Automatic Verification of Loop Invariants

Olivier Ponsini, Hélène Collavizza, Carine Fédèle, Claude Michel, Michel Rueher

Automatic Verification of Loop Invariants[*]

Olivier Ponsini
olivier.ponsini@unice.fr

Hélène Collavizza
helene.collavizza@unice.fr

Carine Fédèle
carine.fedele@unice.fr

Claude Michel
claude.michel@unice.fr

Michel Rueher
michel.rueher@unice.fr

University of Nice–Sophia Antipolis, I3S/CNRS
BP 145, 06903 Sophia Antipolis Cedex, France

ABSTRACT

Loop invariants play a major role in program verification and drastically speed up processes like automatic test case generation. Though various techniques have been applied to automatic loop invariants generation, most interesting ones often generate only candidate invariants. Thus, a key issue, to take advantage of these invariants in a verification process, is to check that these candidate loop invariants are actual invariants. This paper introduces an original technique based on constraint programming for automatic verification of inductive loop invariants. This new approach is efficient to detect spurious invariants and nicely performs verification of valid invariants under boundedness restrictions. First experiments on classical benchmarks are very promising.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

1. INTRODUCTION

Software errors are an ever-increasing issue. As stated in a 2002 U.S NIST report [30], few products are shipped with levels of errors as high as in software — and this has a huge economical cost. Indeed, verification of software usually requires considerable human effort, and automated tools are needed to ease this tedious task. The main obstacle to automation lies in iteration, that is loop constructs of programming languages. Loops are difficult to reason about because the number of iterations they describe cannot always be statically determined. Moreover, this number may depend on the input data, thus defining so many combinations that the problem remains intractable.

One solution to this problem is to reason about loops independently of the number of iterations: *loop invariants* are logical statements that describe properties of a loop holding for all possible executions of the loop. As such, they play a major role in program verification. For instance, a sufficiently strong loop invariant can avoid unfolding the loop in bounded model-checking approaches. In some other verification approaches, e.g., theorem prover based [2, 10], it is even mandatory to provide such invariants. Loop invariants are also useful for testing, e.g., they can improve test-case generation [12].

Due to the importance and difficulty of the problem, there exists a large body of work on automatically generating sound invariants from program source code. Early works date back to the seventies (e.g., [32, 18]). Abstract interpretation based analyses [5, 6] proved to be particularly fruitful (e.g., the recent work of [22, 26]). Lately, some authors proposed replacing the fix-point computation of abstract interpretation by decision procedures, especially based on constraint solving [4, 27, 29]. Yet, automatic generation of correct by construction invariants remains challenging. Often, proposed tools show poor performances: they are not efficient and/or the generated invariants are too weak for most practical purposes.

In contrast, some recent approaches relax the constraint of soundness of the invariants so as to efficiently produce interesting *candidate* invariants. Verification that the candidates are sound invariants is postponed to a second step, and sometimes relies on another dedicated tool. Information from this verification step may be used to refine new candidates. For instance, the Daikon [9] tool implements this approach: based on a dynamic analysis, it infers invariant candidates from a set of execution traces of a given program. Candidates are formed from a predefined finite set of common patterns. The candidates hold over the observed executions, but another tool is needed to ensure they are true for all possible program executions. Various static analyses generate candidate invariants too. They all resort to heuristics. Some of these heuristics are long known and can be found for example in [31, 19, 21, 13], others are original work. For instance, the authors in [16] combine several analyses and heuristics to produce candidate loop invariants. Then, a proof planner tries to prove they hold. The undischarged goals in the proof are used for strengthening the candidates. The method of [20] focuses on a particular case of loops where the number of iterations is controlled

[*]This work was partially supported by the ANR-07-SESUR-003 project CAVERN and the ANR-07-TLOG-022 project TESTEC.

by lower and upper bounds: the "FOR-loops". The candidates are built from the given program postcondition by applying a weakening heuristic. Next, a verification condition ensuring their soundness is synthesized to be used in a theorem prover. Again in the setting of theorem proving, [28] adds new inference rules to the KeY system in order to produce candidate invariants. According to the authors, these rules may generate spurious invariants, which consequently need to be checked. In [25], the authors propose an iterative method for invariant generation. Starting from the postcondition and applying heuristics, each iteration generates one candidate invariant. Then, the Java PathFinder model checker decides the validity of the candidate. When the candidate is invalid, a counter-example is provided by the model checker that helps strengthening the next candidate. Finally, [11] details heuristics to generate candidate invariants from postconditions of programs. The authors suggest to use external tools like SMT solvers to decide the soundness of the candidates.

All these methods that produce candidate loop invariants need some kind of decision procedure to ensure the candidates are indeed invariant. This is also true for user provided loop invariants. Handcrafted invariants may contain errors, and if so, the error must be detected before further usage of the invariant. Conversely, user invariants may be a desired specification from which a programmer writes the loop body code: a decision procedure is then needed to verify that the written code meets the loop invariant. In this paper, we propose to use constraint solvers for the automatic verification of candidate loop invariants in single-loop imperative procedures. Our approach is based on a translation of Java-like methods annotated with JML statements into a constraint problem. This translation was described in [3] and implemented in the CPBPV system. Besides the implementation in CPBPV of a more generic handling of JML "exist" and "for all" quantifiers, our contributions are:

- the use of constraint solvers as a bounded decision procedure for inductive loop invariant verification;
- a set of experiments on classical benchmarks.

Our approach efficiently refutes spurious inductive loop invariants and produces a counter-example that is a complete test case leading to the violation of the invariant. Proving a valid invariant holds is sensitive to the size of the program variables domain and the size of arrays: it ranges from fast on small domains (typically 8-bit integers) to intractable on larger domains. Constraint programming (CP) is only a complete decision procedure over a finite subset of the integers, that is why we have to bound the domains. CP is less powerfull than theorem provers and, thus, CP cannot prove a property in general, but CP can prove properties under boundedness restrictions and disprove properties that a theorem prover cannot handle.

The next section introduces our approach on a typical example. Then, Section 3 gives the details of our method. The results of our experiments are shown in Section 4. Related work is presented in Section 5. Next, we discuss the advantages and limits of our approach in Section 6, and we conclude in Section 7.

2. MOTIVATING EXAMPLE

Listing 1: Sum of the first n integers.

```
1  /*@ requires n >= 0;
2    @ ensures \result == (n*(n+1))/2;
3    @*/
4  public static int sumN(int n) {
5    int i=0, s=0;
6    /*@ loop_invariant
7      @  (s == (i*(i-1))/2) && (i <= n+1);
8      @*/
9    /* erroneous invariant
10    *    (s == (i*(n-1))/2) && (i <= n+1);
11    */
12   while(i <= n) {
13     s = s + i;
14     i = i + 1;
15   }
16   return s;
17 }
```

In this section, we illustrate our approach on the example of the sum of the first n integers. A program computing this sum is showed in Listing 1. This is a Java method enhanced with JML specifications : the precondition (`requires` clause) and the post-condition (`ensures` clause). We suppose we need to verify the inductive loop invariant given at line 7. This invariant states that at each iteration of the loop body, variable `s` stores the sum of the first integers up to `i`−1 and that `i` will not increase over `n`+1. From axiomatic logic [15], we know that a loop invariant is inductive if it holds before entering the while loop (this is the base case), and if holding before the while loop it also holds after one execution of the loop body (the inductive case). In order to ensure these two cases, we build two constraint systems.

For the base case, the formula to be verified is built from the variables of the program ($\vec{x}$), the precondition (Pre), a suitable logical encoding of the initializations ($Init$) occurring before the loop (line 5), and the loop invariant (I) such that:

$$\forall \vec{x}(Pre \land Init \implies I) \tag{1}$$

which is equivalent to (applying a double negation):

$$\neg(\exists \vec{x}(Pre \land Init \land \neg I)).$$

The existential quantification of this latter formula suggests how a constraint solver can be used as a bounded decision procedure for the formula: we translate Pre, $Init$, and $\neg I$ into constraints over the variables $\vec{x}$, then the solver searches for a solution to $\exists \vec{x}(Pre \land Init \land \neg I)$. If no solution is found, then Formula (1) is true, which means that the invariant holds after the initializations for all valuation of the method input data. Otherwise, a solution is found, then Formula (1) is false and so is the spurious invariant; moreover, the solution is a counter-example providing a valuation of the method input and local data that falsifies the invariant. This counter-example is a test case that helps to precisely understand why the program does not meet the invariant. It may be used to correct the invariant — or the program, depending on context.

The translation from JML annotated Java programs into constraints is detailed in Section 3.2. For this example, it is straightforward and leads to the following constraint set over variables n, i, and s whose domain is the one of the Java integers:

$$\{n \geq 0, i = 0, s = 0, (s \neq (i * (i-1))/2) \vee (i > n+1)\}.$$

Several constraint solvers can be used to handle this system. In our implementation, we rely on two solvers from IBM/Ilog: Cplex[1], a MILP solver, and CP Optimizer, a nonlinear solver over finite domains. The strategy is to call the fast linear solver as often as possible and to resort to the more time demanding nonlinear solver only when necessary (more details are given in [3]). Here, the base case constraint system is solved in under a hundredth of a second (Section 4.3 contains the complete benchmark results). There is no solution, hence the candidate invariant is valid for the base case.

The same principles apply for the inductive case which implies checking the following formula:

$$\neg(\exists(\vec{x}\vec{x}')(I \wedge \mathit{Cond} \wedge \mathit{Body} \wedge \neg I[\vec{x}'/\vec{x}]))$$

where *Cond* is the loop condition, *Body* a logical encoding of the loop body statements, and $\vec{x}'$ are fresh variables introduced by the encoding of the body statements. Introduction of new variables is explained in Section 3.2.2, but roughly, it allows[2] $I[\vec{x}'/\vec{x}]$ to correctly reflect the values of the program variables after one execution of the loop body. In the example, it leads to the following constraint set over program variables n, i, and s, and fresh variables s_1, and i_1:

$$\left\{ \begin{array}{l} s = (i * (i-1))/2, i \leq n+1, i \leq n, s_1 = s + i, \\ i_1 = i + 1, (s_1 \neq (i_1 * (i_1 - 1))/2) \vee (i_1 > n+1) \end{array} \right\}.$$

There is no solution to this constraint system and, to ensure this, the solver enumerates on some variable domains. For 8 bit integers, it takes a few hundredths of second; for 16 bit integers, it takes 2.6 hours. However, let us consider the erroneous invariant given as a comment at line 10, where an occurrence of variable `i` has been replaced by variable `n` in the first conjunct. For 32-bit integers, our tool correctly shows the proposed invariant is true for the base case in a quarter of a second, and refutes it on the inductive case in again 0.25 s. The counter-example produced is for $n = 0$, $s = 1073741824$, $i = -2147483648$, $s_1 = -1073741824$, and $i_1 = -2147483647$.

These results illustrate the strong points and limits of our approach: refutation of spurious invariants is fast and produces a test case; proof of valid invariants is fast on small integer domains, but can be much longer on large domains.

3. PROPOSED APPROACH

The principle is to transform loop invariant verification in single-loop programs into assertion verification in loop-free programs. Then, we consider each execution path of the loop-free programs, i.e., each path through the programs' control flow graph. Number and length of execution paths are finite since programs are loop-free. Before and after each statement of a given path, we represent the possible program states with a finite set of finite-domain variables and constraints, i.e., relations on variables. Rules define how each statement modifies the set of possible program states by adding new constraints and variables. Let p be a program point and C_p the constraints that describe the possible states at point p. Then, proving that an assertion A holds at p is showing that $C_p \wedge \neg C_A$, where C_A is the translation into constraints of A, has no solution, i.e., there is no assignment of the program variables that violates the assertion.

The proposed approach takes place in a forward analysis framework: program paths are analyzed starting from the program beginning. It is based on our previous work on constraint-based bounded verification of Java programs [3], which was implemented in the CPBPV system. We first describe how loop invariant verification in single-loop programs reduces to assertion verification in loop-free programs. Next, we briefly present the relevant key points of the interpretation of program states as constraints, as done in CPBPV. Finally, we give a more detailed exposition of the treatment of JML quantifiers.

3.1 Constraint-based inductive loop invariant verification

In axiomatic logic [15], the Hoare triple to prove partial correctness of a while loop obeys the following rule:

$$\frac{\{b \wedge I\}\ \texttt{Body}\ \{I\}}{\{I\}\ \texttt{while(b) Body}\ \{\neg b \wedge I\}} \qquad (2)$$

In its premise, this rule also implicitly defines the inductive loop invariant I: it is an assertion such that every program state satisfying $b \wedge I$ leads, when executing `Body`, to a state satisfying I. In a single-loop program, `Body` is necessarily loop-free. Thus, verifying that I is an inductive invariant for the loop `while(b) Body` amounts to verifying that, being given the precondition $b \wedge I$, the assertion I holds after execution of the loop-free program `Body`.

In our constraint-based representation of states, starting from the states satisfying $b \wedge I$, the set of possible program states after executing `Body` is:

$$\{(\vec{x}, \vec{x}') \in D^n \times D^{n'} \mid C_{b \wedge I} \wedge C_{\texttt{Body}}\}$$

where $C_{b \wedge I}$, and $C_{\texttt{Body}}$ are translations into constraints of $b \wedge I$, and `Body` respectively; D is the variables' domain (typically 32-bit signed integers for Java `int` type); $\vec{x}$ is a tuple corresponding to the n program variables (Java method parameters and local variables); and $\vec{x}'$ a tuple corresponding to the n' fresh variables introduced by the translation of `Body` (see Section 3.2.2). In order to verify that a given assertion holds in a set of states, we proceed by refutation: we check that no state of the set satisfies the negation of the assertion. Hence, the following formula states that I is an inductive invariant for the loop `while(b) Body`:

$$\neg\left(\exists(\vec{x}\vec{x}')(C_{b \wedge I} \wedge C_{\texttt{Body}} \wedge \neg C_{I[\vec{x}'/\vec{x}]})\right) \qquad (3)$$

We refer to this formula as the *inductive case* of the loop invariant verification. It is built using CPBPV and its satisfiability is checked by off-the-shelf constraint-programming solvers called from CPBPV.

[1] Cplex is an optimization software package based on the simplex method and on MILP (Mixed-integer linear programming) techniques. Cplex solves integer programming problems, very large linear programming problems and quadratic programming problems. See `http://www-01.ibm.com/software/integration/optimization/cplex`

[2] Expression $t[y/x]$ denotes the substitution of y to x in t.

In addition, to be useful in the context of a particular program containing the loop statement, the loop invariant must also hold before the loop statement so that the conclusion of the axiomatic rule (2) applies. For this, we examine each execution path `Path` of the program leading to the loop statement. Since we only consider here programs with one loop, these paths are loop-free. Thus, verifying that I holds before the loop statement amounts to verifying that, for all `Path`, the assertion I holds after execution of the loop-free program `Path` being given the program precondition *Pre* (which may be empty).

In our constraint-based representation of states, starting from the states satisfying *Pre*, the set of possible program states after executing `Path` is:

$$\{(\vec{x}, \vec{x}') \in D^n \times D^{n'} \mid C_{Pre} \wedge C_{\texttt{Path}}\}$$

where C_{Pre}, and $C_{\texttt{Path}}$ are translations into constraints of *Pre*, and `Path` respectively; and $\vec{x}'$ is a tuple corresponding to the n' fresh variables introduced by the translation of `Path`. Again proceeding by refutation, the following formula states that I holds after executing path `Path`:

$$\neg\left(\exists(\vec{x}\vec{x}')(C_{Pre} \wedge C_{\texttt{Path}} \wedge \neg C_{I[\vec{x}'/\vec{x}]})\right) \qquad (4)$$

Checking this formula for each execution path leading to the loop statement is the *base case* of the loop invariant verification. Like the inductive case, it is built and checked using CPBPV and constraint-programming solvers.

3.2 Constraint interpretation of program execution paths

CPBPV uses constraint stores to represent both an execution path in a Java method and its specification, i.e., method precondition and postcondition. Execution paths are explored nondeterministically and on-the-fly. One strength of the tool is to contain the combinatorial explosion of possible paths by pruning unreachable execution paths as soon as the corresponding constraint stores are inconsistent. CPBPV imposes bounds on the domain of variables, which are all signed integers, and on the number of elements in arrays. Since we only deal here with loop-free paths, we are not concerned by bounds on the length of the paths. CPBPV was designed for partial correctness verification. The partial correctness, under the boundedness restrictions, of a Java method is ensured if each constraint store, built from the precondition, a path in the method, and the negation of the postcondition, has no solution. To search for a solution, CPBPV calls several constraint solvers in sequence, starting with the fastest ones, but likely to find a spurious solution, up to more costly exact solvers if necessary.

A fragment of the programming language syntax accepted by CPBPV is presented in Section 3.2.1. For each programming language syntactic construct, a rule defines the transition between program states. Section 3.2.2 shows the main rules needed to understand this paper. The interested reader may refer to [3] for the complete set of rules and a more detailed description of CPBPV.

3.2.1 Syntax

We begin with the abstract syntax of programs, which are lists L of statements S, where A is the set of arrays and V the set of variables:

$$\begin{array}{l} L ::= S; L \mid \epsilon \\ S ::= A[E] \leftarrow E \mid V \leftarrow E \mid \textit{if } B\ S \mid \textit{while } B\ S \mid \{L\} \\ S ::= \textit{return } E \mid \textit{assert}(B) \mid \textit{enforce}(B) \end{array}$$

Next are the Boolean expressions B and integer expressions E:

$$\begin{array}{l} B ::= E > E \mid E \geq E \mid E = E \mid E \neq E \mid E \leq E \mid E < E \\ B ::= \neg B \mid B \wedge B \mid B \vee B \mid B \implies B \mid \textit{true} \mid \textit{false} \\ E ::= V \mid A[E] \mid E + E \mid E - E \mid E \times E \mid E/E \end{array}$$

Here is the abstract syntax of constraints C built from the constraint expressions E^+:

$$\begin{array}{l} C ::= E^+ > E^+ \mid E^+ \geq E^+ \mid E^+ = E^+ \mid E^+ \neq E^+ \\ C ::= E^+ \leq E^+ \mid E^+ < E^+ \mid true \mid false \\ C ::= \neg C \mid C \wedge C \mid C \vee C \mid C \implies C \end{array}$$

Finally, the expressions E^+ built from the constraint variables V^+, and the constraint arrays A^+ are described by:

$$\begin{array}{l} E^+ ::= V^+ \mid A^+[E^+] \mid E^+ + E^+ \mid E^+ - E^+ \\ E^+ ::= E^+ \times E^+ \mid E^+/E^+ \end{array}$$

3.2.2 Semantics

Before giving a few examples of the semantics rules that derive program states from program statements, we introduce the representation of program states and some notations.

Program states. As there is no notion of sequence in a constraint store, we cannot use a single constraint variable to represent the different values a program variable may take along an execution path. A classical solution is to introduce new variables and transform the original program so that every program variable is assigned once. This technique is similar to Static Single Assignment (SSA) transformation [7]. However, in CPBPV, this transformation is done on-the-fly while analyzing a given execution path and amounts to simple substitutions of the newly introduced variables in program expressions. The new variables are named from the program variable they replace by adding a subscript number incremented each time the corresponding program variable is assigned. We note $\sigma[x_i/x]$ the substitution function σ where x is now mapped to x_i, i.e., $\sigma[x_i/x](x) = x_i$, and $\sigma[x_i/x](y) = \sigma(y)$ if $y \neq x$. Function *inc* increments variables subscript numbers, e.g., $inc(x_i) = x_{i+1}$. We also note ρ a polymorphic function to transform program expressions into constraints. Since the two languages are so similar, this transformation is straightforward and we do not detail it here.

The set of possible states of a program, noted $\langle l, \sigma, cs\rangle$, is defined by a list l of statements that remains to execute, a substitution function σ, and a constraint store cs. A constraint store is made up of a set of constraint variables, in which we distinguish scalar V^+ and array A^+ variables, and a set C of constraints on the variables that reads as the conjunction of the constraints it contains. We note $cs \wedge c$ the addition of the constraint c in the set of constraints of cs. We also note $cs \cup v$ the addition of variable v in the set of scalar or array variables depending on whether v is scalar or array. When l is empty, we may denote the possible program states from the constraint store only, as in Section 3.1.

Rules. Now, we give the semantics rules for conditional statement, and assignments. There are two rules for the conditional statement *if b s* depending on whether the constraint c_b associated with b is consistent or not with the constraint store. When it is consistent, the store is augmented with c_b and the s is executed.

$$\frac{cs \wedge (\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ \ b\ s\ ;\ l, \sigma, cs\rangle \longmapsto \langle s\ ;\ l, \sigma, cs \wedge (\rho\ \sigma\ b)\rangle}$$

When the negation $\neg c_b$ is consistent with the store, then the constraint store is augmented with $\neg c_b$ and s is skipped.

$$\frac{cs \wedge \neg(\rho\ \sigma\ b) \text{ is satisfiable}}{\langle \mathit{if}\ \ b\ s\ ;\ l, \sigma, cs\rangle \longmapsto \langle l, \sigma, cs \wedge \neg(\rho\ \sigma\ b)\rangle}$$

Both rules may apply since the initial set of states $\langle l, \sigma, cs\rangle$ may contain some program states satisfying the condition b and others violating it. In this case, CPBPV analyzes independently the two branches created in the execution path.

An assignment to a scalar variable v introduces a new constraint variable. This new variable will be used in place of v in the remaining program, so the substitution mapping is updated accordingly. Then the new variable is constrained to the value of the assignment right-hand side expression.

$$\frac{v' = \mathit{inc}(\sigma_1\ v) \quad \sigma_2 = \sigma_1[v'/v] \quad c \equiv (\rho\ v') = (\rho\ \sigma_1\ e)}{\langle v \leftarrow e\ ;\ l, \sigma_1, cs\rangle \longmapsto \langle l, \sigma_2, (cs \cup v') \wedge c\rangle}$$

Assignment to an element *elt* of an array a creates a new constraint variables array. All the new constraint variables in the array are set equal to their corresponding element in a, except for the one corresponding to *elt*, which is constrained to the value of the assignment right-hand side expression.

$$\frac{\begin{array}{l} a' = \mathit{inc}(\sigma_1\ a) \\ \sigma_2 = \sigma_1[a'/a] \\ c_1 \equiv (\rho\ a'[\rho\ \sigma_1\ \mathit{ind}]) = (\rho\ \sigma_1\ e) \\ c_2 \equiv\ \ \forall i \in [0, \mathit{length}(a)[\\ \qquad i \neq (\rho\ \sigma_1\ \mathit{ind}) \implies (\rho\ a'[i]) = (\rho\ (\sigma_1\ a)[i]) \end{array}}{\langle a[\mathit{ind}] \leftarrow e\ ; l, \sigma_1\ , cs\rangle \longmapsto \langle l, \sigma_2, (cs \cup a') \wedge c_1 \wedge c_2\rangle}$$

In this rule, index *ind* does not have to be statically known.

3.3 JML quantifiers

Among the JML quantifiers, we support the universal and existential quantifiers, restricted to quantification over integer values. The JML syntax of the universal quantifier is: `(\forall int` k`;` B_R`;` B_Q`)`, where k is the quantified variable, B_R is the range predicate, and B_Q is the quantified predicate. This means that the quantified predicate holds for all potential values of the quantified variables that satisfy the range predicate. Similarly, the syntax of the existential quantifier is: `(\exists int` k`;` B_R`;` B_Q`)`, which means that B_Q holds for some values of k that satisfy B_R.

In our approach, integers are bounded and so is the quantified variable in a quantifier expression. When a reasonably small bound is known, the quantifier can be eliminated as explained in Section 3.3.1. Otherwise, Section 3.3.2 gives a general technique to deal with quantifiers.

3.3.1 Known bound quantifiers

Often, a small bound on the quantified variable can be inferred from the range predicate, or the quantified predicate. In particular, when quantification ranges over arrays, range predicates of the form $i_1 \leq k \wedge k \leq i_2$, where i_1 and i_2 are statically known integer values, are very common. In these cases, we expand a quantifier expression, substituting possible values to the quantified variable, as a conjunction of constraints for a universal quantifier, or as a disjunction of constraints for an existential quantifier.

For instance, let us consider the expression `(\forall int` k`;` $0 \leq k \wedge k < t.\texttt{length} - 1$`;` $t[k] \leq t[k+1]$`)` from Listing 5. Since array t has a known bounded size, say 3 for this example, we can expand the expression into:

$$(t[0] \leq t[1]) \wedge (t[1] \leq t[2]).$$

Let us turn to another example from Listing 5: `(\forall int` k`;` $0 \leq k \wedge k < \mathit{left}$`;` $t[k] \neq x$`)`. We do not know *a priori* any interesting bound for the variable *left*. However, k indexes the array t in the quantified predicate, thus, k should always be less than $t.\texttt{length}$. This also gives an upper bound for *left* that we can check by adding the assertion $\mathit{left} \leq t.\texttt{length}$ to the constraint store. Hence, the quantified expression can be expanded into (with $t.\texttt{length} = 3$ as before):

$$\begin{array}{rl} & \mathit{left} = 1 \implies t[0] \neq x \\ \wedge & \mathit{left} = 2 \implies (t[0] \neq x) \wedge (t[1] \neq x) \\ \wedge & \mathit{left} = 3 \implies (t[0] \neq x) \wedge (t[1] \neq x) \wedge (t[2] \neq x) \end{array}$$

3.3.2 Unknown bound quantifiers

When no other bound than that of the integers domain is known, expanding quantifier expressions is too costly. In the case of universal quantification, we transform the JML form into its logical equivalent: $\forall k(Q)$, with $Q = (B_R \implies B_Q)$. Let cs be the constraint store of the current possible program states, built from the set of variables V and the constraint C, and let σ be the current substitution function. To prove that the assertion $\forall k(Q)$ holds, we proceed by refutation and build a formula similar to Equations (3) and (4):

$$\begin{array}{rl} & \neg\,(\exists \vec{v}(C \wedge \neg(\forall k(\sigma Q)))) \\ \equiv & \neg\,(\exists \vec{v}(C \wedge (\exists k(\sigma \neg Q)))) \end{array}$$

Without loss of generality, we assume k does not appear in C, then we can add k to V and rewrite the last formula into:

$$\neg\,(\exists \vec{v}(C \wedge (\sigma \neg Q)))$$

Finally, this formula can be solved by a constraint solver.

If we try to do the same with existential quantification, we end up with a formula like:

$$\begin{array}{rl} & \neg\,(\exists \vec{v}(C \wedge \neg(\exists k(\sigma Q)))) \\ \equiv & \neg\,(\exists \vec{v} \forall k(C \wedge (\sigma \neg Q))) \end{array}$$

Because of the presence of a universal quantifier, we cannot directly solve this formula with a constraint-programming solver. Nevertheless, we can also rewrite it as:

$$\begin{array}{rl} & \neg\,(\exists \vec{v}(C \wedge \neg(\exists k(\sigma Q)))) \\ \equiv & \forall \vec{v}(\neg C \vee \exists k(\sigma Q)) \\ \equiv & \forall \vec{v}(C \implies \exists k(\sigma Q)) \end{array}$$

The last formula states that for all tuple solution of C, there must exist a k that is solution of Q. We can solve this formula with constraint-programming solvers by applying the following strategy:

1. solve $\exists \vec{v}(C)$ enumerating all the solutions;
2. for each solution found, report the values in σQ to obtain Q', and solve $\exists k(Q')$.

Depending on the number of solution of the first resolution, this strategy can be computationally expensive, if not intractable.

We have presented here the treatment of quantifiers that appear in assertions and postconditions. A similar reasoning applies to JML quantifiers that appear in preconditions or in loop invariants. However, universal instead of existential quantification is the computationally expensive case, since we do not negate the quantifier expression in these cases as we do in assertions and postconditions.

4. EXPERIMENTS

We performed experiments on a set of Java programs known in software verification to be challenging; some of them are inspired by the gallery of certified programs of Why [10] (`http://why.lri.fr/examples`). For each one, we give the specification of the program, as pre and postconditions in JML notation. In this notation, the keyword `\result` denotes the return value of the associated method. As a reference, we also provide a handcrafted loop invariant sufficient to imply the post condition.

4.1 Program set

The first program, in Listing 1, computes the sum of the first n integers. Although, the code itself only contains linear arithmetic expressions, the specification requires nonlinear operations (multiplications between variables), as well as the loop invariant. Listing 2 is a slight variation of the previous program, it computes the sum of the integers from p to n. This variation is interesting because it introduces a second unknown bound, p, in the summation.

Listing 2: Sum of the integers from p to n.

```
/*@ requires n >= 0 && p >= 0 && p <= n;
  @ ensures \result == ((n*(n+1))/2)
  @                    - (((p-1)*p)/2); @*/
public static int sumPN(int p, int n) {
  int i=p, s=0;
  /*@ loop_invariant
    @   (s == ((i*(i-1))/2) - (((p-1)*p)/2))
    @  && (i <= n+1); @*/
  while(i <= n) {
    s = s + i;
    i = i + 1;
  }
  return s;
}
```

Our third example, in Listing 3, computes an integer approximate square root such that $\text{isqrt}(x) * \text{isqrt}(x) \leq x < (\text{isqrt}(x) + 1) * (\text{isqrt}(x) + 1)$. This time, not only the specification contains nonlinear expressions, but also the code does in the assignments to variable `z`.

Listing 3: Integer square root.

```
/*@ requires x >= 0;
  @ ensures \result * \result  <= x
  @  && x < (\result + 1) * (\result + 1); @*/
public static int sqrt(int x) {
  int y, z;
  if(x == 0) {
    return 0;
  } else if(x <= 3) {
    return 1;
  } else {
    y = x;
    z = (x+1)/2;
    /*@ loop_invariant
      @   (x > 3)
      @  && (z > 0)
      @  && (y > 0)
      @  && (z == (x/y+y)/2)
      @  && (x < (y+1)*(y+1))
      @  && (x < (z+1)*(z+1)); @*/
    while(z < y) {
      y = z;
      z = ((x/z) + z)/2;
    }
    return y;
  }
}
```

The next example, in Listing 4, computes the same approximate square root as the previous example, except that the program does not contain any nonlinear expression.

Listing 4: Integer square root without nonlinear expressions.

```
/*@ requires x >= 0;
  @ ensures \result >= 0
  @  && (\result * \result) <= x
  @  && x < ((\result + 1) * (\result + 1)); @*/
public static int sqrt_bis(int x) {
  int count = 0, sum = 1;
  /*@ loop_invariant
    @   count >= 0
    @  && x >= (count*count)
    @  && sum == ((count+1)*(count+1)); @*/
  while(sum <= x) {
    count = count + 1;
    sum = sum + 2*count + 1;
  }
  return count;
}
```

The last example, in Listing 5, introduces two important features: arrays, and quantification. This example only illustrates universal quantification, but we also handle existential quantification. The program performs a binary search in the given array `t`, looking for the value `x`. If the value `x` is in the array, the method returns the index of an element of the array whose value is `x`; otherwise, it returns −1.

Listing 5: Binary search.

```
/*@ requires
  @   (\ forall  int  k;  0<=k && k<t.length−1;
  @       t [k]<=t[k+1]);
  @ ensures
  @   \result  != −1 ==> t[\result] == x
  @  && \result == −1 ==>
  @         (\ forall  int  k;  0<=k && k<t.length; t[k]!=x);
  @*/
public static int bsearch(int[] t, int x) {
  int result = -1, mid = 0, left = 0;
  int right = t.length - 1;
  /*@ loop_invariant
    @   (\ forall  int  k;  0<=k && k<t.length−1;
    @       t [k]<=t[k+1])
    @  && (\forall  int  k;  0<=k && k<left; t[k]!=x)
    @  && (\forall  int  k;  right<k && k<t.length;
    @         t [k]!=x)
    @  && result != −1 ==> t[result] == x
    @  && left >= 0 && right <= t.length − 1; @*/
  while((result == -1) && (left <= right)) {
    mid = (left + right)/2;
    if(t[mid] == x) {
      result = mid;
    } else if(t[mid] > x) {
      right = mid - 1;
    } else {
      left = mid + 1;
    }
  }
  return result;
}
```

4.2 Candidate invariants

For our experiments, candidate invariants come from four different sources:

Daikon This tool infers candidate invariants from program execution traces. We produced execution traces by running one hundred random tests on each Java program. Each set of traces was analyzed by the Daikon tool which generated several candidate invariants guaranteed to hold on every trace of a set. Daikon has no predefined mechanism for loop invariant generation; but the reference manual suggests to place a call to a procedure doing nothing but returning immediately so that invariants could be collected at the call point. We placed such calls at the head and foot of the loop body. The candidate invariants were obtained by merging the assertions generated at both call points. In case of conflict between assertions, e.g., $x = 1$ at the head and $x < 1$ at the foot, we formed a new assertion satisfying both conflicting assertions, e.g., $x \leq 1$.

InvGen This tool [14] produces correct-by-construction invariants. As such, these invariants do not have to be verified, but they contribute an unbiased source of correct invariants. We used the C front-end to InvGen by means of a quick translation from Java. InvGen does not handle nonlinearity and arrays, so we could only apply it on three of our five program examples.

Heuristics This set of candidate invariants were made by applying well-known heuristics to program postconditions, such as "*replacing a constant in the postcondition by a variable*" [13] or changing the relational operators. This mimics some candidate invariant generation techniques (references are given in Section 1) for which no implementation was available and simple to reuse.

User This set of candidate invariants is made up of the invariants as used in manual proofs of the programs. Hence, they are strong enough to imply the program postcondition. These invariants are given in JML notation in the program listings.

The complete list of invariants is available at this URL: `http://users.polytech.unice.fr/~rueher/invariants/`.

4.3 Results

All benchmarks were run on a 64 bit Linux *quad-core* Intel Xeon (3.16 GHz) platform with 16 GB of RAM. However, our tool was run on a single core and did not take advantage of the three supplementary cores. Memory was never a concern and we stayed far below the platform capacity. Implementation in CPBPV is based on Cplex. Cplex works with floating point numbers and thus is unsafe. To address this problem, we use the simple and cheap procedure introduced by Neumaier et al. [23] to get rigorous answers.

Execution times for each source of invariants are gathered in Table 1. They are further detailed between the valid invariants (True) and the spurious ones (False). Table 1 contains:

- the number (nb) of invariants for the considered category;
- the number and percentage of invariants verified in less than 1 second each (< 1 s);
- the number and percentage of invariants verified in between 1 second and 1 minute each (< 1 min);
- the number and percentage of invariants verified in between 1 and 10 minutes (< 10 min);
- the number and percentage of invariants whose verification timed out — beyond 10 minutes, we did not record the precise time and considered it as a time-out (TO).

A large majority of invariants are verified in less than a second, at the exception of valid invariants provided by the user or heuristically produced from the postcondition. These valid invariants convey more meaning about the loop and the program than those generated by Daikon or InvGen do. They are not "simple" consequences of the program constraints and require enumerating over the program variable domains. Moreover, all the candidates generated by Daikon and InvGen are linear, whereas the valid invariants that times out are mostly nonlinear. Spurious invariants predominates in the generated candidates, whether it is by heuristics or by Daikon. The good news is that they are mostly very quickly discarded by our method. This shows that the proposed approach can be efficiently used as a filter for the candidate invariants automatically generated. The

Table 1: Execution times according to invariants' source and validity.

		Time				
Source		nb	< 1 s	< 1 min	< 10 min	TO
Daikon	True	7	7 (100%)	0	0	0
	False	41	37 (90.2%)	0	0	4 (9.8%)
	Total	48	44 (91.7%)	0	0	4 (8.3%)
InvGen	True	3	3 (100%)	0	0	0
Heuristics	True	7	4 (57.1%)	0	0	3 (42.9%)
	False	118	103 (87.3%)	3 (2.5%)	1 (0.9%)	11 (9.3%)
	Total	125	107 (85.6%)	3 (2.4%)	1 (0.8%)	14 (11.2%)
User	True	7	0	2 (28.6%)	0	5 (71.4%)

few spurious invariants that times out are nonlinear, or relate to arrays. In this latter case, the number of variables weakly constrained to enumerate over (here 10) may explain the bad performance.

Table 2 gathers the data according to the validity of the candidate loop invariants, and to the linearity of the constraint problem built to check this validity. This table confirms our previous observations. The approach performs very well in detecting spurious invariants. Results are more contrasted with valid invariants: from the one hand, complex invariants as produced by hand, involving several program variables and logical operators, may cause the resolution to time-out, on the other hand, simpler valid invariants can be checked as quickly as spurious ones. As regards linearity, most of the time-outs occur on nonlinear constraint systems. Nonlinear solvers may clearly be less efficient than linear ones, however, our approach may still performs well in presence of nonlinear spurious invariants. From the 82 nonlinear problems solved in less than 1 second, 81 involve spurious invariants: as a percentage of the nonlinear spurious invariants, almost 82% are refuted in less than a second.

The figures given in the first two tables are for 32-bit integers. Table 3 shows how execution times vary when we reduce the size of the integers to 16 bits and 8 bits. A smaller integer domain improves the performances of the approach and drastically decreases the number of time-out. Of course, it can only increase confidence in a candidate invariant, not ensure its validity. Yet, it may be enough to discard a spurious invariant, and it is sound then: with 8-bit integers, 99.4% of the spurious invariants are discarded in less than one second each.

5. RELATED WORK

Section 1 briefly presented invariant generation. Compared to correct invariant generation based on constraints, as far as constraint solving is concerned, the problem of invariant verification is simpler:

- verification only deals with the program's unknowns, whereas correct generation needs to deal with the unknown parameters of the invariant pattern in addition;
- verification of a linear invariant for a linear program can be done with a fast linear solver, whereas correct generation of a linear invariant for a linear program can still require solving a prohibitive nonlinear system.

Few works specifically deal with verifying candidate loop invariants. Close to our work is [8] in which candidate program invariants generated by Daikon are checked using constraint solving. The considered invariants are akin to program postconditions, not loop invariants. As the authors point out, their treatment of loops results in unusual constraint systems making resolution inefficient. Beyond the peculiarities that differentiate our translation of programs into constraints and theirs, our work focuses on loop invariant verification and thus avoids the complications due to loops. Indeed, our results show that constraint solvers can be very efficient in this context. Another approach to the static verification of candidate program invariants, again produced by Daikon, was investigated in [24]. In this approach, the static checker ESC/Java is called on the program source code annotated with candidate invariants. As the proof engine of ESC/Java is an automatic theorem prover, when a proof fails, one cannot tell whether the candidate invariant is false or additional lemmas are needed to complete the proof. Moreover, as stated by the authors, ESC/Java itself is unsound, so even if the proof succeeds, the candidate invariant may still be spurious. Our work also shares ideas with [17] in interpreting the semantics of Java programs as constraint systems. But it is mainly focused on heap and shape analyses (cyclic lists...), and it uses a SAT solver to solve Boolean constraints.

6. DISCUSSION

The results in Section 4.3 show that our approach performs especially well on spurious candidate loop invariants. Linear problems are also dealt with efficiently thanks to the highly optimized Cplex solver from IBM/Ilog, even in cases where some enumeration is required to fully explore the search space. In presence of nonlinearity, we still have very good time performances in two situations:

- a linear subset of the nonlinear system is sufficient to prove that the invariants hold: if the linear subset is unsatisfiable, so is the complete nonlinear system;
- the candidate is spurious: the nonlinear solver does not need to fully explore the search space and may quickly find a counter-example.

The longest timings occur when the nonlinear solver enumerates on the program variable domains to prove a valid candidate.

Table 2: Execution times according to validity and linearity.

	nb	< 1 s	< 1 min	< 10 min	TO
True	24	14 (58.3%)	2 (8.3%)	0	8 (33.4%)
False	159	140 (88.1%)	3 (1.9%)	1 (0.6%)	15 (9.4%)
Linear	76	72 (94.8%)	2 (2.6%)	0	2 (2.6%)
Nonlinear	107	82 (76.6%)	3 (2.8%)	1 (0.9%)	21 (19.7%)

Table 3: Execution times varying the integer domain size.

	nb	< 1 s	< 1 min	< 10 min	TO
8 bits	183	176 (96.2%)	5 (2.7%)	0	2 (1.1%)
16 bits	183	166 (90.7%)	9 (4.9%)	1 (0.6%)	7 (3.8%)
32 bits	183	154 (84.1%)	5 (2.7%)	1 (0.6%)	23 (12.6%)

Quantified assertions also have an impact on the performances. If we are able to infer the domain of the quantified variable, and it is acceptably small, we eliminate quantification by expanding assertions with guarded constraints. This is often possible on quantification iterating over array elements — since we impose a bound on array lengths, e.g., in the binary search program of Listing 5. Otherwise, unexpandable existential quantification in preconditions and unexpandable universal quantification in loop invariants require to solve two constraint systems: the second one being solved again for each solution of the first one. Most of the time, this is not reasonably tractable. In a similar way, assertions containing alternating existential and universal quantifiers also need the resolution of several constraint systems depending on the solutions of another resolution. Hence, the same conclusion applies and in practice our tool does not handle nicely alternation of quantifiers.

Our approach is bounded on the size of the integers and on the number of elements in arrays. As for the integers, this is not a limitation since on hardware integers are also bounded, and Java specifies the `int` data type is a 32-bit integer. Table 3 shows our approach efficiently checked most of the loop invariants for 32-bit integers. Moreover, refuting an invariant can often be done on a smaller domain: if a counter-example is found, it still holds in an including domain. However, our approach does not handle overflows. In Java, arrays are indexed by `int` values, so this also sets a bound on the number of elements in arrays. Yet, this is rather large and not tractable by our approach in the general case. Nevertheless, in practice, there are numerous situations where a smaller bound on the size of the arrays is known. Furthermore, even when a smaller bound is not known, verifying for some small array sizes increases confidence in the code, just as testing does. Verification of out of bounds array accesses is not yet implemented, but this can be simply done by adding new constraints on the index expressions.

Finally, other verification tools able to deal with sufficiently expressive program properties could also be used to verify loop invariants. As discussed in Section 5, tools based on an undecidable logic, such as ESC/Java [2] or Why [10], usually do not disprove a spurious candidate. Indeed, a failed proof attempt does not imply the proof is impossible. In contrast, boundedness is an adequate setting for refutation, and constraint programming, on which our approach is based, handles finite domains very efficiently. Other bounded model checking tools, like CBMC [1], should be able to disprove a spurious candidate too. However, CBMC computes only an error path: unlike our approach, it does not provide a counter-example with values for the input data when the invariant does not hold. A more detailed comparison remains to be done, nevertheless this paper demonstrates the feasibility and efficiency of an approach based on constraint solvers.

7. CONCLUSION

In this paper, we showed that constraint solvers could be used as an efficient bounded decision procedure to verify candidate loop invariants. A remarkable feature of our approach is to be able to refute spurious candidates, and not only prove valid ones. Moreover, a counter-example is then provided, which is a complete test case for the violated property. We presented some results on classical benchmarks which show that our approach is very efficient at refuting spurious candidates or when only linear expressions are involved. The approach is still exploitable in other situations.

In our work, we focused on single-loop programs, but the approach can be extended to programs with multiple loops, including nested loops. Yet, in such programs, multiple candidate invariants are mutually dependent. The impact on efficiency of this extension has to be studied in a future work.

Candidate loop invariants may come from various sources, e.g., user, or static and dynamic analyses. In particular, methods starting from the user provided program postcondition seem promising. Indeed, the postcondition already captures some of the semantics of the program and helps in finding invariants strong enough to prove the postcondition itself. In addition, we believe that the complete test case provided as counter-example for spurious invariants may greatly help in refining candidates. The next step of our work will be to build upon these methods to provide sound loop invariants to our bounded program verification tool CPBPV. Ideally, these loop invariants could replace the CPBPV loop unfolding process, or at least strengthen the constraint system at each unfolding.

8. REFERENCES

[1] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 168–176. Springer-Verlag, 2004.

[2] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Int. Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.

[3] H. Collavizza, M. Rueher, and P. V. Hentenryck. CPBPV: A constraint-programming framework for bounded program verification. *Constraint Journal*, 15(2):238–264, april 2010.

[4] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Int. Conf. on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 420–432, 2003.

[5] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Int. Symp. on Programming (ISOP'76)*, pages 106–130. Dunod, 1976.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symp. on Principles of Programming Languages (POPL'78)*, pages 84–96, 1978.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[8] T. Denmat, A. Gotlieb, and M. Ducassé. Proving or disproving likely invariants with constraint reasoning. In *Int. Workshop on Logic Programming Environments (WLPE'05)*, pages 1–13, 2005.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, dec 2007.

[10] J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Int. Conf. on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 173–177. Springer-Verlag, 2007.

[11] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. `http://arxiv.org/abs/0909.0884`, 2010.

[12] C. Gladisch. Verification-based test case generation for full feasible branch coverage. In *Int. Conf. on Software Engineering and Formal Methods (SEFM'08)*, pages 159–168. IEEE Computer Society, 2008.

[13] D. Gries. *The Science of Programming*. Springer, 1981.

[14] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 634–640. Springer-Verlag, 2009.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12(10):576–580, 1969.

[16] A. Ireland, B. J. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. In *Mexican Int. Conf. on Artificial Intelligence (MCAI'04)*, volume 2972 of *LNCS*, pages 190–201. Springer-Verlag, 2004.

[17] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Int. Symp. on Software Testing and Analysis (ISSTA'00)*, pages 14–25. ACM, 2000.

[18] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[19] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

[20] S. Kauer and J. F. Winkler. Mechanical inference of invariants for for-loops. *Journal of Symbolic Computation*, 2009.

[21] B. Meyer. A basis for the constructive approach to programming. In *IFIP Congress 1980*, pages 293–298, 1980.

[22] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[23] A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer programming. *Mathematical Programming*, 99(2):283–296, 2004.

[24] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Int. Symp. on Software Testing and Analysis (ISSTA'02)*, pages 232–242. ACM, 2002.

[25] C. S. Păsăreanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Int. SPIN Workshop on Model Checking Software (SPIN'04)*, volume 2989 of *LNCS*, pages 164–181. Springer-Verlag, 2004.

[26] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64:54–75, 2007.

[27] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *Symp. on Principles of Programming Languages (POPL'04)*, pages 318–329. ACM, 2004.

[28] P. H. Schmitt and B. Weiß. Inferring invariants by symbolic execution. In *Int. Verification Workshop (VERIFY'07)*, volume 259 of *CEUR Workshop Proc.*, pages 195–210. CEUR-WS.org, 2007.

[29] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *Conf. on Programming Language Design and Implementation (PLDI'09)*, pages 223–234. ACM, 2009.

[30] The Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards and Technology, 2002. `http://www.nist.gov/public_affairs/releases/n02-10.htm`.

[31] B. Wegbreit. Heuristic methods for mechanically deriving inductive assertions. In *Int. Joint Conf. on Artificial Intelligence (IJCAI'73)*, pages 524–536, 1973.

[32] B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–112, 1974.