Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain

K. Rustan M. Leino and Francesco Logozzo

Microsoft Research, Redmond, WA, USA
{leino,logozzo}@microsoft.com

Abstract. This paper presents a new technique for combining the inference power of abstract interpretation with the precision and flexibility of an automatic satisfiability-modulo-theories theorem prover.

0 Introduction

Reasoning about the correctness of programs is hard. The hard parts include a program's use of loops and recursion, whose correctness relies on invariants. In this paper, we present a technique that combines the automatic discovery of invariants and the precise reasoning about programs using those invariants. More precisely, we build in a widening operator into a satisfiability-modulo-theories (SMT) solver, which allows the SMT solver to apply the widening operator when the verification process discovers the need to further approximate loop invariants. An important consequence of this design is that it lets the SMT solver and the abstract-interpretation machinery that typically surrounds the widening operator to share in a common implementation. We also show how the basic technique can be extended to infer invariants in a goal-directed way.

To put this into more context, let us consider some common approaches to program verification.

0.0 Some Approaches to Program Verification

One approach to program verification is *verification-condition generation*. It encodes a program's proof obligations as logical formulas, *verification conditions* (VCs), which are then passed to a mechanical theorem prover. If the prover can establish the validity of the formulas, then the program has been shown to be correct. For many verification tasks, an SMT solver makes a good choice as the theorem prover, because it is automatic (*i.e.*, it works without user interaction), supports common theories (*e.g.*, arithmetic, functions and equality, plus support of quantifiers), and, in the event that it cannot establish the validity of the formula, outputs counterexamples that can be turned into error messages about the program (*cf.* [10]). In its classical form, verification-condition generation assumes the program to contain assertional specifications, in particular preconditions, postconditions, and loop invariants.

Another approach to program verification is *abstract interpretation* [5], a general static-analysis technique that can infer invariants of a program and be applied to verify properties of the program. In the classical setting, verification through abstract interpretation works in two phases. In the first phase, the program is analyzed to infer invariants. In the second phase, those invariants are propagated to all program points and it is checked that the proof obligations at each program point are never violated. Analogous to the way an SMT solver uses various theories, an abstract interpreter uses *abstract domains*, each of which supports a *widening operator* that abstracts over details in order to compute invariants as post-fix points in the abstract-domain lattice.

Advantages of the VC generation approach include the precision offered by the theorem prover, and the beautiful Nelson-Oppen way of combining decision procedures for theories [19]. Another advantage is the flexibility offered by quantifiers when encoding verification conditions: these let a verification tool introduce function symbols and give a (possibly partial) axiomatization of them, without necessitating any changes to the theorem prover. A drawback of the approach is that it does not automatically infer the invariants that are needed to make the proofs go through.

Advantages of the abstract interpretation approach include a uniform view of inference and checking, and control over how to represent and approximate the properties of interest. Another advantage is that it presents a generalized setting for talking about static approximation (*i.e.*, the abstract domain) and dynamic approximation (*e.g.*, the widening operator). A drawback of the approach is that its implementation is not straightforward: it needs the design of an abstract domain, transfer functions that consider the effect of program statements on elements of the abstract domain, and convergence operators like widening. In addition, deciding on the direction of the analysis (forward, backward, or some combination thereof) impacts precision, and the forming of initial abstract-domain elements (a process called *seeding*) during an abstract interpretation usually has to be done in an *ad hoc* manner. Finally, there is no "user-programmable abstract domain"; for example, in the abstract interpretation literature, quantifiers have been considered just for restricted classes of problems, *e.g.*, [20, 8].

A hybrid approach to program verification combines the two approaches above, integrating an abstract interpreter with a theorem prover. For instance, one can apply the first phase of an abstract interpreter to infer loop invariants, instrument the program with the inferred invariants, and then continue as in the classical VC generation approach [1]. The major drawback of this hybrid approach is the "duplication of knowledge": despite the fact that the theorem prover and the abstract interpreter share many algorithms (for instance, the closure/decision procedures for linear arithmetic), most of the code and "knowledge" must be duplicated in (i) the abstract domains and the theorem prover theories; (ii) the transfer functions and the background axioms; and (iii) symbolic constants that may be used in the transfer functions and the single-assignment variables introduced during VC generation [7, 2]. A second drawback is that there is no feedback from the theorem prover to the abstract interpreter that would allow it to refine its inference.

The second drawback of the simple hybrid approach is addressed by our previous technique of "*loop invariants on demand*", a powerful integration between the theorem prover and the analyzer that focuses the refinement of loop invariants using counterexamples generated by the theorem prover [11]. Our previous technique works like this. First, we run a "cheap" static analysis on the program to infer simple loop invariants. Second, we instrument the program with those invariants and some special predicate symbols (loop predicates), which can seen as a kind of second-order predicate for loop invariants. Third, we generate verification conditions and pass them to an automatic

```
// assume $M$ to be an array of length $l$ and $0 \leq p < l$
$M[p] := 0;$
while $(M[p] < 10)$ {
   $M[p] := M[p] + 1;$    $(*)$
}
assert $M[p] = 10;$
```

Fig. 0. Discovering the loop invariant $0 \leq M[p] \leq 10$ in a classical setting involves the use of the reduced product of an abstract domain for tracking the shape of the memory, and a numerical domain for tracking the value of the memory location $M[p]$. With our technique, we exploit the knowledge already inside the theorem prover to infer the right invariant.

theorem prover. If the theorem prover can complete the proof with this information, then we are done. Otherwise, the theorem prover returns a monome that represents a path in the program that (may) lead to an error. We analyze the monome, extract the loop predicates, project the monome onto the entry points of the loops, and run a more precise static analysis just on these loops. We then obtain more precise loop invariants, and pass these back to the theorem prover, iterating the process until a proof is found, no further refinement of the loop invariants is possible, or some bound on the number of iterations is reached.

The main advantage of "loop invariants on demand" is that it provides an effective way of combining backward and forward analyses of the code through a form of dynamic trace partition [16, 9]. However, it still does not solve the issue of "duplication of knowledge" and code, since the theorem prover and static analyzer are two separate entities. Furthermore, (i) the static analyzer is (still) unaware of quantifiers and (ii) there are many low-level details, for instance the different representations of variables and functions in the abstract interpreter and the theorem prover, that make the implementation tricky.

In this paper, we introduce a new technique for performing a goal-directed inference of loop invariants directly inside an SMT solver. The starting point is a VC encoding along the lines of "loop invariants on demand". Given an SMT solver, we then define a widening operator over first-order formulae. This affords us a uniform handling of variables, constants, and functions, but more importantly it factors out the "duplication of knowledge" issue. Furthermore, it exploits the theorem prover's precision in handling symbolic expressions and its flexible support of quantifiers. We illustrate the main ideas behind our technique with the next two examples. The examples show that we can make use of quantifiers, and that the goal-directed technique provides a generic framework for seeding.

0.1 Running Examples

Example 0. Our first running example is the program in Fig. 0. It shows a loop that updates an element of an array. The program involves the essence of a pointer program, where the array M represents memory and p represents a pointer into memory (like `*p` in C). Proving the correctness of the program involves the use of the loop invariant $0 \leq M[p] \leq 10$. Discovering such an invariant, in a classical setting, requires an abstract domain that is (at least) the combination of an alias analysis and a numerical

```
// assume $A$ to be an array of length $l$
$s := 0;\ i := 0;$
while $(i < l)$ { $s := A[i] + s;\ \ i := i + 1;$ }
assert $s = \sum_{k=0}^{<l} A[k];$
```

Fig. 1. Proving the assertion involves discovering the loop invariant $0 \leq i \leq l \wedge s = \sum_{k=0}^{<i} A[k]$. While the first invariant can be discovered using standard numerical abstractions, the relation between s and the values in the array requires the design of an *ad hoc* abstract domain. Furthermore, the choice of the abstract domain must be fixed before running the analysis. Our technique leads to a dynamic and postcondition-driven choice of the abstract domain.

analysis, because (i) it must capture the shape of the memory, to infer that p always denotes the same index (*i.e.*, same memory location of) M, and that only the location $M[p]$ is modified, whereas all the others remains unchanged; and (ii) it must correctly approximate the integer value stored at location $M[p]$. Examples of such domains are in [15, 4].

Standard VC-generation handling of arrays takes care of the aliasing issues: assignment (*) in Fig. 0 is treated as $M := \mathsf{store}(M, p, M[p] + 1)$, the term $M[p]$ is represented by some equivalence class, say β, in the SMT solver's theory of functions and equality, and the array property

$$(\forall\, m, i, j, v \bullet\ \ i = j \ \Rightarrow\ \mathsf{store}(m, i, v)[j] = v\ ) \qquad (0)$$

gets used to figure out the change of the value of $M[p]$. With the technique of this paper, incorporating a widening operator into the theory of arithmetic then performs the necessary inference. For example, using the theory of linear inequalities enriched with a widening with thresholds $\nabla_{threshold}$, [3], we can infer, after two iterations of the loop, the invariant $\{\beta = 1\}\nabla_{threshold}\{\beta = 0\} = \{0 \leq \beta \leq 10\}$, which the theorem prover uses to prove the program correct. □

Example 1. Our second running example is shown in Fig. 1. We want to prove that the assertion after the loop always holds. This is tricky, because we have to infer the loop invariant $s = \mathsf{sum}(A, 0, i)$. $\mathsf{sum}(A, 0, i)$ is a predicate that, intuitively, axiomatizes the sum of the first i elements of the array A.

One way of doing it is by providing an *ad hoc* abstract domain and the relative transfer functions. Let us call this domain ArrSum. The abstract elements in ArrSum are maps from variables to terms of the form $\mathsf{sum}(a, 0, n)$ (the sum of the first n elements of the array a). While the domain operations over ArrSum are straightforward, the transfer functions must be carefully designed and coded to handle, for instance, the assignments that involve array elements. However, ArrSum is quite a specialized construction, and even if implemented in an analyzer, it is not clear *when* the analyzer must use it for analyzing a piece of code. For example, in the code of Fig. 1, the need for keeping track of the sum of the elements of the array arises only when the assertion is hit, *i.e.*, after the loop. A forward static analysis alone may not figure it out until the loop exit is reached. For instance, when analyzing the code using Octagons, the static analyzer cannot prove the postcondition, so (i) it must recognize that the problem is because Octagons cannot express such a property; (ii) it must choose the right abstract domain, *i.e.*, ArrSum; and (iii) it must analyze the method from scratch using ArrSum. This wastes both time and precision, since there is no communication between the first and second analyses.

With the technique in this paper, we do not need to hard-code the transfer functions or specify *a priori* the abstract domain. In a sense, the technique is able to dynamically choose the right domain and transfer functions by means of quantifier instantiation. For instance, the following partial axiomatization of the sum of the array:

$$(\,\forall\, a, l, h \bullet\ h \leq l \ \Rightarrow\ \mathsf{sum}(a, l, h) = 0\,) \tag{1}$$
$$(\,\forall\, a, l, h \bullet\ l \leq h \ \Rightarrow\ \mathsf{sum}(a, l, h+1) = \mathsf{sum}(a, l, h) + a[h]\,) \tag{2}$$

in combination with our technique is enough to infer the right loop invariant and hence to prove the program correct [12]. Roughly, the theorem prover first attempts to prove the assertion without looking inside the body of the loop. It fails, but its state now contains (i) the postcondition involving $\mathsf{sum}(A, 0, n)$; and (ii) a predicate encoding the need for a stronger loop invariant. We apply the widening as described later in the paper which causes the instantiation of the two quantifiers above, in particular obtaining the loop invariant $s = \mathsf{sum}(A, 0, i)$, which is enough for the theorem prover to complete the proof. $\square$

1 Preliminaries

We describe our technique for a simple imperative language. We prescribe the generation of verification conditions for this language using the standard approach of translating it into an intermediate language and then applying weakest preconditions [13, 1]. We adapt this process to instrument the verification conditions with information that will be used by our invariant-generation machinery.

1.0 Source Language

We consider the source language whose grammar is given in Fig. 2. The source language includes support for specifications via the $\mathbf{assert}\ E$ statement: if the expression E evaluates to false, then the program fails. The assignment statement $E_0 := E_1$ sets the memory location represented by E_0 to the value of the expression E_1. The statement $\mathbf{havoc}\ x$ non-deterministically assigns a value to x. Sequential composition, conditionals, and loops are the usual ones. Note that we assume loops to be uniquely determined by labels ℓ taken from a set $\mathcal{W}$: Given a label ℓ, the function $\mathsf{LookupWhile}(\ell)$ returns the while loop associated with that label.

1.1 Single-Assignment Form

We translate the source language of Fig. 2 into an intermediate form so as to get rid of (i) loops and (ii) assignments. To achieve (i), the translation replaces an arbitrary number of iterations of a loop with something that describes the effect that these iterations have on the variables, namely the loop invariant. To achieve (ii), the translation uses a variant of static single assignment (SSA) [0,7], introducing new variables (*inflections*)

$Stmt ::=$	$\mathbf{assert}\ Expr;$	(assertion)
$\mid$	$Expr := Expr;$	(assignment)
$\mid$	$\mathbf{havoc}\ x;$	(set x to an arbitrary value)
$\mid$	$Stmt\ Stmt$	(sequential composition)
$\mid$	$\mathbf{if}\ (Expr)\ \{Stmt\}\ \mathbf{else}\ \{Stmt\}$	(conditional)
$\mid$	$\mathbf{while}^{\ell}\ (Expr)\ \{Stmt\}$	(loop)

Fig. 2. The grammar of the source language.

$Cmd ::=$	$\mathbf{assert}^{L} Expr$	(assert)	$L ::= init \mid rec \mid \epsilon$
$\mid$	$\mathbf{assume}\ Expr$	(assume)	
$\mid$	$Cmd\ ;\ Cmd$	(sequence)	
$\mid$	$Cmd \,\square\, Cmd$	(non-deterministic choice)	

Fig. 3. The intermediate language. We label assertions with strings so as to distinguish between assertions to be checked at the entry point of a loop ($\mathbf{assert}^{init}$), after each loop iteration ($\mathbf{assert}^{rec}$), and user-supplied assertions from the source language ($\mathbf{assert}$, ϵ being the empty string).

that stand for the values of program variables at different source-program locations and that within any one execution path has only one value.

The statements of the intermediate language are given by the grammar in Fig. 3. Our intermediate language is that of passive commands, *i.e.*, assignment-free and loop-free statements [7].

The $\mathbf{assert}$ and $\mathbf{assume}$ statements first evaluate the expression $Expr$. If it evaluates to $true$, then the execution continues. If the expression evaluates to $false$, the $\mathbf{assert}$ statement causes the program to fail (the program *goes wrong*) and the $\mathbf{assume}$ statement causes the program to *block* (which implies that there is no possibility of the program subsequently going wrong). The intermediate language also supports sequential composition and non-deterministic choice.

The translation from a source-language program S to an intermediate-language program is given by the following function (id denotes the identity map):

$$\mathsf{translate}(S) \;=\; \mathbf{let}\ (\mathrm{C}, m) := \mathsf{tr}(S, id)\ \mathbf{in}\ \mathrm{C}$$

The verification condition is constructed by taking the weakest precondition (see Fig. 4) of the intermediate-language program, and adding an antecedent Q that contains the program's precondition and various background axioms (such as (0), (1), and (2)):

$$Q \;\Rightarrow\; \mathsf{wp}(\mathsf{translate}(S),\ true)$$

The definition of the function tr is in Fig. 5. The function takes as input a program in the source language and a renaming function from program variables to their pre-state inflections, and it returns a program in the intermediate language and a renaming function from program variables to their post-state inflections. The rules in Fig. 5 are described as follows.

The translation of an $\mathbf{assert}$ just renames the variables in the asserted expression to their current inflections. One of the goals of the passive form is to get rid of the assignments. Given an assignment $x := \mathrm{E}$ in the source language, we generate a fresh

$$\mathsf{wp}(\mathbf{assert}^L E,\ \phi) \ = \ E \wedge \phi \qquad \mathsf{wp}(C_0\ ;\ C_1,\ \phi) \ = \ \mathsf{wp}(C_0,\ \mathsf{wp}(C_1,\ \phi))$$
$$\mathsf{wp}(\mathbf{assume}\, E,\ \phi) \ = \ E \Rightarrow \phi \qquad \mathsf{wp}(C_0 \,\Box\, C_1,\ \phi) \ = \ \mathsf{wp}(C_0,\ \phi) \wedge \mathsf{wp}(C_1,\ \phi)$$

Fig. 4. Weakest preconditions of intermediate-language statements.

variable for x (intuitively, the value of x after the assignment), apply the renaming function to E, and output an $\mathbf{assume}$ statement that binds the new variable to the renamed expression. For instance, a statement that assigns to y in a state where the current inflection of y is y_0 is translated as follows, where y_1 is a fresh variable that denotes the inflection of y after the statement:

$$\mathsf{tr}(y := y + 4,\ [y \mapsto y_0]) \ = \ (\mathbf{assume}\ y_1 = y_0 + 4,\ [y \mapsto y_1])$$

An array assignment $a[\mathrm{E}_0] := \mathrm{E}_1$ is, as usual, treated as a shorthand for $a := \mathsf{store}(a, \mathrm{E}_0, \mathrm{E}_1)$. The translation of $\mathbf{havoc}\ x$ just binds x to a fresh variable, without introducing any assumptions about the value of this fresh variable. The translation of sequential composition yields the composition of the translated statements and the post-renaming of the second statement. The translations of the conditional and the loop are trickier.

For the conditional, we translate the two branches to obtain two translated statements and two renaming functions. Then we consider the set of all the variables on which the renaming functions disagree (intuitively, they are the variables modified in one or both the branches of the conditional), and we assign them fresh names. These names will be the inflections of the variables after the conditional statement. Then, we generate the translation of the true (resp. false) branch of the conditional by assuming the guard (resp. the negation of the guard), followed by the respective translated statement and an $\mathbf{assume}$ statement that equates the fresh variables with the inflections at the end of the true (resp. false) branch. Finally, we use the non-deterministic choice operator to complete the translation of the whole conditional statement.

For the loop, we first identify the assignment targets of the loop body (defined in Fig. 6), generate fresh names for them, and translate the loop body. Then, we generate a fresh predicate symbol indexed by the loop identifier (J_ℓ), which intuitively stands for the invariant of the loop $\mathsf{LookupWhile}(\ell)$. We also generate a fresh predicate symbol F_ℓ, which stands for the continuation of the loop; we will make this point clearer in Section 3. If the label ℓ is clear from context, as it is in our single-loop examples, we omit it. We output a sequence that is made up by: (i) an assertion that the loop invariant holds at the loop entry point; (ii) an assumption that the loop invariant holds after an arbitrary number of executions of the loop body (intuition: we have performed an arbitrary number of loop iterations and the renaming function n gives the current inflections of the variables); and (iii) a non-deterministic choice between two cases: (a) the loop condition evaluates to $true$, we execute a further iteration of the body, we check that the loop invariant holds ($\mathbf{assert}\ J_\ell\langle\mathsf{range}(n_C)\rangle$), and we stop checking ($\mathbf{assume}\ \mathit{false}$); or (b) the loop condition evaluates to false and the loop execution terminates.

Please note that we tacitly assume a total order on variables, so that, *e.g.*, the sets $\mathsf{range}(m)$ and $\mathsf{range}(n)$ can be isomorphically represented as lists of variables. We will use the list representation in our examples.

```
$\mathsf{tr} \in \mathit{Stmt} \times (\mathtt{Vars} \to \mathtt{Vars}) \to \mathit{Cmd} \times (\mathtt{Vars} \to \mathtt{Vars})$
$\mathsf{tr}(\mathbf{assert}\ \mathrm{E};,\ m)\ =\ (\mathbf{assert}\ m(\mathrm{E}),\ m)$
$\mathsf{tr}(\mathtt{x} := \mathrm{E};,\ m)\ =\ (\mathbf{assume}\ \mathtt{x}' = m(\mathrm{E}),\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\mathsf{a}[\mathrm{E}_0] := \mathrm{E}_1;,\ m)\ =\ (\mathbf{assume}\ \mathsf{a}' = \mathsf{store}(a, m(\mathrm{E}_0), m(\mathrm{E}_1)),\ m[\mathsf{a} \mapsto \mathsf{a}'])$
                                    where $\mathsf{a}'$ is a fresh variable
$\mathsf{tr}(\mathbf{havoc}\ \mathtt{x};,\ m)\ =\ (\mathbf{assume}\ \mathit{true},\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\mathrm{S}_0\ \mathrm{S}_1,\ m)\ =\ \mathbf{let}\ (\mathrm{C}_0, n_0) := \mathsf{tr}(\mathrm{S}_0, m)\ \mathbf{in}$
                         $\mathbf{let}\ (\mathrm{C}_1, n_1) := \mathsf{tr}(\mathrm{S}_1, n_0)\ \mathbf{in}$
                         $(\mathrm{C}_0\ ;\ \mathrm{C}_1,\ n_1)$
$\mathsf{tr}(\mathbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \mathbf{else}\ \{\mathrm{S}_1\},\ m)\ =$
    $\mathbf{let}\ (\mathrm{C}_0, n_0) := \mathsf{tr}(\mathrm{S}_0, m)\ \mathbf{in}$
    $\mathbf{let}\ (\mathrm{C}_1, n_1) := \mathsf{tr}(\mathrm{S}_1, m)\ \mathbf{in}$
    $\mathbf{let}\ \mathtt{V} := \{\mathtt{x} \in \mathtt{Vars} \mid n_0(\mathtt{x}) \neq n_1(\mathtt{x})\}\ \mathbf{in}$
    $\mathbf{let}\ \mathtt{V}'\ \mathbf{be}$ fresh variables for the variables in $\mathtt{V}$ $\mathbf{in}$
    $\mathbf{let}\ \mathrm{D}_0 :=\ \mathbf{assume}\ m(\mathrm{E})\ ;\ \mathrm{C}_0\ ;\ \mathbf{assume}\ \mathtt{V}' = n_0(\mathtt{V})\ \mathbf{in}$
    $\mathbf{let}\ \mathrm{D}_1 :=\ \mathbf{assume}\ \neg m(\mathrm{E})\ ;\ \mathrm{C}_1\ ;\ \mathbf{assume}\ \mathtt{V}' = n_1(\mathtt{V})\ \mathbf{in}$
    $(\mathrm{D}_0 \,\square\, \mathrm{D}_1,\ m[\mathtt{V} \mapsto \mathtt{V}'])$
$\mathsf{tr}(\mathbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\},\ m)\ =$
    $\mathbf{let}\ \mathtt{V} := \mathsf{targets}(\mathrm{S})\ \mathbf{in}$
    $\mathbf{let}\ \mathtt{V}'\ \mathbf{be}$ fresh variables for the variables in $\mathtt{V}$ $\mathbf{in}$
    $\mathbf{let}\ n := m[\mathtt{V} \mapsto \mathtt{V}']\ \mathbf{in}$
    $\mathbf{let}\ (\mathrm{C}, n_C) := \mathsf{tr}(\mathrm{S}, n)\ \mathbf{in}$
    $\mathbf{let}\ J_\ell\ \mathbf{be}$ a fresh predicate symbol $\mathbf{in}$
    $\mathbf{let}\ F_\ell\ \mathbf{be}$ a fresh predicate symbol $\mathbf{in}$
    $(\mathbf{assert}^{init}\ J_\ell\langle\mathsf{range}(m)\rangle\ ;$
     $\mathbf{assume}\ J_\ell\langle\mathsf{range}(n)\rangle \wedge F_\ell\ ;$
     $(\mathbf{assume}\ n(\mathrm{E})\ ;\ \mathrm{C}\ ;\ \mathbf{assert}^{rec}\ J_\ell\langle\mathsf{range}(n_C)\rangle\ ;\ \mathbf{assume}\ \mathit{false}$
      $\square\ \ \mathbf{assume}\ \neg n(\mathrm{E})),\ \ n)$
```

Fig. 5. The function that translates from the source program to our intermediate language.

For example, applying translate to the program in Fig. 0 results in the intermediate-language program shown in Fig. 7.

2 Widening in an SMT Solver

To prove the program in Fig. 0 correct, we use an automatic theorem prover, and more precisely a *satisfiability modulo theories* (SMT) solver. We transform the program into passive form and generate the weakest preconditions, as described in the previous section, and then pass the resulting formula to the theorem prover. In order to prove the assertion after the loop, we need to provide a loop invariant, *i.e.*, a value for the predicate J_ℓ. Our goal is to have the SMT solver automatically find a sound yet precise approximation for J_ℓ. For this aim, we need to (i) make explicit the assumptions about the theorem prover; (ii) define a widening operator inside the theorem prover; and (iii) define a strategy for the application of the widening operator.

$\mathsf{targets} \in \mathit{Stmt} \to \mathsf{P}(\mathtt{Vars})$
$\mathsf{targets}(\mathbf{assert}\ \mathrm{E};\,) \;=\; \emptyset$
$\mathsf{targets}(\mathtt{x} := \mathrm{E};\,) \;=\; \mathsf{targets}(\mathtt{x}[\mathrm{E}_0] := \mathrm{E}_1;\,) \;=\; \mathsf{targets}(\mathbf{havoc}\ \mathtt{x};\,) \;=\; \{\mathtt{x}\}$
$\mathsf{targets}(\mathrm{S}_0\ \mathrm{S}_1) \;=\; \mathsf{targets}(\mathbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \mathbf{else}\ \{\mathrm{S}_1\}) \;=\; \mathsf{targets}(\mathrm{S}_0) \cup \mathsf{targets}(\mathrm{S}_1)$
$\mathsf{targets}(\mathbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\}) \;=\; \mathsf{targets}(\mathrm{S})$

Fig. 6. The assignment targets, that is, the set of variables assigned in a source statement.

```
$\mathbf{assume}\ M_0[p] = 0$ ;  $\mathbf{assert}^{init}\ J\langle M_0\rangle$ ;  $\mathbf{assume}\ J\langle M_1\rangle \wedge F$ ;
( $\mathbf{assume}\ M_1[p] + 1 \le 10$ ;  $\mathbf{assume}\ M_2 = \mathsf{store}(M_1, p, M_1[p] + 1)$ ;
  $\mathbf{assert}^{rec}\ J\langle M_2\rangle$ ;  $\mathbf{assume\ false}$ ;
$\Box$
  $\mathbf{assume}\ 10 \le M_1[p]$ ;
) ;
$\mathbf{assert}\ M_1[p] = 10$ ;
```

Fig. 7. The intermediate-language program obtained as a translation of the source-language program in Fig. 0. J is a predicate symbol corresponding to the loop invariant, F is a predicate standing for the semantics of the program bottom up from the exit point up to the loop entry point.

2.0 The SMT Solver

An SMT solver decides validity, w.r.t. to some set of theories, of first-order formulae with equalities. We assume an SMT solver to be a triple $\langle E, \{\, T_i \mid 0 \le i\}, \vdash\rangle$, where E is a system for tracking equivalence relations (*e.g.*, an e-graph [18]), T_i is a set of first-order theories (*e.g.*, linear arithmetic or the fact-generating "theory" of quantifiers) and $\vdash$ is the decision procedure.

2.1 The Widening

A widening is an operator for extrapolating the limit of a (possibly infinite) sequence. The simplest (and least precise) widening is the one that extrapolates the sequence with the greatest element of the abstract domain. Stated differently, it is the one that answers "I do not know". For an abstract domain whose elements can be viewed as a set of constraints, a traditional schema for defining a widening (i) considers two successive sets of constraints in the sequence, say C_n and C_{n+1}; and (ii) keeps those of the constraints in C_n that are "stable", that is, that also hold in C_{n+1}. In other words, in its computation of the limit, this schema looks at consecutive elements of the sequence and removes from the limit any constraints that are not stable. For instance, the traditional definition of the widenings on the numerical domains Polyhedra [6], Octagons [17], and Intervals [5] are instances of this schema, [14]. We seek the definition of such a kind of widening inside an SMT solver.

First, we require that all the theories be extended with a widening specific to the theory. We write T_i^{∇} to denote the extension of a first-order theory T_i that uses widenings. The extension is always possible because, as mentioned earlier, one can always use the "I do not know" widening. However, one may consider using more precise widenings for certain theories.

```
$\Sigma \mathbin{\nabla} \Upsilon = \textbf{for each } c \in \Sigma \textbf{ do}$
    $\textbf{if } \Upsilon \vdash c \textbf{ then keep } c \textbf{ else keep } \bigwedge_{0 \leq i} c \mathbin{\nabla_i} (\Upsilon \downarrow i)$
```

Fig. 8. The widening over first-order formulae. $\Upsilon \downarrow i$ denotes the projection of the formulae in Υ over the language understood by the theory T_i^{∇}. ∇_i is the widening of the theory T_i^{∇}.

```
$\textbf{for each } \ell \in \mathcal{W} \textbf{ do } J_\ell := \textbf{false}$ ;
$\textbf{repeat}$
  Try to prove the formula ;
    0. if error on $\textbf{assert}^{init}\ J_\ell\langle \mathbf{x_0} \rangle$  then $J_\ell := J_\ell \vee \textbf{rename}(\pi_{\mathbf{x_0}}(\Sigma), \mathbf{x_0}, \mathbf{x})$ ;
    1. if error on $\textbf{assert}^{rec}\ J_\ell\langle \mathbf{x_2} \rangle$   then $J_\ell := J_\ell \mathbin{\nabla} \textbf{rename}(\pi_{\mathbf{x_2}}(\Sigma), \mathbf{x_2}, \mathbf{x})$ ;
    2. if error on $\textbf{assert } P$            then report "**error**" and exit ;
    3. if no error                     then report "**correct**" and exit ;
```

Fig. 9. The driver algorithm for the SMT solver. Σ denotes the state of the theorem prover, $:=$ denotes destructive assignment, π denotes projection, the list $\mathbf{x_0}$ stands for the inflections of the (list of) variables $\mathbf{x}$ at the entry point of loop ℓ, and the list $\mathbf{x_2}$ stands for the inflections of the (list of) variables $\mathbf{x}$ at the end of the body of loop ℓ.

Example 2. Consider the theory of linear inequalities and suppose the implementation uses a list of linear constraints. The widening of two sets of linear constraints, say ∇_{la}, can be implemented by converting the constraints into the dual representation as vertices and generators, and then applying the original Cousot-Halbwachs widening on polyhedra [6]. Alternatively, since the conversion into the dual form can be expensive, cheaper solutions can be used, *e.g.*, projecting the constraints onto octagons and then applying the widening of this domain [17]. $\square$

Second, we define the widening inside the SMT solver as in Fig. 8. Let Σ and Υ be two sets of formulae. Intuitively Σ is the set of formulae (whose conjunction represents) "the current approximation of the loop invariant" and Υ is the set of formulae (whose conjunction represents) "after one more iteration". If a formula $c \in \Sigma$ holds in Υ, then it is stable over loop iterations, so we keep it. Otherwise, for each theory T_i^{∇}, we apply the widening of c and (the projection over the) theory of Υ.

2.2 The Fixpoint Computation Strategy

Having defined a widening operator, we now define how to apply it. Stated differently, we need an algorithm that drives the theorem prover to the application of the widening, so as to compute fixpoints. The driver is defined in Fig. 9. For now, we assume that the solver reaches case 2 only if there is no other path in the formula for which cases 0 and 1 apply; we will remove this assumption in the next section.

In essence, the algorithm invokes the SMT solver using successively weaker definitions of each inferred loop invariant. If the SMT solver complains that it cannot prove the current approximation of a loop invariant to hold, the algorithm weakens the loop invariant and reruns the SMT solver. The algorithm assigns to the loop predicate J_ℓ its current invariant for loop ℓ; to represent the program variables in this formula, we will use the pre-inflected forms of the program variables.

In more detail, the algorithm initializes all the loop predicates to $\mathbf{false}$. In other words, the fixpoint computation starts from bottom. Then we ask the SMT solver to prove the formula. We consider a case for each of the four different outcomes.

Outcome 0: the solver complains about the "$init$" assertion of a loop ℓ. Intuitively, the current loop invariant is too strong to accommodate all paths that lead to the entry of the loop. So we weaken the loop invariant J_ℓ to include the prover's knowledge at the entry point of the loop ℓ. This is done by (i) projecting the state Σ of the prover onto the inflections of the variables $\mathbf{x}$ at the entry point: $\pi_{\mathbf{x_0}}(\Sigma)$; (ii) changing the variables back to their pre-inflected form, that is, renaming the $\mathbf{x_0}$ back to $\mathbf{x}$; and (iii) joining the result to the previous J_ℓ. Note and recall, the free variables of the predicate J_ℓ are written in the pre-inflected form $\mathbf{x}$.

Outcome 1: the solver complains about the "rec" assertion of a loop ℓ. This means that the prover is unable to prove that the current loop-invariant approximation is maintain by the loop body. Here, we do not use the logical disjunction as in case 0, because that would not guarantee termination of our analysis. Instead, we use the widening operator ∇ of Fig. 8.

Outcome 2: the solver complains about an assertion that originates from the source program. Since the iterations of our algorithm only weakens the approximated loop invariants, further iterations of the algorithm would never mask this failing assertion. Thus, we report an error and exit. Note, however, that the condition may or may not be an error, because of imprecision in the theorem prover's decision procedures (for example, for quantifiers) or because of precision lost in widening operations.

Outcome 3: the solver finds the current formula to be valid. Then the program is correct (since our analysis and the SMT solver are sound), and we can report it to the user and exit.

A note: For outcome 2, it would be nice if the theorem prover were as specific as possible about the counterexample it has found, since this will allow a more concrete error message. But for outcomes 0 and 1, we are interested in as general a counterexample as possible, since that will reduce the number of iterations of our algorithm.

2.3 Application to the First Example

We briefly sketch how our technique works on the code in Fig. 0. First, the program is translated into the single-assignment form in Fig. 7. Second, we generate the weakest preconditions from the passive program and add formula (0) as an antecedent; call the result ϕ. Third, we feed ϕ to the prover. For the sake of simplicity, herein we reason on the passive program and not on the formula ϕ, the two representations being isomorphic for our purposes. We apply the driver in Fig. 9. We set J_ℓ to $\mathbf{false}$ and run the prover. It returns, complaining about the assertion $\mathbf{assert}^{init}\ J\langle M_0\rangle$. Case 0 applies, and $\pi_{\mathbf{M_0}}(\Sigma)$ yields $M_0[p] = 0$, so we update J to be the predicate $M[p] = 0$. We run the prover again, and this time it complains about the recursive assertion $\mathbf{assert}^{rec}\ J\langle M_2\rangle$. Case 1 applies, so we instantiate the widening operator in Fig. 8. The state of the theorem prover up to this point includes the constraints

$$\{\ \text{formula (0)},\ 0 \leq M_1[p],\ M_1[p] + 1 \leq 10,\ M_2 = \mathsf{store}(M_1, p, M_1[p] + 1)\ \}$$

Projecting this state onto M_2 yields $1 \leq M_2[p] \leq 10$, because by instantiating the quantifier (0), it follows that $M_2[p] = \mathsf{store}(M_1, p, M_1[p] + 1)[p] = M_1[p] + 1$. From renaming, we get $1 \leq M[p] \leq 10$. The widening is then $\{M[p] = 0\}\nabla_{la}\{1 \leq M[p] \leq 10\}$. $M[p]$ is represented in the e-graph of the theorem prover with some symbolic name β. Thus, if we assume that the widening of linear arithmetic, ∇_{la} is with thresholds, we have

$$\{\beta = 0\} \nabla_{la} \{1 \leq \beta \leq 10\} = \{0 \leq \beta \leq 10\}$$

Consequently, the predicate J is updated to $0 \leq M[p] \leq 10$, the prover is run again, and this time it can prove the formula to be valid, so that it reaches case 3 and exits.

3 Goal-Directed Seeding

In the previous section, we saw how to exploit the symbolic capabilities and the quantifiers in theorem provers for inferring non-trivial loop invariants involving array updates, and hence memory manipulation. However, we want to go further, and in particular we want to be able to handle more properties, expressed as (partial) axiomatizations. Let us consider the example in Fig. 1. We would like to use the axioms (1) and (2) to get the correct loop invariant, so as to prove the user-supplied assertion correct. The technique of the previous section is not powerful enough to do so, essentially because the algorithm in Fig. 9 is a *forward* algorithm. It does not infer any invariant about s, since the need for an invariant that correlates s and sum will be discovered only *after* the loop. Hence, the theorem prover cannot prove the user assertion, so the algorithm reaches case 2, and thus ends up reporting a spurious warning.

To overcome this problem, we must be able to refine the information about loop invariants using information coming from the future. In other words, we need to tag entry points of loops with information about what will happen after the loop, so that, if needed, we can exploit this information to refine the loop invariant. We do it in two steps. First, we annotate loop assumptions with a predicate symbol F_ℓ, *i.e.*, a placeholder that we use to encode the execution of the program *after* the loop. Second, we refine case 2 of Fig. 9, so that now, when an assertion is violated, instead of immediately reporting an error and exiting, we (i) check the counterexample to see if it contains any F_ℓ, *i.e.*, witnesses of loops whose invariants can be refined with information coming from the continuation of the loop; (ii) try to refine the loop invariants J_ℓ for the affected loops; and (iii) give another change to the theorem prover with the new facts.

3.0 Enrich

We refine the case 2 of the driver as in Fig. 10. $\mathsf{Enrich}(\ell)$ is a procedure that rolls back the loop invariant inferred for the loop ℓ, and gives a new change to the fixpoint computation for inferring a new invariant, by having some additional knowledge coming from the "future".

The pseudo-code for Enrich is in Fig. 11. The first step of Enrich is to "un-negate" the failed assertion. If an $\mathbf{assert}\ P$ fails, then the monome μ contains the negation of P. Since P may be arbitrarily complex, in particular if it contains conjuncts (that

```
2'. if error on $\mathbf{assert}\ P$
      if want to try again
         $\mathbf{let}\ \mu\ \mathbf{be}$ the counterexample generated from the theorem prover
         Chose an $F_\ell \in \mu$
            $\mathsf{Enrich}(\mu, \ell)$
      else
         report “error” and exit ;
```

Fig. 10. The refinement of the case 2 that introduces a goal-directed inference.

```
$\mathsf{Enrich}(\mu, \ell) =$
   $a$. un-negate the failing assertion in $\mu$. Call the result $\Sigma$ ;
   $b$. roll back to a pre-widening value for $J_\ell$ ;
   $c$. $J_\ell := J_\ell \wedge (\mathbf{rename}(\pi_\Sigma(\mathbf{x_1}), \mathbf{x_1}, \mathbf{x}) \nabla J_\ell)$ ;
```

Fig. 11. The definition of function Enrich. The list $\mathbf{x_1}$ stands for the inflections of the (list of) variables $\mathbf{x}$ after an arbitrary number of loop iterations.

negated became disjuncts), “un-negating” it may be difficult. We believe it is still possible, however—for example, the theorem prover may be able to take a snapshot of its state before it attempts to prove an assertion.

The second step is to roll back the updates of J_ℓ to before we applied the widening. Intuitively, this means that we want to go back to the state just before analyzing the loop ℓ. Hence, we pick the last update J_ℓ before the application of case 1 in the algorithm above. This rolling back is always possible, provided that the implementation keeps the sequence of updates to J_ℓ (or the disjuncts added via applications of case 0).

The third step essentially pushes (backwards) the failed assertion P to the entry point of the loop, so that facts about “the future” become known during the analysis of the loop body, giving case 1 of the driver in Fig. 9 a chance to infer stronger invariants than in its previous attempts.

3.1 Application to the Second Example

Let us consider the code in Fig. 1. We omit showing the steps of single assignment, inflections, and weakest preconditions, in order to concentrate on how the driver algorithm works. We start the iterations from bottom, so at a first approximation we have $J = \mathbf{false}$. Then, we try to verify the code, the prover complains about the “*init*” assertion, so case 0 applies and we update J as follows:

$$\begin{aligned} J \;:=\;& \mathbf{false} \vee \{s = i = 0 \le l\} \\ =\;& \{s = i = 0 \le l\} \end{aligned}$$

We give the theorem prover another chance, and now it complains about the “*rec*” assertion. We proceed as above and we get the new approximation $J = \{0 \le i \le l\}$. Unfortunately, the widening lost all the information about s, so it is not possible to prove the assertion at the end of the program. The driver reaches the case $2'$, in a state μ containing $\{s_0 = i_0 = 0 \le i_1 \le l,\ l \le i_1,\ s_1 \ne \mathsf{sum}(0, l, A),\ F\}$, where s_0 and i_0 are the inflections of s and i at the entry point of the loop and s_1 and i_1 are the inflections of the same variables after an arbitrary number of iterations. Since μ contains the placeholder F, we apply the Enrich procedure to the (only) loop in the example.

Step a removes the negation of the assertion, and adds the assertion itself. The new state Σ contains:

$$\{\text{ axioms},\ s_0 = i_0 = 0 \le i_1 \le l,\ l \le i_1,\ s_1 = \mathsf{sum}(0, l, a) = \mathsf{sum}(0, i_1, A),\ F\ \}$$

Note, that from the equalities $i_1 = l$ and $s_1 = \mathsf{sum}(0, l, a)$ and formula (1), the SMT solver can deduce the equality $s = \mathsf{sum}(0, i_1, A)$.

Step b rolls back J to the last state before the application of a widening:

$$J = \{s = i = 0 \le l\}$$

This means that we, once again, have as much information as possible about s from the execution paths that reach the loop. Step c applies the widening to get the new value for J:

$$\begin{aligned} J \ &:= \ \{0 \le i \le l,\ i = l,\ s = \mathsf{sum}(0, l, A) = \mathsf{sum}(0, l, A)\} \ \nabla \ J \\ &= \ \{0 \le i \le l,\ s = \mathsf{sum}(0, i, A)\} \end{aligned}$$

which, together with the two axioms (1) and (2) let the theorem prover conclude the proof.

4 Conclusions

We have presented a technique that moves the inference of a program's loop invariants into the theorem prover that is attempting to prove the program correct. This has the benefits of making the inference more demand driven, reducing the duplication of work in the implementations of the abstract interpreter and theorem prover, and providing an easy way to extend abstract interpreters with information specified in quantifiers.

The work is still at an early stage. For example, we have not yet implemented the technique, we have not formally proved any soundness theorems, and we have not evaluated the complexity of our widening operator, which would benefit from an efficient way to enumerate the constraints $c \in \Sigma$ in the prover. Nevertheless, we are encouraged by the ease with which our second example goes through; in that example, the loop-invariant inference makes use of the partial axiomatization of sum that is specified by quantifiers (previously not incorporated in abstract interpreters), and it automatically seeds the abstract domain with an appropriate application of function sum (in contrast to the typical mode of requiring each abstract domain to design its own seeding). We hope that the discussion at the workshop will lead to greater insights.

Acknowledgments We are grateful to the referees for their careful readings and useful comments.

References

0. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *15th POPL*, pages 1–11. ACM, January 1988.
1. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
2. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87. ACM, September 2005.
3. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03*, pages 196–207. ACM, June 2003.
4. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI '05*, volume 3385 of *LNCS*. Springer, January 2005.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252. ACM, January 1977.
6. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97. ACM, 1978.
7. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *28th POPL*, pages 193–205. ACM, January 2001.
8. Sumit Gulwani and Ashish Tiwari. Static analysis of heap-manipulating low-level software. Technical report, Microsoft Research, MSR-TR-2006-160, 2006.
9. Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS '98*, volume 1503 of *LNCS*, pages 200–215. Springer, 1998.
10. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*. Springer, 2000.
11. K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS '05*, volume 3780 of *LNCS*. Springer, November 2005.
12. K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs. Manuscript KRML 175, May 2007. Submitted.
13. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *FTfJP 1999*, Technical Report 251. Fernuniversität Hagen, May 1999.
14. Francesco Logozzo. Approximating module semantics with constraints. In *SAC 2004*, pages 1490–1495. ACM, March 2004.
15. Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In *VMCAI '07*, volume 4349 of *LNCS*. Springer, January 2007.
16. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP 2005*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.
17. Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
18. Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Also available as Technical Report CSL-81-10, Xerox PARC, June 1981.
19. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
20. Francesco Ranzato and Francesco Tapparo. An abstract interpretation perspective on linear vs. branching time. In *APLAS '05*, volume 3780 of *LNCS*. Springer, November 2005.