Constraint-based Invariant Inference over Predicate Abstraction

Sumit Gulwani[1], Saurabh Srivastava[2,⋆], and Ramarathnam Venkatesan[3]

[1] Microsoft Research, Redmond, sumitg@microsoft.com
[2] University of Maryland, College Park, saurabhs@cs.umd.edu
[3] Microsoft Research, Redmond, venkie@microsoft.com

Abstract. This paper describes a constraint-based invariant generation technique for proving the validity of safety assertions over the domain of predicate abstraction in an interprocedural setting. The key idea of the technique is to represent each invariant in bounded DNF form by means of boolean indicator variables, one for each predicate p and each disjunct d denoting whether p is present in d or not. The verification condition of the program is then encoded by means of a boolean formula over these boolean indicator variables such that any satisfying assignment to the formula yields the inductive invariants for proving the validity of given program assertions.

This paper also describes how to use the constraint-based methodology for generating weakest preconditions for safety assertions. An interesting application of weakest precondition generation is to produce most-general counterexamples for safety assertions. We also present preliminary experimental evidence demonstrating the feasibility of this technique.

1 Introduction

Predicate abstraction [1] is a commonly used technique for proving program properties. This involves over-approximating the set of reachable states of the program using formulas with boolean structure over a given set of predicates. This over-approximation is usually computed using fixed-point based techniques like abstract interpretation or model checking. One of the main advantages of the predicate abstraction domain is that it can represent disjunctions as opposed to other abstract domains like polyhedron domain. However, this expressiveness comes with disadvantages: First, the abstract state can have size exponential in the number of predicates. Second, the abstract domain has exponential height. The naive fixed-point computation process seems expensive especially when the final inductive invariants required to prove a given property are typically simple and small in size compared to the potential worst-case exponential representation.

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

In this paper, we describe a technique for discovering inductive invariants over predicate abstraction that exploits the observation that the inductive invariants required for proving a given assertion typically require a small representation, instead of the worst-case exponential representation. In particular, we describe the inductive invariants using a bounded boolean structure over a given set of predicates, say DNF formulas with at most k disjuncts, where k is some small constant.[4] To achieve completeness, we can iteratively increase the value of k.

Our technique is based on the following observation: Any DNF formula with k disjuncts over a set of n predicates can be described by a truth-value assignment to $k \times n$ boolean (indicator) variables, one for each predicate p and each disjunct d denoting whether predicate p is in disjunct d. The key idea of our technique is to establish boolean constraints between the boolean indicator variables corresponding to the invariants at neighboring program locations by using the predicate cover operation[5] (The predicate cover of a formula F is the weakest formula over a set of predicates that implies F). The boolean constraint thus obtained encodes the verification condition of the program. A satisfying assignment to this boolean formula yields the inductive invariants sufficient to establish the validity of given assertions. Unsatisfiability of the boolean formula denotes that there are no inductive invariants over our choice of template structure (DNF formula with k disjuncts over the given set of predicates) to validate the assertions in the program. The size of the generated boolean formula is linear in the size of the program and the size of the predicate cover, and polynomial in the number of the predicates. Finding a satisfying assignment to the boolean formula can take exponential time in the worst-case and in theory we have still not gotten rid of the exponential factor. However, this methodology allows a direct way to leverage the engineering advances of the SAT solvers. It is noteworthy that the last decade has witnessed a revolution in SAT solvers enabling solving of industrial sized satisfiability instances.

Our constraint-based technique offers three main advantages over the fixed-point computation based method. First, it is goal-directed and hence has the potential to be more efficient. Secondly, it does away with the iterative process of computing fixed-points, which is expensive, especially when performed on abstractions with exponential-height lattices, like predicate abstraction. Thirdly, it cleanly splits the reasoning required of SMT formulas generated during predicate abstraction into two parts: Theory-based reasoning using predicate cover operation over small and mostly conjunctive formulas (this encodes the abstract

[4] It may appear that this observation can also be used to obtain a PTIME abstract interpretation [2] based algorithm for discovering inductive invariants with k disjuncts. However, this is not true. The domain of k-DNF formulas does not form a lattice as there is no unique LUB. The domain of formulas whose CNF representation contains at most k disjuncts in each conjunct does form a lattice of polynomial height. However, in that case, each abstract interpretation operation requires reasoning about an SMT formula in CNF form, which is NP-hard.

[5] A fundamental operation used in abstract transformers while performing abstract interpretation over predicate abstraction [1].

program semantics) and SAT-based reasoning over a polynomially-sized boolean formula (this encodes the fixed-point).[6]

Our technique complements abstraction refinement techniques (such as counterexample guided abstraction refinement [3] and interpolation based methods [4]) by equipping them with a more robust invariant generation procedure. Abstraction refinement techniques alleviate the cost involved in predicate abstraction by iteratively refining the abstraction until an inductive invariant can be expressed. Our technique alleviates the cost of reasoning over a given abstraction by off-loading the cost of boolean reasoning and fixed-point computation to a SAT query.

We further show how to generate weakest preconditions using the constraint-based methodology. The key idea is to treat the precondition as an unknown relation and repeatedly search for a precondition that is weaker than the current solution until none exists. We prove that this process requires at most n satisfiability queries, where n is the number of predicates. We then describe an interesting application of weakest precondition generation, namely generating most-general counterexample in case the assertions in the program are not valid.

This paper makes the following technical contributions:

- We show how to model the problem of discovering inductive invariants over predicate abstraction as the problem of finding a satisfying assignment to a boolean formula (Section 3). We also show how to extend this modeling to a context-sensitive interprocedural analysis, which is provably harder than intraprocedural analysis (Section 3.2).
- We show how to model the problem of weakest precondition generation over predicate abstraction as the problem of finding satisfying assignments to (at most) n boolean formulas (Section 4). This procedure can be used to find most-general counterexamples to safety assertions, assuming program termination (Section 4.1).

2 Preliminaries

2.1 Program Model

We consider programs with assignments of the form $x := e$, where x denotes some variable and e denotes some expression. (Note that memory reads/writes can be modeled using this formalism by using select-update expressions.) We also allow for assume and assert statements of the form `assume`(p) and `assert`(p), where p is some predicate. Since we allow for `assume` statements, without loss of generality, we assume that all conditionals in the program are non-deterministic.

[6] It is not difficult to extend our approach to model the process of inductive invariant generation as solving only one polynomially-sized SMT query. However, the result that we present is stronger. It shows how to reduce the problem of inductive invariant generation to the problem of solving several small SMT queries over mostly conjunctive formulas and one polynomially sized SAT query.

2.2 Generating Verification Conditions from a Program

A *cut-set* of a program is a set of program locations (called *cut-points*) such that each cycle in the control flow graph of the program 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.) A *simple path* is any path that starts at a cut-point or program entry $\pi_{\texttt{entry}}$ and ends at a cut-point or program exit $\pi_{\texttt{exit}}$ without passing through any other cut-point.

We associate the program entry and exit locations as well as each cut-point π with a *relation* R^π over program variables that are live at π. The verification condition $\texttt{VC}(\tau)$ of any simple path τ between end-points π_1 and π_2 is given by the following formula:

$$\texttt{VC}(\tau) \quad = \quad R^{\pi_1} \Rightarrow \omega(\tau, R^{\pi_2})$$

The notation $\omega(\tau, R)$ denotes the weakest precondition of path τ (which is a sequence of program instructions) with respect to R and is as defined below:

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

where the notation $[e/x]$ denotes substitution of x by e and may *not* be eagerly carried out across unknown relations R. Observe that the verification condition for any simple path τ between π_1 and π_2 simplifies to the following form:

$$\texttt{VC}(\tau) \quad = \quad R^{\pi_1} \Rightarrow (G_1 \Rightarrow (G_2 \wedge R^{\pi_2}\sigma)) \qquad (1)$$

where σ is some substitution, and G_1 and G_2 are known formulas obtained from the predicates that occur in assume and assert statements (on path τ), respectively, after appropriate substitutions. The following claim holds.

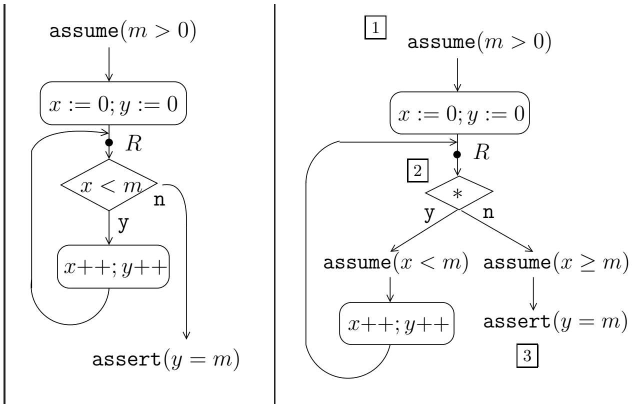
Claim 1. The assertions in the given program are valid iff when we set $R^{\pi_{\texttt{entry}}} = R^{\pi_{\texttt{exit}}} = \texttt{true}$ then there exist relations R^π for all cut-points π such that the verification conditions $\texttt{VC}(\tau)$ hold for every simple path τ.

3 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 inductive invariants R^π at different program points π such that the verification conditions $\texttt{VC}(\tau)$ in Eq. 1 holds for all simple paths τ, which implies the validity of the given assertions (Claim 1). (The issue of discovering counterexamples, in case the assertions are not valid, is addressed in Section 4.1.)

```
loop (int m) {
    x := 0;  y := 0;
    while (x < m) {
        x++;
        y++;
    }
    assert(y = m)
}
```

(a)



$$\begin{array}{l} \boxed{1} \rightarrow \boxed{2}:\ m > 0 \Rightarrow R[y \rightarrow 0, x \rightarrow 0] \\ \boxed{2} \rightarrow \boxed{3}:\ R \wedge x \geq m \Rightarrow y = m \\ \boxed{2} \rightarrow \boxed{2}:\ R \wedge x < m \Rightarrow R[y \rightarrow y+1, x \rightarrow x+1] \end{array}$$

(d)

$$S = \left\{ \begin{array}{c} x \leq y,\ x \geq y,\ x < y, \\ x \leq m,\ x \geq m,\ x < m \\ y \leq m,\ y \geq m,\ y < m \end{array} \right\}$$

(e)

Fig. 1. (a) Iteration over x with an auxiliary variable y (b) The control flow graph (CFG) with the loop invariant marked as R (c) The CFG as modeled in our system. (d) $\mathtt{VC}(\tau)$ corresponding to each simple path τ. (e) The set of predicates S.

Example We first illustrate our constraint-based approach to invariant generation by means of a simple example. Consider the program in Figure 1(a). The program loop iterates using the loop counter x and increments an auxiliary variable y as well. Its control flow graph (CFG) is shown in Figure 1(b). The program is modeled in our system as Figure 1(c). There are three simple paths going from program entry to loop header ($\boxed{1} \rightarrow \boxed{2}$), around the loop ($\boxed{2} \rightarrow \boxed{2}$), and loop header to program exit ($\boxed{2} \rightarrow \boxed{3}$), respectively, and the verification conditions they generate (using Eq. 1) are shown in Figure 1(d). The set of predicates S over which we seek to discover our inductive invariant is shown in Figure 1(e). Let π be the program point at the loop header just *after* the join point. Suppose we make the simplifying assumption that the inductive loop relation R at π is a conjunction of some predicates from S, and we seek to discover those predicates.

The first step is to associate with each predicate $p \in S$ a boolean indicator variable b_p indicating p's presence or absence in R. Then we consider each verification condition $\mathtt{VC}(\tau)$ (derived from a simple path τ using Eq. 1) in turn and generate constraints on the indicator variables:

– *Loop entry* ($\boxed{1} \rightarrow \boxed{2}$): The verification condition is $m > 0 \Rightarrow R[y \rightarrow 0, x \rightarrow 0]$, for which we generate the constraint

$$\neg b_{x<y} \wedge \neg b_{x \geq m} \wedge \neg b_{y \geq m} \qquad \text{(Ex-1)}$$

denoting that the predicates $x < y$ and $x \geq m$ and $y \geq m$ cannot be in R since they are not implied by the verification condition for loop entry.

– *Loop exit* ($\boxed{2} \rightarrow \boxed{3}$): The verification condition is $R \wedge x \geq m \Rightarrow y = m$, for which we generate the constraint

$$(b_{y \geq m} \wedge b_{y \leq m}) \vee b_{x<m} \vee (b_{x \leq y} \wedge b_{y \leq x} \wedge b_{x \leq m}) \qquad \text{(Ex-2)}$$

denoting that either both $y \geq m$ and $y \leq m$ belong to R, or $x < m$ belongs to R, or the three predicates $x \leq y$, $y \leq x$, and $x \leq m$ belongs to R. Observe that these are the only three non-trivial[7] ways in which we can prove $y = m$ under the assumption $x \geq m$. In general, these different ways are computed by using the predicate cover operation.

– *Inductive* ($\boxed{2} \rightarrow \boxed{2}$): The verification condition is $R \wedge x < m \Rightarrow R[y \rightarrow y+1, x \rightarrow x+1]$, for which we generate the constraint

$$(b_{y \leq m} \Rightarrow b_{y<m} \vee (b_{x \leq y} \wedge b_{y \leq x})) \wedge \neg b_{x<m} \wedge \neg b_{y<m} \qquad \text{(Ex-3)}$$

denoting that if $y \leq m$ belongs to R, then either $y < m$ or $x \leq y \wedge y \leq x$ should also belong to R, and that the predicates $x < m$ and $y < m$ cannot be in R. The reader can easily check that this verification condition allows any other predicate p to be in R because $p \wedge x < m \Rightarrow p[y \rightarrow y+1, x \rightarrow x+1]$. These constraints are generated by considering each predicate p, finding the weakest conditions $\delta(p)$ over the set of predicates under which $p \wedge x < m \Rightarrow p[y \rightarrow y+1, x \rightarrow x+1]$ and then generating the constraint that $b_p \Rightarrow \delta(p)$. For the predicate $b_{x<m}$ and $b_{y<m}$, $\delta(p)$ is `false` and hence we generate the constraints $\neg b_{x<m}$ and $\neg b_{y<m}$. For the predicate $b_{y \leq m}$, $\delta(p)$ is $b_{y<m} \vee (b_{x \leq y} \wedge b_{y \leq x})$. For all other predicates, $\delta(p)$ is `true`.

Putting Eq. (Ex-1), (Ex-2), (Ex-3) together we get a SAT formula (over the boolean indicator variables) that encodes the verification condition of the program. The reader can verify that $b_{x \geq y} = b_{x \leq y} = b_{x \leq m} =$ `true` (and all others `false`) is a satisfying solution. This corresponds to R being $(x = y \wedge x \leq m)$.

3.1 Formal constraint generation

We now formally present our constraint-based methodology for discovering the inductive invariants R^π when they can be described using a k-DNF formula over a given set of predicates S. (We use k-DNF form for simplicity. Our methodology can also be applied to other boolean structures that are representable by a bounded number of boolean variables.) In such a case, we can represent R^π by $k \times n$ *boolean indicator variables* $b^\pi_{i,p}$ (where $1 \leq i \leq k$, $p \in S$, $n = |S|$), where the boolean variable $b^\pi_{i,p}$ denotes whether predicate p is present in the i^{th} disjunct of the invariant R^π at program point π. We show how to encode the verification condition of the program as a boolean formula ψ over the boolean indicator variables $b^\pi_{i,p}$. The boolean formula ψ is satisfiable iff there exist inductive invariants (in k-DNF form) strong enough to prove the validity of the assertions—this is the key result of the paper.

[7] Trivial expressions in this case are those that imply `false`, e.g., $(x \geq y \wedge x < y)$.

We first show how to encode the verification condition of any simple path τ as a boolean formula $\psi(\tau)$. The following three cases arise, which we consider in increasing order of difficulty:

Case 1: *Path between program entry and a cut-point.* The verification condition in Eq. 1 simplifies to the following form after substituting $R^{\pi_1} = \mathtt{true}$ and expanding R^{π_2} as $\bigvee_{j=1}^{k} R_j^{\pi_2}$, where each $R_j^{\pi_2}$ is conjunction of some predicates from S.

$$G_1 \Rightarrow \left(G_2 \wedge \bigvee_{j=1}^{k} R_j^{\pi_2} \sigma \right)$$

The above constraint restricts how strong R^{π_2} can be. In particular, if $p_1 \in R_1^{\pi_2}, \ldots, p_k \in R_k^{\pi_2}$, then it must be the case that $G_1 \Rightarrow \bigvee_{j=1}^{k} p_j \sigma$. Hence, we can rewrite the above constraint as:

$$(G_1 \Rightarrow G_2) \wedge \bigwedge_{p_1,..,p_k \in S} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow (G_1 \Rightarrow \bigvee_{j=1}^{k} p_j \sigma) \right) \tag{2}$$

This can be encoded as the following boolean constraint $\psi(\tau)$ over boolean indicator variables $b_{i,p}^{\pi_2}$.

$$\psi(\tau) \quad = \quad D(G_1, G_2) \wedge \bigwedge_{p_1,..,p_k \in S} \left((\bigwedge_{j=1}^{k} b_{j,p_j}^{\pi_2}) \Rightarrow D(G_1, \bigvee_{j=1}^{k} p_j \sigma) \right) \tag{3}$$

where $D(A, B)$ denotes the boolean formula `true` if $A \Rightarrow B$ and `false` otherwise.

Case 2: *Path between a cut-point and program exit.* The verification condition in Eq. 1 simplifies to the following form after substituting $R^{\pi_2} = \mathtt{true}$ and expanding R^{π_1} as $\bigvee_{j=1}^{k} R_j^{\pi_1}$, where each $R_j^{\pi_1}$ is conjunction of some predicates from S.

$$\left(\bigvee_{i=1}^{k} R_i^{\pi_1} \right) \Rightarrow (G_1 \Rightarrow G_2) \quad \text{or, equivalently,} \quad \bigwedge_{i=1}^{k} (R_i^{\pi_1} \Rightarrow (G_1 \Rightarrow G_2))$$

The above constraint restricts how weak $R_i^{\pi_1}$ can be. We can encode the above constraint as a boolean formula over the variables $b_{i,p}^{\pi}$ by considering the predicate cover[8] of $G_1 \Rightarrow G_2$. To recall, the predicate cover of a formula F over a set of predicates S, denoted by $C_S(F)$, is the weakest formula over

[8] It is a fundamental operation used in the abstract transformers while performing abstract interpretation over predicate abstraction [1].

predicates from S that implies F. Let $\phi_S(F, i, \pi)$ denote the boolean formula over boolean variables $b^{\pi}_{i,p}$ obtained after replacing each predicate p in $C_S(F)$ by $b^{\pi}_{i,p}$. The verification condition above can now be encoded as the following boolean constraint $\psi(\tau)$ over boolean indicator variables $b^{\pi_1}_{i,p}$.

$$\psi(\tau) \quad = \quad \bigwedge_{i=1}^{k} \phi_S(G_1 \Rightarrow G_2, i, \pi_1) \tag{4}$$

Case 3: *Path between two adjacent cut-points.* We now combine the key ideas that we used in the above two cases to handle this more general case. The verification condition in Eq. 1 has the following form (after expanding R^{π_1} as $\bigvee\limits_{i=1}^{k} R^{\pi_1}_i$ and R^{π_2} as $\bigvee\limits_{j=1}^{k} R^{\pi_2}_j$, where each $R^{\pi_1}_i$ and $R^{\pi_2}_j$ is a conjunction of some predicates from S).

$$\left(\bigvee_{i=1}^{k} R^{\pi_1}_i\right) \Rightarrow \left(G_1 \Rightarrow (G_2 \wedge \bigvee_{j=1}^{k} R^{\pi_2}_j \sigma\right)$$

$$\text{or, equivalently,} \quad \bigwedge_{i=1}^{k} \left(R^{\pi_1}_i \Rightarrow \left(G_1 \Rightarrow (G_2 \wedge \bigvee_{j=1}^{k} R^{\pi_2}_j \sigma_\tau)\right)\right) \tag{5}$$

The above constraint can be rewritten as (using the same logic used in generating the constraint in Eq. 2):

$$\bigwedge_{i=1}^{k} \bigwedge_{p_1,..,p_k \in S} \left((\bigwedge_{j=1}^{k} b^{\pi_2}_{j,p_j}) \Rightarrow \left(R^{\pi_1}_i \Rightarrow (G_1 \Rightarrow (G_2 \Rightarrow \bigvee_{j=1}^{k} p_j \sigma_\tau)) \right) \right)$$

The verification condition above can be encoded as the following boolean constraint $\psi(\tau)$ over boolean indicator variables $b^{\pi_1}_{i,p}$ and $b^{\pi_2}_{i,p}$ (using the same logic used in generating the constraint in Eq. 4):

$$\psi(\tau) = \bigwedge_{i=1}^{k} \bigwedge_{p_1,..,p_k \in S} \left((\bigwedge_{j=1}^{k} b^{\pi_2}_{j,p_j}) \Rightarrow \phi_S \left((G_1 \Rightarrow (G_2 \wedge \bigvee_{j=1}^{k} p_j \sigma)), i, \pi_1 \right) \right) \tag{6}$$

Observe that the constraints are generated locally from the verification condition of each simple path. Hence, the constraint based technique has the potential for efficient incremental verification (verification of a modified version of an already verified program) with support of an incremental SAT solver.

Example Appendix A gives examples of each of the above cases over Figure 1(a).

The desired boolean formula ψ is now given by the conjunction of formulas $\psi(\tau)$ for all simple paths τ in the program. Since ψ encodes the entire verification condition of the program, it is easy to see that the following claim holds.

Claim 2. The boolean formula ψ is satisfiable iff there exist inductive invariants (in k-DNF form) strong enough to prove the validity of the given assertions.

3.2 Interprocedural Analysis

The ω computation described in Section 2.2 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 number of different calling contexts can potentially be exponential in the number of predicates over program inputs. A standard way 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. A pair (A_i, B_i) denotes that whenever the procedure is called in a context that satisfies A_i, the procedure ensures that the outputs satisfy the constraint B_i. However, the efficient construction of relevant pre/postcondition pairs is unclear.

In this section, we show that the constraint-based approach is particularly suited to discovering 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 in an assume-guarantee style reasoning. Our encoding requires the summary to be only as precise as is required for verification of the given assertions.

Procedure bodies: Without loss of generality, let 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 there are q interesting calling contexts for the 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 the constraint system is satisfiable.) We can summarize the behavior of each procedure using q tuples (A_i, B_i) for $1 \leq i \leq q$, where A_i is some (unknown) relation over $\boldsymbol{x}$, and B_i is some (unknown) relation over $\boldsymbol{x}$ and $\boldsymbol{y}$. We ensure this by generating constraints for each i as below:

$$\texttt{assume}(A_i);\ S;\ \texttt{assert}(B_i) \tag{7}$$

Procedure calls: For simplicity, we assume that the cut-set includes all program locations before any procedure call. For any simple path τ that starts with a procedure call $\boldsymbol{v} := P(\boldsymbol{u})$, let τ_i denote the simple path in which the procedure call is replaced by the following code fragment, where $\boldsymbol{t}$ is a fresh set of variables.

$$\texttt{assert}(A_i[\boldsymbol{u}/\boldsymbol{x}]); \texttt{assume}(B_i[\boldsymbol{u}/\boldsymbol{x}, \boldsymbol{t}/\boldsymbol{y}]); \boldsymbol{v} := \boldsymbol{t}; \tag{8}$$

The boolean formula $\psi(\tau_i)$ that encodes the verification condition of the simple path τ_i can be computed using the method described in Section 3. The formula that encodes the verification condition corresponding to τ is $\psi(\tau) = \bigvee_{i=1}^{q} \psi(\tau_i)$.

Example In Appendix B we illustrate the technique over examples from [5, 6].

4 Weakest Precondition Inference

Given a program with some assertions, the problem of weakest precondition generation is to infer the weakest precondition $R^{\pi_{\texttt{entry}}}$ such that whenever the program is run in a state that satisfies $R^{\pi_{\texttt{entry}}}$, the assertions in the program hold. This weakest precondition inference problem is harder than program verification[9], and relatively few techniques exist for it. Since a precise solution is undecidable, we work with a relaxed notion of weakest precondition. For a given template structure, we say that A is a weakest precondition if A is a precondition that fits the template and there does not exist a weaker precondition than A with similar properties.

In this section, we present a constraint-based approach to inferring weakest preconditions under the given template. In particular, we show how to generate a conjunctive weakest precondition for a given program with assertions. A k-DNF weakest precondition can then be obtained by taking disjunctions of k disjoint conjunctive weakest preconditions, generated iteratively. Our constraint-based approach permits an elegant weakest precondition inference technique based on the monotonicity of implication for CNF formulae over a given set of predicates.

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

We use an iterative approach to generating a conjunctive weakest precondition as follows. We add the constraint that the precondition $R^{\pi_{\texttt{entry}}}$ should be weaker than the current solution S to the verification condition (in Eq. 1). This constraint is encoded by the boolean formula $(\gamma_1 \wedge \neg\gamma_2)$. Here, γ_1 and γ_2 are boolean formulae over the boolean variables $b_p^{\pi_{\texttt{entry}}}$ that encode the constraints $S \Rightarrow R^{\pi_{\texttt{entry}}}$ and $R^{\pi_{\texttt{entry}}} \Rightarrow S$, respectively and are computed using the technique described in Section 3.

Once a weakest conjunctive precondition has been found, we repeat the process to generate other weakest conjunctive preconditions. In order to ensure that we get a precondition that is disjoint from the weakest preconditions already found, we add an additional constraint $\neg\gamma_3$, where γ_3 is the boolean formula over the boolean variables $b_p^{\pi_{\texttt{entry}}}$ that encodes the constraint $R^{\pi_{\texttt{entry}}} \Rightarrow \bigvee_i T_i$, where T_i are the conjunctive weakest preconditions that have already been discovered.

Example We again consider Figure 1(a) but with line 1 ($x := 0; y := 0$) removed and infer weakest preconditions between x, y, m using the predicate set S shown in Figure 1(e). We generate two conjunctive weakest preconditions: $(y = m \wedge x \geq m)$ and $(x = y \wedge x < m)$; their disjunction yields the weakest precondition.

[9] A weakest precondition generator can be used to solve the program verification problem by simply checking whether the weakest precondition generated for given assertions is `true` or not.

```
err (int m) {
  while (x < m) {
    x++;
    y++;
    assert(y < m)
  }
}
```

(a)

```
err (int m) {
  error := 0;
  while (x < m) {
    x++; y++;
    if (y ≥ m)
      error := 1; goto L;
  }
  L: assert(error = 1)
}
```

(b)

Fig. 2. (a) Example with safety assertion $y < m$. (b) Instrumented program. We compute $(x<m \wedge y \geq x)$ as the most-general counterexample that violates the assertion.

4.1 Most-General Counterexample Inference

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 given assertions. In this section, we show how to find *most-general* counterexamples to given assertions.

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 violation of some reachable safety assertion. Generating a most-general counterexample is more desirable than generating a concrete counterexample, and can aid in, say, program debugging. For example, it is more useful to know there is an assertion failure whenever $x < y$ as opposed to knowing that there is an assertion failure when $x = 0 \wedge y = 3$.

We show next how to find a most-general counterexample using the techniques discussed in Section 4 under the assumption that the given program is terminating, i.e., the program exit is always reached. The basic idea is to reduce the problem to that of finding the weakest precondition for some safety property. This reduction involves constructing another program from the given program `Prog` using the following transformation, $T_{\texttt{err}}(\texttt{Prog})$: We introduce a new variable `error` that is set to 0 at the beginning of the program. Whenever violation of the given safety property occurs (i.e., the negation of any of the safety assertions holds), the variable `error` is set to 1 and the control jumps to the end of the program. We assert that `error` $= 1$ at the end of the program, and remove the original safety assertions from the program.

Claim 3. Let `Prog` be a terminating program with some safety assertions. Then, `Prog` has an assertion violation iff the assertions in program $T_{\texttt{err}}(\texttt{Prog})$ hold.

The significance of Claim 3 is 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 Verification							Weakest Precondition Inference							
	Number of				Time for			Number of					Time for		
Program	n	k	vars	clauses	CG	CNF	SAT	n	k	vars	clauses	sol	CG	CNF	SAT
counter	12	1	12	21	0.23	0.14	0.04	12	1	24	1345	1	0.23	0.44	0.05
ex1a [7]	12	1	12	22	0.23	0.15	0.04	14	1	28	1857	2	0.25	0.67	0.07
lockstep	5	1	5	8	0.23	0.11	0.03	9	1	18	584	2	0.23	0.29	0.05
nested	16	1	32	62	0.23	0.26	0.04	18	1	54	2866	2	0.23	1.52	0.09
twoloop	20	1	40	79	0.23	0.36	0.04	20	1	60	3778	3	0.23	1.86	0.16
ex2 [8]	12	2	24	72	0.23	0.14	0.04	12	2	36	4588	1	0.23	1.16	0.11
ex1b [7]	20	2	40	1704	0.23	10.68	0.06	20	2	60	7548	1	0.23	14.22	0.09
ex3 [9]	20	2	40	1782	0.23	8.53	0.06	13	2	39	2031	4	0.23	3.90	0.14

Table 1. Results for (a) Program verification (b) Weakest Precondition Inference

program. We need to track the new boolean variable `error` in the transformed program and therefore add $\texttt{error} = 1$ and $\texttt{error} = 0$ to the predicate set.

Example Consider the program shown in Figure 2(a), which we instrument with the error variable to obtain the program in Figure 2(b). Our weakest precondition inference generates $(x < m \wedge y \geq x)$ as the most general counterexample that violates the assertion $y < m$. Note that we need k to be at least 2 since the inductive invariant (at the loop header) for establishing the counterexample is $(x < m \wedge y \geq x) \vee (\texttt{error} = 1 \wedge y \geq x))$.

Observe the importance of introducing the error variable. An alternative 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, running our tool with such an alternative transformation yields two conjunctive weakest preconditions—$(x \geq m)$ and $(x < m \wedge y \geq m - 1)$ of which the former does not describe a counterexample, while the latter does not describe the weakest conjunctive counter-example.

5 Experiments

In this section we demonstrate the viability of a constraint-based approach by uniformly discovering invariants for programs for which specialized techniques [7–9] have been proposed.

The results of invariant generation for program verification are shown in Table 1(a). The first set of columns indicate the programs[10], the parameters (number of disjuncts k, and size of predicate set n), and the number of variables and clauses in the CNF formula. The second set indicates the time (in seconds) on generating the program constraints (CG), generating the CNF formula (CNF) and solving the resulting SAT instance (SAT). We use Z3 [10] as our SAT solver.

[10] Available at http://research.microsoft.com/users/sumitg/benchmarks/pa.html

The first set of examples require conjunctive invariants ($k = 1$). The first program (*counter*) is a loop iteration with a counter from $1 \ldots m$. The second (*lockstep*, shown in Figure 1) is also a counter iteration but with another variable counting in lock-step. The third (*nested*) consists of two nested counter loops. The next program (*twoloop*) consists of two counter loops one after the other. The last two examples need two invariants, one at each loop header.

The second set requires disjunctive invariants ($k = 2$) and are from recent work on sophisticated invariant generation techniques like CFG elaboration (*ex2* [8]), probabilistic inference (*ex3* [9]), and sophisticated widening (*ex1b* [7]).

Our technique uniformly discovers invariants over predicate abstraction for all these examples. Our base predicates are difference constraints over the program variables with small constants. Program parsing and constraint generation takes 0.23s. Our preliminary tool uses an unoptimized implementation of predicate cover and therefore spends most of its time in CNF generation, which can be improved easily. Solving the resulting CNF constraints takes 0.04s on average. Our preliminary tool also shows a noticeable overhead when a disjunctive invariant at the loop header causes case enumeration during CNF generation (in *ex3* [9] and *ex1b* [7]). However, even for large SAT instances in these cases, solutions are generated by the solver in very reasonable time, demonstrating the viability of a constraint-based approach.

5.1 Weakest Precondition Inference

The SAT solver that we used tends to generate a *maximally-false* satisfying assignment to a satisfiable boolean formula, as a result of which we obtained conjunctive weakest preconditions in the first query and did not have to iterate n times. A satisfying assignment A to a boolean formula is *maximally-false* if by changing the truth values of any of the boolean variables from `true` to `false` in assignment A transforms A to an unsatisfying assignment.

We exploit this property by adding an additional constraint to the system, which in practice improves performance. The added clauses constrains the weakest precondition to be saturated, i.e., for all predicates p_1, p_2, p_3, if $p_1 \wedge p_2 \Rightarrow p_3$, we add the constraint:

$$b_{p_1}^{\pi_{\texttt{entry}}} \wedge b_{p_2}^{\pi_{\texttt{entry}}} \Rightarrow b_{p_3}^{\pi_{\texttt{entry}}} \tag{9}$$

Claim 4. A *maximally-false* satisfying assignment to the boolean formula ψ (that encodes the verification condition of the program) along with the constraint in Eq. 9 yields a conjunctive weakest precondition.

The results for weakest precondition inference are shown in Table 1(b). Our tool generates all weakest preconditions when multiple incomparable ones exist. Therefore, in addition to the number of predicates n, disjuncts k, variables and clauses in the CNF, we also report the number of solutions generated. In such cases, we report the cumulative time required for generating all solutions.

Due to the tendency of the SAT-solver to generate a maximally-false assignment, our tool produced valid weakest preconditions in the first iteration for all but two programs, *ex2* [8] and *twoloop*, each of which required two iterations.

6 Related Work

Constraint-based techniques have been recently used for discovering linear arithmetic invariants (conjunctive invariants [11–14] as well as disjunctive invariants [15] in the context of verifying safety properties as well as discovering ranking functions for proving termination [16, 17]). Constraint-based techniques have also been applied for discovering non-linear polynomial invariants [13, 18] and invariants in the combined theory of linear arithmetic and uninterpreted functions [19]. In contrast, this paper extends the applicability of constraint-based methodology to the important domain of predicate abstraction, where the predicates can be from any theory. There are two key technical differences between the earlier work that focused on arithmetic invariants and the current work based on predicate abstraction: (a) The key principle behind a constraint-based methodology is to convert universal quantification into existential quantification in the verification condition. In this respect, the earlier work uses Farkas' lemma, while the current work uses the predicate cover operation. (b) The earlier work translates the problem of discovering arithmetic invariants into solving polynomial constraints[11], while in contrast the proposed technique translates the problem of discovering invariants over a given set of predicates into solving a SAT constraint. The latter is more desirable since we have good off-the-shelf SAT solvers.

Constraint-based techniques, being goal-directed, work naturally in program verification mode where the task is to discover inductive loop invariants for verification of given assertions. As a result, earlier work on constraint-based techniques (with the exception of [15]) has been limited to program verification as opposed to other program analysis problems such as weakest precondition generation. This paper demonstrates the applicability of constraint-based methodology to the problem of weakest precondition generation, which in turn can be used for generation of most-general counterexamples (assuming program termination). The technique used for weakest precondition generation in [15] is based on binary search over arithmetic coefficients from a bounded range, while the technique used for weakest precondition generation in this paper relies on monotonicity of implication of a conjunctive set of predicates.

SATURN [20] also uses SAT-solving, but for bug-finding in loop-free programs. (Programs with loops are modeled by unrolling loops.) Theoretically, it is well known that loop-free programs can be modeled as Boolean circuits. SATURN's contribution is primarily engineering-based; it illustrates that the SAT queries that are generated from real programs with complicated constructs can be efficiently solved in practice. In sharp contrast, we focus on invariant inference for correctness proofs and show how programs with loops can be abstracted as Boolean circuits. Additionally, our work finds *most-general* bugs in programs with loops.

[11] [15] further proposes solving the quadratic inequalities using bit-vector modeling, thus effectively translating into SAT constraints; however, this reduction to SAT is artificial in that it is used only to approximate the SMT query, because current solvers cannot generate models for SMT queries that involve multiplication.

7 Conclusions and Future Work

In this paper we present a constraint-based technique for discovering inductive program invariants over predicate abstraction. We show how to push the burden of fixed-point computation as well as the boolean reasoning to a SAT-solver by encoding program verification conditions as SAT-constraints over boolean indicator variables. A solution to the SAT instance maps directly to inductive program invariants that prove the validity of given program assertions. We lift the base verification procedure to the interprocedural setting and additionally infer weakest preconditions that can be used for most-general bug finding.

We present encouraging preliminary results using a prototype implementation. Integration with abstraction refinement procedures and model checking frameworks for evaluation over industrial-sized programs remains future work.

References

1. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification. (1997) 72–83
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by constr. or approx. of fixpoints. In: POPL. (1977) 238–252
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
4. McMillan, K.L.: Appl. of craig interpolants in model checking. In: TACAS. (2005)
5. Seidl, H., Flexeder, A., Petter, M.: Interprocedurally analysing linear inequality relations. In: ESOP. (2007) 284–299
6. Müller-Olm, M., Seidl, H., Steffen, B.: Interprocedural analysis (almost) for free. In: Technical Report 790, Fachbereich Informatik, Universitt Dortmund. (2004)
7. Gulavani, B.S., Rajamani, S.K.: Counterexample driven refinement for abstract interpretation. In: TACAS. (2006) 474–488
8. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: SAS. (2006) 3–17
9. Gulwani, S., Jojic, N.: Program verification as prob. inference. In: POPL. (2007)
10. de Moura, L.M., Bjørner, N.: Eff. E-Matching for SMT solvers. In: CADE. (2007)
11. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV. (2003) 420–432
12. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS. (2004) 53–68
13. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: POPL. (2004) 318–329
14. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: VMCAI. (2005) 25–41
15. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. (2008) 281–292
16. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI. (2004) 239–251
17. Bradley, A.R., Manna, Z., Sipma, H.B.: Lin. ranking with reach. In: CAV. (2005)
18. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Deduction and Applications. (2005)
19. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: VMCAI. Volume 4349 of LNCS. (2007) 378–394
20. Xie, Y., Aiken, A.: Saturn: A sat-based tool for bug det. In: CAV. (2005) 139–143

A Example: Formal constraint generation

In this section, we illustrate the general framework for constraint generation (discussed in Section 3.1) on the example in Figure 1. The task here is to find the inductive invariant R at the loop join point π. Remember that we are considering invariants with one disjunct, i.e., $k = 1$ over the set of predicates S in Figure 1(e). For $p \in S$ we use indicator variables b_p (simplifying the notation because both the program point and disjunct are unique) to denote whether p is in R or not. Recall our definition of $D(A, B)$ as being `true` iff $A \Rightarrow B$. We describe some of the constraint that we get from each of the three verification conditions below:

- *Loop entry (Case 1):* The loop entry verification condition is $m > 0 \Rightarrow R[y \rightarrow 0, x \rightarrow 0]$, and therefore $G_1 = (m > 0)$ and $G_2 = \texttt{true}$. $D(m > 0, \texttt{true})$ is obviously `true` and the verification condition from Eq. 3 simplifies to:

$$\psi(\tau) = \bigwedge_{p \in S} (b_p \Rightarrow D(m > 0, p[x \rightarrow 0, y \rightarrow 0]))$$

  For all $p \in \{x \geq y, x \leq y, x \leq m, y \leq m, x < m, y < m\}$ the term $D(m > 0, p[y \rightarrow 0, x \rightarrow 0])$ reduces to `true` and therefore no constraints are imposed on the corresponding indicator variables. On the other hand, for the remaining predicates the term reduces to `false` and the following verification condition is generated $\psi(\tau) = ((b_{x \geq m} \Rightarrow \texttt{false}) \wedge (b_{y \geq m} \Rightarrow \texttt{false}) \wedge (b_{x<y} \Rightarrow \texttt{false}))$, or equivalently $\neg b_{x \geq m} \wedge \neg b_{y \geq m} \wedge \neg b_{x<y}$.
- *Loop exit (Case 2):* The verification condition is $R \wedge x \geq m \Rightarrow y = m$ and therefore $G_1 = (x \geq m)$ and $G_2 = (y = m)$. The verification condition, from Eq. 4, is

$$\psi(\tau) = \phi_S(x \geq m \Rightarrow y = m, 1, \pi)$$

  The predicate cover for the formula $F = (x \geq m \Rightarrow y = m)$ over the predicate set S is the following:

$$C_S(F) = (x \leq m \wedge x \leq y \wedge x \geq y) \vee (y \geq m \wedge y \leq m) \vee (x < m)$$

  Notice that expressions such as $y \geq x \wedge y \leq m$ imply the formula but are not the weakest (they are implied by some element above) and therefore do not appear in $C_S(F)$.
  Given the predicate cover $C_S(F)$ as above and the definition of ϕ_S, we can encode the verification condition as:

$$\psi(\tau) = (b_{x \leq m} \wedge b_{x \leq y} \wedge b_{x \geq y}) \vee (b_{y \geq m} \wedge b_{y \leq m}) \vee b_{x<m}$$

- *Inductive (Case 3):* The verification condition is $R \wedge x < m \Rightarrow R[y \rightarrow y+1, x \rightarrow x+1]$ and therefore $G_1 = x < m$ and $G_2 = \texttt{true}$. The unmodified constraint for the inductive case from Eq. 5 is $R \Rightarrow (x < m \Rightarrow (\texttt{true} \wedge R\sigma))$ that reduces to $R \Rightarrow (x < m \Rightarrow R\sigma)$, where σ is $[y \rightarrow y + 1, x \rightarrow x + 1]$. Applying the first step of the reduction from Case 3, we get:

$$\bigwedge_{p \in S} (b_p \Rightarrow (R \Rightarrow (x < m \Rightarrow p\sigma))$$

Applying the next step we get the following form of Eq. 6:

$$\psi(\tau) = \bigwedge_{p \in S} (b_p \Rightarrow \phi_S\left((x < m \Rightarrow p_j\sigma), 1, \pi\right))$$

For each of the predicates p in the following cases, we consider the predicate cover for $F_p = (x < m \Rightarrow p\sigma)$:

- $p = (y \leq m)$: The predicate cover $C_S(F_{y \leq m})$ is $(y < m) \vee (x \leq y \wedge x \geq y)$. Therefore, the term corresponding to $y \leq m$ is $b_{y \leq m} \Rightarrow (b_{y<m} \vee (b_{x \leq y} \wedge b_{x \geq y}))$.
- $p \in \{x < m, y < m\}$: For any p here, the weakest precondition that implies F_p is just `false`. Therefore, we generate the $b_p \Rightarrow$ `false`, or equivalently $\neg b_p$.
- $p \in \{x \geq y, x \leq y, x < y, x \leq m, x \geq m, y \geq m\}$: Consider $p = (x \geq y)$ for which $F_p = (x < m \Rightarrow (x \geq y)\sigma)$. The weakest precondition that ensures F_p is $x \geq y$ itself, and the corresponding constraint $b_{x \geq y} \Rightarrow b_{x \geq y}$ is trivial. Similarly, all other elements here can be easily verified to generate trivial constraints which we omit from the output.

B Example: Interprocedural Analysis

Consider the program shown in Figure 3(a). The instrumented procedures are shown in Figure 3(a'). We assume that there is only one summary of relevance ($q = 1$) and therefore attempts to solve the constraint system for the pre/postcondition pair (A_1, B_1) for the procedure `Add`. The input relation A_1 is constrained to contain predicates from $S_{\texttt{Add},1}$ only while the output relation B_1 can contain predicates from $S_{\texttt{Add},1} \cup S_{\texttt{Add},2}$.

Each call to `Add` is instrumented in accordance with Eq. (8). Then the procedure body of `Add` (for which procedure summaries are being computed) is wrapped in accordance with Eq. (7). Our algorithm verifies the assertion by generating the summary $(i \geq 0, \texttt{ret} = i + j)$ for procedure `Add`. This example illustrates that only relevant summaries need to be computed. The true branch of the conditional inside `Add` has can be summarized as $(i < 0, \texttt{ret} = j)$. The system would be unsatisfiable if an additional invocation site existed with the context $i < 0$. But the constraint system is satisfiable with $q = 1$ and this additional summary is not required for verification of this example.

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

(a)

```
IP() {
   x := 5;  y := 3;
   {
      /*out := Add(x, y);*/
      assert(A1[x/i, y/j]);
      assume(B1[x/i, y/j, t/ret]);
      out := t;
   }
   assert(out = 8);
}
```

```
Add(int i, j) {
   assume(A1)
   if i ≤ 0
      ret := j;
   else
      b := i − 1;
      c := j + 1;
      {
         /* ret := Add(b, c); */
         assert(A1[b/i, c/j]);
         assume(B1[b/i, c/j, t'/ret]);
         ret := t';
      }
   assert(B1)
}
```

(a')

$$S_{\mathtt{Add},1} = \left\{ \begin{array}{c} i \leq 0, i \geq 0, j \leq 0, j \geq 0, \\ i \leq j, i \geq j \end{array} \right\}$$

$$S_{\mathtt{Add},2} = \left\{ \begin{array}{c} \mathtt{ret} \leq i, \mathtt{ret} \geq i, \mathtt{ret} \leq j, \\ \mathtt{ret} \leq 0, \mathtt{ret} \geq 0, \mathtt{ret} \geq j, \\ \mathtt{ret} \leq i + j, \mathtt{ret} \geq i + j, \\ \mathtt{ret} \leq i - j, \mathtt{ret} \geq i - j, \end{array} \right\}$$

(b)

Fig. 3. (a) Interprocedural analysis example taken from [5, 6] (a') Instrumented program (b) Predicate sets.