

Proving or Disproving Likely Invariants with Constraint Reasoning [★]

Tristan Denmat¹, Arnaud Gotlieb², and Mireille Ducassé¹

¹ IRISA/INSA

² IRISA/INRIA

Campus universitaire de Beaulieu 35042 Rennes Cedex, France
{denmat,gotlieb,ducasse}@irisa.fr

Abstract. A program invariant is a property that holds for every execution of the program. Recent work suggest to infer *likely*-only invariants, via dynamic analysis. A likely invariant is a property that holds for some executions but is not guaranteed to hold for all executions. In this paper, we present work in progress addressing the challenging problem of automatically verifying that likely invariants are actual invariants. We propose a constraint-based reasoning approach that is able, unlike other approaches, to both prove or disprove likely invariants. In the latter case, our approach provides counter-examples. We illustrate the approach on a motivating example where automatically generated likely invariants are verified.

1 Introduction

A program invariant is a property that holds over every execution of the program. Examples of program invariants include loop invariants presented by Hoare in the weakest precondition calculus [12] or pre-post conditions of the design by contracts approach [14]. Invariants have proved to be crucial in various fields of software engineering such as specification refinement, software evolution or software verification. Unfortunately, writing invariants is a tedious task and few programmers write program invariants by themselves.

In order to palliate this problem, a trend of research aims at inferring invariants *a posteriori*. In this case, invariants correspond to the actual behaviors of programs, not to their intended behaviors.

A common approach is to use static analysis, which infers invariants from the source code. For example, abstract interpretation-based analyses

[★] In A. Serebrenik and S. Muñoz-Hernández (editors), Proceedings of the 15th Workshop on Logic-based methods in Programming Environments, October 2005, Spain. Computer Research Repository (<http://www.acm.org/corr/>), cs.SE/0508108; whole proceedings: cs.PL/0508078.

generate different kinds of invariants, depending on the abstract domain used : intervals [3], polyhedra [4] or octagons [15], to name a few. These methods generate sound invariants but the abstractions used to address problems of termination and complexity may lead to a weak accuracy.

Ernst et al. introduced Daikon, a tool performing dynamic inference of properties using actual values computed during program executions [6]. The advantage is that the generated properties are in general more precise than those generated with a static inference. The drawback of this method is that the properties may not hold for particular executions. They are therefore *likely* only invariants. Proving likely invariants to be correct would make them sound, while being in general more precise than statically inferred invariants.

In this paper, we present work in progress regarding a constraint-based approach to verify likely invariants by refutation. We have restrained the presentation to the validation of likely invariants generated by Daikon. Nevertheless, others likely invariants can be checked with this approach. For example, a user could test his program against properties that he knows it is supposed to have. The idea of the approach is, firstly, to generate a constraint system, CS , modeling an imperative program. To do this, we use the translation of an imperative program into CLP(FD) presented by Gotlieb et al. [8], which has already proved to be useful in structural testing [9]. This transformation can deal with a large subset of C/C++ language, including floating point numbers [1] and a restricted class of pointers [10]. Then, we transform a likely invariant into a constraint I . Finally, we try to find a solution of the constraint system $CS \wedge \neg I$: if the constraint solver finds a solution, then the likely invariant is spurious. If the solver finds that there is no solution, then the likely invariant is an invariant. Unfortunately, the resolution might not terminate or might take too long. In these cases, nothing can be concluded. From a declarative point of view, this approach is very similar to the verification of program based on Horn Logic Denotations [11]. The difference is the use of constraint logic to express the semantics of an imperative language instead of pure Horn logic. When running the verification, Horn logic leads to a generate-and-test method, whereas constraint logic leads to a propagate-generate-and-test method. We expect our approach to be more efficient because the propagation should reduce the number of test cases.

The different steps of our approach are detailed on a motivating example. Three likely invariants are generated by a dynamic inference. By applying the method presented here, two of them are disproved and the other is proved.

The contribution of the approach, as illustrated on our example, is to be able to both prove or disprove some likely invariants. In the literature, similar techniques are dedicated to either one or the other. Jackson and Vaziri use a constraint solving-based approach that only allows them to disprove likely invariants [13]. Nimmer and Ernst present an experiment to prove the correctness of likely invariants using the static checker ESC-Java [2,16]. When ESC-Java fails to prove a likely invariant, it might be due to the lack of an assertion or precondition rather than to an actual error. Because of this point, ESC-Java cannot disprove spurious likely invariants.

Section 2 briefly describes the work of Ernst et al. on dynamic inference of likely invariants. Section 3 presents our motivating example. The dynamic analysis of Ernst et al. is used to infer invariants on this program. Section 4 summarizes the translation of an imperative program into a constraint system. Section 5 illustrates how we suggest to refute or prove a likely invariant using constraint solving. Section 6 discusses difficulties encountered with our approach. Finally, section 7 concludes the paper.

2 Dynamic inference of invariants

This section briefly describes the seminal work of Ernst et al. on dynamic inference of likely invariants [6].

Previous work about the inference of program invariants used static analyses. Results of such analyses are sound, which is very important for program invariants. The counterpart is that the approximations and complex algorithms required to achieve soundness may lead to a weak accuracy.

Ernst et al. propose a compromise where the soundness of the results is not guaranteed in order to gain accuracy. They use dynamic analyses that compute likely invariants from data collected during executions. The underlying idea is that, if a property holds over many executions, then it has good chances to be an invariant.

Daikon is a tool that implements the dynamic inference of likely invariants in four steps. Firstly, the program is instrumented to automatically trace values of variables of interest during execution. Secondly, a test suite is executed on this new program. The data collected during these executions are stored in a database. Thirdly, the set of potential likely invariants is generated. Daikon uses a pool of relationships to automatically generate all potential invariants between variables that can be compared.

```

int foo (int n, int r){
  int s = 0;
  while (n > 0) {
    n --;
    if (s == 0){
      s = 1;
      r ++;
    }
    else {
      s = 0;
      r --;
    }
  }
  return r;
}

```

Fig. 1. A toy example : the *foo* program

Comparability between variables is discussed in [7]. Examples of possible invariants are equalities with a constant (e.g. $x = a$), non-linear relationships among variables (e.g. $z = gcd(x, y)$) or ordering relationships between variables (e.g. $x > y$). Additional relationships involving at most three variables are trivial to add. Finally, the set of possible invariants so-generated is checked against the execution data stored in the database. Possible invariants that are not falsified during this checking are reported to be likely invariants.

In practice, the complexity of the Daikon algorithm tends to be proportional to the number of detected invariants. A lot of research is done around Daikon to improve the efficiency and accuracy of the inference.

3 Running example

This section presents an example of dynamic inference of likely invariants as presented in section 2. Figure 1 shows the *foo* C program. This program takes two input values : n and r . It returns r if n is negative. Else, it returns r if n is even and $r + 1$ if n is odd.

We have used Daikon, the tool presented in section 2, to infer likely program invariants of the *foo* program. We used an all-branch covering test suite of 25 test cases. In these test cases, the loop is unfolded from 0 to 454 times. With this test suite, the inference configured in the default mode resulted in three likely invariants at the exit point of the program :

1. $orig(r) = 0 \implies return = 0$

2. $return = 0 \implies orig(r) = 0$
3. $return \geq orig(r)$

In these likely invariants, $orig(r)$ corresponds to the value of variable r at the entry point of the program and $return$ is the value returned by the program. These likely invariants are not trivial, as they represent a partial specification of a loop. In particular, likely invariant 3 is complicated to infer statically. Indeed, it requires to detect that the executed branch of the conditional alternates at each loop unfolding in such a way that the value of r cannot become lower than $orig(r)$. Likely invariants 1 and 2 are also difficult to infer as they can be seen as a disjunction of two properties. For example, likely invariant 1 is actually $orig(r) \neq 0 \vee return = 0$.

4 Translation of an imperative program into a constraint system

This section describes the first step of our approach to validate likely invariants, namely translating an imperative program into constraint logic programming on finite domains (CLP(FD)). More details about the transformation can be found in [8].

CLP(FD) is an extension of logic programming. In CLP(FD) programs, logical variables are assigned a *domain* and relations between variables are described with *constraints*. A solution to a CLP(FD) program is a valuation of every variable in its own domain such that no constraint is falsified. Solutions are found using two mechanisms : *propagation* and *enumeration*. Propagation uses domain information of each variable to reduce domains of other variables. When no more propagation can be done, enumeration, also called labeling, assigns values to variables to find a solution. Note that each time a variable is assigned a value, a new propagation phase takes this new information into account.

The goal of the transformation described in the following is to generate a CLP(FD) constraint between the input and output variables of an imperative program. Values for which this constraint is satisfied are those who correspond to an existing execution of the program. More formally, if In is the list of input variables of the program and Out the list of output variables, a constraint $clp_prog(In, Out)$ is generated. If the pair (I, O) is a solution of clp_prog then the execution of the original program on inputs I returns values O .

The translation uses the SSA-form as an intermediary form of the program. The instructions of the intermediary program are transformed

into constraints. In particular, specific operators are designed to deal with control structures.

4.1 The SSA-form

The SSA-form is an intermediate representation of imperative programs which prepares the translation into CLP(FD). It has originally been presented by Cytron et al. to optimize compilers [5]. The SSA form is a semantically equivalent version of a program where each variable has a unique definition and every use of this variable is reached by the definition.

The SSA-form is relevant here because logical variables in CLP(FD) programs can be assigned only once whereas, in imperative programs not in SSA-form, variables can be assigned many times.

Every program can be transformed into SSA by renaming the uses and definitions of the variables. For example $i = i + 1; j = j * i$ is transformed into $i_2 = i_1 + 1; j_2 = j_1 * i_2$. At the junction nodes of the control structures, SSA introduces special assignments, called ϕ -functions, to merge several definitions of the same variable: $\vec{v}_2 = \phi(\vec{v}_0, \vec{v}_1)$ assigns the values of \vec{v}_0 in \vec{v}_2 if the flow comes from the first branch of the decision, \vec{v}_1 otherwise. In the case of conditional structures, \vec{v}_0 and \vec{v}_1 are respectively the vectors of defined variables in the *then* and *else* branches. \vec{v}_2 is the vector of these variables out of the conditional structure. Depending on the validity of the condition, $\vec{v}_2 = \vec{v}_0$ or $\vec{v}_2 = \vec{v}_1$.

4.2 Instructions as CLP(FD) constraints

The instructions of the original program are transformed into constraints between logical variables. Type declarations are translated into domain constraints. For example, the declaration of a signed integer x is translated into: $X \in -2^{31}..2^{31} - 1$ where X is a logical FD_variable.

Assignments and decisions are translated into arithmetical constraints. For example, assignment $x = x + 1$ is converted into the SSA form $x_2 = x_1 + 1$ and further translated into $X_2 = X_1 + 1$ where X_1, X_2 are logical FD_variables.

The main difficulty is to transform control structures into constraints. As described in the following, two specific operators are used.

Conditional statements The conditional statement is treated with a specific combinator `ite/6`. Arguments of `ite/6` are the variables that

appear in the ϕ -functions and the constraints generated from the different parts of the original conditional statement. Note that other combinators may be nested into the arguments of **ite/6**. The SSA **if_else** statement :

$$\mathbf{if}(exp) \{stmt\} \mathbf{else} \{stmt\} \vec{v}_2 = \phi(\vec{v}_0, \vec{v}_1)$$

is translated into $\mathbf{ite}(C_{Cond}, \vec{v}_0, \vec{v}_1, \vec{v}_2, C_{Then}, C_{Else})$ where C_{Cond} is a constraint generated by the analysis of exp and C_{Then} (resp. C_{Else}) is a set of constraints generated by the analysis of the *then* branch (resp. *else* branch).

The combinator **ite/6** is defined as :

Definition 1 ite/6

$$\begin{aligned} \mathbf{ite}(C_{Cond}, \vec{v}_0, \vec{v}_1, \vec{v}_2, C_{Then}, C_{Else}) : - \\ C_{Cond} \longrightarrow C_{Then} \wedge \vec{v}_2 = \vec{v}_0, \\ \neg C_{Cond} \longrightarrow C_{Else} \wedge \vec{v}_2 = \vec{v}_1, \\ \neg(C_{Cond} \wedge C_{Then} \wedge \vec{v}_2 = \vec{v}_0) \longrightarrow \neg C_{Cond} \wedge C_{Else} \wedge \vec{v}_2 = \vec{v}_1, \\ \neg(\neg C_{Cond} \wedge C_{Else} \wedge \vec{v}_2 = \vec{v}_1) \longrightarrow C_{Cond} \wedge C_{Then} \wedge \vec{v}_2 = \vec{v}_0, \\ (C_{Cond} \wedge C_{Then} \wedge \vec{v}_2 = \vec{v}_0) \vee (\neg C_{Cond} \wedge C_{Else} \wedge \vec{v}_2 = \vec{v}_1). \end{aligned}$$

This definition uses *guarded-constraints*. A guarded-constraint $head \longrightarrow tail$ rewrites into $tail$ if the constraint $head$ is entailed by the constraint store. The first two guarded-constraints straightforwardly result from the operational semantics of the **if_else** statement whereas the third and the fourth correspond to a backward reasoning. In this case, values of \vec{v}_2 are used to deduce information concerning the flow. The last constraint contains the constructive disjunction operator \vee . This operator removes from the domains of the variables the values that are removed whatever the executed part of the disjunction is. For example, if the constraint $\mathbf{ite}(\dots, [X_0], [X_1], [X_2], X_0 = 1, X_1 = 3)$ stands, the constructive disjunction operator deduces that $X_2 \in \{1, 3\}$.

Iterative statements The SSA **while** statement

$$\vec{v}_2 = \phi(\vec{v}_0, \vec{v}_1) \mathbf{while}(exp) \{stmt\}$$

is treated with the recursive specific combinator $\mathbf{w}(C_{Cond}, \vec{v}_0, \vec{v}_1, \vec{v}_2, C_{Body})$ where C_{Cond} is a constraint generated by the analysis of exp and C_{Body} is a set of constraints generated by the analysis of $stmt$.

Definition 2 $w/5$

$$\begin{aligned}
\mathbf{w}(C_{Cond}, \vec{v}_0, \vec{v}_1, \vec{v}_2, C_{Body}) : - \\
& C_{Cond} \longrightarrow (C_{Body} \wedge \mathbf{w}(C'_{Cond}, \vec{v}_1, \vec{v}_3, \vec{v}_2, C'_{Body})), \\
& \neg C_{Cond} \longrightarrow \vec{v}_2 = \vec{v}_0, \\
& \neg(C_{Cond} \wedge C_{Body}) \longrightarrow (\neg C_{Cond} \wedge \vec{v}_2 = \vec{v}_0), \\
& \neg(\neg C_{Cond} \wedge \vec{v}_0 = \vec{v}_2) \longrightarrow (C_{Cond} \wedge C_{Body} \wedge \\
& \qquad \qquad \qquad \mathbf{w}(C'_{Cond}, \vec{v}_1, \vec{v}_3, \vec{v}_2, C'_{Body})).
\end{aligned}$$

Note that combinator $w/5$ is dynamic : new variables and new constraints are generated during its evaluation. In particular, the vector \vec{v}_3 is a vector of fresh variables. The first and the last guarded constraints both make a recursive call to w . The parameters of this new w are not C_{Cond} and C_{Body} but new constraints C'_{Cond} and C'_{Body} where some variables have been substituted by variables of \vec{v}_1 and \vec{v}_3 to model the fact that the loop has already been entered once.

The first two guarded-constraints are deduced from the operational semantics of the **while** statement. The third constraint tells that, if the constraints extracted from the body are proved to be contradictory with the current constraint system then the loop cannot be entered. The last constraint models the fact that, if any variable possesses distinct values before and after the execution of the **while** statement, then the loop must be entered at least once.

4.3 Translation of the *foo* program into constraints

This section presents the translation of the *foo* program of Figure 1 into a constraint system. By applying the translation described above, the constraint system presented in Figure 2 is generated.

For the sake of clarity, we omit the translation into SSA-form. That is why the constraint system presented on Figure 2 does not explicitly show all the SSA-names. In fact, the variable names that are in the parameters of the w and ite operators must be considered only as syntactical names. Depending on the cases, these names are replaced by logical variables that are in the vectors $\vec{V}_{old}, \vec{V}_{new}, \vec{V}_{final}, \vec{V}_{then}, \vec{V}_{else}$ or \vec{V}_{f_ite} . Constraints that correspond to the type declarations of variables are also omitted.

As the transformation faithfully models the operational semantics of C programs, the constraint system can be executed just like the original C program. For example, if we instantiate N_0 to 5 and R_0 to 3, constraint propagation leads to the instantiation of RET to 4, which is the result of the original program on the same entries.

<pre> int foo (int n, int r){ int s = 0; while (n > 0) { n --; if (s == 0){ s = 1; r ++; } else { s = 0; r --; } } return r; } </pre>	<pre> foo([N₀, R₀],[RET]):- S₀ = 0, w(n > 0, $\overrightarrow{V_{old}}$, $\overrightarrow{V_{new}}$, $\overrightarrow{V_{final}}$, [n = n - 1, ite(s = 0, $\overrightarrow{V_{then}}$, $\overrightarrow{V_{else}}$, $\overrightarrow{V_{f_ite}}$, [s = 1, r = r + 1], [s = 0, r = r - 1]]), RET = R_{final}. </pre>
--	--

Fig. 2. Translation of the *foo* program into a constraint system

5 Validation of likely invariants

In this section, we informally introduce a method to prove or disprove likely invariants. Section 5.1 explains how we transform the problem of invariant validation into a constraint satisfaction problem and Section 5.2 illustrates the behavior of constraint solvers for the running example.

5.1 A constraint solving problem

Section 4 presented a model of an imperative program as a constraint system. This constraint system, denoted by CS , is a relation between the input variables and the output variables. If (X, Y) is a solution of CS , X and Y being respectively input and output values, then there exists a finite execution of the original program starting with input X and returning Y .

A likely invariant, denoted by I , can be seen as one more constraint. This new constraint should be implied by CS if I really is an invariant. We want to prove

$$CS \models I$$

Such a proof can be established by refutation using constraint solving :

$$CS \models I \Leftrightarrow Sol(CS \wedge \neg I) = \emptyset$$

In this equation, $Sol(CS \wedge \neg I)$ denotes the set of solutions of the constraint system $CS \wedge \neg I$.

When solving the refutation request $CS \wedge \neg I$, there are three cases :

1. there exists a solution (X, Y) , which means that the execution starting from X and terminating in Y does not verify the likely invariant I . Thus, I is spurious and (X, Y) is a counter-example.
2. there is no solution to this problem. It means that I really is an invariant.
3. the user runs out of patience. It can be due either to a too long computation or a non-terminating computation. Nothing can be concluded.

As already mentioned in the introduction, the method presented by Nimmer and Ernst [16] can prove that a program verifies a likely invariant. However, if no proof can be established, it might be due to the fact that there is not enough axioms. For example, loop invariants must be provided by users in order to soundly prove properties [2]. On the contrary, in the work of Jackson and Vaziri [13], it is possible to find a counter-example that does not verify the property. However, if none can be found, it can be due either to the fact that the likely invariant is indeed an invariant or to the inaccuracy of the under-approximation. For example, as the number of loop unfoldings is bounded by a value k , there might exist a counter-example that unfolds $k + 1$ times a loop.

In other words, at the question *does the program verify the property ?*, Nimmer and Ernst answer “yes” or “maybe”, Jackson and Vaziri answer “no” or “maybe” and our method answers “yes”, “no” or “maybe”.

5.2 Validation of the invariants of the running example

In this section, we illustrate our approach on the running example. The first likely invariant inferred by Daikon for the *foo* program is

$$orig(r) = 0 \implies return = 0.$$

As explained in the previous paragraph, the first step of the validation consists in adding the negation of the likely invariant to the program. The request sent to the solver is therefore

$$: -foo([N_0, R_0], RET), R_0 = 0, RET \neq 0. \quad (1)$$

After propagation the solver answers :

$$N_0 \in [inf, sup], RET \in [inf, -1] \cup [1, sup], R_0 = 0 \quad (2)$$

The propagation alone did not allow the solver to find inconsistencies in the constraint system. Nothing can be deduced concerning the invariant unless concrete values for N_0 and RET are found. An enumeration step on variables N_0 and RET must be done. Note that variables need to have a domain for labeling. As the logical variables correspond to integers in the original imperative program, their bounds are MIN_INT and MAX_INT . The request is now :

$$: -domain([N_0, RET], MIN_INT, MAX_INT), foo([N_0, R_0], RET), \\ R_0 = 0, RET \setminus = 0, labeling([N_0, RET]). \quad (3)$$

After propagation and enumeration, the solver finds a solution

$$N_0 = 1, R_0 = 0, RET = 1. \quad (4)$$

It means that the execution of the original program with input $n = 1, r = 0$ returns $ret = 1$. This execution is a counter-example of the likely invariant $orig(r) = 0 \implies return = 0$. It is therefore disproved.

The second likely invariant inferred by Daikon for the *foo* program is

$$return = 0 \implies orig(r) = 0.$$

In the same way as above, a counter-example is found :

$$n = 1, r = -1, return = 0.$$

The second likely invariant is therefore also disproved.

The third likely invariant inferred by Daikon for the *foo* program is

$$return \geq orig(r).$$

Repeating the operations previously detailed, the following request is sent to the constraint solver :

$$: -foo([N_0, R_0], RET), \\ R_0 > RET. \quad (5)$$

This time, without any enumeration, the constraint solver answers “no”, meaning that there is no solution to this problem. The third likely invariant is therefore proved to be an invariant.

The behavior of the \mathbf{w} operator on the latter refutation is as follows. Initially, the \mathbf{w} operator is instantiated to

$$\mathbf{w}(N_0 > 0, [R_0, N_0, S_0], [R_1, N_1, S_1], [RET, N_2, S_2], C_{Body})$$

We have not expanded the constraint system of the body for readability reasons. The fourth guarded constraint of the \mathbf{w} operator instantiated for the `foo` program is logically equivalent to what follows.

$$N_0 > 0 \vee (R_0 \neq RET) \longrightarrow (N_0 > 0 \wedge C_{Body} \wedge \mathbf{w}(N_1 > 0, [R_1, N_1, S_1], [R_3, N_3, S_3], [RET, N_2, S_2], C'_{Body})).$$

As $R_0 > RET$ (constraint 5), it is impossible for R_0 to be equal to RET . The guard of the previous constraint is entailed. The loop must therefore be entered and constraints of C_{Body} are set up

$$N_1 = N_0 - 1. \tag{6}$$

As $S_0 = 0$ (first constraint of the `foo` program), the `ite` operator set up constraints corresponding to the *then* branch

$$S_1 = 1 \tag{7}$$

$$R_1 = R_0 + 1 \tag{8}$$

Due to constraints 5 and 8 the following property is true

$$R_1 > R_0 > RET, \tag{9}$$

therefore, it is impossible to have $R_1 = RET$. Consequently, the loop is unfold again. Values $[R_3, N_3, S_3]$ are constrained by clones of constraints 6 and 8. The same reasoning applies until propagation deduces that n cannot be greater than 0. At the beginning, n is in the interval $[MIN_INT, MAX_INT]$ so after MAX_INT iterations n is in the interval $[MIN_INT, 0]$ because of constraint 6 and all its clones. Thus,

$$N_{MAX_INT} \leq 0 \tag{10}$$

At this point, $R_{MAX_INT} > RET$. The second guarded-constraint of the \mathbf{w} operator instantiated for the `foo` program is :

$$\neg N_k > 0 \longrightarrow [R_k, N_k, S_k] = [RET, N_2, S_2]$$

When $k = MAX_INT$, the guard is entailed because of constraint 10. Consequently, the constraint

$$RET = R_{MAX_INT} \quad (11)$$

is set up. It makes the constraint store unsatisfiable, and this is detected by the constraint solver. As a consequence, the third invariant is proved to be true.

6 Discussion

The previous section presented three examples of validation of likely invariants by constraint solving. Two likely invariants were disproved by the exhibition of a counter-example. The last one was proved to be an invariant.

A point that we have not developed yet is the case where the resolution does not terminate or is too long. There are two main reasons why these cases can happen. The first reason is due to the loops. Indeed, as the model we use describes the operational semantics of a program, if the original program does not terminate, then the resolution will not terminate.

The second reason is a problem of propagation in the constraint system. As presented in section 4, the operators *ite* and *w* are defined via guarded-constraints. Consequently, if the entailment of none of the guards can be deduced from the current store of constraints, then the resolution of the constraint system suspends. The problem is that our system is very specific and usual methods of entailment-checking are inefficient in this context : domains of variables are very large, constraint store is dynamic and constraints used can be non-linear.

The consequence of this lack of propagation is that, in bad cases, almost all the possible values of input variables will have to be enumerated to prove or disprove likely invariants. In such a case, our approach becomes a generate-and-test method, which is intractable when the domains of input variables are large. Future work will consist in improving the propagation inside our specific constraint system.

7 Conclusion

In this paper, we have presented an approach to verify the correctness of likely invariants using constraint solving. We have illustrated its principles on a toy example.

The originality of this method is that some likely invariants are disproved and others are proved. This differs from other methods that are dedicated to only one of these capabilities. Methods using under-approximations can only disprove likely invariants whereas methods using over-approximation can only prove likely invariants. We are not using any approximation, it allows us to prove and disprove but prevents us to guarantee termination and good performances. Consequently, the key point of our approach is to have a good propagation inside the constraint system in order to reduce as much as possible the number of cases where we cannot conclude.

Acknowledgments We thank the anonymous referees for their helpful comments.

References

1. B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability Journal*, 2005.
2. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
3. P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
4. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of Symposium on Principles of Programming Languages*, pages 84–96. ACM, 1978.
5. 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.
6. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
7. M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the International Conference on Software Engineering*, pages 449–458. IEEE, 2000.
8. A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 53–62. ACM, 1998.
9. A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *First International Conference on Computational Logic*, pages 399–413. Springer, 2000.
10. A. Gotlieb, T. Denmat, and B. Botella. Goal-oriented test data generation for programs with pointer variables. In *Proceedings of the International Computer Software and Applications Conference*. IEEE, 2005.

11. G. Gupta. Horn logic denotations and their applications. In K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm : A 25-Year Perspective*, pages 127–159. Springer, 1999.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
13. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–25. ACM, 2000.
14. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
15. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proceedings of the European Symposium on Programming*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.
16. J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 229–239. ACM, 2002.