Automatic Verification of Loop Invariants

Olivier Ponsini, Hélène Collavizza, Carine Fédèle, Claude Michel and Michel Rueher
University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
email: firstname.lastname@unice.fr

***Abstract*—Loop invariants play a major role in program verification. 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 a new technique based on constraint programming for automatic verification of inductive loop invariants. This approach is efficient to detect spurious invariants and is also able to verify valid invariants under boundedness restrictions. First experiments on classical benchmarks are very promising.**

I. Introduction

A major obstacle to automatic software verification lies in iteration, that is loop constructs of programming languages. Loops are difficult to reason about because the number of iterations cannot always be statically determined. A 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 proving [1], it is even mandatory to provide such invariants. Loop invariants are also useful for testing, e.g., they can improve test-case generation [2].

Despite numerous works, automatic generation of sound invariants from program source code remains challenging: proposed tools are not efficient and-or generated invariants are too weak for most practical purposes. Some recent approaches relax the soundness constraint so as to efficiently produce interesting *candidate* invariants by analysis of execution traces [3] or by static analysis based on heuristics [4]–[7]. Verifying that the candidates are sound invariants is postponed to a second step and requires some kind of decision procedure. Such a decision procedure is also useful 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 use constraint solvers for automatic verification of candidate loop invariants in single-loop imperative procedures. We use the translation of Java-like methods annotated with JML statements into constraint problems introduced in CPBPV [8], a constraint programming framework for bounded program verification. Our contributions are:

- the use of constraint solvers as a bounded decision procedure for checking the validity of inductive loop invariant;
- a set of experiments on classical benchmarks inspired by the gallery of Why [1], using existing tools (Daikon and InvGen) for invariant generation.

In addition, we implemented a more generic handling of JML "exist" and "for all" quantifiers than the one of CPBPV. Our approach efficiently refutes spurious inductive loop invariants and produces a counter-example that is a real test case leading to the violation of the invariant. Proving a valid invariant holds is sensitive to the size of the program variable domains and the size of arrays: it ranges from fast on small domains to intractable on larger domains.

Checking candidate invariants is close to the more general problem of proving verification conditions in theorem proving. However, the boundedness hypotheses inherent to our approach render our validity checks unsuitable to theorem proving, only refutation results could be useful in this context.

The next section introduces our approach on a typical example. Then, Section III gives some details of our approach. Experimental results are shown in Section IV. Related work is discussed in Section V, and Section VI concludes.

II. Motivating examples

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 Fig. 1. This is a Java method enhanced with JML specifications : the precondition (`requires` clause) and the post-condition (`ensures` clause). We suppose that we need to verify the inductive loop invariant given at line 6. 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 [9], 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). To check whether an invariant holds, we build two constraint systems corresponding to these two cases.

For the base case, the formula to be verified is built from the variables of the program ($\vec{x}$), the precondition (Pre), a suitable

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

Fig. 1. Sum of the first n integers

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

logical encoding of the initializations ($Init$) occurring before the loop (line 5), and the loop invariant (I) such that:

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

which is equivalent to (applying a double negation):

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

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 $\exists \vec{x}(Pre \wedge Init \wedge \neg I)$. If no solution is found, then Formula (1) is true, which means that the invariant holds after the initializations for all valuations of the method input data. If a solution is found, then Formula (1) is false and so is the 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 understand why the program does not meet the invariant. It may be used to correct the invariant or the program.

The translation from JML annotated Java programs in this example is straightforward and leads to the following constraint set over variables n, i, and s, whose domain is the one of the Java integers:

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

This constraint system is trivially inconsistent. Our approach relies on two constraint solvers: a MILP (Mixed-integer linear programming) solver and a constraint programming (CP) solver over finite domains. Here, the CP solver detects the inconsistency at once. Since there is no solution, the candidate invariant is valid for the base case.

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

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

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. Roughly, introduction of new variables allows $I[\vec{x}'/\vec{x}]$ 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:

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

This constraint system is nonlinear and cannot be easily simplified. The CP solver can still detect instantly the inconsistency on small variable domains, but it will need more time on larger domains since some enumeration is required.

As another example, let us consider the erroneous invariant given as a comment at line 8, where an occurrence of variable `i` has been replaced by variable `n` in the first conjunct. The CP solver shows instantly—for 32-bit integer variable domains—that the proposed invariant is true for the base case and false for the inductive case. Our system produces the counter-example: $n = 0$, $s = 1073741824$, $i = -2147483648$, $s_1 = -1073741824$, and $i_1 = -2147483647$.

These examples illustrate the strong points 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.

III. 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 programs, i.e., each path through the programs' control flow graph. Number and length of execution paths are finite because 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. Proving that an assertion A holds at p is done by refutation: it amounts to show 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 [8], which was implemented in the CPBPV system. The interested reader may refer to this work for the programming language syntax accepted by CPBPV and complete set of transition rules between program states. Here, we first recall the relevant key points of the interpretation of program states as constraints, as done in CPBPV, for loop invariant verification in single-loop programs. Next, we describe how we handle JML quantifiers.

A. Constraint-based verification

We use 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. To contain the combinatorial explosion of possible paths, we prune unreachable execution paths as soon as the corresponding constraint stores are inconsistent. We impose bounds on the domain of variables, which are all signed (up to) 32-bit integers, and on the number of elements in arrays. As for integers, this is not a limitation because hardware integers are also bounded and the Java **int** data type is a 32-bit integer. However, our approach does not handle overflows. As for array size, in practice, there are numerous situations where a bound is known. Furthermore, even when a bound is unknown, verifying for some small array sizes increases confidence in the code, just as testing does. Verifying out of bounds array accesses is not yet implemented, but this can be done by adding new constraints on the index expressions. Since we only deal here with loop-free paths, we are not concerned by bounds on the length of the paths. To search for a solution, we call 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.

B. 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. The meaning is 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 III-B1. Otherwise, Section III-B2 gives a general technique to deal with quantifiers.

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.length` $- 1$`;` $t[k] \leq t[k+1]$`)`. 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: `(\forall` **`int`** k`;` $0 \leq k \wedge k <$ *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`.length`. This also gives an upper bound for *left* that we can check by adding the assertion *left* $\leq t$`.length` to the constraint store. Hence, the quantified expression can be expanded into (with t`.length` $= 3$ as before):

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

2) Unknown bound quantifiers: When no other bound than that of the integer 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 C be the current conjunction of program constraints, built from the set of variables $\vec{v}$. To prove that the assertion $\forall k(Q)$ holds, we proceed by refutation and build the formula:

$$\neg\,(\exists \vec{v}(C \wedge \neg(\forall k(Q)))) \quad \equiv \quad \neg\,(\exists \vec{v}(C \wedge (\exists k(\neg Q))))$$

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:

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

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:

$$\neg\,(\exists \vec{v}(C \wedge \neg(\exists k(Q)))) \quad \equiv \quad \neg\,(\exists \vec{v} \forall k(C \wedge (\neg Q)))$$

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:

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

The last formula states that for all tuple solution of C, there must exist a k that is solution of Q. We can check 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 Q to obtain Q', and solve $\exists k(Q')$.

Depending on the number of solutions of the first resolution, this strategy can be computationally expensive.

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

IV. 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 [1]. In our implementation, we rely on two constraint solvers from IBM/Ilog: Cplex[1], an optimization software package based

[1]Cplex works with floating point numbers and thus requires to use the simple procedure introduced by Neumaier et al. [10] to get rigorous answers.

on the simplex method and on MILP (Mixed-integer linear programming) techniques, and CP Optimizer, a constraint 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 CP solver only when necessary.

A. Program set and candidate invariants

The first program 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. The second program is a slight variation of the previous one, 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. Our third program computes an integer approximate square root. This time, not only the specification contains nonlinear expressions, but also the code does. The fourth program computes the same approximate square root as the previous example, except that the program does not contain any nonlinear expression. The last program performs a binary search and introduces two important features: arrays, and quantification.

Candidate invariants come from four different sources:

- **Daikon** infers candidate invariants from program execution traces. We produced 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 the traces.
- **InvGen** [11] produces correct-by-construction invariants. As such, these invariants do not have to be verified, but they contribute an unbiased source of correct invariants. InvGen does not handle nonlinearity and arrays, so we could only apply it on three of our five program.
- The **heuristics** set of candidate invariants was made by applying well-known heuristics to program postconditions, such as "*replacing a constant by a variable*" [12] or changing the relational operators. This set is close to invariants generated by some techniques for which no implementation was available and simple to reuse.
- The **user** set of candidate invariants is made up of the invariants as used in manual proofs of the programs. They are strong enough to imply the program postcondition.

We checked by hand the validity of all the invariants to classify them. The complete list of invariants is available at: http://users.polytech.unice.fr/~rueher/invariants/.

B. 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 system 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. The figures, except otherwise specified, are for 32-bit integers. Execution times for each source of invariants are gathered in Table I. They are further detailed between the valid invariants (True) and the spurious ones (False). Table I contains:

TABLE I
EXECUTION TIMES ACCORDING TO INVARIANTS' SOURCE AND VALIDITY

Source		#	Time < 1 s	Time < 1 min	Time TO
Daikon	True	7	7 (100%)	0	0
	False	41	37 (90.2%)	0	4 (9.8%)
	Total	48	44 (91.7%)	0	4 (8.3%)
InvGen	True	3	3 (100%)	0	0
Heuristics	True	7	4 (57.1%)	0	3 (42.9%)
	False	118	104 (88.1%)	3 (2.5%)	11 (9.4%)
	Total	125	107 (85.6%)	3 (2.4%)	15 (12%)
User	True	7	0	1 (14.3%)	6 (85.7%)

TABLE II
EXECUTION TIMES ACCORDING TO VALIDITY AND LINEARITY

	#	< 1 s	< 1 min	TO
True	24	14 (58.3%)	1 (4.2%)	9 (37.5%)
False	159	141 (88.7%)	3 (1.9%)	15 (9.4%)
Linear	76	73 (96.1%)	1 (1.3%)	2 (2.6%)
Nonlinear	107	82 (76.6%)	3 (2.8%)	22 (20.6%)

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

A large majority of invariants are checked 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 time out are mostly nonlinear. Spurious invariants predominate in the generated candidates, whether it is by heuristics or by Daikon. They are mostly quickly discarded by our method, which shows that the proposed approach can be efficiently used as a filter for the candidate invariants automatically generated. The few spurious invariants that time 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 II 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 that the approach performs well in detecting spurious invariants. Results are more contrasted with valid invariants: on the one hand, complex invariants such 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. Regarding linearity, most of the time-outs occur on nonlinear constraint systems. The CP solver is clearly less efficient than the MILP solver, however, our approach may still perform well in presence of nonlinearity in two situations:

TABLE III
EXECUTION TIMES VARYING THE INTEGER DOMAIN SIZE

	#	< 1 s	< 1 min	TO
8 bits	183	177 (96.8%)	3 (1.6%)	3 (1.6%)
16 bits	183	166 (90.7%)	8 (4.4%)	9 (4.9%)
32 bits	183	155 (84.7%)	4 (2.2%)	24 (13.1%)

- 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: CP Optimizer does not need to fully explore the search space and may quickly find a counter-example (82% of the nonlinear spurious invariants are refuted in less than a second).

The longest execution times occur when the CP solver enumerates on the variable domains to prove a valid candidate.

Table III 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-outs. Of course, it can only increase confidence in a candidate, 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.

V. RELATED WORK

Few works specifically deal with verifying candidate loop invariants. Close to our work is [13] 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, they handle loops with unusual constraint systems that are very hard to solve. 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 [14]. In this approach, the static checker ESC/Java is called on the program source code annotated with candidate invariants. As stated by the authors, ESC/Java itself is unsound, so even if the proof succeeds, the candidate invariant may still be spurious. Moreover, 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. Similarly, other tools based on an undecidable logic, such as Why [1], usually do not disprove a spurious candidate. Indeed, a failed proof attempt does not imply the proof is impossible. In contrast, constraint programming cannot prove a property in general, but it can prove properties under boundedness restrictions and disprove properties that a theorem prover cannot handle.

Our work shares ideas with [15], [16] which also interpret semantics of programs as constraint systems. In [15], focus is on heap and shape analyses. Gulwani et al. [16] address interprocedural verification, weakest precondition and strongest postcondition inference but user-provided hints are needed for generating the constraint system. Both works use SAT solvers.

Other bounded model checking tools, like CBMC [17], should be able to disprove a spurious candidate like we do. 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.

VI. CONCLUSION

In this paper, we showed that constraint solvers could be used as an efficient bounded decision procedure to verify loop invariants. A remarkable feature of our approach is to be able to quickly 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.

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 must be studied in a future work.

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 exploit this property to provide sound loop invariants to our bounded program verification tool CPBPV and improve its loop unfolding process.

REFERENCES

[1] J.-C. Filliâtre and C. Marché, "The why/krakatoa/caduceus platform for deductive program verification," in *CAV*, ser. LNCS, vol. 4590, 2007, pp. 173–177, http://why.lri.fr/examples.

[2] C. Gladisch, "Verification-based test case generation for full feasible branch coverage," in *SEFM*. IEEE CS, 2008, pp. 159–168.

[3] 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*, vol. 69, no. 1–3, pp. 35–45, dec 2007.

[4] A. Ireland, B. J. Ellis, and T. Ingulfsen, "Invariant patterns for program reasoning," in *MCAI*, ser. LNCS, vol. 2972, 2004, pp. 190–201.

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

[6] P. H. Schmitt and B. Weiß, "Inferring invariants by symbolic execution," in *VERIFY*, ser. CEUR Workshop Proc., vol. 259, 2007, pp. 195–210.

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

[8] H. Collavizza, M. Rueher, and P. V. Hentenryck, "CPBPV: A constraint-programming framework for bounded program verification," *Constraint Journal*, vol. 15, no. 2, pp. 238–264, apr 2010.

[9] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communication of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[10] A. Neumaier and O. Shcherbina, "Safe bounds in linear and mixed-integer programming," *Mathematical Programming*, vol. 99, no. 2, pp. 283–296, 2004.

[11] A. Gupta and A. Rybalchenko, "InvGen: An efficient invariant generator," in *CAV*, ser. LNCS, vol. 5643, 2009, pp. 634–640.

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

[13] T. Denmat, A. Gotlieb, and M. Ducassé, "Proving or disproving likely invariants with constraint reasoning," in *WLPE*, 2005, pp. 1–13.

[14] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," in *ISSTA*. ACM, 2002, pp. 232–242.

[15] D. Jackson and M. Vaziri, "Finding bugs with a constraint solver," in *ISSTA*. ACM, 2000, pp. 14–25.

[16] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *PLDI*. ACM, 2008, pp. 281–292.

[17] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS, vol. 2988, 2004, pp. 168–176.