Assertion-based Loop Invariant Generation

Mikoláš Janota*

School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie

Abstract. Many automated techniques for invariant generation are based on the idea that the invariant should show that something "bad" will not happen in the analyzed program. In this article we present an algorithm for loop invariant generation in programs with assertions using a weakest precondition calculus. We have realized the algorithm in the extended static checker ESC/Java2. Challenges stemming from our initial experience with the implementation are also discussed.

1 Introduction

Automated invariant generation techniques face two major challenges. The first challenge is the invariant itself, i.e., which formulas should be suggested as invariants. The second challenge is time complexity. Programs in practice require nontrivial, e.g., quantified, invariants; to be able to reason about such invariants, techniques commonly rely on an automated theorem prover. Since invocations of a theorem prover are time expensive, it is necessary to carefully limit the number of calls to the prover. In this article we introduce a light-weight technique that derives loop invariants from assertions, which express desired properties of the program. We have implemented the algorithm in the extended static checker ESC/Java2 [11]. The implementation operates on the intermediate representation of the program, where program annotations are translated into assertions and assumptions. Thus, the implementation takes into account both the program code and its specification.

The rest of the paper is structured as follows. Section 2 introduces the problem and its context. Section 3 describes the basic version of the loop invariant generation algorithm and Sections 4 illustrates the algorithm on an example. Section 5 describes an extension of the algorithm. Section 6 discusses the implementation and Section 7 lists related work. Finally, Section 8 concludes and proposes future work.

* This work was funded by Science Foundation Ireland under grant number 03/CE2/I303-1, "LERO: the Irish Software Engineering Research Centre." This work was partially supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

2 Background

The Java Modeling Language (JML) is a first–order logic annotation language for Java programs [12]. JML is embedded into Java programs as a special form of comments. The sign '@' indicates that a particular comment is a JML annotation. Figure 1 illustrates the use of JML.

```
//@ requires a != null;
//@ requires (\forall int x; (0 <= x & x < a.length) ==> a[x] != null);
void setToZero(int [][] a) {
  /*@ loop_invariant
    @   (\forall int x; (0 <= x & x < a.length) ==> a[x] != null);
    @ loop_invariant a != null;
    @ loop_invariant i >= 0; */
  for (int i = 0; i < a.length; i++) {
    /*@ loop_invariant j >= 0;
      @ loop_invariant a != null;
      @ loop_invariant
      @   (\forall int x; (0 <= x & x < a.length) ==> a[x] != null); */
    for (int j = 0; j < a[i].length; j++)
      a[i][j] = 0;
  }
}
```

Fig. 1. JML-annotated Java method.

The goal of ESC/Java2 is to determine whether JML-annotated Java code conforms to its annotations. For the given annotated program, ESC/Java2 generates a logic formula, called *verification condition* (VC), using a weakest precondition or a strongest postcondition calculus. Subsequently, it sends the VC to an automated theorem prover. If the theorem prover does not prove the VC valid, ESC/Java2 provides warnings describing how the program violates the JML-specification or how the program might cause run-time exceptions (e.g., by null-pointer dereferencing).

ESC/Java2 performs the translation from a JML-annotated program to the VC in several stages. For the space limitations we do not provide details for the whole translation process here and the interested reader is referred to the relevant documentation [7]. For the understanding of the article, however, it suffices to present one of the intermediate representations called *guarded command language* [13].

In the rest of the paper we will assume that we have the following. A set of program variables Var and a language of side-effect free expressions $\mathcal{E}$. We intentionally leave the grammar of expressions unspecified as the presented algorithm does not depend on their structure. A first-order logic language that contains at least the boolean expressions in $\mathcal{E}$, propositional connectives, and

the constants true, false. A theory T, known as *background predicate*, axiomatizing the semantics of expressions and we will write $T \models f$ to denote that the formula f is valid in the theory T. Additionally, we assume that an automated theorem prover is available. The prover accepts queries of the form $T \models f$ and responds whether the query was proven valid or not.

The following defines the grammar of the guarded command language:

$$\begin{aligned} cmd := \; & x \leftarrow expr \mid \mathbf{assume}\, f \mid \mathbf{assert}\, f \mid \mathbf{skip} \mid \\ & cmd \,[]\, cmd \mid cmd; cmd \mid \{\mathcal{I}\}\mathbf{while}\ expr\ \mathbf{do}\ cmd \end{aligned}$$

where $x \in Var$, $expr \in \mathcal{E}$, and $\mathcal{I}, f$ are logic formulas where the only free variables are in Var.

Informally, if the execution reaches $\mathbf{assume}\, f$ then f can be assumed and, if f does not hold, the execution blocks. If $\mathbf{assert}\, f$ is reached and f does not hold, an error occurs, in this case we say that the program *went wrong*. The command $C_1 \,[]\, C_2$ is a nondeterministic *choice* between C_1 and C_2; $C_1; C_2$ is a *sequence* of commands, $\{\mathcal{I}\}\mathbf{while}\ c\ \mathbf{do}\ B$ is a loop with the invariant $\mathcal{I}$, condition c and body B representing a program that repeats B while c holds. The command $x \leftarrow expr$ *assigns* the value of the expression $expr$ to the variable x.

In ESC/Java2, the guarded command language is used to represent Java code and its pertaining JML annotations[1]. For example, a call to a method is translated to an $\mathbf{assert}\, P$ representing the precondition of the called method and an $\mathbf{assume}\, Q$ representing its postcondition; an if statement is translated to a choice between the "then" and the "else" branch; the domains of variables contain Java primitive types and the model of the heap.

To formally capture the semantics of the guarded command language we will define a weakest precondition predicate transformer. We will do this is two steps, the first step is called *desugaring* and it defines the semantics of loops in terms of other commands. The second step defines a weakest precondition calculus on the desugared form of the guarded command language.

We will use the following auxiliary functions. The function *havoc* resets values of the given variables:

$$havoc(\{x_1, \ldots, x_n\}) \equiv x_1 \leftarrow v'_1; \ldots; x_n \leftarrow v'_n$$

where $v'_1, \ldots, v'_2$ are fresh variables. And, let $targets(C)$ be a function that returns all variables that might be modified by the command C. The desugaring is captured by the function desugar defined as follows:

$$\begin{aligned} &\text{desugar}\ (\{\mathcal{I}\}\mathbf{while}\ c\ \mathbf{do}\ B) \equiv \\ &\qquad \mathbf{assert}\, \mathcal{I}; \\ &\qquad \text{havoc}(\text{targets}(B)); \mathbf{assume}\, \mathcal{I}; \\ &\qquad ((\mathbf{assume}\, c; \text{desugar}(B); \mathbf{assert}\, \mathcal{I}; \mathbf{assume}\, \text{false}) \,[]\, (\mathbf{assume}\, \neg c)) \\ &\text{desugar}(C_1 \,[]\, C_2) \equiv \text{desugar}(C_1) \,[]\, \text{desugar}(C_2) \\ &\text{desugar}(C_1; C_2) \equiv \text{desugar}(C_1); \text{desugar}(C_2) \\ &\text{desugar}(C) \equiv C, \text{ all other commands} \end{aligned}$$

[1] ESC/Java2 supports full Java 1.4 source code.

$$
\begin{array}{ll}
\mathrm{wlp}(\textbf{skip}, N, W) & \equiv N \\
\mathrm{wlp}(x \leftarrow expr, N, W) & \equiv N[x \mapsto expr] \\
\mathrm{wlp}(\textbf{assume}\, f, N, W) & \equiv f \Rightarrow N \\
\mathrm{wlp}(\textbf{assert}\, f, N, W) & \equiv (f \Rightarrow N) \wedge (\neg f \Rightarrow W) \\
\mathrm{wlp}(C_1; C_2, N, W) & \equiv \mathrm{wlp}(C_1, \mathrm{wlp}(C_2, N, W), W) \\
\mathrm{wlp}(C_1 \,[\!]\, C_2, N, W) & \equiv \mathrm{wlp}(C_1, N, W) \wedge \mathrm{wlp}(C_2, N, W)
\end{array}
$$

Fig. 2. Weakest precondition calculus.

Intuitively, the initial assertion in the desugaring of a loop is a check for the invariant when the execution reaches the loop. The left branch of the choice command represents an arbitrary iteration of the loop and the **assume** $\neg c$ represents the termination of the loop.

The predicate transformer $wlp(C, N, W)$ defined in Figure 2 captures the semantics of the desugared guarded command language. For a command C and predicates N and W, if $wlp(C, N, W)$ holds then N holds if C terminates normally, W holds if C goes wrong or C does not terminate at all. For example, **assume** false never terminates or goes wrong.

In this context, the verification condition is defined so that the given program does not go wrong for any possible output. This is captured by the following definition.

Definition 21 *1. For a program C, the* verification condition *is the formula:*

$$\mathrm{wlp}(desugar(C), true, false)$$

2. A program C conforms to its specification *if and only if:*

$$T \models \mathrm{wlp}(desugar(C), true, false)$$

3 Invariant Inference from Assertions

The presented technique is motivated by a simple observation. In the subcommand of the desugared version of a loop representing an arbitrary iteration all we know about the loop's targets is whatever is in the loop invariant. Since the ultimate goal of the verification process is to show that the given program conforms to its specification, we need to ensure that the loop invariant is strong enough to prove that the assertions inside the loop do not go wrong (see Sections 4, 5 for examples of assertions and invariants).

To derive such loop invariants, the algorithm back-propagates assertions outwards to the outermost loop using the weakest precondition calculus. To explain the algorithm, we introduce the following terminology. We split the loops in the analyzed program into *layers*. The layer 0 contains exactly the loops that are not inside any other loop in the program. Loops in the layer i are exactly those loops nested in i other loops. We will refer to the loops in the layer 0 as *loop-nests* and we will use L_i to denote that the loop L_i is in the layer i. We will say that a

```
INFER-INVARIANTS(L_0, ..., L_i : loops, assert f : command hosted by L_i)
    loc := location of the given assertion in L_i
    I := f
    preserves := true
    for k := i downto 0
          do I := back-propagate I from loc to entry of L_k
             if (L_k preserves the invariant I)
                then add I to the invariant of L_k
                else  preserves := false
                      break
             if k ≠ 0
                then loc := location of the entry of L_k in L_{k-1}

    succeeded := preserves ∧ (I holds at the entry of L_0)
    if ¬succeeded
       then remove the added invariants
```

Fig. 3. Deriving invariants from an assertion.

.

loop L_i *hosts* the command C if and only if L_i contains C and there is no loop L_k with $k > i$ that contains C. We will say that a loop L *preserves* a formula f to express that if f held before an arbitrary iteration of L then it will hold once the iteration terminates; we will discuss the exact meaning of this term and back-propagation in Section 3.1.

The heart of the algorithm is the derivation of invariants from a given assertion. Consider the following. An assertion **assert** f is hosted by a loop L_i. And let $L_0, \ldots, L_i$ be a sequence of nested loops, i.e., L_k hosts L_{k+1} for $k \in 0 \ldots i-1$. In this scenario, we use the asserted formula f to infer invariants for the loops $L_0, \ldots, L_i$. This inference process starts by back-propagating the formula f to the top of L_i yielding a formula f_i. Subsequently, we determine whether f_i is preserved by L_i. If that is true, the process continues by back-propagating f_i to the top of L_{i-1}, yielding a formula f_{i-1}. This is repeated until the outermost loop L_0 is reached with the formula f_0. Finally, to verify that the suggested formulas are indeed invariants, we test whether f_0 holds at the entry of the loop-nest L_0. This process is captured by the pseudo-code in Figure 3.

Up to this point we have described how to infer an invariant from a single assertion. The whole analysis infers invariants from all assertions that are contained in any loop starting from the assertions in the innermost layers. This is captured by the following pseudo-code.

```
ANALYZE(C : command)
    for each loop-nest L_0 in C, using breadth-first search
         do ANALYZE(L_0)
```

ANALYZE($L_0, \ldots, L_i : loop$)
  for each loop K hosted in L, using breadth-first search
    do ANALYZE($L_0, \ldots, L_i, K$)
  for each assert f hosted in L, using breadth-first search
    do INFER-INVARIANTS($L_0, \ldots, L_i,$ **assert** f)

So far we have not explained the following: how expressions are back-propagated, how it is determined that a formula preserves a loop, and that a formula holds at the entry of a loop-nest. The following section provides details on these subjects.

3.1 Algorithm Details

For the purpose of this section we consider a control flow graph representation of the desugared guarded command language. We require that the control flow graphs have the following properties. Each node in the graph is labeled either with the command **skip**, **assume** f, **assert** f, or $x \leftarrow e$. The graph is directed acyclic and it has exactly one entry node, a node that dominates all the other nodes in the graph. Any subgraph G_L resulting from the desugaring of a loop has one entry node and one exit node. The entry node dominates all nodes in G_L and any path from any node in G_L to a node that is not in G_L contains the exit node (a postdominator). It is easy to observe that it is possible to construct such a graph for any desugared command. For example, the graph of the choice command has a "diamond" structure where the top and bottom tips are labeled with **skip** and the sides are graphs representing each choice.

In the context of a control flow graph G, back-propagation of the formula f from the node r to the node n is computed by the following function:

$$\begin{aligned} pre_G(n, r, f) &\equiv f, \text{if } n = r \\ &\equiv \text{wlp}(C_n, \textstyle\bigwedge_{c \text{ is a child of } n \text{ in } G} pre_G(c, r, f), \text{true}), \text{ otherwise} \end{aligned}$$

where C_n denotes the command labeling the node n. In plain English, the property of the function pre_G is that if $pre_G(n, r, f)$ held in n and the normal execution reached r then f holds.

Now we can describe back-propagation in an iteration of the loop in procedure INFER-INVARIANTS (Figure 3). For a loop $L_k \equiv \{f\}$**while** c **do** B it constructs a graph G with the entry node n_k of the command desugar(**assume** $c; B$). It identifies the entry node n_{k+1} of the subgraph of G that resulted from the desugaring of the loop L_{k+1}. Finally, it sets the suggested invariant $\mathcal{I}$ to $pre_G(n_k, n_{k+1}, \mathcal{I})$.

To determine whether the given invariant $\mathcal{I}$ holds at the entry of a loop-nest L_0, we construct a graph G with the entry node n from the desugared version of the whole program being analyzed, identify the entry node n_0 of the subgraph of G resulting from L_0, and send the query $T \models pre_G(n, n_0, \mathcal{I})$ to the theorem prover.

To determine whether the loop $L \equiv \{\mathcal{I}\}$**while** c **do** B preserves a formula f we send the following query to the theorem prover:

$$T \models f \Rightarrow \text{wlp}(C; \text{havoc}(\text{targets}(B)); \textbf{assume}\, f; \textbf{assume}\, c; \text{desugar}(B), f, \text{true})$$

where C is the loop's *context*, i.e., the preceding commands in the desuagared version of the program. For example, the inner loop in the program

$$C_0; (\{\mathcal{I}_0\}\textbf{while } c_0 \textbf{ do } (C_1; (\{I_1\}\textbf{while } c_1 \textbf{ do } B)))$$

has the following context:

$$\text{desugar}(C_0); \text{havoc}(\text{targets}(C_1; \{I_1\}\textbf{while } c_1 \textbf{ do } B)); \textbf{assume } (\mathcal{I}_0 \wedge c_0)$$

4 Example of Back-propagation

This section illustrates the analysis presented in the last section on an example. Consider the program in Figure 4, written in pseudo-code, which takes a two-dimensional array as its input and sets all its elements to zero.

```
 1: INPUT: a: array[1...N][1...M] of ℕ
 2: VAR i, j: ℕ;
 3: i ← 1;
 4: while i ≤ N do
 5:    j ← 1;
 6:    while j ≤ M do
 7:       ASSERT 1 ≤ i ∧ i ≤ N;
 8:       ASSERT 1 ≤ j ∧ j ≤ M;
 9:       a[i][j] ← 0;
10:       j ← j + 1;
11:    end while
12:    i ← i + 1;
13: end while
```

Fig. 4. Example of code with assertions.

The assignment to an element of the array requires that the values of i and j are in the bounds of the array a. This is captured by the two assertions. We will illustrate the steps of the algorithm on the first assertion.

The algorithm first back-propagates the assertion to the top of the inner loop, which results in the following formula:

$$(j \leq M) \Rightarrow (1 \leq i \wedge i \leq N)$$

Subsequently, the algorithm tests whether this formula is preserved by the body of the inner loop. This follows from the fact that the inner loop does not change the value of i.

In the next step the algorithm propagates the previously obtained formula to the top of the outer loop. This results in the following:

$$(i \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq i \wedge i \leq N)$$

This can be simplified to the equivalent formula:

$$(i \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq i)$$

Further, the algorithm tests whether the outer loop preserves the simplified formula. This follows from our previous result that the inner loop preserves the desired property and that i is increased.

The final step tests whether the suggested invariant is established by the code preceding the outer loop, i.e., the validity:

$$T \models (1 \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq 1)$$

this immediately follows from the reflexivity of $\leq$.

At this stage we know that

$$(i \leq N) \Rightarrow (1 \leq M) \Rightarrow (1 \leq i)$$

is an invariant of the outer loop. And,

$$(j \leq M) \Rightarrow (1 \leq i \wedge i \leq N)$$

is an invariant for the inner loop. Together, these invariants guarantee that the first assertion is not violated (does not go wrong).

5 Invariant Alterations

This section describes an extension of the presented invariant inference technique called *invariant alterations*. It is obvious that in many cases the invariant suggested by the procedure Infer-Invariants (Figure 3) will not be strong enough to prove that it is preserved by the loop even if it is an invariant of that loop. A possible improvement is to alter the suggested invariant based on some heuristic.

The following is an example of how to obtain a set of alterations from a formula f:

$$\{(\forall v' \bullet f[v \mapsto v']) \bullet \ v \text{ is free in } f \text{ and } v' \text{ is a fresh variable}\}$$

All these alterations are stronger than the original formula f. Therefore, if we can show that one of these alterations is an invariant, it will still guarantee that the original assertion does not go wrong.

The rest of this section illustrates the use of alterations on an example. Consider the Java code in Figure 1 without the loop invariants. The assertions that this code yields are captured in the pseudo-code in Figure 5. Please note that any access to a pointer or array has to be prepended with the pertaining assertion, including the loop guards. The precondition of the method is translated into **assume** commands.

```
1: INPUT: a: array[][] of int
2: VAR i, j: int;
3: assume a ≠ null;
4: assume ∀m : int•m ≥ 0 ∧ m < a.length ⇒ a[m] ≠ null;
5: i ← 0;
6: while  assert a ≠ null; i < a.length do
7:    j ← 0;
8:    while assert (a ≠ null ∧a[i] ≠ null ∧0 ≤ i ∧ i < a.length); j < a[i].length do
9:       assert 0 ≤ i ∧ i < a.length;
10:      assert 0 ≤ j ∧ j < a[i].length;
11:      assert a ≠ null ∧a[i] ≠ null
12:      a[i][j] ← 0;
13:      j ← j + 1;
14:   end while
15:   i ← i + 1;
16: end while
```

Fig. 5. Example of code with assertions.

Analogously to the process in the previous example, we obtain the following invariants. Invariants $a \neq \text{null}$ is inferred for both loops. Further, the invariants for the outer loop will be:

$$a \neq \text{null} \wedge i < a.\text{length} \Rightarrow 0 \leq i$$
$$(j \geq 0 \wedge j < a[i].\text{length}) \wedge (0 \leq i \wedge i < \text{a.length}) \Rightarrow a[i] \neq \text{null}$$

And the invaraints for the inner loop will be:

$$a \neq \text{null} \Rightarrow 0 \leq i$$
$$a \neq \text{null} \wedge 0 \leq i \Rightarrow i < a.\text{length}$$
$$a \neq \text{null} \wedge 0 \leq i \wedge i < a.\text{length} \wedge a[i] \neq \text{null} \wedge j < a[i].\text{length} \Rightarrow 0 \leq j$$

The interesting case is the non-nullness of $a[i]$. The algorithm first suggests the following invariant for the inner loop:

$$(0 \leq j \wedge j < a[i].\text{length}) \wedge (0 \leq i \wedge i < \text{a.length}) \Rightarrow a[i] \neq \text{null}$$

which is indeed preserved by the inner loop. However, it is not preserved by the outer loop. If we perform the alteration by quantifying over i, we obtain the following:

$$\forall i' \bullet (0 \leq j \wedge j < a[i'].\text{length}) \wedge (0 \leq i' \wedge i' < a.\text{length}) \Rightarrow a[i'] \neq \text{null}$$

which is preserved by both loops and hence, it is inserted as an invariant for both loops.

6 Implementation

We have implemented the technique described by the method Analyze (Section 3) with the extension of the alterations described in Section 5. The implementation is build as a subcomponent of ESC/Java2, and utilizes the automated

theorem prover Simplify [6]. Based on our initial experiments with the implementation we have added the following enhancements to the algorithm.

Heuristic back-propagation. The back-propagation realized as described in Section 3.1 generates large formulas. Therefore we approximate the transformer wlp as follows. When we back-propagate a formula $\mathcal{I}$ over a node labeled with the command **assume** f or **assert** f, we first substitute the expression f in $\mathcal{I}$ for true, and then apply wlp only if $\mathcal{I}$ and f share at least one free variable.

Simplifications. Even with the heuristic back-propagation, the invariants often contain redundant information. To alleviate this problem we have implemented several straight-forward formula propositional simplifications and the following rule:

$$\frac{\forall x \bullet x = E \Rightarrow F, \text{ where } x \text{ is not free in E}}{F[x \mapsto E]}$$

Assertion breaking. Assertions often come as conjuncts of simpler expressions, e.g., $0 \leq j \wedge j < l$, and sometimes only one of the literals leads to a successful invariant discovery. Therefore, it has proven useful to break the assertion into the individual literals and back-propagate these individually.

The technique has proven successful in finding simple invariants; in particular invariants that guarantee that a certain variable is nonnegative, certain variable is non-null, or that a certain variable has the desired type (required by typecasting). On the other hand, for the technique be useful in practice we believe that taking into account assertions outside loops is necessary.

7 Related Work

One of the first documented implementation of invariant generation utilizing a theorem prover was realized by Suzuki and Ishihata [14]. Similarly to our work, their algorithm aims to prove that a given program does not violate array bounds.

The predominant approach to invariant generation is *abstract interpretation* introduced by Cousot and Cousot [5]. A popular variant of abstract interpretation is *predicate abstraction* [9], where the abstract space is formed by boolean expressions on a finite set of predicates.

One of the key issues in predicate abstraction is to come up with the appropriate set of predicates. An example of a technique that discovers predicates automatically is *counterexample refinement* [10]. This technique tries to refine the model whenever the current model is shown to be too coarse, i.e., when the model contains an error-trace in the abstract space that does not have a corresponding trace in the concrete space. This approach is similar to ours in the sense that is driven by the undesired behavior.

The counterexample refinement has proven suitable for implementation as it was applied to verification of C programs in the tools *BLAST*[2] and *SATABS*[3] [4]

Flanagan and Qadeer utilized predicate abstraction for loop invariant generation in ESC/Java [8]. In this work Flanagan and Qadeer enriched the technique by introducing skolem constants to be able to infer quantified invariants.

Abstract interpretation is used in the Spec# programming system[4] to infer loop invariants [1]; the implementation operates on the intermediate representation in the language Boogie PL. Chang and Leino, members of the Spec# team, in [3] describe how abstract interpretation can be used to infer object invariants. Further, the authors in [2] propose a technique that allows combining different abstractions.

8 Conclusion and Future Work

We have introduced a technique for loop invariant generation from assertions in the context of an automated theorem prover and a weakest precondition calculus. An advantage of the technique is that it requires relatively small number of calls to the theorem prover, compared to predicate abstraction for example. Moreover, it takes into account user's specifications, which focuses the technique on the relevant space of loop invariants. The technique is easy to implement as it does not rely on the underlying theories, such as arithmetics. On the other hand, this property is at the same time a disadvantage since exploiting the information specific to a particular domain enables inferring stronger invariants. It appears that the main disadvantage, however, is the "snowball effect" caused by the weakest precondition calculus increasing the size of the back-propagated formula.

We propose the flowing challenges for future work.

Invariant simplifications. How can we identify irrelevant parts of the suggested invariant and generate simpler invariants?

Invariant alterations. What are the alternations useful in practice? Can other invariant inference techniques be exploited?

Loop postcondition. In the presented work we utilize only the assertions that are inside the investigated loop. This approach could be leveraged by taking into account the assertions after the loop, i.e., the desired postcondition of the loop.

Feedback to the user. The presented implementation operates on the intermediate language. Therefore, the inferred invariants are difficult to interpret for the user. Thus, it is desirable to be able to translate the inferred invariants to the JML notation.

[2] http://mtc.epfl.ch/software-tools/blast
[3] http://www.verify.ethz.ch/satabs
[4] http://research.microsoft.com/specsharp

References

1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceeding of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, 2004.
2. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Proceeding of 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005.
3. Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants, 2005.
4. Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25, 2004.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press.
6. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
7. Extended Static Checker for Java version 2 (ESC/Java2). At `http://secure.ucd.ie/products/opensource/ESCJava2/`.
8. Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2002.
9. Suzanne Graf and Hassen Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of 9th International Conference on Computer Aided Verification (CAV'97)*. Springer–Verlag, 1997.
10. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2002.
11. Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, January 2005.
12. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.
13. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May 1999.
14. Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, 1977.