Program Verification using Templates over Predicate Abstraction

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

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Abstract

We address the problem of automatically generating invariants with quantified and boolean structure for proving the validity of given assertions or generating pre-conditions under which the assertions are valid. We present three novel algorithms, having different strengths, that combine template and predicate abstraction based formalisms to discover required sophisticated program invariants using SMT solvers.

Two of these algorithms use an iterative approach to compute fixed-points (one computes a least fixed-point and the other computes a greatest fixed-point), while the third algorithm uses a constraint based approach to encode the fixed-point. The key idea in all these algorithms is to reduce the problem of invariant discovery to that of finding *optimal* solutions for unknowns (over conjunctions of some predicates from a given set) in a template formula such that the formula is valid.

Preliminary experiments using our implementation of these algorithms show encouraging results over a benchmark of small but complicated programs. Our algorithms can verify program properties that, to our knowledge, have not been automatically verified before. In particular, our algorithms can generate full correctness proofs for sorting algorithms (which requires nested universally-existentially quantified invariants) and can also generate preconditions required to establish worst-case upper bounds of sorting algorithms. Furthermore, for the case of previously considered properties, in particular sortedness in sorting algorithms, our algorithms take less time than reported by previous techniques.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants, Logics of Programs, Mechanical Verification, Pre- and Post-conditions; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness Proofs, Formal Methods

General Terms Languages, Verification, Theory, Algorithms

Keywords Predicate abstraction, Quantified Invariants, Template Invariants, Iterative Fixed-point, Constraint-based Fixed-point, Weakest Preconditions, SMT Solvers.

* The work was supported in part by CCF-0430118 and in part done during an internship at Microsoft Research.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

1. Introduction

There has been a traditional trade-off between automation and precision for the task of program verification. At one end of the spectrum, we have fully automated techniques like data-flow analysis [19], abstract interpretation [5] and model checking [8] that can perform iterative fixed-point computation over loops, but are limited in the kind of invariants that they can discover. At the other end, we have approaches based on verification condition generation that can be used to establish sophisticated properties of a program using SMT solvers [25, 32], but require the programmer to provide all the sophisticated properties along with loop invariants, which are usually even more sophisticated. The former approach enjoys the benefit of complete automation, while the latter approach enjoys the benefit of leveraging the engineering and state-of-the-art advances that are continually being made in SMT solvers. In this paper, we explore the middle ground, wherein we show how to use SMT solvers as black-boxes to discover sophisticated inductive loop invariants, using only a little help from the programmer in the form of templates and predicates.

We take inspiration from recent work on template-based program analysis [26, 27, 4, 18, 1, 13, 12, 14] that has shown promise in discovering invariants that are beyond the reach of fully-automated techniques. The programmer provides hints in the form of a set of invariant templates with holes/unknowns that are then automatically filled in by the analysis. However, most of existing work in this area has focused on quantifier-free numerical invariants and depends on specialized non-linear solvers to find solutions to the unknowns. In contrast, we focus on invariants that are useful for a more general class of programs. In particular, we consider invariants with arbitrary but pre-specified logical structure (involving disjunctions and universal and existential quantifiers) over a given set of predicates. One of the key features of our template-based approach is that it uses the standard interface to an SMT solver, allowing it to go beyond numerical properties and leverage ongoing advances in SMT solving.

Our templates consist of formulas with arbitrary logical structure (quantifiers, boolean connectives) and unknowns that take values over some conjunction of a given set of predicates (Section 2). Such a choice of templates puts our work in an unexplored space in the area of predicate abstraction, which has been highly successful in expressing useful non-numerical and disjunctive properties of programs. The area was pioneered by Graf and Seïdi [10], who showed how to compute quantifier-free invariants (over a given set of predicates). Later, strategies were proposed to discover universally quantified invariants [9, 21, 17] and disjunctions of universally quantified invariants in the context of shape analysis [22]. Our work extends the field by discovering invariants that involve an arbitrary (but pre-specified) quantified structure over a given set of predicates. Since the domain is finite, one can potentially search over all possible solutions, but this naive approach would be infeasible.

(a) $\texttt{InsertionSort}(\texttt{Array } A, \texttt{int } n)$

1 $i := 1;$
2 $\texttt{while } (i < n)$
3 $\quad j := i - 1;\ \texttt{val} := A[i];$
4 $\quad \texttt{while } (j \geq 0 \wedge A[j] > \texttt{val})$
5 $\quad\quad A[j+1] := A[j];$
6 $\quad\quad j := j - 1;$
7 $\quad A[j+1] := \texttt{val};$
8 $\quad i := i + 1;$
9 $\texttt{Assert}(\forall y \exists x : (0 \leq y < n)$
10 $\quad\quad \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x < n))$

User Input:
Invariant Template: $v_1 \ \wedge\ (\forall y : v_2 \Rightarrow v_3) \ \wedge\ (\forall y \exists x : v_4 \Rightarrow v_5)$
Predicate Set: $\texttt{AllPreds}(\{x, y, i, j, n\}, \{0, \pm 1\}, \{\leq, \geq, \neq\}) \ \cup$
$\quad \texttt{AllPreds}(\{\texttt{val}, A[t], \tilde{A}[t] \mid t \in \{i, j, x, y, n\}\}, \{0\}, \{=\})$

Tool Output: Proof of validity of assertion:
Outer Loop Invariant:
$\forall y : (i \leq y < n) \Rightarrow (\tilde{A}[y] = A[y]) \ \wedge$
$\forall y \exists x : (0 \leq y < i) \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x < i)$
Inner Loop Invariant:
$\texttt{val} = \tilde{A}[i] \wedge -1 \leq j < i \ \wedge$
$\forall y : (i < y < n) \Rightarrow \tilde{A}[y] = A[y] \ \wedge$
$\forall y \exists x : (0 \leq y < i) \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x \leq i \wedge x \neq j + 1)$

(b) $\texttt{SelectionSort}(\texttt{Array } A, \texttt{int } n)$

$i := 0;$
$\texttt{while } (i < n - 1)$
$\quad \texttt{min} := i;\ j := i + 1;$
$\quad \texttt{while } (j < n)$
$\quad\quad \texttt{if } (A[j] < A[\texttt{min}])\ \texttt{min} := j;$
$\quad\quad j := j + 1;$
$\quad \texttt{Assert}(i \neq \texttt{min});$
$\quad \texttt{if } (i \neq \texttt{min})\ \texttt{swap } A[i] \texttt{ and } A[\texttt{min}];$
$\quad i := i + 1;$

User Input:
Invariant Template: $v_0 \wedge (\forall k : v_1 \Rightarrow v_2) \wedge (\forall k : v_3 \Rightarrow v_4) \wedge (\forall k_1, k_2 : v_5 \Rightarrow v_6)$
Predicate Set: $\texttt{AllPreds}(\{k, k_1, k_2, i, j, \texttt{min}, n\}, \{0, 1\}, \{\leq, \geq, >\}) \ \cup$
$\quad \texttt{AllPreds}(\{A[t] \mid t \in \{k, k_1, k_2, i, j, \texttt{min}, n\}\}, \{0, 1\}, \{\leq, \geq\})$

Tool Output: Assertion valid under following precondition.
Precondition Required:
$\forall k : (0 \leq k < n - 1) \Rightarrow A[n-1] < A[k]$
$\forall k_1, k_2 : (0 \leq k_1 < k_2 < n - 1) \Rightarrow A[k_1] < A[k_2]$
Outer Loop Invariant:
$\forall k_1, k_2 : (i \leq k_1 < k_2 < n - 1) \Rightarrow A[k_1] < A[k_2]$
$\forall k : i \leq k < n - 1 \Rightarrow A[n-1] < A[k]$
Inner Loop Invariant:
$\forall k_1, k_2 : (i \leq k_1 < k_2 < n - 1) \Rightarrow A[k_1] < A[k_2]$
$\forall k : (i \leq k < n - 1) \Rightarrow A[n-1] < A[k]$
$j > i \wedge i < n - 1 \wedge \forall k : (i \leq k < j) \Rightarrow A[\texttt{min}] \leq A[k]$

Figure 1. (a) Verifying that Insertion Sort preserves all its input elements (b) Generating a precondition under which Selection Sort exhibits its worst-case number of swaps. (For any set of program variables Z, any constants C and any relational operators R, we use the notation $\texttt{AllPreds}(Z, C, R)$ to denote the set of predicates $\{z - z' \texttt{ op } c,\ \ z \texttt{ op } c \mid z, z' \in Z, c \in C, \texttt{op} \in R\}$.)

We therefore present three novel algorithms for efficiently discovering inductive loop invariants that prove the validity of assertions in a program, given a suitable set of invariant templates and a set of predicates. Two of these algorithms use standard iterative techniques for computing fixed-point as in data-flow analysis or abstract interpretation. One of them performs a forward propagation of facts and computes a least fixed-point, and then checks whether the facts discovered imply the assertion or not (Section 4.1). The other algorithm performs a backward propagation of facts starting from the given assertion and checks whether the precondition discovered is `true` or not (Section 4.2). The third algorithm uses a constraint-based approach to encode the fixed-point as a SAT formula such that a satisfying assignment to the SAT formula maps back to a proof of validity for the assertion (Section 5). The worst-case complexity of these algorithms is exponential only in the maximum number of unknowns at two neighboring points as opposed to being exponential in the total number of unknowns at all program points for the naive approach. Additionally, in practice we have found them to be efficient and having complementary strengths (Section 7).

The key operation in these algorithms is that of finding *optimal solutions* for unknowns in a template formula such that the formula is valid (Section 3). The unknowns take values that are conjunctions of some predicates from a given set of predicates, and can be classified as either positive or negative depending on whether replacing them by a stronger or weaker set of predicates makes the formula stronger or weaker respectively. We describe an efficient, systematic, search process for finding optimal solutions to these unknowns. Our search process uses the observation that a solution for a positive (or negative) unknown remains a solution upon addition (or deletion) of more predicates.

One of the key aspects of our algorithms is that they can be easily extended to discover *maximally-weak* preconditions[1] that ensure validity of given assertions. This is unlike most invariant generation tools that cannot be easily extended to generate (weak) preconditions. Automatic precondition generation not only reduces the annotation burden on the programmer in the usual case, but can also help identify preconditions that are not otherwise intuitive.

This paper makes the following contributions:

- We present a template based formalism to discover invariants with arbitrary (but pre-specified) logical structure over a given set of predicates.
- We present three novel fixed-point computation algorithms, each having different strengths, given a set of templates and predicates. Two iteratively propagate facts and one encodes the fixed-point as a constraint.
- We show how to generate maximally-weak preconditions for ensuring the validity of assertions that hold only under certain preconditions.
- We present preliminary experimental evidence that these algorithms can verify (using off-the-shelf SMT solvers) program properties not automatically verified before. They also take less time than previously reported for properties analyzed before by alternate techniques. We also compare the properties of our algorithms against each other.

1.1 Motivating Examples

Checking Validity of Assertions Consider, for example, the inplace `InsertionSort` routine in Figure 1(a) that sorts an array A of length n. The assertion at Line 9 asserts that no elements in array A are lost, i.e., the array A at the end of the procedure contains all elements from array $\tilde{A}$, where $\tilde{A}$ refers to the state of array A at the beginning of the procedure. The assertion as well as the loop invariants required to prove it are $\forall\exists$ quantified, and we do not know of any automated tool that can automatically discover such invariants for array programs.

In this case, the user can easily guess that the loop invariants would require a $\forall\exists$ structure to prove the assertion on Line 9. Ad-

[1] A precondition is maximally-weak if no other strictly weaker precondition exists in the template that also ensures the validity of the assertions.

ditionally, the user needs to guess that an inductive loop invariant may require a $\forall$ fact (to capture some fact about array elements) and a quantified-free fact relating non-array variables. The quantified facts contain an implication as is there in the final assertion. The user also needs to provide the set of predicates. In this case, the set consisting of inequality and disequality comparisons between terms (variables and array elements that are indexed by some variable) of appropriate types suffices. This choice of predicates is quite natural and has been used in several works based on predicate abstraction. Given these user inputs, our tool then automatically discovers the non-trivial loop invariants mentioned in the figure.

Our tool eases the task of validating the assertion by requiring the user to only provide a template in which the logical structure has been made explicit, and provide some over-approximation of the set of predicates. Guessing the template is a much easier task than providing the precise loop invariants, primarily because these templates are usually uniform across the program and depend on the kind of properties to be proved.

Precondition Generation Consider, for example, the in-place `SelectionSort` routine in Figure 1(b) that sorts an array A of length n. Using bound analysis [11], it is possible to prove that the worst-case number of array swaps for this example is $n-1$. On the other hand, suppose we want to verify that the worst-case number of swaps, $n-1$, can indeed be achieved by some input instance. This problem can be reduced to the problem of validating the assertion at Line 7. If the assertion holds then the swap on Line 8 is always executed, which is $n-1$ times. However, this assertion is not valid without an appropriate precondition (e.g., consider a fully sorted array when the swap happens in no iteration). We want to generate a precondition that does not impose any constraints on n while allowing the assertion to be valid—this would provide a proof that `SelectionSort` indeed admits a worst-case of $n-1$ array swaps.

In this case, the user can easily guess that a quantified fact ($\forall k_1, k_2$ that compares the elements at locations k_1 and k_2) capturing some sortedness property will be required. However, this alone is not sufficient. The user can then iteratively guess and add templates until a precondition is found. (The process can probably be automated.) Two additional quantified facts and an unquantified fact suffice in this case. The user also supplies a predicate set consisting of inequality and disequality comparisons between terms of appropriate type. The non-trivial output of our tool is shown in the figure.

Our tool automatically infers the maximally-weak precondition that the input array should be sorted from $A[0]$ to $A[n-2]$, while the last entry $A[n-1]$ contains the smallest element. Other sorting programs exhibit their worst-case behaviors usually when the array is reverse-sorted (For selection sort, a reverse sorted array is not the worst case; it incurs only $\frac{n}{2}$ swaps!). By automatically generating this non-intuitive maximally-weak precondition our tool provides a significant insight about the algorithm and reduces the programmer's burden.

2. Notation

We often use a set of predicates in place of a formula to mean the conjunction of the predicates in the set. In our examples, we often use predicates that are inequalities between a given set of variables or constants. For some subset V of the variables, we use the notation Q_V to denote the set of predicates $\{v_1 \leq v_2 \mid v_1, v_2 \in V\}$. Also, for any subset V of variables, and any variable j, we use the notation $Q_{j,V}$ to denote the set of predicates $\{j < v, j \leq v, j > v, j \geq v \mid v \in V\}$.

2.1 Templates

A *template* τ is a formula over unknown variables v_i that take values over (conjunctions of predicates in) some subset of a given set of predicates. We consider the following language of templates:

$$\tau \;::=\; v \;\mid\; \neg\tau \;\mid\; \tau_1 \vee \tau_2 \;\mid\; \tau_1 \wedge \tau_2 \;\mid\; \exists x : \tau \;\mid\; \forall x : \tau$$

We denote the set of unknown variables in a template τ by $\mathtt{U}(\tau)$. We say that an unknown $v \in \mathtt{U}(\tau)$ in template τ is a *positive (or negative) unknown* if τ is monotonically stronger (or weaker respectively) in v. More formally, let v be some unknown variable in $\mathtt{U}(\tau)$. Let σ_v be any substitution that maps all unknown variables v' in $\mathtt{U}(\tau)$ that are different from v to some set of predicates. Let $Q_1, Q_2 \subseteq Q(v)$. Then, v is a positive unknown if

$$\forall \sigma_v, Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \;\;\Rightarrow\;\; (\tau\sigma_v[v \mapsto Q_2] \Rightarrow \tau\sigma_v[v \mapsto Q_1])$$

Similarly, v is a negative unknown if

$$\forall \sigma_v, Q_1, Q_2 : (Q_1 \Rightarrow Q_2) \;\;\Rightarrow\;\; (\tau\sigma_v[v \mapsto Q_1] \Rightarrow \tau\sigma_v[v \mapsto Q_2])$$

We use the notation $\mathtt{U}^+(\tau)$ and $\mathtt{U}^-(\tau)$ to denote the set of all positive unknowns and negative unknowns respectively in τ.

If each unknown variable in a template/formula occurs only once, then it can be shown that each unknown is either positive or negative. In that case, the sets $\mathtt{U}^+(\tau)$ and $\mathtt{U}^-(\tau)$ can be computed using structural decomposition of τ as follows:

$$\begin{array}{rcl|rcl}
\mathtt{U}^+(v) & = & \{v\} & \mathtt{U}^-(v) & = & \emptyset \\
\mathtt{U}^+(\neg\tau) & = & \mathtt{U}^-(\tau) & \mathtt{U}^-(\neg\tau) & = & \mathtt{U}^+(\tau) \\
\mathtt{U}^+(\tau_1 \wedge \tau_2) & = & \mathtt{U}^+(\tau_1) \cup \mathtt{U}^+(\tau_2) & \mathtt{U}^-(\tau_1 \wedge \tau_2) & = & \mathtt{U}^-(\tau_1) \cup \mathtt{U}^-(\tau_2) \\
\mathtt{U}^+(\tau_1 \vee \tau_2) & = & \mathtt{U}^+(\tau_1) \cup \mathtt{U}^+(\tau_2) & \mathtt{U}^-(\tau_1 \vee \tau_2) & = & \mathtt{U}^-(\tau_1) \cup \mathtt{U}^-(\tau_2) \\
\mathtt{U}^+(\forall X : \tau) & = & \mathtt{U}^+(\tau) & \mathtt{U}^-(\forall X : \tau) & = & \mathtt{U}^-(\tau) \\
\mathtt{U}^+(\exists X : \tau) & = & \mathtt{U}^+(\tau) & \mathtt{U}^-(\exists X : \tau) & = & \mathtt{U}^-(\tau)
\end{array}$$

EXAMPLE 1. *Consider the following template τ with unknown variables $v_1, \ldots, v_5$.*

$$\begin{array}{l}
(v_1 \;\wedge\; (\forall j : v_2 \Rightarrow sel(A,j) \leq sel(B,j)) \;\wedge \\
\qquad (\forall j : v_3 \Rightarrow sel(B,j) \leq sel(C,j))) \;\;\Rightarrow \\
\qquad\qquad (v_4 \;\wedge\; (\forall j : v_5 \Rightarrow sel(A,j) \leq sel(C,j)))
\end{array}$$

Then, $\mathtt{U}^+(\tau) = \{v_2, v_3, v_4\}$ and $\mathtt{U}^-(\tau) = \{v_1, v_5\}$.

2.2 Program Model

We assume that a program `Prog` consists of the following kind of statements s (besides the control-flow).

$$s \;::=\; x := e \;\mid\; \mathtt{assert}(\phi) \;\mid\; \mathtt{assume}(\phi)$$

In the above, x denotes a variable and e denotes some expression. Memory reads and writes can be modeled using memory variables and select/update expressions. Since we allow `assume` statements, without loss of any generality, we can treat all conditionals in the program as non-deterministic.

We now set up a formalism in which different templates can be associated with different program points, and different unknowns in templates can take values from different sets of predicates. Let C be some cut-set of the program `Prog`. (A cut-set of a program is a set of program points, called cut-points, such that any cyclic path in `Prog` passes through some cut-point.) Every cut-point in C is labeled with an invariant template. For simplicity, we assume that C also consists of program entry and exit locations, which are labeled with an invariant template that is simply `true`. Let $\mathtt{Paths}(\mathtt{Prog})$ denote the set of all tuples $(\delta, \tau_1, \tau_2, \sigma_t)$, where δ is some straight-line path between two cut-points from C that are labeled with invariant templates τ_1 and τ_2 respectively. Without loss of any generality, we assume that each program path δ is in static single assignment (SSA) form, and the variables that are live at start of path δ are the original program variables, and the SSA versions of the variables that are live at the end of δ are given by the mapping σ_t, while σ_t^{-1} denotes the reverse mapping.

We use the notation $\mathtt{U}(\mathtt{Prog})$ to denote the set of unknown variables in the invariant templates at all cut-points of $\mathtt{Prog}$.

EXAMPLE 2. *Consider the following program* $\mathtt{ArrayInit}$ *(used as a running example) that initializes all array elements to* 0.

```
  ArrayInit(Array A, int n)
1   i := 0;
2   while (i < n)
3       A[i] := 0;
4       i := i + 1;
5   Assert(∀j : 0 ≤ j < n ⇒ sel(A, j) = 0);
```

Consider the cut-set C for program $\mathtt{ArrayInit}$ *that consists of only the program location 2 besides the entry and exit locations. Let the program location 2 be labeled with the invariant template $\forall j : v \Rightarrow \mathtt{sel}(A, j) = 0$, which has one negative unknown v. Then,* $\mathtt{Paths}(\mathtt{ArrayInit})$ *consists of the following tuples.*

Entry Case $(i := 0, \mathtt{true}, \forall j : v \Rightarrow \mathtt{sel}(A, j) = 0, \sigma_t)$*, where σ_t is the identity map.*

Exit Case $(\mathtt{assume}(i \geq n), \forall j : v \Rightarrow \mathtt{sel}(A, j) = 0, \forall j : 0 \leq j < n \Rightarrow \mathtt{sel}(A, j) = 0, \sigma_t)$*, where σ_t is the identity map.*

Inductive Case $(\mathtt{assume}(i < n); A' := \mathtt{upd}(A, i, 0); i' := i + 1, \forall j : v \Rightarrow \mathtt{sel}(A, j) = 0, \forall j : v \Rightarrow \mathtt{sel}(A', j) = 0, \sigma_t)$*, where $\sigma_t(i) = i', \sigma_t(A) = A'$.*

2.3 Invariant Solution

The *verification condition* of any straight-line path δ (a sequence of statements s) in SSA form between two program points labeled with invariant templates τ_1 and τ_2 is given by

$$\mathtt{VC}(\langle \tau_1, \delta, \tau_2 \rangle) \quad = \quad \tau_1 \Rightarrow \mathtt{WP}(\delta, \tau_2)$$

where the weakest precondition $\mathtt{WP}(\delta, \phi)$ of formula ϕ with respect to path δ is as follows:

$$\begin{array}{rcl} \mathtt{WP}(\mathtt{skip}, \phi) & = & \phi \\ \mathtt{WP}(s_1; s_2, \phi) & = & \mathtt{WP}(s_1, \mathtt{WP}(s_2, \phi)) \\ \mathtt{WP}(\mathtt{assert}(\phi'), \phi) & = & \phi' \wedge \phi \\ \mathtt{WP}(\mathtt{assume}(\phi'), \phi) & = & \phi' \Rightarrow \phi \\ \mathtt{WP}(x := e, \phi) & = & (x = e) \Rightarrow \phi \end{array} \tag{1}$$

Observe that the correctness of Eq. 1 in the definition of the weakest precondition above relies on the fact that the statements on path δ are in SSA form. (Note that it is important for the path δ to be in SSA form since otherwise we will have to address the issue of substitution in templates, as the only choice for $\mathtt{WP}(x := e, \phi)$ when the path δ is in non-SSA form would be $\phi[e/x]$.

DEFINITION 1 (Invariant Solution). *Let Q be a* predicate-map *that maps each unknown v in any template invariant in program* $\mathtt{Prog}$ *to some set of predicates $Q(v)$. Let σ map each unknown v in any template invariant in program* $\mathtt{Prog}$ *to some subset of $Q(v)$. We say that σ is an* invariant solution *for* $\mathtt{Prog}$ *over Q if the following formula* $\mathtt{VC}(\mathtt{Prog}, \sigma)$*, which denotes the verification condition of the program* $\mathtt{Prog}$ *w.r.t. σ, is valid.*

$$\mathtt{VC}(\mathtt{Prog}, \sigma) \stackrel{def}{=} \bigwedge_{(\delta, \tau_1, \tau_2, \sigma_t) \in \mathtt{Paths}(\mathtt{Prog})} \mathtt{VC}(\langle \tau_1 \sigma, \delta, \tau_2 \sigma \sigma_t \rangle)$$

EXAMPLE 3. *Consider the program* $\mathtt{ArrayInit}$ *described in Example 2. Let Q map unknown v in the invariant template at cut-point location 2 to $Q_{j,\{0,i,j\}}$. Let σ map v to $Q_0 = \{0 \leq j, j < i\}$. Then, σ is an invariant solution for* $\mathtt{ArrayInit}$ *over Q since the verification condition* $\mathtt{VC}(\mathtt{ArrayInit}, \sigma)$ *of the program* $\mathtt{ArrayInit}$*, which is given by the conjunction of the following formulas, is valid.*

- $i = 0 \;\Rightarrow\; (\forall j : Q_0 \Rightarrow \mathtt{sel}(A, j) = 0)$
- $(i \geq n \;\wedge\; (\forall j : Q_0 \Rightarrow \mathtt{sel}(A, j) = 0)) \;\Rightarrow\; (\forall j : 0 \leq j \leq n \Rightarrow \mathtt{sel}(A, j) = 0)$
- $(i < n \;\wedge\; A' = \mathtt{upd}(A, i, 0) \;\wedge\; i' = i + 1 \;\wedge\; (\forall j : Q_0 \Rightarrow \mathtt{sel}(A, j) = 0)) \;\Rightarrow\; (\forall j : Q_0 \sigma_t \Rightarrow \mathtt{sel}(A', j) = 0)$

where $\sigma_t(i) = i'$ and $\sigma_t(A) = A'$.

Sections 4 and 5 describe algorithms for generating an invariant solution given program $\mathtt{Prog}$ and an appropriate predicate-map Q.

3. Optimal Solutions

In this section, we present the core operation of generating an *optimal solution* that is used by our algorithm to perform local reasoning about program paths (which are encoded as formulae). Separating local reasoning from fixed-point computation (that we address later) is essential because it is not possible, in general, to encode a program with loops as a single SMT constraint.

DEFINITION 2 (Optimal Solution). *Let ϕ be a formula with unknowns $\{v_i\}_i$ where each v_i is either positive or negative. Let Q map each unknown v_i to some set of predicates $Q(v_i)$. A map $\{v_i \mapsto Q_i\}_i$ is a* solution *(for ϕ over domain Q) if the formula ϕ is valid after each v_i is replaced by Q_i, and $Q_i \subseteq Q(v_i)$. A solution $\{v_i \mapsto Q_i\}_i$ is* optimal *if replacing Q_i by a strictly weaker or stronger subset of predicates from $Q(v_i)$, depending on whether v_i is negative or positive, results in a map that is no longer a solution.*

EXAMPLE 4. *Consider the following formula ϕ with one negative unknown η.*

$$i = 0 \;\Rightarrow\; (\forall j : \eta \Rightarrow \mathtt{sel}(A, j) = 0)$$

Let $Q(\eta)$ be $Q_{j,\{0,i,n\}}$. There are four optimal solutions for ϕ over Q. These map the negative unknown variable η to $\{0 < j \leq i\}$, $\{0 \leq j < i\}$, $\{i < j \leq 0\}$, and $\{i \leq j < 0\}$ respectively.

Since the naive exponential search for optimal solutions to a formula would be too expensive, here we present a systematic search, that we found to be efficient in practice.

The procedure described in Figure 2 returns the set of all optimal solutions for an input formula ϕ over domain Q. It makes use of an operation $\mathtt{OptimalNegativeSolutions}(\phi, Q)$ (discussed later) that returns the set of all optimal solutions for the special case when ϕ consists of only negative unknowns. To understand how the procedure $\mathtt{OptimalSolutions}$ operates, it is illustrative to think of the simple case when there is only one positive variable ρ. In this case, the algorithm simply returns the conjunction of all those predicates $q \in Q(\rho)$ such that $\phi[\rho \mapsto \{q\}]$ is valid. Observe that such a solution is an optimal solution, and this procedure is much more efficient than naively trying out all possible subsets and picking the maximal ones.

EXAMPLE 5. *Consider the following formula ϕ with one positive unknown ρ.*

$$(i \geq n) \wedge (\forall j : \rho \Rightarrow \mathtt{sel}(A, j) = 0)) \;\Rightarrow\; (\forall j : 0 \leq j < n \Rightarrow \mathtt{sel}(A, j) = 0)$$

Let $Q(\rho)$ be $Q_{j,\{0,i,n\}}$. There is one optimal solution for ϕ over Q, namely

$$\rho \mapsto \{0 \leq j, j < n, j < i\}$$

OptimalSolutions(ϕ, Q)
1 Let $\mathtt{U}^{+}(\phi)$ be $\{\rho_1, .., \rho_a\}$.
2 Let $\mathtt{U}^{-}(\phi)$ be $\{\eta_1, .., \eta_b\}$.
3 $S := \emptyset$;
4 foreach $\langle q_1, .., q_a\rangle \in Q(\rho_1) \times .. \times Q(\rho_a)$:
5 $\quad \phi' := \phi[\rho_i \mapsto \{q_i\}]_i$;
6 $\quad T :=$ OptimalNegativeSolutions(ϕ', Q);
7 $\quad S := S \cup \{\sigma \mid \sigma(\rho_i) = \{q_i\}, \sigma(\eta_i) = t(\eta_i), t \in T\}$;
8 $R := \{\mathtt{MakeOptimal}(\sigma, S) \mid \sigma \in S\}$;
9 while any change in R:
10 $\quad$ foreach $\sigma_1, \sigma_2 \in R$
11 $\quad\quad \sigma :=$ Merge(σ_1, σ_2, S); if $(\sigma = \bot)$ continue;
12 $\quad\quad$ if $\not\exists \sigma' \in R : \bigwedge_{i=1}^{a} \sigma'(\rho_i) \Rightarrow \sigma(\rho_i) \wedge \bigwedge_{i=1}^{b} \sigma(\eta_i) \Rightarrow \sigma'(\eta_i)$
13 $\quad\quad\quad R := R \cup \{\mathtt{MakeOptimal}(\sigma, S)\}$;
14 return R;

MakeOptimal(σ, S)
$T := \{\sigma' \mid \sigma' \in S \wedge \bigwedge_{i=1}^{b} \sigma(\eta_i) \Rightarrow \sigma'(\eta_i)\}$
foreach $\sigma' \in T$:
$\quad \sigma'' :=$ Merge(σ, σ', S)
$\quad$ if $(\sigma'' \neq \bot)$ $\sigma := \sigma''$;
return σ

Merge(σ_1, σ_2, S)
Let σ be s.t. $\sigma(\rho_i) = \sigma_1(\rho_i) \cup \sigma_2(\rho_i)$ for i = 1 to a
$\quad$ and $\sigma(\eta_i) = \sigma_1(\eta_i) \cup \sigma_2(\eta_i)$ for i = 1 to b
$T := \{\sigma' \mid \sigma' \in S \wedge \bigwedge_{i=1}^{b} \sigma(\eta_i) \Rightarrow \sigma'(\eta_i)\}$
if $\bigwedge_{q_1 \in \sigma(\rho_1), .., q_a \in \sigma(\rho_a)} \exists \sigma' \in T$ s.t. $\bigwedge_{i=1}^{a} \sigma'(\rho_i) = \{q_i\}$ return σ
else return $\bot$

Figure 2. Procedure for generating optimal solutions given a template formula ϕ and a predicate-map Q.

This is computed by the algorithm in Figure 2 as follows. At the end of the first loop (Lines 4-7), the set S contains three solutions:

1: $\rho \mapsto \{0 \le j\}$ $\quad$ *2:* $\rho \mapsto \{j < n\}$ $\quad$ *3:* $\rho \mapsto \{j < i\}$

The set R at the end of line 8 contains only one optimal solution:

$$\rho \mapsto \{0 \le j, j < n, j < i\}$$

The set R is unchanged by second loop (Lines 9-13), simply because it contains only one optimal solution, while any change to R would require R to contain at least two optimal solutions.

Now, consider the case, of one positive and one negative variable. In this case, the algorithm invokes OptimalNegativeSolutions to find an optimal set of negative solutions for the negative variable η, for each choice of predicate $q \in Q(\rho)$ for the positive variable ρ and stores these solutions in set S (Lines 4-7). After this, it groups together all those solutions in S that match on the negative variable to generate a set R of optimal solutions (Line 8). (Recall, from Defn 2, that in an optimal solution a positive variable is mapped to a maximal set of predicates, while a negative variable is mapped to a minimal set.) It then attempts to generate more optimal solutions by merging the solutions for *both* the positive and negative variables of the optimal solutions in R (Lines 9-13).

EXAMPLE 6. *Consider the following formula ϕ with one positive unknown ρ and one negative unknown η.*

$$(\eta \wedge (i \ge n) \wedge (\forall j : \rho \Rightarrow \mathit{sel}(A, j) = 0)) \;\Rightarrow\; (\forall j : j \le m \Rightarrow \mathit{sel}(A, j) = 0)$$

Let $Q(\eta)$ and $Q(\rho)$ both be $Q_{\{i,j,n,m\}}$. There are three optimal solutions for ϕ over Q, namely

1: $\rho \mapsto \{j \le m\}$, $\eta \mapsto \emptyset$
2: $\rho \mapsto \{j \le n, j \le m, j \le i\}$, $\eta \mapsto \{m \le n\}$
3: $\rho \mapsto \{j \le i, j \le m\}$, $\eta \mapsto \{m \le i\}$

These are computed by the algorithm in Figure 2 as follows. At the end of the first loop (Lines 4-7), the set S contains the following four solutions:

1: $\rho \mapsto \{j \le m\}$, $\eta \mapsto \emptyset$
2: $\rho \mapsto \{j \le n\}$, $\eta \mapsto \{m \le n\}$
3: $\rho \mapsto \{j \le i\}$, $\eta \mapsto \{m \le i\}$
4: $\rho \mapsto \{j \le i\}$, $\eta \mapsto \{m \le n\}$

The set R at the end of line 8 contains the following three optimal solutions:

1: $\rho \mapsto \{j \le m\}$, $\eta \mapsto \emptyset$
2: $\rho \mapsto \{j \le n, j \le m, j \le i\}$, $\eta \mapsto \{m \le n\}$
3: $\rho \mapsto \{j \le i, j \le m\}$, $\eta \mapsto \{m \le i\}$

The set R is unchanged by second loop (Lines 9-13).

The extension to multiple positive variables involves considering a choice of all tuples of predicates of appropriate size (Line 4), while the extension to multiple negative variables is not very different.

OptimalNegativeSolutions This operation requires performing theory based reasoning over predicates, for which we use an SMT solver as a black box. Of several ways to implement OptimalNegativeSolutions, we found it effective to implement OptimalNegativeSolutions(ϕ, Q) as a breadth first search on the lattice of subsets ordered by implication with $\top$ and $\bot$ being $\emptyset$ and the set of all predicates, respectively. We start at $\top$ and keep deleting the subtree of every solution discovered until no more elements remain to be searched. Furthermore, to achieve efficiency, one can truncate the search at a certain depth. (We observed that the number of predicates mapped to a negative variable in any optimal solution in our experiments was never greater than 4.) To achieve completeness, the bounding depth can be increased iteratively after a failed attempt.

4. Iterative Propagation Based Algorithms

In this section, we present two iterative propagation based algorithms for discovering an inductive invariant that establishes the validity of assertions in a given program.

The key insight behind these algorithms is as follows. Observe that the set of elements that are instantiations of a given template with respect to a given set of predicates, ordered by implication, forms a pre-order, but not a lattice. Our algorithms performs a standard data-flow analysis over the powerset extension of this abstract domain (which forms a lattice) to ensure that it does not miss any solution. Experimental evidence shows that the number of elements in this powerset extension never gets beyond 6. Each step involves updating a fact at a cut-point by using the facts at the neighboring cut-points (preceding/succeeding cut-points in case of forward/backward data-flow respectively). The update is done by generating the verification condition that relates the facts at the neighboring cut-points with the template at the current cut-point, and updating using the solutions obtained from a call to OptimalSolutions.

The two algorithms differ in whether they perform a forward or backward dataflow and accordingly end up computing a least or greatest fixed point respectively, but they both have the following property.

THEOREM 1. *Given a program Prog and a predicate map Q, the algorithms in Figure 3 output an invariant solution, if there exists one.*

`LeastFixedPoint`$(\texttt{Prog}, Q)$

1 `Let` σ_0 `be s.t.` $\sigma_0(v) \mapsto \emptyset$, `if` v `is negative`
$\sigma_0(v) \mapsto Q(v)$, `if` v `is positive`
2 $S := \{\sigma_0\}$;
3 `while` $S \neq \emptyset \wedge \forall \sigma \in S : \neg\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$
4 `Choose` $\sigma \in S, (\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ `s.t.`
$\neg\texttt{Valid}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t\rangle))$
5 $S := S - \{\sigma\}$;
6 `Let` $\sigma_p = \sigma \mid_{\texttt{U(Prog)}-\texttt{U}(\tau_2)}$ `and` $\theta := \tau_2\sigma \Rightarrow \tau_2$.
7 $S := S \cup \{\sigma'\sigma_t^{-1} \cup \sigma_p \mid$
$\sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\rangle) \wedge \theta, Q\sigma_t)\}$
8 `if` $S = \emptyset$ `return ''No solution''`
9 `else return` $\sigma \in S$ `s.t. Valid(VC(Prog,`σ`))`

(a) Least Fixed Point Computation

`GreatestFixedPoint(Prog)`

1 `Let` σ_0 `be s.t.` $\sigma_0(v) \mapsto Q(v)$, `if` v `is negative`
$\sigma_0(v) \mapsto \emptyset$, `if` v `is positive`
2 $S := \{\sigma_0\}$;
3 `while` $S \neq \emptyset \wedge \forall \sigma \in S : \neg\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$
4 `Choose` $\sigma \in S, (\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ `s.t.`
$\neg\texttt{Valid}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t\rangle))$
5 $S := S - \{\sigma\}$;
6 `Let` $\sigma_p = \sigma \mid_{\texttt{U(Prog)}-\texttt{U}(\tau_1)}$ `and` $\theta := \tau_1 \Rightarrow \tau_1\sigma$.
7 $S := S \cup \{\sigma' \cup \sigma_p \mid$
$\sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1, \delta, \tau_2\sigma\sigma_t\rangle) \wedge \theta, Q)\}$
8 `if` $S = \emptyset$ `return ''No solution''`
9 `else return` $\sigma \in S$ `s.t. Valid(VC(Prog,`σ`))`

(b) Greatest Fixed Point Computation

Figure 3. Iterative Algorithms for generating an invariant solution given program `Prog` and predicate-map Q.

For notational convenience, we present the algorithms slightly differently. Each of these algorithms (described in Figure 3) involve maintaining a set of candidate-solutions at each step. A *candidate-solution* σ is a map of the unknowns v in all templates to some subset of $Q(v)$, where Q is the given predicate-map. The algorithms make progress by choosing a candidate-solution, and replacing it by a set of weaker or stronger candidate-solutions (depending on whether a forward/least fixed-point or backward/greatest fixed-point technique is used) using the operation `OptimalSolutions` defined in Section 3. The algorithms return an invariant solution whenever any candidate solution σ becomes one (i.e., $\texttt{Valid}(\texttt{VC}(\texttt{Prog}, \sigma))$), or fail when the set of candidate-solutions becomes empty. We next discuss the two specific variants along with an example.

4.1 Least Fixed-point

This algorithm (Figure 3(a)) starts with the singleton set containing the candidate solution that maps each negative unknown to the empty set (i.e., `true`) and each positive unknown to the set of all predicates. In each step, the algorithm chooses a σ, which is not an invariant solution. It must be the case that there exists $(\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ such that $\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t\rangle)$ is not valid. Furthermore, this is because $\tau_2\sigma$ is a too strong instantiation for τ_2. The algorithm replaces the candidate solution σ by the solutions $\{\sigma'\sigma_t^{-1} \cup \sigma_p \mid \sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\rangle) \wedge \theta, Q\sigma_t)\}$, where σ_p is the projection of the map σ onto the unknowns in the set $\texttt{U}(\texttt{Prog}) - \texttt{U}(\tau_2)$ and θ (defined as $\tau_2\sigma \Rightarrow \tau_2$) ensures that only stronger solutions are considered.

EXAMPLE 7. *Consider the* `ArrayInit` *program from Example 2. Let* $Q(v) = Q_{j,\{0,i,n\}}$. *In the first iteration of the while loop,* S *is initialized to* σ_0*, and in Line 4 there is only one triple in* $\texttt{Paths}(\texttt{ArrayInit})$ *whose corresponding verification condition is inconsistent, namely* $(i := 0, \textit{true}, \forall j : v \Rightarrow \texttt{sel}(A, j) = 0, \sigma_t)$*, where* σ_t *is the identity map. Line 7 results in a call to* `OptimalSolutions` *on the formula* $\phi = (i = 0) \Rightarrow (\forall j : v \Rightarrow \texttt{sel}(A, j) = 0)$*, the result of which has already been shown in Example 4. The set* S *now contains the following candidate solutions after the first iteration of the while loop.*

1: $v \mapsto \{0 < j \leq i\}$ *2:* $v \mapsto \{0 \leq j < i\}$
3: $v \mapsto \{i < j \leq 0\}$ *4:* $v \mapsto \{i \leq j < 0\}$

Of these, the candidate-solution $v \mapsto \{0 \leq j < i\}$ *is a valid solution and hence the while loop terminates after one iteration.*

4.2 Greatest Fixed-point

This algorithm (Figure 3(b)) starts with the singleton set containing the candidate solution that maps each positive unknown to the empty set (i.e., `true`) and each negative unknown to the set of all predicates. In each step, the algorithm chooses a σ, which is not an invariant solution. It must be the case that there exists $(\delta, \tau_1, \tau_2, \sigma_t) \in \texttt{Paths}(\texttt{Prog})$ such that $\texttt{VC}(\langle \tau_1\sigma, \delta, \tau_2\sigma\sigma_t\rangle)$ is not valid. Furthermore, this is because $\tau_1\sigma$ is a too weak instantiation for τ_1. The algorithm replaces the candidate solution σ by the solutions $\{\sigma' \cup \sigma_p \mid \sigma' \in \texttt{OptimalSolutions}(\texttt{VC}(\langle \tau_1, \delta, \tau_2\sigma\sigma_t\rangle) \wedge \theta, Q)\}$, where σ_p is the projection of the map σ onto the unknowns in the set $\texttt{U}(\texttt{Prog}) - \texttt{U}(\tau_1)$ and θ (defined as $\tau_1 \Rightarrow \tau_1\sigma$) ensures that only weaker solutions are considered.

EXAMPLE 8. *Consider the* `ArrayInit` *program from Example 2. Let* $Q(v) = Q_{j,\{0,i,n\}}$. *In the first iteration of the while loop,* S *is initialized to* σ_0*, and in Line 4 there is only one triple in* $\texttt{Paths}(\texttt{ArrayInit})$ *whose corresponding verification condition is inconsistent, namely* $(\texttt{assume}(i \geq n), \forall j : v \Rightarrow \texttt{sel}(A, j) = 0, \forall j : 0 \leq j < n \Rightarrow \texttt{sel}(A, j) = 0, \sigma_t)$*, where* σ_t *is the identity map. Line 7 results in a call to* `OptimalSolutions` *on the formula* $\phi = (i \geq n) \wedge (\forall j : v \Rightarrow \texttt{sel}(A, j) = 0) \Rightarrow (\forall j : 0 \leq j < n \Rightarrow \texttt{sel}(A, j) = 0)$*, whose output is shown in Example 5. This results in* S *containing only the following candidate-solution after the first iteration of the while loop.*

$$v \mapsto \{0 \leq j, j < n, j < i\}$$

The candidate-solution $v \mapsto \{0 \leq j, j < n, j < i\}$ *is a valid solution and hence the while loop terminates after one iteration.*

5. Constraint Based Algorithm

In this section, we show how to encode the verification condition of the program as a boolean formula such that a satisfying assignment to the boolean formula corresponds to an inductive invariant that establishes the validity of assertions in a given program.

For every unknown variable v and any predicate $q \in Q(v)$, we introduce a boolean variable b_q^v to denote whether or not the predicate q is present in the solution for v. We show how to encode the verification condition of the program `Prog` using a boolean formula $\psi_{\texttt{Prog}}$ over the boolean variables b_q^v. The boolean formula $\psi_{\texttt{Prog}}$ is constructed by making calls to the theorem proving interface `OptimalNegativeSolutions` and has the following property.

THEOREM 2. *The boolean formula* $\psi_{\texttt{Prog}}$ *(Eq. 2) is satisfiable iff there exists an invariant solution for program* `Prog` *over predicate-map* Q.

5.1 Notation

Given a mapping $\{v_i \mapsto Q_i\}_i$ (where $Q_i \subseteq Q(v_i)$), let $\texttt{BC}(\{v_i \mapsto Q_i\}_i)$ denote the boolean formula that constrains the unknown variable v_i to contain all predicates from Q_i.

$$\texttt{BC}(\{v_i \mapsto Q_i\}_i) = \bigwedge_{i, q \in Q_i} b_q^{v_i}$$

5.2 Boolean Constraint Encoding for Verification Condition

We first show how to generate the boolean constraint $\psi_{\delta,\tau_1,\tau_2}$ that encodes the verification condition corresponding to any tuple $(\delta,\tau_1,\tau_2,\sigma_t) \in \texttt{Paths}(\texttt{Prog})$. Let τ_2' be the template that is obtained from τ_2 as follows. If τ_2 is different from τ_1, then τ_2' is same as τ_2, otherwise τ_2' is obtained from τ_2 by renaming all the unknown variables to fresh unknown variables with `orig` denoting the reverse mapping that maps the fresh unknown variables back to the original. (We rename to ensure that each occurrence of an unknown variable in the formula $\texttt{VC}(\langle\tau_1,\delta,\tau_2'\rangle)$ is unique. Note that each occurrence of an unknown variable in the formula $\texttt{VC}(\langle\tau_1,\delta,\tau_2\rangle)$ is not unique when τ_1 and τ_2 refer to the same template, which is the case when the path δ goes around a loop).

A simple approach would be to use `OptimalSolutions` to compute all valid solutions for $\texttt{VC}(\langle\tau_1,\delta,\tau_2'\rangle)$ and encode their disjunction. But because both τ_1,τ_2' are uninstantiated unknowns, the number of optimal solutions explodes. We describe below an efficient construction that involves only invoking `OptimalNegativeSolutions` over formulae with a smaller number of unknowns (the negative) for a small choice of predicates for the positive variables.

Let $\rho_1,..,\rho_a$ be the set of positive variables and let $\eta_1,..,\eta_b$ be the set of negative variables in $\texttt{VC}(\langle\tau_1,\delta,\tau_2'\rangle)$. Consider any positive variable ρ_i and any $q_j \in Q'(\rho_i)$, where Q' is the map that maps an unknown v that occurs in τ_1 to $Q(v)$ and an unknown v that occurs in τ_2 to $Q(v)\sigma_t$. Consider the partial map σ_{ρ_i,q_j} that maps ρ_i to $\{q_j\}$ and ρ_k to $\emptyset$ for any $k \neq i$. Let $S^{\rho_i,q_j}_{\delta,\tau_1,\tau_2}$ be the set of optimal solutions returned after invoking the procedure `OptimalNegativeSolutions` on the formula $\texttt{VC}(\langle\tau_1,\delta,\tau_2'\rangle)\sigma_{\rho_i,q_j}$ as below:

$$S^{\rho_i,q_j}_{\delta,\tau_1,\tau_2} = \texttt{OptimalNegativeSolutions}(\texttt{VC}(\langle\tau_1,\delta,\tau_2'\rangle)\sigma_{\rho_i,q_j}, Q')$$

Similarly, let S_{δ,τ_1,τ_2} denote the set of optimal solutions returned after invoking the procedure `OptimalNegativeSolutions` on the formula $\texttt{VC}(\langle\tau_1,\delta,\tau_2'\rangle)\sigma$, where σ is the partial map that maps ρ_k to $\emptyset$ for all $1 \leq k \leq a$.

$$S_{\delta,\tau_1,\tau_2} = \texttt{OptimalNegativeSolutions}(\texttt{VC}(\langle\tau_1,\delta,\tau_2'\rangle)\sigma, Q')$$

The following Boolean formula $\psi_{\delta,\tau_1,\tau_2,\sigma_t}$ encodes the verification condition corresponding to $(\delta,\tau_1,\tau_2,\sigma_t)$.

$$\psi_{\delta,\tau_1,\tau_2,\sigma_t} = \left(\bigvee_{\{\eta_k\mapsto Q_k\}_k \in S_{\delta,\tau_1,\tau_2}} \texttt{BC}(\{\texttt{orig}(\eta_k)\mapsto Q_k\sigma_t^{-1}\}_k)\right) \wedge$$

$$\bigwedge_{\rho_i,q_j\in Q'(\rho_i)} \left(b^{\texttt{orig}(\rho_i)}_{q_j\sigma_t^{-1}} \Rightarrow \bigvee_{\{\eta_k\mapsto Q_k\}_k \in S^{\rho_i,q_j}_{\delta,\tau_1,\tau_2}} \texttt{BC}(\{\texttt{orig}(\eta_k)\mapsto Q_k\sigma_t^{-1}\}_k)\right)$$

The verification condition of the entire program is now given by the following boolean formula $\psi_{\texttt{Prog}}$ that is the conjunction of the verification condition of all tuples $(\delta,\tau_1,\tau_2,\sigma_t) \in \texttt{Paths}(\texttt{Prog})$.

$$\psi_{\texttt{Prog}} = \bigwedge_{(\delta,\tau_1,\tau_2,\sigma_t)\in\texttt{Paths(Prog)}} \psi_{\delta,\tau_1,\tau_2,\sigma_t} \quad (2)$$

EXAMPLE 9. *Consider the* `ArrayInit` *program from Example 2. Let $Q(v) = Q_{j,\{0,i,n\}}$. The above procedure leads to generation of the following constraints.*

Entry Case *The verification condition corresponding to this case contains one negative variable v and no positive variable. The set S_{δ,τ_1,τ_2} is same as the set S in Example 7, which contains 4 optimal solutions. The following boolean formula encodes this verification condition.*

$$(b^v_{0\leq j} \wedge b^v_{j<i}) \vee (b^v_{0<j} \wedge b^v_{j\leq i}) \vee (b^v_{i\leq j} \wedge b^v_{j<0}) \vee (b^v_{i<j} \wedge b^v_{j\leq 0}) \quad (3)$$

Exit Case *The verification condition corresponding to this case contains one positive variable v and no negative variable. We now consider the set $S^{v,q}_{\delta,\tau_1,\tau_2}$ for each $q \in Q(v)$. Let $P = \{0 \leq j, j < i, j \leq i, j < n, j \leq n\}$. If $v \in P$, the set $S^{v,q}_{\delta,\tau_1,\tau_2}$ contains the empty mapping (i.e., the resultant formula when v is replaced by q is valid). If $v \in Q(v) - P$, the set $S^{v,q}_{\delta,\tau_1,\tau_2}$ is the empty-set (i.e., the resultant formula when v is replaced by q is not valid). The following boolean formula encodes this verification condition.*

$$\bigwedge_{q\in P}(b^v_q \Rightarrow \mathit{true}) \wedge \bigwedge_{q\in Q(v)-P}(b^v_q \Rightarrow \mathit{false})$$

which is equivalent to the following formula

$$\neg b^v_{0<j} \wedge \neg b^v_{i<j} \wedge \neg b^v_{i\leq j} \wedge \neg b^v_{n<j} \wedge \neg b^v_{n\leq j} \wedge \neg b^v_{j<0} \wedge \neg b^v_{j\leq 0} \quad (4)$$

Inductive Case *The verification condition corresponding to this case contains one positive variable v and one negative variable v' obtained by renaming one of the occurrences of v. Note that S_{δ,τ_1,τ_2} contains a singleton mapping that maps v' to the empty-set. Also, note that $S^{v,j\leq i}_{\delta,\tau_1,\tau_2}$ is the empty-set, and for any $q \in Q(v') - \{j \leq i\}$, $S^{v,q}_{\delta,\tau_1,\tau_2}$ contains at least one mapping that maps v' to the singleton $\{q\sigma_t\}$. Hence, the following boolean formula encodes this verification condition.*

$$(b^v_{j\leq i} \Rightarrow \mathit{false}) \wedge \bigwedge_{q\in Q(v')-\{j\leq i\}} \left(b^v_q \Rightarrow (b^v_q \vee \ldots)\right)$$

which is equivalent to the formula

$$\neg b^v_{j\leq i} \quad (5)$$

The boolean assignment where $b^v_{0\leq j}$ and $b^v_{j<i}$ are set to `true`*, and all other boolean variables are set to* `false` *satisfies the conjunction of the boolean constraints in Eq. 3,4, and 5. This implies the solution $\{0 \leq j, j < i\}$ for the unknown v in the invariant template.*

6. Maximally-Weak Precondition Inference

In this section, we address the problem of discovering *maximally-weak* preconditions that fit a given template and ensure that all assertions in a program are valid.

DEFINITION 3. (Maximally-Weak Precondition) *Given a program* `Prog` *with assertions, invariant templates at each cutpoint, and a template τ_e at the program entry, a solution σ for the unknowns in the templates assigns a maximally-weak precondition to τ_e if*

- *σ is a valid solution, i.e. $\mathtt{Valid}(\mathit{VC}(\mathtt{Prog},\sigma))$.*
- *For any solution σ', it is not the case that $\tau_e\sigma'$ is strictly weaker than $\tau_e\sigma$, i.e., $\forall\sigma' : (\tau_e\sigma \Rightarrow \tau_e\sigma' \wedge \tau_e\sigma' \not\Rightarrow \tau_e\sigma) \Rightarrow \neg\mathtt{Valid}(\mathit{VC}(\mathtt{Prog},\sigma'))$.*

The greatest fixed-point algorithm described in Section 4.2 already computes one such maximally-weak precondition. If we change the condition of the while loop in line 3 in Figure 3(b) to $S \neq \emptyset \wedge \exists\sigma \in S : \neg\texttt{Valid}(\texttt{VC}(\texttt{Prog},\sigma'))$, then S at the end of the loop contains all maximally-weak preconditions.

The constraint based algorithm described in Section 5 is extended to generate maximally-weak preconditions using an iterative process by first generating any precondition, and then encoding an additional constraint that the precondition should be strictly weaker than the precondition that was last generated, until any such precondition can be found. The process is repeated to generate other maximally-weak preconditions by encoding an additional constraint that the precondition should not be stronger than any previously found maximally-weak precondition. See [30] for details.

A dual notion can be defined for *maximally-strong* postconditions, which is motivated by the need to discover invariants as op-

Benchmark	Assertion proved
Merge Sort (inner)	$\forall y \exists x : 0 \leq y < m \Rightarrow A[y] = C[x] \wedge 0 \leq x < t$ $\forall y \exists x : 0 \leq y < n \Rightarrow B[y] = C[x] \wedge 0 \leq x < t$
Other Sorting	$\forall y \exists x : 0 \leq y < n \Rightarrow \tilde{A}[y] = A[x] \wedge 0 \leq x < n$

Table 1. The assertions proved for verifying that sorting programs preserve the elements of the input. $\tilde{A}$ is the array A at the input.

Benchmark	Precondition inferred
Selection Sort	$\forall k : 0 \leq k < n-1 \Rightarrow A[n-1] < A[k]$ $\forall k_1, k_2 : 0 \leq k_1 < k_2 < n-1 \Rightarrow A[k_1] < A[k_2]$
Insertion Sort	$\forall k : 0 \leq k < n-1 \Rightarrow A[k] > A[k+1]$
Bubble Sort (flag)	$\forall k : 0 \leq k < n-1 \Rightarrow A[k] > A[k+1]$
Quick Sort (inner)	$\forall k_1, k_2 : 0 \leq k_1 < k_2 \leq n \Rightarrow A[k_1] \leq A[k_2]$

Table 2. The preconditions inferred by our algorithms resulting in the worst case upper bounds runs for sorting programs.

Benchmark	Preconditions inferred under given postcondition
Partial Init	*pre:* (a) $m \leq n$ (b) $\forall k : n \leq k < m \Rightarrow A[k] = 0$ *post:* $\forall k : 0 \leq k < m \Rightarrow A[k] = 0$
Init Synthesis	*pre:* (a) $i = 1 \wedge max = 0$ (b) $i = 0$ *post:* $\forall k : 0 \leq k < n \Rightarrow A[max] \geq A[k]$
Binary Search	*pre:* $\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]$ *post:* $\forall k : 0 \leq k < n \Rightarrow A[k] \neq e$
Merge	*pre:* $\forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1]$ $\forall k : 0 \leq k < m \Rightarrow B[k] \leq B[k+1]$ *post:* $\forall k : 0 \leq k < t \Rightarrow C[k] \leq C[k+1]$

Table 3. Given a functional specification (post), the maximally-weak preconditions (pre) inferred by our algorithms for functional correctness. The code for these examples is listed in Figure 10.

posed to verifying given assertions. It can be shown that the least fixed-point based algorithm in Section 4.1 already computes one such maximally-strong solution. Furthermore, the constraint based algorithm can be extended to generate maximally-strong postconditions. See [30] for details.

7. Evaluation

We have built a prototype implementation using the Phoenix compiler framework [24] as the frontend parser for ANSI-C programs and Z3 [25, 7] as our SMT solver. Our implementation is approximately 15K lines of non-blank, non-comment, C# code.

Since quantification makes reasoning undecidable, SMT solvers require additional help in the presence of quantified facts. We have developed a wrapper interface that automatically constructs patterns for quantifier instantiation (used for E-matching [6]) and introduces explicit skolemization functions. Also, to support linked lists, we augment Z3's support for select/update with axioms for reachability. See [31, 30] for details.

We evaluated the performance of our algorithms, using a 2.5GHz Intel Core 2 Duo machine with 4GB of memory.

7.1 Templates and Predicates

Our tool takes as input a program and a global set of templates and predicates. The global template is associated with each loop header (cut-point) and the global set of predicates with each unknown in the templates. We use a global set to reduce annotation burden but possibly at the cost of efficiency. For each benchmark program, we supplied the tool with a set of templates, whose structure is very similar to the program assertions (usually containing one unquantified unknown and a few quantified unknowns, as in Figure 1) and a set of predicates consisting of inequality relations between relevant program and bound variables.

7.2 Verifying standard benchmarks

Simple array/list manipulation: We present the performance of our algorithms on simple but difficult programs manipulating arrays and lists that have been previously considered by alternative techniques. By adding axiomatic support for reachability, we were able to verify simple list programs illustrating our extensibility. Table 4 presents the benchmark examples, the time in seconds taken by each of our algorithm (least fixed-point, greatest fixed-point and constraint-based) and the time reported by previous techniques[2].

For the appropriately named Consumer Producer [17], we verify that only values produced are consumed. For the appropriately named Partition Array [2, 17], we verify that the output arrays are partitions of the input. For List Init [12], we verify that the output list is initialized and for List Insert/Delete [12] that they maintain the initialization.

Sortedness property: We choose sorting for our benchmark comparisons because these are some of the hardest verification instances for array programs and have been attempted by previous techniques. We verify sortedness for all major sorting procedures.

Table 6, columns 1–5, presents the benchmark examples, the time taken in seconds by our algorithms (least fixed-point, greatest fixed-point and constraint-based) to verify that they indeed output a sorted array and previously reported timings[2]. We evaluate over selection, insertion and bubble sort (one that iterates n^2 times irrespective of array contents, and one that maintains a flag checking if the array is already sorted). For quick sort and merge sort we consider their partitioning and merge steps, respectively.

We do not know of a single technique that can uniformly verify all sorting benchmarks as is possible here. In fact, the missing results indicate that previous techniques are not robust and are specialized to the reasoning required for particular programs. In contrast, our tool successfully verified all programs that we attempted. Also, on time, we outperform the current state-of-the-art.

7.3 Proving $\forall\exists$, worst-case bounds and functional correctness

We now present analyses for which no previous techniques are known. We handle three new analyses: $\forall\exists$ properties for verifying that sorting programs *preserve* elements, maximally-weak preconditions for *worst case upper bounds* and *functional correctness*.

$\forall\exists$ ***properties:*** We prove that the sorting programs do not lose any elements of the input[3]. The proof requires discovering $\forall\exists$ invariants (Table 1). The runtimes are shown in Table 6, columns 6–8. Except for two runs that timeout, all three algorithms efficiently verify all instances.

[2] We warn the reader that the numbers for previous techniques are potentially incomparable because of the differences in experimental setups and because some techniques infer predicates, possibly using templates.

[3] Similar $\forall\exists$ invariants can be used to prove that no elements are gained. Together, these invariants prove that the output array is a permutation of the input for the case when the elements in the input array are distinct.

Benchmark	LFP	GFP	CFP	Previous
Consumer Producer	0.45	2.27	4.54	45.00 [17]
Partition Array	2.28	0.15	0.76	7.96 [17], 2.4 [2]
List Init	0.15	0.06	0.15	24.5 [12]
List Delete	0.10	0.03	0.19	20.5 [12]
List Insert	0.12	0.30	0.25	23.9 [12]

Table 4. Time (secs) for verification of data-sensitive array/list programs.

Benchmark	GFP
Partial Init	0.50
Init Synthesis	0.72
Binary Search	13.48
Merge	3.37

Table 5. Time (secs) for preconditions for functional correctness.

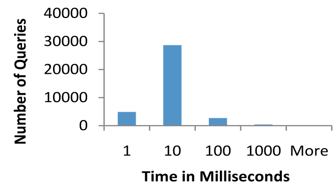

Figure 4. Most SMT queries take less than 10ms.

	Sortedness				∀∃			Upper
Benchmark	LFP	GFP	CFP	Previous	LFP	GFP	CFP	Bound
Selection Sort	1.32	6.79	12.66	na[4]	22.69	17.02	timeout	16.62
Insertion Sort	14.16	2.90	6.82	5.38 [15][4]	2.62	94.42	19.66	39.59
Bubble Sort (n^2)	0.47	0.78	1.21	na	5.49	1.10	13.74	0.00
Bubble Sort (flag)	0.22	0.16	0.55	na	1.98	1.56	10.44	9.04
Quick Sort (inner)	0.43	4.28	1.10	42.2 [12]	1.89	4.36	1.83	1.68
Merge Sort (inner)	2.91	2.19	4.92	334.1 [12]	timeout	7.00	23.75	0.00

Table 6. Time (secs) for sorting programs. We verify sortedness, preservation (∀∃) and infer preconditions for the worst case upper bounds.

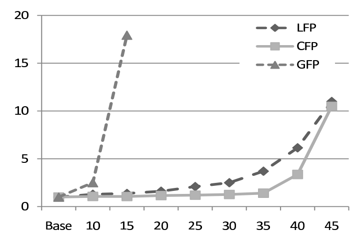

Figure 5. LFP and CFP remain relatively robust to irrelevant predicates.

Worst case upper bounds: We have already seen that the worst case input for Selection Sort involves a non-trivial precondition that ensures that a swap occurs every time it is possible (line 7 of Figure 1). For Insertion Sort we assert that the copy operation in the inner loop is always executed. For the termination checking version of Bubble Sort we assert that after the inner loop concludes the swapped flag is always set. For the partitioning procedure in Quick Sort (that deterministically chooses the leftmost element as the pivot), we assert that the pivot ends up at the rightmost location. All of these assertions ensure the respective worst case runs occur.

We generate the maximally-weak preconditions for each of the sorting examples as shown in Table 2. Notice that the inner loop of merge sort and the n^2 version of bubble sort always perform the same number of writes and therefore no assertions are present and the precondition is `true`. The time taken is shown in Table 6, column 9, and is reasonable for all instances.

Functional correctness: Often, procedures expect conditions to hold on the input for functional correctness. These can be met by initialization, or by just assuming facts at entry. We consider the synthesis of the maximally-weakest such conditions. Table 3 lists our benchmarks and the interesting non-trivial[5] preconditions (*pre*) we compute under the functional specification (*post*) supplied as postconditions. Table 5 lists the time taken to compute the preconditions.

Partial Init initializes the locations $0 \ldots n$ while the functional specification expects initialization from $0 \ldots m$. Our algorithms, interestingly, generate two alternative preconditions, one that makes the specification expect less, while the other expects locations outside the range to be pre-initialized. Init Synthesis computes the index of the maximum array value. Restricting to equality predicates we compute two orthogonal preconditions[6] that correspond to the missing initializers. Binary Search is the standard binary search for the element e with the correctness specification that if the element was not found in the array, then the array does not contain the element. We generate the intuitive precondition that the input array must have been sorted. Merge Sort (inner) outputs a sorted array. We infer that the input arrays must have been sorted for the procedure to be functionally correct.

7.4 Properties of our algorithms

Statistical properties: We statistically examined the practical behavior our algorithms to explain why they work well despite the theoretical bottlenecks. We accumulated the statistics over all analyses and for all relevant modes (iterative and constraint-based).

First, we measured if the SMT queries generated by our system were efficiently decidable. Figure 4 shows that almost all of our queries take less than 10ms. By separating fixed-point computation from reasoning about local verification conditions, we have brought the theorem proving burden down to the realm of current solvers.

Second, because our algorithms rely on `OptimalNegativeSolutions` and `OptimalSolutions`, it is therefore important that in practice they return a small number of optimal solutions. In fact, we found that on most calls they return a single optimal solution (Figure 6 and 7) and never more than 6. Therefore there are indeed a small number of possibilities to consider when they are called (in Figures 2 and 3 and in the constraint encoding). This explains the efficiency of our local reasoning in computing the best abstract transformer.

Third, we examine the efficiency of the fixed-point computation (iterative) or encoding (constraint-based) built from the core procedures. For the iterative approaches, we reached a fixed-point in a median of 4 steps with the number of candidates remaining small, at around 8 (Figure 8). This indicates that our algorithms perform a very directed search for the fixed-point. For the constraint-based approach, the number of clauses in the SAT formula never exceeds 500 (Figure 9) with a median size of 5 variables. This explains the efficiency of our fixed-point computation.

[4] [12] and [17] present timing numbers for the *inner loops* that are incomparable to the numbers for the entire sorting procedure that we report here. For the inner loops of selection sort and insertion sort, our algorithms run in time 0.34(LFP), 0.16(GFP), 0.37(CFP) for selection sort compared to 59.2 [12] and in time 0.51(LFP), 1.96(GFP), 1.04(CFP) for insertion sort compared to 35.9 [12] and 91.22 [17].

[5] We omit other non-interesting trivial preconditions for lack of space.

[6] Notice that the second precondition is indeed the maximally-weakest for the specification, even though max could be initialized out of bounds. If we expected to strictly output an array index and not just the location of the maximum, then the specification should have contained, $0 \leq max < n$.

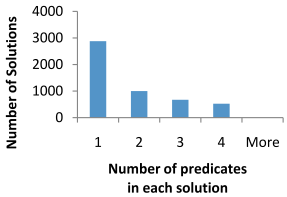

Figure 6. `OptimalNegativeSolutions` mostly returns solutions with one predicate.

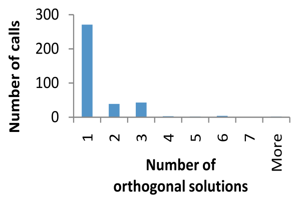

Figure 7. `OptimalSolutions` mostly returns a single solution.

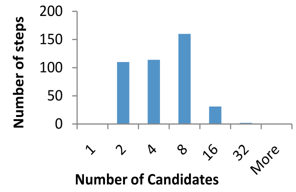

Figure 8. The number of candidates in iterative schemes remains mostly below 8.

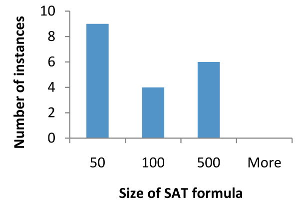

Figure 9. The number of clauses in the SAT formulae are always less than 500.

Robustness: Our algorithms use a global set of user specified predicates. We evaluated the robustness of our algorithms over the sortedness analysis by adding irrelevant predicates. Figure 5 shows how the performance degrades, as a factor of the base performance and averaged over all sorting examples, as irrelevant predicates are introduced. The constraint-based approach is much more robust than the iterative schemes and, remarkably, only shows degradation past 35 irrelevant predicates. On the other hand, greatest fixed-point cannot handle more than 15 and least fixed-point shows steady decrease in performance.

7.5 Discussion

Our benchmark programs pose a spectrum of analysis challenges. The experiments corroborate the intuition that a universal panacea capable of addressing all these challenges probably does not exist. No single technique (forward or backward iterative or bi-directional constraint-based) addresses all the challenges, but between them they cover the space of reasoning required. Therefore in practice, a combination will probably be required for handling real world instances.

We have also identified the different strengths that each algorithm demonstrates in practice. We found that for maximally-weak precondition inference, the iterative greatest fixed-point approach is more efficient than the constraint-based approach. In a similar setting of computing maximally-strong postcondition, the iterative least fixed-point is expected to be more efficient, as is indicated by its performance in our experiments. A constraint-based encoding is not suitable in an unconstrained problem where the number of possibilities grows uncontrollably. On the other hand, when the system is sufficiently constrained, for example when verifying sortedness or preservation, the constraint-based approach is significantly more robust to irrelevant predicates, followed by least fixed-point and lastly greatest fixed-point.

8. Related Work

Template-based analyses The template-based approach used in this work is motivated by recent work on using templates to discover precise program properties, such as numerical invariants [26, 27, 4, 18, 1, 13], quantifier-free invariants over predicate abstraction [14], and universally quantified invariants (over arrays) [12]. All these techniques differ in expressivity of the templates, as well as the algorithm and underlying technology used to solve for the unknowns in the templates. In terms of expressivity, our templates, which are based on logical structure over predicate abstraction, are more precise than the quantifier-free templates of [14], and orthogonal to the templates used in any other approach.

All techniques, with the exception of [12], employ a *constraint-based approach* to encode fixed point, reducing invariant generation to the task of solving a constraint. In particular, [14] uses the notion of predicate cover of a quantifier-free formula to reduce the problem to SAT solving, while the remaining techniques use Farkas' lemma to reduce the problem to solving non-linear constraints, which are then solved by either SAT solvers after bit-blasting [13] or by using specialized non-linear solvers [26, 27, 4, 18, 1]. On the other hand, [12] uses an iterative least-fixed point approach; however it requires non-standard under-approximation analyses. In contrast, we present both iterative and constraint-based algorithms built on the power of SMT solvers, and preliminary experimental results indicate that each has its own strengths.

Predicate Abstraction The form of our templates (in which unknowns range over conjunctions of predicates as opposed to numerical templates in which unknowns range over constant coefficients) is motivated by recent advances in using predicate abstraction to express and discover disjunctive properties of programs. The important body of work [10, 9, 21, 17, 22] leading up to our work has been discussed earlier (Section 1). In that dimension, we extend the field to discovering invariants that involve an arbitrary (but pre-specified) quantified structure over a given set of predicates. Another significant difference is that our technique (the iterative greatest fixed-point version, which works in a backward direction) generate maximally-weak preconditions, while the other predicate abstraction techniques that we know of, with the exception of [14], do not generate preconditions, primarily because most of them work in the forward direction. [14] presents a constraint based approach to generating preconditions for quantifier-free templates. In contrast, our quantified templates are not only more expressive, but our experimental results show that the constraint-based approach does not lend itself well to generating preconditions because of too many choices that become possible in an unconstrained system.

We do not consider the orthogonal problem of computing the right set of predicates (e.g., [3, 16]) and leave the interesting avenue of combining our work with predicate discovery for future work.

Sketching In the domain of program synthesis, combinatorial search based algorithms [29, 28] are distantly related. They also use templates, but for program statements. It will be interesting to apply the ideas presented here to template based program synthesis.

Others [23] describes how to use decision procedures to compute best abstract transformers over domains other than predicate-abstraction domains. Our iterative algorithms accomplish this for arbitrary logical structure over predicate abstraction.

Kovács and Voronkov [20] describe a technique for generating invariants with quantifier alternations using a saturation-based theorem prover. Their technique relies on an underlying procedure for generating invariants over scalar loop variables and an instrumented loop counter. Any skolemizations functions are removed by the introduction of quantifier alternations. They discover quantified invariants for a couple of examples, e.g., array partitioning and initialization, but the completeness of the approach is unclear and it is unclear whether it can be adapted to prove given assertions as opposed to generating arbitrary invariants.

```
Partial Init(Array A, int m)
  i := 0;
  while (i < n)
      A[i] := 0;
      i++;
  Assert(∀k : 0 ≤ k < m ⇒
                          A[k] = 0)
```

```
Init Synthesis(Array A, int n)
  while (i < n)
      if (A[max] < A[i])
          max := i;
      i++;
  Assert(∀k : 0 ≤ k < n ⇒
                  A[max] ≥ A[k])
```

```
Binary Search(Array A, int n)
  low := 0; high := n − 1;
  while (low ≤ high
      Assume(low ≤ mid ≤ high)
      if (A[mid] < e)
          low := mid + 1;
      else if (A[mid] > e)
          high := mid − 1;
      else return;
  Assert(∀k : 0 ≤ k < n : A[k] ≠ e)
```

```
Merge(Array A, int n)
  i := j := t := 0;
  while (i < n ∧ j < m)
      if (A[i] ≤ B[j])
          C[t++] := A[i++];
      else
          C[t++] := B[j++];
  while (i < n)
      C[t++] := A[i++];
  while (j < m)
      C[t++] := B[j++];
  Assert(∀k : 0 ≤ k < t − 1 ⇒
                  C[k] ≤ C[k + 1])
```

Figure 10. Benchmark examples for weakest preconditions for functional correctness.

9. Conclusions and Future Work

In this paper, we address the problem of inferring expressive program invariants over predicate abstraction for verification and also for inferring maximally-weak preconditions. We present the first technique that infers $\forall/\forall\exists$-quantified invariants for proving the full functional correctness of all major sorting algorithms. Additionally, we present the first technique that infers maximally-weak preconditions for worst case upper bounds and functional correctness.

We present three fixed-point computing algorithms (two iterative and one constraint-based) that use a common interface to SMT solvers to construct invariants as instantiations of templates with arbitrary quantification and boolean structure. Our algorithms compute greatest and least fixed-point solutions that induce maximally-weak precondition and maximally-strong postcondition analyses.

We have implemented our algorithms in a tool that uses off-the-shelf SMT solvers. Our tool uniformly and efficiently verifies (sortedness and preservation) properties of all major sorting algorithms and we have also used it for establishing worst case bounds and maximally-weak preconditions for functional correctness. We are unaware of any other technique that performs these analyses.

Today, SMT solvers support a variety of theories, and we have verified simple linked list programs. Next, we intend to use our algorithms to verify full functional correctness of list/tree and other data structure operations (e.g. insertion in AVL/Red-Black trees). Future work includes integrating our algorithms with predicate-discovery techniques, and extending ideas to program synthesis.

A. Code listing for example benchmarks

Figure 10 shows the benchmark examples for which we generate the weakest preconditions (Table 3) for functional correctness.

Acknowledgments

The authors would like to thank Bor-Yuh Evan Chang, Jeff Foster, Aswin Sankaranarayanan and the other anonymous reviewers for their insightful comments and suggestions for improvements to earlier versions of the paper. Also, we greatly appreciate the continued support by the Z3 team at Microsoft Research, specifically Nikolaj Bjørner and Leonardo de Moura, for their help in interfacing with their solver.

References

[1] Dirk Beyer, Thomas Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394, 2007.

[2] Dirk Beyer, Tom Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[3] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[4] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

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

[6] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for smt solvers. In *CADE*, pages 183–198, 2007.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: Efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, April 2008.

[8] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[9] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[10] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, pages 72–83, 1997.

[11] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.

[12] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.

[13] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.

[14] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, pages 120–135, 2009.

[15] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.

[16] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[17] Ranjit Jhala and Ken McMillan. Array abstraction from proofs. In *CAV*, 2007.

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

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

[20] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, 2009.

[21] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. on Computational Logic*, 9(1), 2007.

[22] Andreas Podelski and Thomas Wies. Boolean heaps. In *SAS*, 2005.

[23] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic impl. of the best transformer. In *VMCAI*, pages 252–266, 2004.

[24] Microsoft Research. Phoenix. `http://research.microsoft.com/Phoenix/`.

[25] Microsoft Research. Z3. `http://research.microsoft.com/projects/Z3/`.

[26] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.

[27] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In *SAS*, pages 53–68, 2004.

[28] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, June 2007.

[29] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit A. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, Oct 2006.

[30] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. Technical Report MSR-TR-2008-173, Nov 2008.

[31] Saurabh Srivastava, Sumit Gulwani, and Jeffrey Foster. VS^3: SMT-solvers for program verification. In *CAV*, 2009.

[32] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.