \\ & \{e_j + e_\ell \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid j \neq \ell \wedge 1 \leq j, \ell \leq n\} \end{aligned} \tag{4}$$

Geometric Interpretation: The neighborhood structure described above has a nice geometric interpretation. The neighbors of a convex region $\bigwedge_i e_i \geq 0$ are obtained by slightly moving any of the hyper-planes $e_j \geq 0$ parallel to itself, or by slightly rotating any of the hyper-planes $e_j \geq 0$ along its intersection with any other hyper-plane $e_\ell \geq 0$.

We extend the neighborhood structure defined above to relations in DNF form (in which, without loss of generality, we assume that all disjuncts are disjoint from each other) as:

$$\mathtt{N}(\bigvee_{i=1}^{m} I_i) \quad = \quad \{I'_j \vee \bigvee_{i \neq j} I_i \mid 1 \leq j \leq m; I'_j \in \mathtt{N}(I_j)\}$$

Notice how the above choice of the neighborhood structure helps avoid the repeated iteration over the preconditions $i \geq j + 127, i \geq j + 126, \ldots, i \geq j$ (as alluded in Section 4 on Page 4) to obtain the weakest precondition $i \geq j$ for the example in Figure 4(a). None of these preconditions except $i \geq j$ is locally pointwise-weakest with respect to the above neighborhood structure. Hence, the use of the above neighborhood structure requires only one iteration of the while loop in the algorithm in Figure 6 for obtaining weakest precondition for the example in Figure 4(a).

However, a locally pointwise-weakest relation with respect to the neighborhood structure defined above may not be a pointwise-weakest relation. For example, consider the program in Figure 7(a). The relations $x \leq 0$ and $x \leq 8$ are both locally pointwise-weakest relations (with respect to the above neighborhood structure) at program entry. However, only the relation $x \leq 8$ is a pointwise-weakest relation at program entry (and hence a weakest precondition). Hence, use of the above neighborhood structure requires two iterations of the while loop in the algorithm in Figure 6 for obtaining weakest precondition for the example in Figure 7(a).

5. Strongest Postcondition

The problem of strongest postcondition is to generate the most precise invariants at a given cut-point. Just as in the weakest precondition case, we work with a relaxed notion of strongest postcondition, wherein we are interested in finding a strongest postcondition, whose proof of correctness is expressible in the given template structure.

Our technique for generating strongest postcondition is very similar to the weakest precondition inference technique described in the previous section. We use the algorithm described in Figure 8

```
SPost(Input: Neighborhood structure N)
foreach cutpoint π in topological order:
    I := true;
    while exists a relation I' at π s.t.
      (a)  ⋀_{π' ∈ Predecessors(π)} VC(π', π)[I_π ← I']
      (b)  I' ⇒ I but I ⇏ I'
      (c)  ⋀_{I'' ∈ N(I')} ( ⋁_{π' ∈ Predecessors(π)} ¬VC(π', π) ) [I_π ← I'']
    do {I := I'};
    I_π := I;
Output I_{π_exit};
```

Figure 8. An iterative algorithm for computing a strongest post-condition based on an input neighborhood structure N.

(in place of the algorithm described in Figure 6) with the following neighborhood structure (in place of the one mentioned in Eq. 4).

$$\mathtt{N}(\bigwedge_{i=1}^{n} e_i \geq 0) \quad = \quad \{e_j - 1 \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid 1 \leq j \leq n\} \ \cup$$
$$\{e_j - e_\ell \geq 0 \wedge \bigwedge_{i \neq j} e_i \geq 0 \mid j \neq \ell \wedge 1 \leq j, \ell \leq n\}$$

Examples For a more general version of the procedure in Figure 2, wherein we replace the constant 50 by a symbolic constant m that is asserted to be non-negative, our tool generates the post-condition $x = 2m + 2$.

For the procedure in Figure 7(b), our tool generates the strongest postcondition $s + d + t \geq 0 \wedge d \leq s + 5t$.

6. Applications

In earlier sections, we have described constraint-based techniques for verification of safety properties. In this section, we show how to apply those techniques for finding counterexamples to safety properties, verification of termination (which is a liveness property), and finding counterexamples to termination.

6.1 Termination and Bounds Analysis

The termination problem involves checking whether the given procedure terminates under all inputs. In this section, we show how to use the constraint-based approach to solve a harder problem, namely bounds analysis. The problem of bounds analysis is to find a worst-case bound on the running time of a procedure (say in terms of the number of instructions executed) expressed in terms of its inputs.

We build on the techniques that reduce the bounds analysis problem to discovering invariants of a specific kind [20]. The key idea is to compute bounds on loop iterations and number of recursive procedure call invocations. Each of these can be bounded by appropriately instrumenting counter variables and estimating bounds on counter variables. In particular, the number of loop iterations of a while loop "while c do S" can be bounded by computing upper bound on the instrumented variable i inside the loop in the following code fragment: "$i := 0$; while c do { i++; S; }". The number of recursive procedure call invocations of a procedure "$P(x)$ { S }" can be bounded similarly by computing an upper bound on global variable i inside procedure P in the following code fragment: "$P(x)$ { $i := 0$; $P'(x)$; }; $P'(x')$ { i++; $S[x'/x]$; }".

CLAIM 1. *Let P be a given program. Let P' be the transformed program obtained after instrumenting counters that keep track of loop iterations and recursive call invocations and introducing partial assertions at appropriate locations that assert that the counters are bounded above by some function of the inputs. The program P terminates iff the assert statements in P' are satisfied.*

```
Loop(int n,m,x0,y0) {
   assert(x0 < y0);
   x := x0; y := y0;
   while (x < y)
      x := x + n;
      y := y + m;
}
```

Original Program

```
Loop(int n,m,x0,y0) {
   assert(x0 < y0);
   x := x0; y := y0; i := 0;
   while (x < y)
      i++;
      assert(i ≤ f(n,m,x0,y0));
      x := x + n;
      y := y + m;
}
```

Instrumented Program

Figure 9. Discovering weakest preconditions for termination.

Invariant generation tools such as abstract interpretation can be used to compute bounds on the counter variables (as proposed in [20]). We show instead that a constraint-based approach is particularly suited for discovering these invariants since they have a specified form and involve linear arithmetic. We introduce assert statements with templates $i < a_0 + \sum_i a_i x_i$ (at the counter increment i++ site in case of loops and at the end of the procedure in case of recursive procedures) for bounding the counter variable.

Besides the counter instrumentation strategy mentioned above, [20] also describes some other counter instrumentation strategies that can be used to compute non-linear bounds as a composition of linear bounds on multiple instrumentation counters. Such techniques can also be used in our framework to compute non-linear bounds using linear templates.

Additionally, the constraint-based approach solves an even harder problem, namely inferring preconditions under which the procedure terminates and inferring a bound under that precondition. For this purpose, we simply run the tool in weakest precondition inference mode. This is particularly significant when procedures are expected to be called under certain preconditions and would not otherwise terminate under all inputs. We are not aware of any technique that can compute such conditional bounds.

Example For the example in Figure 9, our tool computes the weakest precondition $n \geq m + 1 \wedge x_0 \leq y_0 - 1$ and the bound $y_0 - x_0$. The latter requires discovering the inductive loop invariant $i \leq (x - x_0) - (y - y_0)$.

6.2 Counterexamples for Safety Properties

Since program analysis is an undecidable problem, we cannot have tools that can prove correctness of all correct programs or find bugs in all incorrect programs. Hence, to maximize the practical success rate of verification tools, it is desirable to search for both proofs of correctness as well as counterexamples in parallel. Earlier, we showed how to find proofs of correctness of safety and termination properties. In this section, we show how to find *most-general* counterexamples to safety properties.

The problem of generating a most-general counterexample for a given set of (safety) assertions involves finding the most general characterization of inputs that leads to the violation of some reachable safety assertion. We show how to find such a characterization using the techniques discussed in Section 4 and Section 6.1.

The basic idea is to reduce the problem to that of finding the weakest precondition for an assertion. This reduction involves constructing another program from the given program P using the following transformations:

B1 Instrumentation of program with an error variable: We introduce a new error variable that is set to 0 at the beginning of the program. Whenever violation of any assertion occurs (i.e., the negation of that assertion holds), we set the error variable to 1

```
  Bug1(int y, n) {
1    x := 0;
2    if (y < 9)
3      while (x < n)
4        assert(x < 200);
5        x := x + y;
6    else
7      while (x ≥ 0)
8        x++;
  }
```

(a) Original Program

```
   Bug1(int y, n) {
1    x := err := i1 := i2 := 0;
2    if (y < 9)
3      while (x < n)
4        i1++;
5        assert(i1 ≤ f1(n,y));
6        if (x ≥ 200)
7          err := 1; goto L;
8        x := x + y;
9    else
10     while (x ≥ 0)
11       i2++;
12       assert(i2 ≤ f2(n,y));
13       x++;
14 L: assert(err = 1);
   }
```

(b) Instrumented Program

Figure 10. A most-general counterexample that leads to violation of the safety assertion in the original program is $(n \geq 200 + y) \wedge (0 < y < 9)$. Our tool discovers this by instrumenting the program appropriately and then running our weakest precondition algorithm.

and jump to the end of the program, where we assert that the error variable is equal to 1. We remove the original assertions from the program.

B2 Instrumentation to ensure termination of all loops: For this we use the strategy described in Section 6.1, wherein we instrument the program with counter variables and introduce assertion templates that assert that the counter variable is upper bounded by some function of loop inputs or procedure inputs.

CLAIM 2. *Let P be a program with some safety assertions. Let P' be the program obtained from program P by using the transformation described above. Then, P has an assertion violation iff the assertions in program P' hold.*

Claim 2 holds and its significance lies in the fact that now we can use weakest precondition inference (Section 4) on the transformed program to discover most-general characterization of inputs under which there is a safety violation in the original program.

Example The program shown in Figure 10(a) is instrumented using transformations B1 and B2 and the resulting program is shown in Figure 10(b). Our tool discovers the precondition $(n \geq 200 + y) \wedge (9 > y > 0)$. The loop invariant (at line 3) that establishes all assertions in the instrumented program is $(n \geq 200 + y) \wedge (i \leq x) \wedge (9 > y > 0) \wedge (x < n)$. Note that this invariant implies the instantiation n for the loop bound function $f_1(n, y)$. On the other hand the precondition $y < 9$ implies that the loop on line 10 is unreachable, and hence any arbitrary f_2 suffices.

Observe the importance of transformation B1. An alternative to transformation B1 that one might consider is to simply negate the original safety assertion instead of introducing an error variable. This is incorrect for two reasons: (a) It is too stringent a criterion because it insists that in each iteration of the loop the original assertion does not hold, (b) It does not ensure reachability and allows for those preconditions under which the assert statement is never executed. In fact, when we run our tool with such an alternative transformation on the example in Figure 10(a), we obtain $n \leq 0$ as the weakest precondition.

Also, observe the importance of transformation B2. If we do not perform transformation B2 on the example in Figure 10(a), then the tool comes up with the following weakest precondition: $y \leq 0$. Note that under this precondition, the assertion at the end of the program always holds since that location is unreachable.

```
NT1(int x, y) {
  while (x ≥ 0)
    x := x + y;
    y++;
}
```

(a)

```
NT2(int i) {
  even := 0;
  while (i ≥ 0)
    if (even = 0)
      i--;
    else
      i++;
    even := 1 − even;
}
```

(b)

Figure 11. Non-termination examples taken from [22].

Note that the transformation B2 does not ensure termination of all loops in the original program. The transformation B2 ensures termination of only those loops that are reachable under the to-be-discovered weakest precondition and that too in the program obtained after transformation B1, which introduces extra control-flow that causes loops to terminate as soon as the violation of any safety property occurs. For example, the loop on line 10 in Figure 10(b) is unreachable under the discovered preconditions and therefore any arbitrary function f_2 suffices.

6.3 Counterexamples for Termination

The problem of generating a most-general counterexample for program termination involves finding the most-general characterization of inputs that leads to non-termination of the program. Without loss of generality we assume that the program has at most one exit point.

CLAIM 3. *Let P be a given program with a single exit point. Let P' be the program obtained from P by adding the assert statement "`assert(false)`" at the end of the program. Then, P is non-terminating iff the assert statement in P' is satisfied.*

The significance of Claim 3 lies in the fact that now we can use weakest precondition inference (Section 4) on the transformed program to discover most-general characterization of inputs under which the original program is non-terminating.

Examples Consider the example shown in Figure 11(a). If we instrument `assert(false)` at the end of the program, then our tool generates the precondition $x \geq 0 \wedge y \geq 0$, which is one of the weakest affine conditions under which the program is non-terminating.

Now consider the program shown in Figure 11(b). If we instrument `assert(false)` at the end of this program, then our tool generates the precondition $i \geq 1$. Notice that the loop guard $i \geq 0$ is not sufficient to guarantee non-termination.

7. Experiments

In previous sections, we have shown how to reduce various program analysis problems to the problem of solving SAT constraints. We now present encouraging experimental results illustrating that SAT solvers can in fact solve the constraints generated from our chosen set of examples in a reasonable amount of time. Our examples are drawn from benchmarks used by state-of-the-art alternative techniques.

Our reduction technique is parameterized by several parameters (such as the cut-set, the number of bits used in bit-vector modeling, and the size of templates in terms of the number of conjuncts and disjuncts) whose choice presents a completeness/efficiency trade-off. An increase in size of these parameters increases the chance that the required invariant/pre-condition would fit the template, but at the cost of generating a bigger SAT formula.

Name	Time (secs)	Num. Clauses
cegar1 [19]	0.08	5 K
cegar2 [19]	0.80	50 K
barbr [18]	0.41	76 K
berkeley [18]	3.00	441 K
bk-nat [18]	5.30	174 K
seesaw [18]	3.23	70 K
hsortprime [18]	0.51	54 K
lifnatprime [18]	1.27	51 K
swim [18]	1.63	45 K
cars [18]	2.93	86 K
ex1 [18]	0.10	10 K
ex2 [18]	0.75	92 K
fig1a [18]	0.14	20 K
fig2 [18]	0.56	239 K
fig3 [18]	16.60	547 K
w1 [5], pg12	0.14	25 K
w2 [5], pg12	1.80	165 K

(a)

Name	Time (secs)	Num. Clauses
Fig 3(a), [39]	0.57	63 K
a1 [31], pg9	9.90	174 K
a2 [29], pg2	0.50	75 K
mergesort	0.19	43 K
quicksort	0.45	133 K
fibonacci	11.00	90 K
Fig 3(b)	72.00	558 K

(b)

Name	Time (secs)	Num. Clauses
[22], pg3	0.80	42 K
Fig 11(b) [22]	0.40	57 K
Fig 11(a) [22]	0.60	43 K
Fig 4(a)	14.40	119 K
Fig 4(b)	80.00	221 K
Fig 7(a)	0.50	50 K
Fig 9	11.60	118 K
Fig 10	68.00	135 K

(c)

Name	Time (secs)	Num. Clauses
Fig 2 [16, 17]	1.40	107 K
Fig 7(b)	16.10	273 K
w1 [5], pg12	0.60	60 K
speed [17], pg10	18.20	41 K
merge [16], pg 11	3.90	128 K
burner [15], pg14	1.50	91 K

(d)

Table 1. (a) Program verification examples (b) Interprocedural analysis examples (c) Weakest precondition inference (including non-termination and bug-finding) examples. (d) Strongest postcondition inference examples

In our experiments, we used the cut-set suggested by Theorem 1. For discovering the remaining parameters, we used an incremental strategy. We progressively increased the number of bits required for bit-vector modeling by 2 bits (starting from 3 bits for unknown coefficients and 6 bits for unknown constants and 1 bit for the multipliers λ's used in Farkas' lemma). The number of disjuncts and conjuncts were progressively increased by 1 (starting with 1 disjunct and 2 conjuncts). The increment was performed until the SAT solver stopped saying UNSAT. For most of our benchmark examples, our choice of parameters required upto 2 iteration steps. The specific choice of these parameters was motivated by the observation that for most of our examples, the required invariants involved only one disjunct and one bit for the multiplier λ's. We also observed that working with a smaller number of disjuncts and a smaller number of bits for the multiplier λ's is important for efficiency reasons because the size of the generated SAT formula usually blows up with these two particular parameters.

Table 1 describes the results of our analysis on our benchmark examples. It shows the number of clauses in the generated CNF formula (for the choice of parameters for which the SAT solver was able to find a satisfying assignment) and the time taken by the SAT solver (Z3 [12]) to find a satisfying assignment under the discovered parameters.

Table 1(a) shows the time taken by our tool on several program verification examples. Most of these examples are taken from benchmarks used by some state-of-the-art abstraction refinement based techniques [19, 18], which also provide exhaustive comparison against similar techniques. The last two examples `w1` and `w2` are taken from [5]. `w1` is a simple loop iteration but with $x \leq n$ replaced with $x \neq n$ while `w2` is a loop with the guard moved inside a non-deterministic conditional. Standard narrowing is unable to capture the precision lost due to widening in these instances. Our solution times compare favorably against previous techniques.

Table 1(b) shows the time taken by our tool for generating required invariants for establishing validity of assertions in an interprocedural setting for different examples. The first three examples are taken from alternate proposals [29, 31, 39] for discovering linear invariants in an interprocedural setting. We also analyze some recursive procedures (Mergesort, Quicksort and Fibonacci) for discovering invariants that establish upper bounds on the number of recursive procedure call invocations after the respective procedures have been instrumented with the counter instrumentation strategy described in Section 6.1. The invariants for all of these examples required producing one pre/post pair for each procedure. Proving correctness of the McCarthy91 function in Figure 3(b), however, required computing two pre/post pairs.

Table 1(c) shows the time taken by our tool for generating weakest preconditions for respective examples. Our tool implements the methodology described in Section 4.2 for generation weakest precondition. The first three examples in Table 1(c) are taken from [22], and we infer the weakest preconditions that ensure non-termination of these examples. Our most challenging example (Figure 10) takes 68 seconds.

Table 1(d) shows the time taken by our tool for generating strongest postconditions for respective examples taken from benchmarks used by some sophisticated widening techniques [5, 15, 16, 17]. For each of these examples, we compute the strongest linear invariants that hold at the end of the respective procedures. The examples `speed`, `merge`, and `burner` model hybrid automaton for some real systems.

8. Related Work

Constraint solving based techniques Theoretical expositions of program analysis techniques frequently formulate them as constraints (constraint-based CFA, type inference, reachable states in abstract interpretation [11], model checking among others) and typically solve them using fixed-point computation. We instead concentrate on techniques that reduce the analysis problem to constraints that can be solved using SAT/SMT solvers. Constraint-based techniques have been successfully applied for discovering *conjunctive* linear arithmetic invariants [8, 36, 35, 37] in an intraprocedural setting. In contrast, our approach discovers linear arithmetic invariants with arbitrary (but pre-specified) *boolean structure* in a context-sensitive *interprocedural* manner.

Constraint-based techniques have also been applied for discovering non-linear polynomial invariants [24] and invariants in the combined theory of linear arithmetic and uninterpreted functions [3], but again in a conjunctive and intraprocedural setting. It is possible to combine these techniques with our formulation to lift them to disjunctive and context-sensitive interprocedural settings.

Constraint-based techniques, being goal-directed, work naturally in program verification mode where the task is to discover inductive loop invariants for the verification of assertions. Otherwise, there is no guarantee on the precision of invariants generated. [6] describes a simple iterative strategy of rerunning the solver with the additional constraint that the new solution should be stronger than the previous solution. Such a strategy can have a very slow progress. Our approach for strongest postcondition provides a more efficient solution. Additionally, we present a methodology for generating weakest preconditions.

Constraint solvers have been used for finding bugs in loop-free programs [41] (obtained by unrolling loops in programs heuristically). In contrast, our methodology can be used to find a most-general counterexample and also find bugs in programs that require an unbounded or a large number of loop iterations for the bug to manifest.

Abstract interpretation based techniques for discovering linear arithmetic invariants There is a large body of work on sophisticated widening techniques [16, 17], abstraction refinement [40, 18] and specialized extensions (using acceleration [15], trace partitioning and loop unrolling [5]) for discovering conjunctive linear inequality invariants in an intraprocedural setting. Powerset extensions [14, 19] of linear inequalities domain and derived techniques utilizing control flow structure [34, 4] have been proposed for disjunctive invariants. All these are specialized to work for specific classes of programs. In contrast, our approach can uniformly discover precise invariants in all such classes of programs in strongest postcondition setting, while also offering the added advantage of being goal-directed in verification setting.

There has been work on interprocedurally discovering linear equality relationships [33, 29]; however the problem of discovering linear inequalities in an interprocedural setting has not been effectively addressed. Recently, some heuristics have been proposed [39] to discover linear inequality relationships in an interprocedural setting based on extension of an earlier work on transition matrices and postponing conditional evaluation. The precision of these techniques is unclear in the presence of conditionals. Since our approach facilitates disjunctive reasoning, our approach discovers linear inequalities interprocedurally as naturally (and as precisely) as it does so in an intraprocedural setting.

Proofs and counterexamples to termination There is a large body of work on proving termination properties by synthesizing ranking functions [9, 32, 7, 1, 2, 10] using a variety of techniques including those based on constraint solving. We show how to use constraint solving to solve the harder problem of timing analysis. Moreover, we also show how to do conditional termination analysis, wherein we infer preconditions for termination.

Recently, finding counterexamples to termination was addressed in [22]. Their technique finds counterexamples to termination properties by means of identifying *lassos* (linear program paths that end in a non-terminating cycle) using a constraint-based approach to find recurring sets of states. In contrast, our scheme for proving non-termination is based on inferring weakest preconditions, thereby inferring a most-general counterexample to termination.

9. Conclusion and Future Work

This paper describes how to model a wide spectrum of program analysis problems as constraints that can be solved using off-the-shelf constraint solvers. We show how to model the problem of discovering invariants that involve (conjunctions and disjunctions of) linear inequalities (both intraprocedurally and interprocedurally), and apply it to the problem of checking safety properties and timing analysis of programs. We also show how to model the problem of discovering weakest preconditions (and strongest postconditions) and apply it to inferring most-general counterexamples for both safety and termination properties. The constraints that we generate are boolean combinations of quadratic inequalities over integer variables, which we reduce to SAT formulas using bit-vector modeling. We show experimentally that the SAT solver can efficiently solve such constraints generated from hard benchmarks.

The work described here can be extended in two directions. The first one is to extend these techniques to discover a richer class of invariants involving arrays, pointers, and even quantifiers. Secondly, one can also consider new constraint solving techniques, in particular QBF (Quantified Boolean Formula) solvers. This would alleviate the need for applying Farkas' lemma to compile away universal quantification, leading to smaller sized SAT formulas, but those that are universally quantified.

References

[1] I. Balaban, A. Cohen, and A. Pnueli. Ranking abstraction of recursive programs. In *VMCAI*, pages 267–281, 2006.

[2] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance analyses from invariance analyses. In *POPL*, pages 211–224, 2007.

[3] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI'07*, pages 378–394, 2007.

[4] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation.*, LNCS 2566, pages 85–108. Oct. 2002.

[6] A. R. Bradley and Z. Manna. Verification constraint problems with strengthening. In *ICTAC*, pages 35–49, 2006.

[7] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *Proc. 17^{th} Intl. Conference on Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*. Springer Verlag, July 2005.

[8] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

[9] M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 442–454. Springer-Verlag, 2002.

[10] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, pages 1–24, 2005.

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

[12] L. M. de Moura and N. Bjørner. Efficient e-matching for smt solvers. In *CADE*, pages 183–198, 2007.

[13] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[14] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. of Comp. Prg.*, 32(1-3):177–210, 1998.

[15] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *13th International Static Analysis Symposium, SAS'06*, LNCS 4134, Aug. 2006.

[16] D. Gopan and T. W. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.

[17] D. Gopan and T. W. Reps. Guided static analysis. In *SAS*, pages 349–365, 2007.

[18] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. Technical Report TR-07-23, IIT Bombay, 2007.

[19] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, pages 474–488, 2006.

[20] S. Gulwani, K. Mehra, and T. Chilimbi. Statically computing complexity bounds for programs with recursive data-structures. Technical Report MSR-TR-2008-16, Microsoft Research, Jan. 2008.

[21] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. Full version. Technical Report MSR-TR-2008-44, Microsoft Research, Mar. 2008.

[22] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*, 2008.

[23] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[24] D. Kapur. Automatically generating loop invariants using quantifier elimination. In *Deduction and Applications*, 2005.

[25] G. A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[26] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, New York, '74.

[27] Z. Manna and J. McCarthy. Properties of programs and partial function logic. *Machine Intelligence*, 5, 1970.

[28] Z. Manna and A. Pnueli. Formalization of properties of functional programs. *Journal of the ACM*, 17(3):555–569, 1970.

[29] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341, 2004.

[30] M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural analysis (almost) for free. In *Technical Report 790, Fachbereich Informatik, Universitt Dortmund*, 2004.

[31] M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural herbrand equalities. In *ESOP*, pages 31–45, 2005.

[32] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[33] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

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

[35] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.

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

[37] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41, 2005.

[38] A. Schrijver. *Theory of Linear and Integer Programming*. 1986.

[39] H. Seidl, A. Flexeder, and M. Petter. Interprocedurally analysing linear inequality relations. In *ESOP*, pages 284–299, 2007.

[40] C. Wang, Z. Yang, A. Gupta, and F. Ivancic. Using counterex. for improv. the prec. of reachability comput. with polyhedra. In *CAV*, pages 352–365, 2007.

[41] Y. Xie and A. Aiken. Saturn: A sat-based tool for bug detection. In *CAV*, pages 139–143, 2005.