Inferring Loop Invariants using Postconditions

Carlo A. Furia and Bertrand Meyer

May 17, 2010

Abstract

One of the obstacles in automatic program proving is to obtain suitable *loop invariants*. The invariant of a loop is a weakened form of its postcondition (the loop's goal, also known as its contract); the present work takes advantage of this observation by using the postcondition as the basis for invariant inference, using various heuristics such as "uncoupling" which prove useful in many important algorithms. Thanks to these heuristics, the technique is able to infer invariants for a large variety of loop examples. We present the theory behind the technique, its implementation (freely available for download and currently relying on Microsoft Research's Boogie tool), and the results obtained.

1 Overview

Many of the important contributions to the advancement of program proving have been, rather than grand new concepts, specific developments and simplifications; they have removed one obstacle after another preventing the large-scale application of proof techniques to realistic programs built by ordinary programmers in ordinary projects. The work described here seeks to achieve such a practical advance by automatically generating an essential ingredient of proof techniques: *loop invariants*. The key idea is that invariant generation should use not just the text of a loop but its *postcondition*. Using this insight, the gin-pink tool presented here is able to infer loop invariants for non-trivial algorithms including array partitioning (for Quicksort), sequential search, coincidence count, and many others. The tool is available for free download.[1]

1.1 Taking advantage of postconditions

In the standard Floyd-Hoare approach to program proving, loop invariants are arguably the biggest practical obstacle to full automation of the proof process. Given a routine's specification (contract), in particular its postcondition, the proof process consists of deriving intermediate properties, or *verification conditions*, at every point in the program text. Straightforward techniques yield verification conditions for basic instructions, such as assignment, and basic control structures, such as sequence and conditional. The main difficulty is the loop control structure, where the needed verification condition is a *loop invariant*, which unlike the other cases cannot be computed through simple rules; finding the appropriate loop invariant usually requires human invention.

[1] `http://se.inf.ethz.ch/people/furia/`

Experience shows, however, that many programmers find it hard to come up with invariants. This raises the question of devising automatic techniques to infer invariants from the loop's text, in effect extending to loops the mechanisms that successfully compute verification conditions for other constructs. Loops, however, are intrinsically more difficult constructs than (for example) assignments and conditionals, so that in the current state of the art we can only hope for *heuristics* applicable to specific cases, rather than general algorithms guaranteed to yield a correct result in all cases.

While there has been considerable research on loop invariant generation and many interesting results (reviewed in the literature survey of Section 6), most existing approaches are constrained by a fundamental limitation: to obtain the invariant they only consider the *implementation* of a loop. In addition to raising epistemological problems explained next, such techniques can only try to discover relationships between successive loop iterations; this prevents them from discovering many important classes of invariants.

The distinctive feature of the present work is that it uses postconditions of a routine for inferring the invariants of its loops. The postcondition is a higher-level view of the routine, describing its goal, and hence allows inferring the correct invariants in many more cases. As will be explained in Section 3.1, this result follows from the observation that a loop invariant is always a weakened form of the loop's postcondition. Invariant inference as achieved in the present work then relies on implementing a number of *heuristics for mutating postconditions into candidate invariants*; Section 2 presents four such heuristics, such as *uncoupling* and *constant relaxation*, which turn out to cover many practical cases.

1.2 Inferring assertions: the Assertion Inference Paradox

Any program-proving technique that attempts to infer specification elements (such as loop invariants) from program texts faces a serious epistemological objection, which we may call the Assertion Inference Paradox.

The Assertion Inference Paradox is a risk of vicious circle. The goal of program proving is to establish program correctness. A program is correct if its implementation satisfies its specification; for example a square root routine implements a certain algorithm, intended to reach a final state satisfying the specification that the square of the result is, within numerical tolerance, equal to the input. To talk about correctness requires having both elements, the implementation and the specification, and assessing one against the other. But if we infer the specification from the implementation, does the exercise not become vacuous? Surely, the proof will succeed, but it will not teach us anything since it loses the fundamental property of independence between the mathematical property to be achieved and the software artifact that attempts to achieve it — the problem and the solution.

To mitigate the Assertion Inference Paradox objection, one may invoke the following arguments:

- The Paradox only arises if the goal is to prove correctness. Specification inference can have other applications, such as reverse-engineering legacy software.

- Another possible goal of inferring a specification may be to present it to a programmer, who will examine it for consistency with an intuitive understanding of its intended behavior.

- Specification inference may produce an inconsistent specification, revealing a flaw in the implementation.

For applications to program proving, however, the contradiction remains; an inferred specification not exhibiting any inconsistencies cannot provide a sound basis for a proof process.

For that reason, the present work refrains from attempting specification inference for the principal units of a software system: routines (functions, methods) and those at an even higher level of granularity (such as classes). It assumes that these routine specifications are available. Most likely they will have been written explicitly by humans, although their origin does not matter for the rest of the discussion.

What does matter is that once we have routine specifications, it becomes desirable to infer the specifications of all lower-level constructs (elementary instructions and control structures such as conditionals and loops) automatically. At those lower levels, the methodological objection expressed by the Assertion Inference Paradox vanishes: the specifications are only useful to express the semantics of implementation constructs, not to guess the software's intent. The task then becomes: given a routine specification — typically, a precondition and postcondition — derive the proof automatically by inferring verification conditions for the constructs used in the routine and proving that the constructs satisfy these conditions. No vicious circle is created.

For basic constructs such as assignments and conditional instructions, the machinery of Floyd-Hoare logic makes this task straightforward. The principal remaining difficulty is for loops, since the approach requires exhibiting a *loop invariant*, also known as an *inductive assertion*, and proving that the loop's initialization establishes the invariant and that every execution of the body (when the exit condition is not satisfied) preserves it.

A loop invariant captures the essence of the loop. Methodologically, it is desirable that programmers devise the invariant while or before devising the loop. As noted, however, many programmers have difficulty coming up with loop invariants. This makes invariants an attractive target for automatic inference.

In the present work, then, postconditions are known and loop invariants inferred. The approach has two complementary benefits:

- It does not raise the risk of circular reasoning since the specification of every program unit is given from the outside, not inferred.

- Having this specification of a loop's context available gives a considerable boost to loop invariant inference techniques. While there is a considerable literature on invariant inference, it is surprising that none of the references with which we are familiar use postconditions. Taking advantage of postconditions makes it possible — as described in the rest of this paper — to derive the invariants of many important and sometimes sophisticated loop algorithms that had so far eluded other techniques.

2 Illustrative examples

This section presents the fundamental ideas behind the loop-invariant generation technique detailed in Section 4 and demonstrates them on a few examples. It uses an Eiffel-like [30] pseudocode, which facilitates the presentation thanks to the native syntax for contracts and loop invariants.

As already previewed, the core idea is to generate candidate invariants by mutating postconditions according to a few commonly recurring patterns. The patterns capture some basic ways in which loop iterations modify the program state towards achieving the postcondition. Drawing both from classic literature [19, 29] and our own more recent investigations we consider the following fundamental patterns.

Constant relaxation [29, 19]: replace one or more constants by variables.

Uncoupling [29]: replace two occurrences of the same variable each by a different variable.

Term dropping [19]: remove a term, usually a conjunct.

Variable aging: replace a variable by an expression that represents the value the variable had at previous iterations of the loop.

These patterns are then usually used in combination, yielding a number of mutated postconditions. Each of these candidate invariants is then tested for initiation and consecution (see Section 3.1) over any loop, and all verified invariants are retained.

The following examples show each of these patterns in action. The tool described in Sections 4 and 5 can correctly infer invariants of these (and more complex) examples.

2.1 Constant relaxation

Consider the following routine to compute the maximum value in an array.

```
 1  max ($A$: ARRAY [T]; $n$: INTEGER): T
 2     require $A.length = n \geq 1$
 3     local $i$: INTEGER
 4     do
 5        from $i := 0$;  Result := $A[1]$;
 6        until $i \geq n$
 7        loop
 8           $i := i + 1$
 9           if Result $\leq A[i]$ then Result := $A[i]$ end
10        end
11     ensure $\forall j \bullet 1 \leq j \wedge j \leq n \implies A[j] \leq$ Result
```

Lines 5–10 may modify variables i and **Result** but they do not affect input argument n, which is therefore a constant with respect to the loop body. The *constant relaxation* technique replaces every occurrence of the constant n by a variable i. The modified postcondition, $\forall\ j \bullet 1 \leq j \wedge\ j \leq i \implies A[j] \leq$ **Result**, is indeed an invariant of the loop: after every iteration the value of **Result** is the maximum value of array A over range $[1..i]$.

2.2 Variable aging

Sometimes substituting a constant by a variable in the postcondition does not yield any loop invariant because of how the loop body updates the variable. It may happen that the loop body does not "use" the latest value of the substituted variable until the next iteration. Consider for example another implementation of computing the maximum of an array, which increments variable i *after* using it, so that only the range $[1..i-1]$ of array A has been inspected after every iteration.

```
max_v2 ($A$: ARRAY [T], $n$: INTEGER): T
   require $A.length = n \geq 1$
   local $i$: INTEGER
   do
      from $i := 1$;  Result := $A[1]$;
      until $i > n$
      loop
         if Result $\leq A[i]$ then Result := $A[i]$ end
         $i := i + 1$
      end
   ensure $\forall\ j \bullet 1 \leq j \wedge\ j \leq n \implies A[j] \leq$ Result
```

The *variable aging* heuristics handles these cases by introducing an expression that represents the value of the variable at the previous iteration in terms of its current value. In the case of routine *max_v2* it is straightforward that such an expression for variable i is $i-1$. The postcondition can be modified by first replacing variable n by variable i and then by "aging" variable i into $i-1$. The resulting formula $\forall\ j \bullet 1 \leq j \wedge\ j \leq i-1 \implies A[j] \leq$ **Result** correctly captures the semantics of the loop.

Computing the symbolic value of a variable at the "previous" iteration can be quite complex in the general case. In practice, however, a simple (e.g., flow-insensitive) approximation is often enough to get significant results. The experiments of Section 5 provide a partial evidence to support this conjecture.

2.3 Uncoupling

Consider the task (used as part of the Quicksort algorithm) of partitioning an array A of length n into two parts such that every element of the first part is less than or equal to a given *pivot* value and every element of the second part is greater than or equal to it. The following contracted routine specifies and implements such task.

```
partition ($A$: ARRAY [T]; $n$: INTEGER; $pivot$: T): INTEGER
  require $A.length = n \geq 1$
  local $low\_index$, $high\_index$: INTEGER
  do
    from $low\_index := 1$; $high\_index := n$
    until $low\_index = high\_index$
    loop
       from -- no loop initialization
       until $low\_index = high\_index \vee A[low\_index] > pivot$
       loop $low\_index := low\_index + 1$ end
```

```
11            from −− no loop initialization
12            until $low\_index = high\_index \lor pivot > A[high\_index]$
13            loop $high\_index := high\_index - 1$ end
14            $A.swap\ (A,\ low\_index,\ high\_index)$
15        end
16        if $pivot \leq A[low\_index]$ then
17            $low\_index := low\_index - 1$
18            $high\_index := low\_index$
19        end
20        Result := $low\_index$
21    ensure $(\ \forall k \bullet 1 \leq k \land k <$ Result $+ 1 \Longrightarrow A[k] \leq pivot\ )$
22                $\land\ (\ \forall k \bullet$ Result $< k \land k \leq n \Longrightarrow A[k] \geq pivot\ )$
```

The postcondition consists of the conjunction of two formulas (lines 21 and 22). If we try to mutate it by replacing constant **Result** by variable *low_index* or by variable *high_index* we obtain no valid loop invariant. This is because the two clauses of the postcondition should refer, respectively, to portion $[1..low\_index{-}1]$ and $[high\_index{+}1..n]$ of the array. We achieve this by first uncoupling **Result**, which means replacing its first occurrence (in line 21) by variable *low_index* and its second occurrence (in line 22) by variable *high_index*. After "aging" variable *low_index* we get the formula:

$$\begin{gathered}(\ \forall k \bullet 1 \leq k \land k < low\_index \Longrightarrow A[k] \leq pivot\ )\\ \land\ (\ \forall k \bullet high\_index < k \land k \leq n \Longrightarrow A[k] \geq pivot\ )\end{gathered}$$

The reader can check that this indeed a loop invariant of all loops in routine *partition* and that it allows a straightforward partial correctness proof of the implementation.

2.4 Term dropping

The last mutation pattern that we consider consists simply of removing a part of the postcondition. The formula to be modified is usually assumed to be in conjunctive normal form, that is, expressed as the conjunction of a few clauses: then term dropping amounts to removing one or more conjuncts. Going back to the example of *partition*, let us drop the first conjunct in the postcondition. The resulting formula

$$\forall k \bullet \textbf{Result} < k \land k \leq n \Longrightarrow A[k] \geq pivot$$

can be further transformed through constant relaxation, so that we end up with a conjunct of the invariant previously obtained by uncoupling: $\forall k \bullet high\_index < k \land k \leq n \Longrightarrow A[k] \geq pivot$. This conjunct is also by itself an invariant. In this example term dropping achieved by different means the same result as uncoupling.

3 Foundations

Having seen typical examples we now look at the technical choices that support the invariant inference tools. To decouple the loop-invariant generation technique as much as possible from the specifics of any one programming language, we adopt Boogie from Microsoft Research [26] as our concrete programming language; Section 3.2 is then devoted to a concise introduction to the features of Boogie that are essential for the remainder. Sections 3.1 and 3.3 introduce definitions of basic concepts and some notational conventions that will be used. We assume the reader is familiar with standard formal definitions of the axiomatic semantics of imperative programs.

3.1 Invariants

Proving a procedure correct amounts to verifying that:

1. Every computation terminates.
2. Every call of another procedure is issued only when the preconditions of the callee hold.
3. The postconditions hold upon termination.

It is impossible to establish these facts automatically for all programs but the most trivial ones without additional information provided by the user in the form of annotations. The most crucial aspect is the characterization of loops, where the expressive power of universal computation lies. A standard technique to abstract the semantics of any number of iterations of a loop is by means of *loop invariants*.

Definition 1 (Inductive loop invariant)**.** Formula ϕ is an inductive invariant of loop

from *Init* **until** *Exit* **loop** *Body* **end**

iff:

- *Initiation*: ϕ holds after the execution of *Init*
- *Consecution*: the truth of ϕ is preserved by every execution of *Body* where *Exit* does not hold

In the rest of the discussion, inductive invariants will be called just invariants for short. Note, however, that an invariant in the weaker sense of a property that stays true throughout the loop's execution is not necessarily an *inductive* invariant: in

from $x := 1$ **until** *False* **loop** $x := -\ x$ **end**

formula $x \geq -1$ will remain true throughout, but is not considered an inductive invariant because $\{x \geq -1\}\ x := -x\ \{x \geq -1\}$ is not a correct Hoare triple. In the remainder we will deal solely with inductive loop invariants, as is customary in the program proving literature.

From a design methodology perspective, the invariant expresses a weakened form of the loop's postcondition. More precisely [31, 19], the invariant is a form of the loop's postcondition that applies to a subset of the data, and satisfies the following three properties:

1. It is *strong enough* to yield the postcondition when combined with the exit condition (which states that the loop has covered the *entire* data).

2. It is *weak enough* to make it easy to write an algorithm (the loop initialization *Init*) that will satisfy the invariant on a subset (usually empty or trivial) of the data.

3. It is *weak enough* to make it easy to write an algorithm (the loop body *Body*) that, given that the invariant holds on a subset of the data that is not the entire data, extends it to cover a slightly larger subset.

"Easy", in the last two conditions, means "much easier than solving the entire original problem". The loop consists of an approximation strategy that starts with the initialization, establishing the invariant, then retains the invariant while extending the scope by successive approximations to an ever larger set of the input through repeated executions of the loop body, until it hits the exit condition, signaling that it now covers the entire data and hence satisfies the loop's postcondition. This explains that the various strategies of Section 2, such as constant relaxation and uncoupling, are heuristics for mutating the loop's postcondition into a weaker form. The present work applies the same heuristics to mutate postconditions of the *routine* that encapsulates the loop. The connection that exists between the routine's and the loop's postcondition justifies the rationale behind using weakening heuristics as mutation heuristics to generate invariant candidates.

3.2 Boogie

Boogie, now in its second version, is both an intermediate verification language and a verification tool.

The Boogie language combines a typed logical specification language with an in-the-small imperative programming language with variables, procedures, contracts, and annotations. The type system comprises a few basic primitive types as well as type constructors such as one- and two-dimensional arrays. It supports a relatively straightforward encoding of object-oriented language constructs. Indeed, Boogie is part of the Spec# programming environment; mappings have been defined for other programming languages, including Eiffel [40] and C [39]. This suggests that the results described here can be generalized to many other contexts.

The Boogie tool verifies conformance of a procedure to its specification by generating verification conditions (VC) and feeding them to an automated theorem prover (the standard one being Z3). The outcome of a verification attempt can be successful or unsuccessful. In the latter case the tool provides some feedback on what might be wrong in the procedure, in particular by pointing out what contracts or annotations it could not verify. Verification with Boogie is sound but incomplete: a verified procedure is always guaranteed to be correct, while an unsuccessful verification attempt might simply be due to limitations of the technology.

3.2.1 The Boogie specification language

The Boogie specification language is essentially a typed predicate calculus with equality and arithmetic. Correspondingly, formulas — that is, logic expressions — are built by combining atomic constants, logic variables, and program variables with relational and arithmetic operators, as well as with Boolean connectives and quantifiers. For example, the following formula (from Section 2.1) states that no element in array X within positions 1 and n is larger than v: in other words, the array has upper bound v.

$$\forall\ j\ :\ \mathbf{int}\ \bullet\ 1 \le j \land\ j \le n\ \Longrightarrow\ X[j] \le v$$

The syntactic classes *Id*, *Number*, and *Map* represent constant and variable identifiers, numbers, and mappings, respectively.

Complex formulas and expressions can be postulated in *axioms* and parameterized by means of logic *functions*. Functions are a means of introducing encapsulation and genericity for formulas and complex expressions. For example, the previous formula can be parameterized into function *is_upper* with the following signature and definition:

```
function is_upper (m: int, A: array int, low: int, high: int)
   returns ( bool )
   { ∀ j : int • low ≤ j ∧ j ≤ high ⟹ A[j] ≤ m }
```

Axioms constrain global constants, variables, and functions; they are useful to supply Boogie with domain knowledge to facilitate inference and guide the automated reasoning over non-trivial programs. In certain situations it might for example be helpful to introduce the property that if an array has upper bound m over range $[low..high]$ and the element in position $high + 1$ is smaller than m then m is also the upper bound over range $[low..high+1]$. The following Boogie axiom will express this:

```
   axiom ( ∀m: int, A: array int, low: int, high: int •
         is_upper (m, A, low, high) ∧ A[high + 1] < m
               ⟹ is_upper (m, A, low, high+1) )
```

3.2.2 The Boogie programming language

A Boogie program is a collection of *procedures*. Each procedure consists of a signature, a *specification* and (optionally) an *implementation* or body. The signature gives the procedure a name and declares its formal input and output arguments. The specification is a collection of contract clauses of three types: *frame conditions*, *preconditions*, and *postconditions*.

A frame condition, introduced by keyword **modifies**, consists of a list of global variables that can be modified by the procedure; it is useful in evaluating the side-effects of procedure call within any context. A precondition, introduced by the keyword **requires**, is a formula that is required to hold upon procedure invocation. A postcondition, introduced by the keyword **ensures**, is a formula that is guaranteed to hold upon successful termination of the procedure. For example, procedure *max_v2*, computing the maximum value in an array A given its size n, has the following specification.

```
procedure max_v2 (A: array int, n: int) returns (m: int)
   requires n ≥ 1;
   ensures is_max (m, A, 1, n);
```

The implementation of a procedure consists of a declaration of local variables, followed by a sequence of (possibly labeled) program statements. Figure

Statement	::=	Assertion | Modification
	|	ConditionalBranch | Loop
Annotation	::=	**assert** Formula | **assume** Formula
Modification	::=	**havoc** VariableId | VariableId := Expression
	|	**call** [VariableId$^+$:=] ProcedureId (Expression*)
ConditionalBranch	::=	**if** (Formula) Statement* [**else** Statement*]
Loop	::=	**while** (Formula) Invariant* Statement*
Invariant	::=	**invariant** Formula

Figure 1: Simplified abstract syntax of Boogie statements

1 shows a simplified syntax for Boogie statements. Statements of class *Annotation* introduce checks at any program point: an *assertion* is a formula that must hold of every execution that reaches it for the program to be correct and an *assumption* is a formula whose validity at the program point is postulated. Statements of class *Modification* affect the value of program variables, by nondeterministically drawing a value for them (**havoc**), assigning them the value of an expression (:=), or calling a procedure with actual arguments (**call**). The usual conditional **if** statement controls the execution flow. Finally, the **while** statement supports loop iteration, where any loop can be optionally annotated with a number of *Invariants* (see Section 3.1). Boogie can check whether Definition 1 holds for any user-provided loop invariant.

The implementation of procedure *max_v2* is:

```
var i: int;
i := 1;   m := A[1];
while (i ≤  n)
{
  if  (m≤A[i])   { m := A[i]; }
  i := i + 1;
}
```

While the full Boogie language includes more types of statement, any Boogie statement can be desugared into one of those in Figure 1. In particular, the only looping construct we consider is the structured **while**; this choice simplifies the presentation of our loop invariant inference technique and makes it closer as if it was defined directly on a mainstream high-level programming language. Also, there is a direct correspondence between Boogie's **while** loop and Eiffel's **from** ... **until** loop, used in the examples of Section 2 and the definitions in Section 3.1.

3.3 Notational conventions

$\mathsf{subExp}(\phi, SubType)$ denotes the set of sub-expressions of formula ϕ that are of syntactic type $SubType$. For example, $\mathsf{subExp}(\mathit{is\_upper}(v,X,1,n), Map)$ denotes all mapping sub-expressions in $\mathit{is\_upper}(v,X,1,n)$, that is only $X[j]$.

$\mathsf{replace}(\phi, old, new, *)$ denotes the formula obtained from ϕ by replacing every occurrence of sub-expression *old* by expression *new*. Similarly, $\mathsf{replace}(\phi, old, new, n)$ denotes the formula obtained from ϕ by replacing only the n-th occurrence of sub-expression *old* by expression *new*, where the total ordering of sub-expressions is given by a pre-order traversal of the expression parse tree. For example, $\mathsf{replace}(\mathit{is\_upper}(v,X,1,n), j, h, *)$ is:

```
1  find_invariants ( a_procedure: PROCEDURE )
2                     : SET_OF [FORMULA]
3    do
4      Result := $\emptyset$
5      for each post in postconditions(a_procedure) do
6        for each loop in outer_loops(a_procedure) do
7          -- compute all mutations of post
8          -- according to chosen strategies
9          mutations := build_mutations(post, loop)
10         for each formula in mutations do
11           for each any_loop in loops(a_procedure) do
12             if is_invariant(formula, any_loop) then
13               Result := Result $\cup$ $\{formula\}$
```

Figure 2: Procedure find_invariants

$$\forall\ h\ :\ \textbf{int}\ \bullet\ \ low \leq h\ \wedge\ \ h \leq high\ \ \Longrightarrow\ A[h] \leq m$$

while replace($is\_upper(v, X, 1, n), j, h, 4$) is:

$$\forall\ j\ :\ \textbf{int}\ \bullet\ \ low \leq j\ \wedge\ \ j \leq high\ \ \Longrightarrow\ A[h] \leq m$$

Given a while loop ℓ: **while** (...) { *Body* }, targets(ℓ) denotes the set of its *targets*: variables (including mappings) that can be modified by its *Body*; this includes global variables that appear in the **modifies** clause of called procedures.

Given a **procedure** *foo*, variables(*foo*) denotes the set of all variables that are visible within *foo*, that is its locals and any global variable.

A loop ℓ' is *nested* within another loop ℓ, and we write $\ell' \prec \ell$, iff ℓ' belongs to the *Body* of ℓ. Notice that if $\ell' \prec \ell$ then targets(ℓ') $\subseteq$ targets(ℓ). Given a **procedure** *foo*, its *outer* while loops are those in its body that are not nested within any other loop.

4 Generating loop invariants from postconditions

This section presents the loop invariant generation algorithm in some detail.

4.1 Main algorithm

The pseudocode in Figure 2 describes the main algorithm for loop-invariant generation. The algorithm operates on a given procedure and returns a set of formulas that are invariant of *some* loop in the procedure. Every postcondition *post* among all postconditions postconditions(*a_procedure*) of the procedure is considered separately (line 5). This is a coarse-grained yet effective way of implementing the *term-dropping* strategy outlined in Section 2.4: the syntax of the specification language supports splitting postconditions into a number of conjuncts, each introduced by the **ensures** keyword, hence each of these conjuncts is modified in isolation. It is reasonable to assume that the splitting into **ensures** clauses performed by the user separates logically separated portions of the postcondition, hence it makes sense to analyze each of them separately.

```
 1 build_mutations ( post: FORMULA; loop: LOOP )
 2                        : SET_OF [FORMULA]
 3   do
 4    Result := {post}
 5     all_subexpressions  := subExp(post, Id) ∪
 6                            subExp(post, Number) ∪
 7                            subExp(post, Map)
 8    for each constant in  all_subexpressions \ targets(loop) do
 9        for each variable in targets(loop) do
10            Result := Result  ∪
11                coupled_mutations(post, constant, variable)  ∪
12                uncoupled_mutations(post, constant, variable)
```

Figure 3: Procedure build_mutations

This assumption might fail, of course, and in such cases the algorithm can be enhanced to consider more complex combinations of portions of the postcondition. However, one should only move to this more complex analysis if the basic strategy — which is often effective — fails. This enhancement belongs to future work.

The algorithm of Figure 2 then considers every *outer* while loop (line 6). For each of them, it computes a set of mutations of postcondition *post* (line 9) according to the heuristics of Section 2. It then examines each mutation to determine if it is invariant to *any* loop in the procedure under analysis (lines 10–13), and finally it returns the set **Result** of mutated postconditions that are invariants to some loop. For is_invariant($formula, loop$), the check consists of verifying whether initiation and consecution hold for $formula$ with respect to *loop*, according to Definition 1. How to do this in practice is non-trivial, because loop invariants of different loops within the same procedure may interact in a circular way: the validity of one of them can be established only if the validity of the others is known already and *vice versa*.

This was the case of *partition* presented in Section 2.3. Establishing consecution for the modified postcondition in the outer loop requires knowing that the same modified postcondition is invariant to each of the two internal while loops (lines 8–13) because they belong to the body of the outer while loop. At the same time, establishing initiation for the first internal loop requires that consecution holds for the outer while loop, as every new iteration of the external loop initializes the first internal loop. Section 5 discusses a straightforward, yet effective, solution to this problem.

4.2 Building mutated postconditions

Algorithm build_mutations($post, loop$), described in Figure 3, computes a set of modified versions of postcondition formula *post* with respect to outer while loop *loop*. It first includes the unchanged postcondition among the mutations (line 4). Then, it computes (lines 5–7) a list of sub-expressions of *post* made of atomic variable identifiers (syntactic class Id), numeric constants (syntactic class $Number$) and references to elements in arrays (syntactic class Map). Each

```
1 coupled_mutations
2       ( post: FORMULA; constant, variable: EXPRESSION )
3       : SET_OF [FORMULA]
4   do
5     Result := replace($post, constant, variable, *$)
6     aged_variable := aging($variable$, $loop$)
7     Result := Result ∪
8                      replace($post, constant, aged\_variable, *$)
```

Figure 4: Procedure coupled_mutations

of these sub-expressions that *loop* does not modify (i.e., it is not one of its targets) is a *constant* with respect to the loop. The algorithm then applies the *constant relaxation* heuristics of Section 2.1 by relaxing *constant* into any *variable* among the *loop*'s targets (lines 8–9). More precisely, it computes two sets of mutations for each pair $\langle constant, variable\rangle$: in one *uncoupling*, described in Section 2.3, is also applied (lines 12 and 11, respectively).

The fact that any target of the loop is a candidate for substitution justifies our choice of considering only outer while loops: if a loop ℓ' is nested within another loop ℓ then $\mathsf{targets}(\ell') \subseteq \mathsf{targets}(\ell)$, so considering outer while loops is a conservative approximation that does not overlook any possible substitution.

4.3 Coupled mutations

The algorithm in Figure 4 applies the constant relaxation heuristics to postcondition *post* without uncoupling. Hence, relaxing *constant* into *variable* simply amounts to replacing every occurrence of *constant* by *variable* in *post* (line 5); i.e., $\mathsf{replace}(post, constant, variable, *)$ using the notation introduced in Section 3.3. Afterward, the algorithm applies the other *aging* heuristics (introduced in Section 2.2): it computes the "previous" value of *variable* in an execution of *loop* (line 6) and it substitutes the resulting expression for *constant* in *post* (lines 7–8).

While the implementation of function aging could be very complex we adopt the following unsophisticated approach. For every possible acyclic execution path in *loop*, we compute the symbolic value of *variable* with initial value v_0 as a symbolic expression $\epsilon(v_0)$. Then we obtain $\mathsf{aging}(variable, loop)$ by solving the equation $\epsilon(v_0) = variable$ for v_0, for every execution path.[2] For example, if the loop simply increments *variable* by one, then $\epsilon(v_0) = v_0 + 1$ and therefore $\mathsf{aging}(variable, loop) = variable - 1$. Again, while the example is unsophisticated it is quite effective in practice; indeed, most of the times it is enough to consider simple increments or decrements of *variable* to get a "good enough" aged expression.

[2]Note that $\mathsf{aging}(variable, loop)$ is in general a set of expressions, so the notation at lines 6–8 in Figure 4 is a shorthand.

```
 1 uncoupled_mutations
 2       ($post$: FORMULA; $constant$, $variable$: EXPRESSION)
 3       : SET_OF [FORMULA]
 4   do
 5     Result := $\emptyset$; $index := 1$
 6     for each $occurrence$ of $constant$ in $post$ do
 7       Result := Result $\cup$
 8           {replace($post, constant, variable, index$)}
 9       $aged\_variable$ := aging($variable$, $loop$)
10       Result := Result $\cup$
11           {replace($post, constant, aged\_variable, index$)}
12       $index := index + 1$
```

Figure 5: Procedure uncoupled_mutations

4.4 Uncoupled mutations

The algorithm of Figure 5 is a variation of the algorithm of Figure 4 applying the *uncoupling* heuristics outlined in Section 2.3. It achieves this by considering every occurrence of *constant* in *post* separately when performing the substitution of *constant* into *variable* (line 6). Everything else is as in the non-uncoupled case; in particular, aging is applied to every candidate for substitution.

This implementation of uncoupling relaxes one occurrence of a constant at a time. This is not the most general implementation of uncoupling, as in some cases it might be useful to substitute different occurrences of the same constant by different variables. This was the case of *partition* discussed in Section 2.3, where relaxing two occurrences of the same constant **Result** into two different variables was needed in order to get a valid invariant. Section 2.4 showed, however, that the term-dropping heuristics would have made this “double” relaxation unnecessary for the procedure.

5 Implementation and experiments

We developed a command-line tool code-named gin-pink (Generation of INvariants by PostcondItioN weaKening) implementing in Eiffel the loop-invariant inference technique described in Section 4. While we plan to integrate gin-pink into EVE (the Eiffel Verification Environment[3]) where it will analyze the code resulting from the translation of Eiffel into Boogie [40], its availability as a stand-alone tool makes it possible to use it for languages other than Eiffel provided a Boogie translator is available.

gin-pink applies the algorithm of Figure 2 to some selected procedure in a Boogie file provided by the user. After generating all mutated postconditions it invokes the Boogie tool to determine which of them is indeed an invariant: for every candidate invariant I, a new copy of the original Boogie file is generated with I declared as invariant of *all* loops in the procedure under analysis. It then repeats the following until either Boogie has verified all current declarations of I in the file or no more instances of I exist in the procedure:

[3]http://eve.origo.ethz.ch

1. Use Boogie to check whether the current instances of I are verified invariants, that is they satisfy initiation and consecution.

2. If any candidate fails this check, comment it out of the file.

In the end, the file contains all invariants that survive the check, as well as a number of commented out candidate invariants that could not be checked. If no invariant survives or the verified invariants are unsatisfactory, the user can still manually inspect the generated files to see if verification failed due to the limited reasoning capabilities of Boogie.

When generating candidate invariants, gin-pink does not apply all heuristics at once but it tries them incrementally, according to user-supplied options. Typically, the user starts out with just constant relaxation and checks if some non-trivial invariant is found. If not, the analysis is refined by gradually introducing the other heuristics — and thus increasing the number of candidate invariants as well. In the experiments below we briefly discuss how often and to what extent this is necessary in practice.

Experiments Table 1 summarizes the results of a number of experiments with gin-pink with a number of Boogie procedures obtained from Eiffel code. We carried out the experimental evaluation as follows. First, we collected examples from various sources [29, 19, 33, 27, 3] and we manually completed the annotations of every algorithm with full pre and postconditions as well as with any loop invariant or intermediate assertion needed in the correctness proof. Then, we coded and tried to verify the annotated programs in Boogie, supplying some background theory to support the reasoning whenever necessary. The latest Boogie technology cannot verify certain classes of properties without a very sophisticated *ad hoc* background theory or without abstracting away certain aspects of the implementation under verification. For example, in our implementation of Bubblesort, Boogie had difficulties proving that the output is a permutation of the input. Correspondingly, we omitted the (few) parts of the specification that Boogie could not prove even with a detailedly annotated program. Indeed, "completeness" (full functional correctness) should not be a primary concern, because its significance depends on properties of the prover (here Boogie), orthogonal to the task of inferring invariants. Finally, we ran gin-pink on each of the examples after commenting out all annotations except for pre and postconditions (but leaving the simple background theories in); in a few difficult cases (discussed next) we ran additional experiments with some of the annotations left in. After running the tests, we measured the relevance of every automatically inferred invariant: we call an inferred invariant *relevant* if the correctness proof needs it. Notice that our choice of omitting postcondition clauses that Boogie cannot prove does not influence relevance, which only measures the fraction of inferred invariants that are useful for proving correctness.

For each experiment, Table 1 reports: the name of the procedure under analysis; the length in lines of codes (the whole file including annotations and auxiliary procedures and, in parentheses, just the main procedure); the total number of loops (and the maximum number of nested loops, in parentheses); the total number of variables modified by the loops (scalar variables/array or map variables); the number of mutated postconditions (i.e., candidate invariants) generated by the tool; how many invariants it finds; the number and percentage

Procedure	LOC	# lp.	m.v.	cnd.	inv.	rel.	T.	Src.
Array Partitioning (v1)	58(22)	1 (1)	2+1	38	9	3 (33%)	93	
Array Partitioning (v2)	68(40)	3 (2)	2+1	45	2	2(100%)	205	[29]
Array Stack Reversal	147(34)	2 (1)	1+2	134	4	2 (50%)	529	
Array Stack Reversal (ann.)	147(34)	2 (1)	1+2	134	6	4 (67%)	516	
Bubblesort	69(29)	2 (2)	2+1	14	2	2(100%)	65	[33]
Coincidence Count	59(29)	1 (1)	3+0	1351	1	1(100%)	4304	[27]
Dutch National Flag	77(43)	1 (1)	3+1	42	10	2 (20%)	117	[16]
Dutch National Flag (ann.)	77(43)	1 (1)	3+1	42	12	4 (33%)	122	[16]
Longest Common Sub. (ann.)	73(59)	4 (2)	2+2	508	22	2 (9%)	4842	
Majority Count	48(37)	1 (1)	3+0	23	5	2 (40%)	62	[4, 32]
Max of Array (v1)	27(17)	1 (1)	2+0	13	1	1(100%)	30	
Max of Array (v2)	27(17)	1 (1)	2+0	7	1	1(100%)	16	
Plateau	53(29)	1 (1)	3+0	31	6	3 (50%)	666	[19]
Sequential Search (v1)	34(26)	1 (1)	3+0	45	9	5 (56%)	120	
Sequential Search (v2)	29(21)	1 (1)	3+0	24	6	6(100%)	58	
Shortest Path (ann.)	57(44)	1 (1)	1+4	23	2	2(100%)	53	[3]
Stack Search	196(49)	2 (1)	1+3	102	3	3(100%)	300	
Sum of Array	26(15)	1 (1)	2+0	13	1	1(100%)	44	
Topological Sort (ann.)	65(48)	1 (1)	2+4	21	3	2 (67%)	101	[31]
Welfare Crook	53(21)	1 (1)	3+0	20	2	2(100%)	586	[19]

Table 1: Experiments with gin-pink.

of verified invariants that are relevant; the total run-time of gin-pink in seconds; the source (if any) of the implementation and the annotations. The experiments where performed on a PC equipped with an Intel Quad-Core 2.40 GHz CPU and 4 Gb of RAM, running Windows XP as guest operating system on a VirtualBox virtual machine hosted by Ubuntu GNU/Linux 9.04 with kernel 2.6.28.

Most of the experiments succeeded with the application of the most basic heuristics. Procedures *Coincidence Count* and *Longest Common Subsequence* are the only two cases that required a more sophisticated uncoupling strategy where two occurrences of the same constant within the same formula were modified to two different aged variables. This resulted in an explosion of the number of candidate invariants and consequently in an experiment running for over an hour.

A few programs raised another difficulty, due to Boogie's need for user-supplied loop invariants to help automated deduction. Boogie cannot verify any invariant in *Shortest Path*, *Topological Sort*, or *Longest Common Subsequence* without additional invariants obtained by means other than the application of the algorithm itself. On the other hand, the performance with programs *Array Stack Reversal* and *Dutch National Flag* improves considerably if user-supplied loop invariants are included, but fair results can be obtained even without any such annotation. Table 1 reports both experiments, with and without user-supplied annotations.

More generally, Boogie's reasoning abilities are limited by the amount of information provided in the input file in the form of axioms and functions that postulate sound inference rules for the program at hand. We tried to limit this amount as much as possible by developing the necessary theories before tackling invariant generation. In other words, the axiomatizations provided are enough for Boogie to prove functional correctness with a properly annotated program, but we did not strengthen them only to ameliorate the inference of invariants. A richer axiomatization may have removed the need for user-supplied invariants in the programs considered.

6 Discussion and related work

6.1 Discussion

The experiments in Section 5 provide a partial, yet significant, assessment of the practicality and effectiveness of our technique for loop invariant inference. Two important factors to evaluate any inference technique deserve comment: relevance of the inferred invariants and scalability to larger programs.

A large portion of the invariants retrieved by gin-pink are relevant — i.e., required for a functional correctness proof — and complex — i.e., involving first-order quantification over several program elements. To some extent, this is unsurprising because deriving invariants from postconditions ensures by construction that they play a central role in the correctness proof and that they are at least as complex as the postcondition.

As for scalability to larger programs, the main problem is the combinatorial explosion of the candidate invariants to be checked as the number of variables that are modified by the loop increases. In properly engineered code, each routine should not be too large or call too many other routines. The empirical observations mentioned in [24, Sec. 9] seem to support this assumption, which ensures that the candidate invariants do not proliferate and hence the inference technique can scale within reasonable limits. The examples of Section 5 are not trivial in terms of length and complexity of loops and procedures, if the yardstick is well-modularized code. On the other hand, there is plenty of room for finessing the application order of the various heuristics in order to analyze the most "promising" candidates first; the Houdini approach [17] might also be useful in this context. The investigation of these aspects belongs to future work.

6.2 Limitations

Relevant invariants obtained by postcondition mutation are most of the times significant, practically useful, and complementary to a large extent to the categories that are better tackled by other methods (see next sub-section). Still, the postcondition mutation technique cannot obtain *every* relevant invariant. Failures have two main different origins: conceptual limitations and shortcomings of the currently used technology.

The first category covers invariants that are not expressible as mutations of the postcondition. This is the case, in particular, whenever an invariant refers to a local variable whose final state is not mentioned in the postcondition. For example, the postcondition of procedure *max* in Section 2.1 does not mention variable i because its final value n is not relevant for the correctness. Correspondingly, invariant $i \leq n$ — which is involved in the partial correctness proof — cannot be obtained from by mutating the postcondition. A potential solution to these conceptual limitations is two-fold: on the one hand, many of these invariants that escape postcondition mutations can be obtained reliably with other inference techniques that do not require postconditions — this is the case of invariant $i \leq n$ in procedure *max* which is retrieved automatically by Boogie. On the other hand, if we can augment postconditions with complete information about local variables, the mutation approach can have a chance to work. In the case of *max*, a dynamic technique could suggest the supplementary postcondition $i \leq n \wedge i \geq n$ which would give the sought invariant by dropping the second conjunct.

Shortcomings of the second category follow from limitations of state-of-the-art automated theorem provers, which prevent reasoning about certain interesting classes of algorithms. For the sake of illustration, consider the following idealized[4] implementation of Newton's algorithm for the square root of a real number, more precisely the variant known as the Babylonian algorithm (we ignore numerical precision issues) [29]:

```
square_root (a: REAL): REAL
    require $a \geq 0$
    local y: REAL
    do
        from Result := 1; y := a
        until Result = y
        loop
            Result := (Result + y)/2
            y := a / Result
        end
    ensure Result $\geq 0 \wedge$ Result $*$ Result = a
```

Postcondition mutations would correctly find invariant **Result** $* \ y = a$ (by term dropping and uncoupling), but Boogie cannot verify that it is an invariant because the embedded theorem prover Z3 does not handle reasoning about properties of products of numeric variables [27]. If we can verify by other means that a candidate is indeed an invariant, the postcondition mutation technique of this paper would be effective over additional classes of programs.

6.3 Related work

The amount of research work on the automated inference of invariants is formidable and spread over more than three decades; this reflects the cardinal role that invariants play in the formal analysis and verification of programs. This section outlines a few fundamental approaches and provides some evidence that this paper's technique is complementary, in terms of kinds of invariants inferred, to previously published approaches. For more references, in particular regarding software engineering applications, see the "related work" section of [15].

Static methods Historically, the earliest methods for invariant inference where *static* as in the pioneering work of Karr [23]. Abstract interpretation and the constraint-based approach are the two most widespread frameworks for static invariant inference (see also [6, Chap. 12]).

Abstract interpretation is, roughly, a symbolic execution of programs over abstract domains that over-approximates the semantics of loop iteration. Since the seminal work by Cousot and Cousot [10], the technique has been updated and extended to deal with features of modern programming languages such as object-orientation and heap memory-management (e.g., [28, 8]).

Constraint-based techniques rely on sophisticated decision procedures over non-trivial mathematical domains (such as polynomials or convex polyhedra)

[4]The routine does assumes infinite-precision reals and does not terminate.

to represent concisely the semantics of loops with respect to certain template properties.

Static methods are sound — as is the technique introduced in this paper — and often complete with respect to the class of invariants that they can infer. Soundness and completeness are achieved by leveraging the decidability of the underlying mathematical domains they represent; this implies that the extension of these techniques to new classes of properties is often limited by undecidability. In fact, state-of-the-art static techniques can mostly infer invariants in the form of "well-behaving" mathematical domains such as linear inequalities [11, 9], polynomials [38, 37], restricted properties of arrays [7, 5, 20], and linear arithmetic with uninterpreted functions [1]. Loop invariants in these forms are extremely useful but rarely sufficient to prove full functional correctness of programs. In fact, one of the main successes of abstract interpretation has been the development of sound but incomplete tools [2] that can verify the absence of simple and common programming errors such as division by zero or void dereferencing. Static techniques for invariant inference are now routinely part of modern static checkers such as ESC/Java [18], Boogie/Spec# [26], and Why/Krakatoa/Caduceus [22].

The technique of the present paper is complementary to most static techniques in terms of the kinds of invariant that it can infer, because it derives invariants directly from postconditions. In this respect "classic" static inference and our inference by means of postcondition mutation can fruitfully work together to facilitate functional verification; to some extent this happens already when complementing Boogie's built-in facilities for invariant inference with our own technique.

[34, 21, 13, 25, 24] are the approaches that, for different reasons, share more similarities with ours. To our knowledge, [34, 21, 13, 25] are the only other works applying a static approach to derive loop invariants from annotations. [21] relies on user-provided assertions nested within loop bodies and essentially tries to check whether they hold as invariants of the loop. This does not release the burden of writing annotations nested within the code, which is quite complex as opposed to providing only contracts in the form of pre and postconditions. In practice, the method of [21] works only when the user-provided annotations are very close to the actual invariant; in fact the few examples where the technique works are quite simple and the resulting invariants are usually obtainable by other techniques that do not need annotations. [13] briefly discusses deriving the invariant of a *for* loop from its postcondition, within a framework for reasoning about programs written in a specialized programming language. [25] also leverages specifications to derive intermediate assertions, but focusing on lower-level and type-like properties of pointers. On the other hand, [34] derives candidate invariants from postconditions in a very different setting than ours, with symbolic execution and model-checking techniques.

Finally, [24] derives complex loop invariants by first encoding the loop semantics as recurring relations and then instructing a rewrite-based theorem prover to try to remove the dependency on the iterator variable(s) in the relations. It shares with our work a practical attitude that favors powerful heuristics over completeness and leverages state-of-the-art verification tools to boost the inference of additional annotations.

Dynamic methods More recently, dynamic techniques have been applied to invariant inference. The Daikon approach of Ernst et al. [15] showed that dynamic inference is practical and sprung much derivative work (e.g., [35, 12, 36] and many others). In a nutshell, the Daikon approach consists in testing a large number of candidate properties against several program runs; the properties that are not violated in any of the runs are retained as "likely" invariants. This implies that the inference is not sound but only an "educated guess": dynamic invariant inference is to static inference what testing is to program proofs. Nonetheless, just like testing is quite effective and useful in practice, dynamic invariant inference is efficacious and many of the guessed invariants are indeed sound.

Our approach shares with the Daikon approach the idea of guessing a candidate invariant and testing it *a posteriori*. There is an obvious difference between our approach, which retains only invariants that can be soundly verified, and dynamic inference techniques, which rely on a finite set of tests. A deeper difference is that Daikon guesses candidate invariants almost blindly, by trying out a pre-defined set of user-provided templates (including comparisons between variables, simple inequalities, and simple list comprehensions). On the contrary, our technique assumes the availability of contracts (and postconditions in particular) and leverages it to restrict quickly the state-space of search and get to good-quality loop invariants in a short time. As it is the case for static techniques, dynamic invariant inference methods can also be usefully combined with our technique, in such a way that invariants discovered by dynamic methods boost the application of the postcondition-mutation approach.

Program construction Classical formal methods for program construction [14, 19, 29, 32] have first described the idea of deriving loop invariants from postconditions. Several of the heuristics that we discussed in Section 2 are indeed a rigorous and detailed rendition of some ideas informally presented in [29, 19]. In addition, the focus of the seminal work on program construction is to derive systematically an implementation from a complete functional specification. In this paper the goal is instead to enrich the assertions of an already implemented program and to exploit its contracts to annotate the code with useful invariants that facilitate a functional correctness proof.

7 Conclusion and future work

As we hope to have shown, taking advantage of postconditions makes it possible to obtain loop invariants through effective techniques — not as predictable as the algorithms that yield verification conditions for basic constructs such as assignments and conditionals, but sufficiently straightforward to be applied by tools, and yielding satisfactory results in many practical cases.

The method appears general enough, covering most cases in which a programmer with a strong background in Hoare logic would be able at some effort to derive the invariant, but a less experienced one would be befuddled. So it does appear to fill what may be the biggest practical obstacle to automatic program proving.

The method requires that the programmer (or a different person, the "proof engineer", complementing the programmer's work, as testers traditionally do) provide the postcondition for every routine. As has been discussed in Section 1, we feel that this is a reasonable expectation for serious development, reflected in the Design by Contract methodology. For some people, however, the very idea of asking programmers or other members of a development team to come up with contracts of any kind is unacceptable. With such an a priori assumption, the results of this paper will be of little practical value; the only hope is to rely on invariant inference techniques that require the program only (complemented, in approaches such as Daikon, by test results and a repertoire of invariant patterns).

Some of the results that the present approach yields (sometimes trivially) when it is applied manually, are not yet available through the tools used in the current implementation of gin-pink. Although undecidability results indicate that program proving will never succeed in all possible cases, it is fair to expect that many of these limitations — such as those following from Z3's current inability to handle properties of products of variables — will go away as proof technology continues to progress.

We believe that the results reported here can play a significant role in the effort to make program proving painless and even matter-of-course. So in addition to the obvious extensions — making sure the method covers all effective patterns of postcondition mutation, and taking advantage of progress in theorem prover technology — our most important task for the near future is to integrate the results of this article, as unobtrusively as possible for the practicing programmer, in the background of a verification environment for contracted object-oriented software components.

Acknowledgements. A preliminary version of this work has been presented at the IFIP TC2 WG 2.3 meeting in Lachen, Switzerland, March 2010. The authors thank the attendees for their useful comments and criticism.

References

[1] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and Andreas Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM, 2003.

[3] Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL-Boogie — an interactive prover for the Boogie program-verifier. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2008.

[4] Robert S. Boyer and J. Strother Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[5] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, and Tomáš Vojnar. Automatic verification of integer array programs. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2009.

[6] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation.* Springer, 2007.

[7] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

[8] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2005.

[9] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification(CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[11] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96, 1978.

[12] Christoph Csallner, Nikolai Tillman, and Yannis Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 281–290. ACM, 2008.

[13] Guido de Caso, Diego Garbervetsky, and Daniel Gorín. Reducing the number of annotations in a verification-oriented imperative language. In *Proceedings of Automatic Program Verification*, 2009.

[14] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[15] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions of Software Engineering*, 27(2):99–123, 2001.

[16] Jean-Christophe Filliâtre. *The WHY verification tool*, 2009. Version 2.18, `http://proval.lri.fr`.

[17] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe (FME'01)*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[18] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM, 2002.

[19] David Gries. *The science of programming*. Springer-Verlag, 1981.

[20] Thomas A. Henzinger, Thibaud Hottelier, Laura Kovács, and Andrei Voronkov. Invariant and type inference for matrices. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'10)*, Lecture Notes in Computer Science. Springer, 2010.

[21] Mikoláš Janota. Assertion-based loop invariant generation. In *Proceedings of the 1st International Workshop on Invariant Generation (WING'07)*, 2007.

[22] Claude Marché Jean-Christophe Filliâtre. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

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

[24] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In Marsha Chechik and Martin Wirsing, editors, *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, volume 5503 of *Lecture Notes in Computer Science*, pages 470–485. Springer, 2009.

[25] Shuvendu K. Lahiri, Shaz Qadeer, Juan P. Galeotti, Jan W. Voung, and Thomas Wies. Intra-module inference. In Ahmed Bouajjani and Oded Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 2009.

[26] K. Rustan M. Leino. This is Boogie 2. (Manuscript KRML 178) `http://research.microsoft.com/en-us/projects/boogie/`, June 2008.

[27] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09)*, pages 615–622. ACM Press, 2009.

[28] Francesco Logozzo. Automatic inference of class invariants. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2004.

[29] Bertrand Meyer. A basis for the constructive approach to programming. In Simon H. Lavington, editor, *Proceedings of IFIP Congress 1980*, pages 293–298, 1980.

[30] Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, 2nd edition, 1997.

[31] Bertrand Meyer. *Touch of Class: learning to program* well *with objects and contracts.* Springer, 2009.

[32] Carroll Morgan. *Programming from Specifications.* Prentice Hall, 2nd edition, 1994.

[33] Ian Parberry and William Gasarch. *Problems on Algorithms.* `http://www.eng.ent.edu/ian/books/free/`, 2002.

[34] Corina S. Păsăreanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[35] Jeff H. Perkings and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In Richard N. Taylor and Matthew B. Dwyer, editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*, pages 23–32. ACM, 2004.

[36] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 93–104, 2009.

[37] Enric Rodríguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

[38] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 318–329. ACM, 2004.

[39] Wolfram Schulte, S. Xia, Jan Smans, and Frank Piessens. A glimpse of a verifying C compiler (extended abstract). In *C/C++ Verification Workshop*, 2007.

[40] Julian Tschannen. Automatic verification of Eiffel programs. Master's thesis, Chair of Software Engineering, ETH Zürich, 2009.