Program Verification Using Automatic Generation of Invariants * **

Enric Rodríguez-Carbonell, Deepak Kapur

LSI Department, Technical University of Catalonia
Jordi Girona, 1-3 08034 Barcelona (Spain)
`erodri@lsi.upc.es`
Department of Computer Science, University of New Mexico
Albuquerque, NM 87131-0001 (USA)
`kapur@cs.unm.edu`

Abstract. In an earlier paper, an algorithm based on algebraic geometry was developed for discovering polynomial invariants in loops without nesting, not requiring any a priori bound on the degree of the invariants. Polynomial invariants were shown to form an ideal, a basis of which could be computed using Gröbner bases methods. In this paper, an abstract logical framework is presented for automating the discovery of invariants for loops without nesting, of which the algorithm based on algebraic geometry and Gröbner bases is one particular instance. The approach based on this logical abstract framework is proved to be *correct* and *complete*. The techniques have been used with a verifier to automatically check properties of many non-trivial programs with considerable success. Some of these programs are discussed in the paper to illustrate the effectiveness of the method.

1 Introduction

There has recently been a surge of interest in research on automatic generation of loop invariants of imperative programs. This is perhaps due to the successful development of powerful automated reasoning tools including BDD packages, SAT solvers, model checkers, decision procedures for common data structures in applications (such as numbers, lists, arrays, ...), as well as theorem provers for first-order logic, higher-order logic and induction. These tools have been successfully used in application domains such as hardware circuits and designs, software and protocol analysis.

In an earlier paper [RCK04], an approach for generating polynomials as invariants for loops without nesting was presented. An algorithm employing Gröbner basis method was proposed to derive conjunctions of polynomial equalities as loop invariants, without any a priori bound on the degree of the polynomials appearing in the invariants. The main contributions of that paper are:

* This research was partially supported by an NSF ITR award CCR-0113611, the Prince of Asturias Endowed Chair in Information Science and Technology at the University of New Mexico, a Spanish FPU grant ref. AP2002-3693, and the Spanish project MCYT TIC2001-2476-C03-01.

** An extended version of this paper is available at `www.lsi.upc.es/~erodri`.

1. Invariant polynomials are shown to form an *ideal*, a well-known algebraic concept [CLO98]. Consequently, algebraic geometry techniques are brought into play to discover such invariants from a given program without imposing any a priori bound on their degrees.
2. The proposed algorithm for generating polynomial invariants is proved to terminate using algebraic geometry if right-hand sides of assignments are *solvable* mappings either commuting or with positive eigenvalues (see [RCK04] for definitions and details).

In this paper we improve our previous work in a number of aspects:

1. The construction of [RCK04] is generalized and presented in an abstract logical framework, thus highlighting the key properties required for the proposed approach to be applicable to data structures other than numbers. We were thus able to abstract properties needed from algebraic geometry for our results in [RCK04].
2. The abstract framework is based on the forward propagation semantics of program statements. A fixed point computation of formulas approximating the invariant at the loop entry point is carried out by considering all possible execution paths.
3. A procedure for computing loop invariants based on this abstract logical framework is presented. The procedure is proved to be *sound* and *complete*, in the sense that on termination, the procedure generates the strongest possible invariant expressible in the considered language for specifying invariants.
4. The significance of this framework is demonstrated by showing our algorithm in [RCK04] as a nontrivial instance of this abstract procedure.
   In another paper [RCK], we have used the abstract interpretation framework for developing approximations and a widening operator to compute polynomial invariants of a bounded degree, where the bound on their degree is determined by the widening operator. The termination proof of this algorithm is different from the one in [RCK04]; it is based on using the dimension of the vector space generated by polynomials of bounded degree. The advantage of our framework over abstract interpretation is that we are able to ensure that we generate the *strongest* invariant expressible in the language, which is not usually possible in abstract interpretation.
5. The procedure has been implemented and is employed with our tools for program verification to prove the correctness of a number of programs, as shown in a table of examples. Some of these are used for illustrating the key ideas of the approach. Currently, the procedure only generates conjunctions of polynomial equalities as invariants, but plans are underway to generate polynomial inequalities as well.

The rest of the paper is organized as follows. In the next subsection, related work is briefly reviewed. Section 2 introduces the general framework: the programming model is presented, and necessary properties of the language for expressing invariants are studied so that the generic procedure for finding loop invariants can be formulated. In Section 3, we prove that the language of conjunctions of polynomial equalities satisfies all the required properties of the abstract framework and is thus an instance of it. This gives an algorithm for computing invariant polynomial equalities that turns out to be equivalent to that given in [RCK04]. In Section 4 we show that the framework is applicable even when some of the conditions on the language to express invariants are not met. Section 5 is a brief overview of the verifier we have built for proving properties of programs. In Section 6 we give some illustrative examples of program verification using this tool. Finally, Section 7 concludes with a discussion on future research.

1.1 Related Work

The generation of arithmetic invariants between numerical variables is a long researched area. Karr first showed in [Kar76] an algorithm for finding invariant *linear equalities* at any program point of a procedure. This work was extended by Cousot and Halbwachs [CH78], who applied the model of abstract interpretation [CC77] for finding invariant *linear inequalities*. Like our techniques, both methods are based on forward propagation and fixed point computation [Weg75], which points out that our ideas may be useful for accelerating the termination as well as improving the precision in abstract interpretation.

Recently, there has been a renewed surge of interest in automatically deriving invariants of imperative programs. In [CSS03], Colón et al. used non-linear constraint solving based on Farkas' lemma to attack the problem of finding invariant linear inequalities. Extending Karr's work, Müller-Olm and Seidl [MOS04] proposed an interprocedural method for computing invariant polynomial equalities of bounded degree in programs with affine assignments. The same authors [MOS03] developed a complete technique for finding invariants of a prefixed form in procedures with polynomial assignments and disequality guards. Similarly, in [SSM04] a method was proposed for generating polynomials as invariants, which starts with a template polynomial with undetermined coefficients and attempts to find values for the coefficients so that the template is invariant using the Gröbner basis algorithm. Kapur proposed a related approach using quantifier elimination in November 2003 (see [Kap03]).

In [RCK04], we gave an algorithm based on algebraic geometry and not requiring any degree bounds for generating conjunctions of polynomial equalities as loop invariants. This algorithm served as the basis for developing the proposed abstract logical framework of this paper. In that paper, the discussion and proofs extensively use results of polynomial ideal theory and algebraic geometry, because of which they are not likely to be directly applicable to other data structures such as arrays, records, etc. In contrast, this paper presents a logical framework that is likely to be more widely applicable. Finally, in [RCK] we have employed the framework of abstract interpretation to generate polynomial equalities of bounded degree as invariants in general procedures.

2 Abstract Framework

We consider a simple programming language with multiple assignments, nondeterministic conditional statements and loop constructs. Loops are assumed to have the following form:

```
while E(x̄) do
    if C_1(x̄) → x̄ :=f_1(x̄);
    ...
    [] C_i(x̄) → x̄ :=f_i(x̄);
    ...
    [] C_n(x̄) → x̄ :=f_n(x̄);
    end if
end while
```

where $\bar{x} = (x_1, x_2, ..., x_m)$ denotes the tuple of program variables, E, C_i's are boolean expressions and each f_i is an m-tuple of expressions.

2.1 Loop Invariants

A formula expressing a property of a loop (including an invariant of the loop) is specified using the program variables $\bar{x}$ and variables, denoted by $\bar{x}^*$, representing the initial, usually unknown, values of the program variables before entering the loop.

Let $\mathcal{R}$ stand for a subset of a first-order language with equality used for expressing properties of loops. A formula in $\mathcal{R}$ representing an invariant property will be written as $R(\bar{x}, \bar{x}^*)$. Our goal is to capture the semantics of loops using the strongest invariant expressible in the language $\mathcal{R}$. For that we characterize the expressiveness of $\mathcal{R}$ to admit such strongest invariants.

Definition 1. *A formula $R \in \mathcal{R}$ is* invariant *(with respect to another formula $R_0(\bar{x}^*)$ relating initial values of $\bar{x}$) if:*
i) $R_0(\bar{x}^*) \Rightarrow R(\bar{x}^*, \bar{x}^*)$ *and*
ii) $\forall i : 1 \leq i \leq n,\ (R(\bar{x}, \bar{x}^*) \wedge E(\bar{x}) \wedge C_i(\bar{x})) \Rightarrow R(f_i(\bar{x}), \bar{x}^*)$.

To capture the semantics of the loop, we have to compute the strongest possible invariant in the language $\mathcal{R}$:

Definition 2. *The language $\mathcal{R}$ is* expressive *for a loop if $\exists R_\infty \in \mathcal{R}$ such that*

1. *R_∞ is an invariant of the loop and*
2. *for every invariant R of the loop in the language $\mathcal{R}$, $R_\infty(\bar{x}, \bar{x}^*) \Rightarrow R(\bar{x}, \bar{x}^*)$.*

In Section 3, the language of conjunctions of polynomial equalities is introduced for specifying invariants, and it is shown to be expressive for loops with polynomial assignments (when tests are abstracted and considered to be *true*).

2.2 Fixed Point Procedure for Computing Invariants

We give an iterative procedure for computing the strongest invariant R_∞ of a given loop. Assume that the loop test E and each C_i, the tests in the conditional statement, and each assignment mapping f_i are expressible in $\mathcal{R}$. Let R_0 stand for a formula satisfied by the initial values of the variables before entering the loop. Based on the forward propagation semantics of program statements, the procedure below computes successive approximations of the strongest invariant R_∞ until reaching a fixed point.

Forward Propagation Semantics. If $R(\bar{x}, \bar{x}^*)$ holds at the loop entry point, the loop test $E(\bar{x})$ is true and the i-th conditional branch is executed, then the strongest postcondition at the end of the body of the loop is

$$\exists \bar{y}(\bar{x} = f_i(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge E(\bar{y}) \wedge C_i(\bar{y}))\,.$$

Traditionally abstract interpretation uses this one-step forward propagation to compute invariants, employing a widening operator to guarantee termination. In order to accelerate the procedure for finding invariants and avoid the loss of precision involved in widening, we propose instead a many-step forward propagation along the lines of the meta-transitions of Boigelot [Boi99]. While these meta-transitions were originally utilized to compute the exact reach set of a system, we apply accelerations to the more general problem of computing overapproximations of the set of reachable states, i.e. invariants, in a given specification language.

If the i-th branch is executed s times in a row, the strongest postcondition is:

$$\exists \bar{y}\Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\Big)\Big)\,,$$

assuming that the s-th power of f_i is also expressible in $\mathcal{R}$.

Given that the number of iterations s is undetermined, an infinite disjunction is needed to express the relation (which is not a formula anymore in the language unless existential quantifiers are used):

$$\bigvee_{s=1}^{\infty} \Big(\exists \bar{y}\Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\Big)\Big)\Big)\,. \tag{1}$$

In general, there are several branches in a loop and each of the branches can be executed arbitrarily many times. This results in an infinitary formula capturing the program states at the loop entry point after an undetermined branch has been executed arbitrarily many times:

$$R(\bar{x}, \bar{x}^*) \vee \Big(\bigvee_{i=1}^{n} \bigvee_{s=1}^{\infty} \Big(\exists \bar{y}\Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\big)\Big)\Big)\Big)\,.$$

In order to capture the semantics of the loop, this approximation of the invariant is computed iteratively until reaching a fixed point (or going on forever). This is the core of the procedure below.

In a highly powerful language for expressing invariants, the infinite disjunction in the above infinitary formula can perhaps be expressed using an equivalent formula with the help of existential quantifiers. If the language does not permit existential quantifiers or even disjunctions (as will be the case for the language of conjunctions of polynomial equalities), the language must be able to express a sufficiently strong approximation.

Definition 3. *The language* $\mathcal{R}$ *is* disjunctively closed *if* $\forall R, S \in \mathcal{R}$, $\exists T \in \mathcal{R}$, *written as* $R \sqcup S$, *such that*

i) $(R(\bar{x}, \bar{x}^*) \vee S(\bar{x}, \bar{x}^*)) \Rightarrow T(\bar{x}, \bar{x}^*)$,

ii) $\forall T' \in \mathcal{R}$, *if* $(R(\bar{x}, \bar{x}^*) \vee S(\bar{x}, \bar{x}^*)) \Rightarrow T'(\bar{x}, \bar{x}^*)$, *then* $T(\bar{x}, \bar{x}^*) \Rightarrow T'(\bar{x}, \bar{x}^*)$.

The language of first-order predicate calculus with equality is disjunctively closed, as $\vee$ can be taken as $\sqcup$. So is the language of polynomial equalities closed under conjunction: for any R and S that are conjunctions of polynomial equalities, there is another conjunction of polynomial equalities $R \sqcup S$ which is equivalent to $R \vee S$, as shown later. The language of conjunctions of linear inequalities, used in [CH78], is also disjunctively closed: given conjunctions R and S of linear inequalities, $R \sqcup S$ is defined as the convex hull of R and S; unlike the previous cases, $R \sqcup S$ is, in general, not equivalent to $R \vee S$ in this case.

To get approximations in $\mathcal{R}$ of infinitary formulas involving infinite disjunctions as well as existential quantifiers, $\mathcal{R}$ is required to have some additional properties. For the i-th conditional branch we assume that $\exists \varphi_i(R) \in \mathcal{R}$, the strongest formula in the language implied by the formula (1) above:

Definition 4. $\mathcal{R}$ allows existential elimination *if* $\forall R \in \mathcal{R}$, $\forall i : 1 \leq i \leq n$ $\exists T \in \mathcal{R}$, *written as* $\varphi_i(R)$, *such that*

i) $$\bigvee_{s=1}^{\infty} \exists \bar{y} \Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\Big)\Big) \Rightarrow T(\bar{x}, \bar{x}^*)\,.$$

ii) $\forall T' \in \mathcal{R}$ *such that*

$$\bigvee_{s=1}^{\infty} \exists \bar{y} \Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\Big)\Big) \Rightarrow T'(\bar{x}, \bar{x}^*)\,,$$

$$T(\bar{x}, \bar{x}^*) \Rightarrow T'(\bar{x}, \bar{x}^*).$$

In order to check whether the fixed point has already been reached, we additionally need to decide whether two formulas in the language are equivalent:

Definition 5. $\mathcal{R}$ allows equivalence check *if* $\forall R, S \in \mathcal{R}$, *it can be decided whether* $R \Leftrightarrow S$ *or not.*

After replacing disjunctions $\vee$ by $\sqcup$ and eliminating existential quantifiers by means of the φ_i's, we get the procedure below. It starts assigning to the formula variable R an initial formula satisfied by the initial values of the variables in the loop. This variable R stores the formula corresponding to the successive approximations of the invariant.

<u>Invariant Generation Procedure</u>

Input: Assignment mappings $f_1, ..., f_n$
Formula R_0 satisfied by the initial values of program variables
Output: Strongest invariant formula R_∞

var R, R' : formulas in $\mathcal{R}$ **end var**
$R' := false$
$R := (x_1 = x_1^*) \wedge \cdots \wedge (x_m = x_m^*) \wedge R_0$
while $R' \not\Leftrightarrow R$ **do**

$R' := R$
$$R := R \sqcup^{1} \left(\bigsqcup_{i=1}^{n} \varphi_i(R) \right) \tag{2}$$
end while
return R

The following theorem captures the correctness and completeness of the above procedure:

Theorem 1. *Given a loop* $\mathbf{L}$*, let* $\mathcal{R}$ *be a language expressive for* $\mathbf{L}$*, disjunctively closed and admitting existential elimination and equivalence check. Let* R_∞ *stand for the strongest invariant of* $\mathbf{L}$ *in the language. If the invariant generation procedure terminates with output* R*, then* $R(\bar{x}, \bar{x}^*) \Leftrightarrow R_\infty(\bar{x}, \bar{x}^*)$*.*

The proof of the theorem, given in the extended version of the paper, is based on two facts: *i*) if the procedure terminates, R is an invariant of the loop; and *ii*), $R \Rightarrow R_\infty$ holds in all steps of the invariant generation procedure. So, if the procedure terminates, $R \Rightarrow R_\infty$ and $R_\infty \Rightarrow R$, which finally leads to the result that the above procedure on termination indeed computes the strongest invariant of the loop.

The key issue in the procedure is to find the appropiate definition for the $\sqcup$ operator as well as for the φ_i functions in order to ensure termination. In the following section we show a nontrivial instance of language satisfying these requirements, the language of polynomial equalities. Further, in Section 4 we will see how, even if the specification language only satisfies some of the requirements for disjunctive closedness and quantifier elimination, the procedure can still yield useful results on termination.

3 Conjunctions of Polynomial Equalities as Invariants

In this section we show that the language of conjunctions of polynomial equalities, denoted by $\mathcal{P}$, is a particular instance of the abstract framework. Assuming that the guards are ignored (i.e. $E = C_i = true$) and that the assignment mappings are polynomial, it is shown that this language $\mathcal{P}$ satisfies all the requirements discussed above (Section 3.2). The above iterative procedure for computing invariants can be instantiated as well (Section 3.3).

3.1 Preliminaries

Given a field $\mathbb{K}$, let $\mathbb{K}[\bar{z}] = \mathbb{K}[z_1, ..., z_l]$ denote the ring of polynomials in the variables $z_1, ..., z_l$ with coefficients from $\mathbb{K}$. An *ideal* is a set $I \subseteq \mathbb{K}[\bar{z}]$ that contains 0, is closed under addition and such that, if $p \in \mathbb{K}[\bar{z}]$ and $q \in I$, then $pq \in I$. Given a set of polynomials $S \subseteq \mathbb{K}[\bar{z}]$, the *ideal spanned by* S is $\langle S \rangle = \{p \in \mathbb{K}[\bar{z}] \mid \exists k \geq 1 \; p = \sum_{j=1}^{k} p_j q_j \text{ with } p_j \in \mathbb{K}[\bar{z}], q_j \in S\}$. For an ideal I, a set $S \subseteq \mathbb{K}[\bar{z}]$ such that $I = \langle S \rangle$ is called a *basis* of I.

[1] The use of $\sqcup$ approximating $\vee$ here may not be sufficient to guarantee termination. Using a widening operator ∇ instead of $\sqcup$ ([CC77]) as a further approximation of $\vee$ can ensure the termination of the procedure, probably at the cost of completeness.

The *variety* of a set $S \subseteq \mathbb{K}[\bar{z}]$ over $\mathbb{K}^l$ is defined as its set of zeroes, $\mathbf{V}(S) = \{\bar{\alpha} \in \mathbb{K}^l \mid p(\bar{\alpha}) = 0 \; \forall p \in S\}$. On the other hand, if $A \subseteq \mathbb{K}^l$, the ideal $\mathbf{I}(A) = \{p \in \mathbb{K}[\bar{z}] \mid p(\bar{\alpha}) = 0 \; \forall \bar{\alpha} \in A\}$ is the *annihilator* of A.

A mapping $g : \mathbb{K}^l \to \mathbb{K}^l$ is *affine* if it is of the form $g(\bar{z}) = A\bar{z} + b$, where $A \in \mathbb{K}^{l \times l}$ and $b \in \mathbb{K}^l$. In general, a mapping $g \in \mathbb{K}[\bar{z}]^l$ is a *polynomial mapping.*

To each conjunction of polynomial equalities $R \equiv (p_1 = 0 \wedge \cdots \wedge p_k = 0) \in \mathcal{P}$, we associate the ideal $J = \langle p_1, \cdots, p_k \rangle$. Similarly, given an ideal J specified by a finite basis, say B, there is a formula in $\mathcal{P}$ (not necessarily unique) associated with it, written as $\bigwedge_{p \in B}(p = 0)$; depending upon the basis chosen for J, different (but equivalent) formulas can be obtained.

3.2 Properties of $\mathcal{P}$

Expressiveness. Given a loop, the language $\mathcal{P}$ is expressive, i.e., there exists a formula R_∞ in $\mathcal{P}$ such that (i) R_∞ is an invariant of the loop, and (ii) any formula R in $\mathcal{P}$ that is an invariant of the loop is implied by R_∞. The idea of the proof is to take a basis of the ideal generated by all the polynomials that are invariants of the loop. By Hilbert's basis theorem, such an infinite basis has an equivalent finite basis. The conjunction of the polynomial equalities corresponding to the polynomials in the finite basis is precisely R_∞.

Disjunctive Closedness. The language $\mathcal{P}$ is disjunctively closed: if $R \equiv p_1 = 0 \wedge ... \wedge p_k = 0$ and $S \equiv q_1 = 0 \wedge ... \wedge q_l = 0$, there is a conjunction of polynomial equalities $R \sqcup S$ that is equivalent to $R \vee S$. This formula can be constructed by computing a finite basis of the intersection ideal $\langle p_1, ..., p_k \rangle \cap \langle q_1, ..., q_l \rangle$ and taking the corresponding conjunction of polynomial equalities (since ideals of polynomials are always finitely generated by Hilbert's basis theorem).

Existential Elimination. For the i-th conditional branch, we need to show the existence of $\varphi_i(R) \in \mathcal{R}$, the strongest formula in the language implied by the infinitary formula (1) above in Section 2.2. Such a formula in $\mathcal{P}$ can be obtained by computing a finite basis B of the ideal

$$\mathbb{K}[\bar{x}, \bar{x}^*] \bigcap \left(\bigcap_{s=1}^{\infty} \left\langle (-\bar{x} + f_i^s(\bar{y})) \bigcup \left(\bigcup_{p \in \mathbf{IV}(I)} p(\bar{y}, \bar{x}^*) \right) \right\rangle \right), \tag{3}$$

where $I = \langle p_1, ..., p_k \rangle$ is the ideal associated to the formula $R \equiv p_1 = 0 \wedge ... \wedge p_k = 0$ and $-\bar{x} + f_i^s(\bar{y})$ denotes the set of m polynomials corresponding to the projections over each of the m coordinates. Then $\varphi_i(R) \equiv (\bigwedge_{q \in B} \; q = 0)$ is the strongest formula in $\mathcal{P}$ implied by (1).

Equivalence Check. The language $\mathcal{P}$ admits equivalence check: if $R \equiv p_1 = 0 \wedge ... \wedge p_k = 0$ and $S \equiv q_1 = 0 \wedge ... \wedge q_l = 0$, then $R \Leftrightarrow S$ is equivalent to $\mathbf{IV}(p_1, ..., p_k) = \mathbf{IV}(q_1, ..., q_l)$. In case that $\mathbf{IV}(p_1, ..., p_k) = \langle p_1, ..., p_k \rangle$ and $\mathbf{IV}(q_1, ..., q_l) = \langle q_1, ..., q_l \rangle$ (which is common in practice), then obviously $R \Leftrightarrow S$ is equivalent to $\langle p_1, ..., p_k \rangle = \langle q_1, ..., q_l \rangle$, which can be easily checked.

3.3 Generating Conjunctions of Polynomial Equalities as Invariants

The abstract procedure discussed in Section 2.2 can be instantiated to be the algorithm presented in [RCK04] as follows. The assignment labelled as (2) in the abstract procedure is the most non-trivial and complicated to perform: it requires computing a basis of the ideal defined by the expression (3) for each assignment mapping, which involves an infinite intersection of ideals. To compute this infinite intersection, elimination theory is used to eliminate s and possibly other auxiliary variables needed to express the f_i^s's as polynomials. In order to represent the f_i^s's as polynomials, we ask assignment mappings to be *solvable mappings*, a particular case of polynomial mappings; solvable mappings are an extension of affine mappings. The following theorem is proved in [RCK04]; the reader can refer to that paper for details about the theorem as well as the proof.

Theorem 2. *Let* $\mathbf{L}$ *be a loop with tests* $E = C_i = true$ *and assignments* $\bar{x} := f_i(\bar{x})$, $1 \leq i \leq n$. *If each of the assignment mappings* f_i *is a solvable mapping with positive rational eigenvalues, the procedure computes the strongest invariant in at most* $2m + 1$ *steps, where* m *is the number of program variables in* $\mathbf{L}$. *Moreover, if the assignment mappings commute, i.e.* $f_i \circ f_j = f_j \circ f_i$ *for* $1 \leq i, j \leq n$, *then the algorithm terminates in at most* $n + 1$ *steps, where* n *is the number of branches in the non-deterministic conditional statement of the body of* $\mathbf{L}$.

The proof of the first part of the theorem extensively uses algebraic geometry concepts including irreducible decomposition of varieties and their dimension. It is our experience that for instantiating the abstract framework, the most nontrivial task is to find conditions under which the procedure for generating invariants terminates.

4 Heuristic Procedure for Non-expressive Languages

In the previous section, we showed how the language of conjunctions of polynomial equalities satisfies all the conditions required in the abstract logical framework. We thus get a sound and complete algorithm for computing conjunction of polynomial equalities as invariants; further, the invariant generated by the procedure is the strongest possible invariant expressible in this language.

In this section we show that our abstract framework is still useful when the language $\mathcal{R}$ for specifying invariants is not expressive for the loop. Namely, if the language admits equivalence check and the conditions $i)$ of both definitions of disjunctive closedness and existential elimination are satisfied, then the invariant generation procedure can still be formulated and yields correct invariants on termination. The invariant generated by the procedure, however, needs not be the strongest possible one as the non-expressiveness of the language implies that there is no such strongest invariant.

For example, let us consider the first-order language of quantifier-free formulas with polynomial inequalities as atoms, which subsumes conjunctions of linear inequalities [CH78] or polynomial equalities [RCK04,RCK]. This language admits equivalence check and is in fact disjunctively closed; and moreover, it non-trivially satisfies condition $i)$ in the definition of existential elimination, as we can use quantifier elimination [Tar51] to get rid of infinite disjunctions and existential quantifiers. We illustrate this with the following simple program, for which our procedure finds a correct invariant in the language under consideration:

{Pre: $n \geq 0$}
a:=0; **while** $(a+1)^2 \leq n$ **do** a:=$a+1$; **end while**

In this case, given a formula $R = R(a,n,a^*,n^*)$ (where a^*, n^* stand for the initial values of the variables a, n before entering the loop), we can compute the formula for the next iteration by eliminating s, t from

$$\exists s\ (s \geq 0 \wedge R(a-s,n,a^*,n^*) \wedge \forall t\ ((t \geq 0 \wedge t \leq s-1) \Rightarrow (a-t)^2 \leq n))$$

using quantifier elimination. Starting with $R_0(a^*,n^*) \equiv (a^*=0) \wedge (n^* \geq 0)$, after two iterations we get the fixed point $a \geq 0 \wedge a^2 \leq n \wedge n \geq 0 \wedge a^* = 0 \wedge n = n^*$, which is invariant for the loop. Notice that $a \geq 0 \wedge a^2 \leq n \wedge n \geq 0$ contains a non-linear inequality.

However, the first-order quantifier-free language of polynomial inequalities is not expressive in general. The next loop illustrates this fact:

x:=0; **while** *true* **do** x:=$x+1$; **end while**

In this case, the set of reachable states is $\mathbb{N}$. Any invariant formula $R(x,0)$ in the language will hold for all natural numbers. In particular, such a formula will necessarily hold for an interval of real numbers of the form $[x_0,\infty)$, for a certain natural number x_0. Then the formula $R(x,0) \wedge (x = x_0 \vee x \geq x_0+1)$ will also be an invariant in the language, but will be strictly stronger than $R(x,0)$; so there is no strongest invariant. For this example, our invariant generation procedure yields the invariant $x \geq 0$.

5 Verification of Properties of Programs

An implementation in Maple for automatically discovering polynomial loop invariants has been manually interfaced with a prototype of verifier described below. We have successfully used this verifier to automatically prove non-trivial properties of many numerical programs (computing for instance products, divisions, square roots, divisors, ...). Some of these programs are shown in Section 6 to illustrate the power of the techniques.

5.1 Verifier

The verifier takes as input imperative programs with annotated assertions, including preconditions and postconditions. The programming language that it accepts has a similar syntax as that of `C`. It features integer variables, arithmetic operations ($+$, $*$, div, mod, etc.) and function calls. The assertion language is a first-order logic with equality with interpreted function and predicate symbols. Admitted functions are the arithmetic operators of `C` and the gcd, lcm functions, introduced to widen the range of treatable properties. The predicates are equality $=$ and order $>$, $\geq$.

The verifier mainly consists of two components: (i) a *verification condition generator* that produces the formulas that ensure that the desired properties of the program are fulfilled; and (ii) a *theorem prover*, which checks the validity of these formulas.

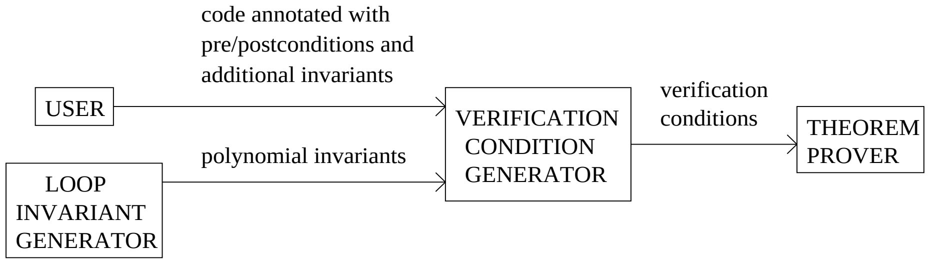

Fig. 1. Scheme of the system for verifying programs

Verification Condition Generator. The verification condition generator is based on Floyd-Hoare-Dijkstra's inductive assertion method. It generates formulas (called *verification conditions*) from the code and the annotations that must be satisfied to guarantee the correctness of the program with respect to the specification. This is done by means of a semantics of language constructs as predicate transformers. Given a program postcondition, this semantics allows to mechanically compute an assertion such that if the precondition implies it, then the postcondition holds on termination of the program.

Theorem Prover. The goal of the theorem prover is to check the validity of the verification conditions.

We initially tried SPASS [WBH+02], a general-purpose theorem prover for first-order-logic with equality. Since the conditions are about integer numbers, SPASS had to be given an axiomatization of the integers explicitly; still, this theorem prover had problems handling formulas requiring algebraic manipulation and knowledge on the integers.

This led us to implement an *ad-hoc* prover in Prolog. In our prover, formulas are proved by simplification using rewriting rules until the tautology *true* is obtained; if this is not the case and the prover cannot rewrite further, then it gives up and the problem of the validity of the formula is unsolved. Strategies for proving formulas are implemented using conditional rewriting rules. This allows us to give the prover a knowledge on numbers that overcomes the power of general theorem provers like SPASS for our concrete application. Our prover has given overall good results, as we shall see in the examples.

5.2 Interface of the Loop Invariant Generator and the Verifier

For the time being, the interface between our implementation in Maple for generating polynomial loop invariants and the verifier is manual; that is to say, the user has to annotate by hand the polynomial invariants obtained by means of the method here described in the code to be verified.

Sometimes an invariant expressed as a conjunction of polynomial equalities is not strong enough to prove the desired properties of a program; then additional invariants are also annotated. Most often there are already methods for automatically finding these assertions, as is the case for linear inequalities [CH78].

6 Examples

In this section we illustrate the power of the proposed techniques for generating polynomial invariants and their effectiveness in proving properties of non-trivial algorithms operating on numbers. Although in some cases the polynomial invariants are not enough to prove the desired properties, we will demonstrate their need in proving properties of programs. For the sake of simplicity, we will focus on the verification conditions that guarantee that the postcondition is met on termination of the program. At the end of the section, a table summarizes the results of applying our tools to a variety of programs. For all the examples here shown, the verifier is powerful enough to check the required properties. The timings are taken in seconds with a Pentium 4 with a 2.5 GHz. processor and 512 MB of memory.

Example 1. The next program is an algorithm for computing the product of two natural numbers A and B. Three of the assignments are non-affine solvable:

function product $(A,\ B$: integer$)$ **returns** q: integer
  $\{$ Pre: $A \geq 0 \wedge B \geq 0\}$
  var a, b, p: integer **end var**
  $(a,\ b,\ p,\ q)$:=$(A,\ B,\ 1,\ 0)$;
  while $(a \neq 0) \wedge (b \neq 0)$ **do**
    if $(a \bmod 2 = 0) \wedge (b \bmod 2 = 0)$
      $\rightarrow (a, b, p, q) := (a \text{ div } 2, b \text{ div } 2, 4p, q)$;
    [] $(a \bmod 2 = 1) \wedge (b \bmod 2 = 0)$
      $\rightarrow (a, b, p, q) := (a - 1, b, p, q + bp)$;
    [] $(a \bmod 2 = 0) \wedge (b \bmod 2 = 1)$
      $\rightarrow (a, b, p, q) := (a, b - 1, p, q + ap)$;
    [] $(a \bmod 2 = 1) \wedge (b \bmod 2 = 1)$
      $\rightarrow (a, b, p, q) := (a - 1, b - 1, p, q + (a + b - 1)p)$;
    end if
  end while
  $\{$ Post: $q = AB\}$

Our algorithm yields the invariant $q + abp = AB$ in 3.32 s. In this case, the verification condition that ensures that the postcondition is met is $(q + abp = AB \wedge (a = 0 \vee b = 0)) \Rightarrow q = AB$ (free variables are implicitly universally quantified); this condition is split into $(q + abp = AB \wedge a = 0) \Rightarrow q = AB$ and $(q + abp = AB \wedge b = 0) \Rightarrow q = AB$, which are reduced to $(q = AB \wedge a = 0) \Rightarrow q = AB$ and $(q = AB \wedge b = 0) \Rightarrow q = AB$ respectively, and then both to *true*. The program is shown to be correct automatically by our system in 0.82 s.

Example 2. The next example, taken from [Dij76], is an extension of Euclid's algorithm for computing the least common multiple of two natural numbers a and b. The invariant generation procedure yields $xu + yv = 2ab$ in 2.02 s.

function lcm $(a, b$: integer$)$ **returns** z: integer
  $\{$ Pre: $a > 0 \wedge b > 0\}$
  var x, y, u, v: integer **end var**
  (x, y, u, v):=(a, b, b, a);

```
{ Inv: gcd(x, y) =gcd(a, b)}
while x ≠ y do
    if x > y → (x, y, u, v):=(x − y, y, u, u + v);
    [] x < y → (x, y, u, v):=(x, y − x, u + v, v);
    end if
end while
{Post : (u + v) div 2 = lcm(a, b)}
```

In this case the auxiliary invariant $\gcd(x, y) = \gcd(a, b)$, which can be obtained automatically by other means ([CP93]), is also needed to prove the post-condition. The verification condition in this case is $(\gcd(x, y) = \gcd(a, b) \wedge xu + yv = 2ab \wedge x = y) \Rightarrow (u + v)$ div $2 = \mathrm{lcm}(a, b)$. Our prover reduces this formula to $\gcd(a, b)\,(u + v) = 2 \gcd(a, b)\,\mathrm{lcm}(a, b) \Rightarrow (u + v)$ div $2 = \mathrm{lcm}(a, b)$ and then to $u + v = 2\,\mathrm{lcm}(a, b) \Rightarrow (u + v)$ div $2 = \mathrm{lcm}(a, b)$, which is trivially valid. The program is shown to be correct automatically in 0.9 s.

Example 3. The following program has been extracted from [Knu69]. It tries to find a divisor d of the natural number N using a parameter D:

```
function divisor (N, D: integer) returns d, r: integer
  { Pre: N > 0 ∧ N mod 2 = 1 ∧ D mod 2 = 1 ∧ D ≥ 2∛n + 1}
  var t, q: integer end var
  (d, r, t, q):=(D, N mod D, N mod (D − 2), 4(N div (D − 2) − N div D));
  { Inv: d mod 2 = 1}
  while d ≤ ⌊√N⌋ ∧ r ≠ 0 do
      (d, r, t, q):=(d + 2, 2r − t + q, r, q);
      if r < 0 → (r, q):=(r + d, q + 4); end if
      if r ≥ d → (r, q):=(r − d, q − 4); end if
      if r ≥ d → (r, q):=(r − d, q − 4); end if
  end while
  { Post: r = 0 ⇒ N mod d = 0}
```

For this program, our invariant generation algorithm yields $d(dq - 4r + 4t - 2q) + 8r = 8N$ as invariant in 24.56 s. In this case we also need the extra invariant $d \bmod 2 \equiv 1$. To prove the postcondition we have to show the validity of $(d(dq - 4r + 4t - 2q) + 8r = 8N \wedge d \bmod 2 \equiv 1 \wedge (r = 0 \vee d > \lfloor\sqrt{N}\rfloor)) \Rightarrow (r = 0 \Rightarrow N \bmod d = 0)$. Our prover reduces this formula into $(d(dq - 4r + 4t - 2q) + 8r = 8N \wedge d \bmod 2 \equiv 1 \wedge r = 0) \Rightarrow N \bmod d = 0$, and then to $(d(dq - 4r + 4t - 2q) = 8N \wedge d \bmod 2 \equiv 1) \Rightarrow N \bmod d = 0$, which is able to prove to be valid. The total time spent on proving the correctness of the program is 2.03 s.

Table of Examples. Table 1 summarizes the results obtained after generating invariants and verifying correctness for a number of programs [2]. There is a row for each program; the columns provide the following information:

1. 1st column is the name of the program; 2nd column states what the program does; 3rd column gives the source where the program was picked from (the entry (∗) is for the examples developed up by the authors).

[2] These programs are available at `www.lsi.upc.es/~erodri`.

2. 4th column gives the number of variables in the program; 5th column gives the number of loops; 6th column is the number of branches for each loop;
3. 7th column gives the number of loop invariants generated for each loop; 8th column is the time taken by the invariant generation.
4. 9th column indicates if any other kind of additional invariants was needed; 10th column is the time taken by the verifier to prove correctness.

Table 1. Table of examples

1	2	3	4	5	6	7	8	9	10
cohencu	cube	[Coh90]	4	1	1	4	2.15	No	0.61
prod4br	product	(∗)	6	1	4	1	3.32	No	0.82
hard	integer division	[SSM04]	6	2	1-2	3-3	7.43	Yes	5.46
divbin	integer division	[Kal90]	5	2	1-2	2-1	4.28	Yes	1.99
dijkstra	integer sqrt	[Dij76]	5	2	1-2	2-1	4.73	Yes	18.88
euclidex2	Bezout's coefs	(∗)	8	1	2	5	3.64	Yes	1.79
lcm2	lcm	[Dij76]	6	1	2	1	2.02	Yes	0.90
fermat2	divisor	[Knu69]	5	1	2	1	2.26	Yes	1.01
factor	divisor	[Knu69]	6	1	4	1	24.56	Yes	2.03

7 Conclusions and Further Research

An abstract framework for automatically discovering invariants for loops without nesting is proposed. A general procedure for that is given if the language used for expressing invariants is *expressive*, *disjunctively closed* and *allows existential elimination* and *equivalence check*. The procedure computes the strongest possible invariant expressible in the language.

It is shown that our earlier results on computing polynomial equalities as invariants are an instance of the abstract logical framework here presented. We are investigating other languages for expressing invariants for which the framework can be adapted. We are particularly interested in the first-order language of polynomial inequalities, which subsumes that of of linear inequalities [CH78].

The framework has been implemented as a part of a verifier for proving properties of programs. The verifier is interfaced with the Maple computer algebra system, which generates conjunctions of polynomial equalities as invariants. The verifier also features a theorem prover able to reason about numbers, so that the formulas representing the desired program properties can be checked to be valid. This scheme has been applied to many non-trivial programs, successfully generating invariants and then verifying properties of these programs.

We believe that the proposed abstract logical framework will also allow us to design expressive languages to specify invariants of loops manipulating data structures such as records, pointers, etc. We regard that the approach will be particularly useful with arrays, since our framework has already been successful in inferring invariants for some loops involving this data structure.

We are also investigating enriching the programming model to consider nested loops as well as procedure calls; the main idea here is to represent all execution paths using regular expressions and define fixed point computations as prescribed by such regular expressions.

Acknowledgements. The authors would like to thank R. Clarisó, G. Godoy, R. Nieuwenhuis and M. Subramaniam for their help and advice.

References

[Boi99] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, Faculté des Sciences Appliquées de l'Université de Liège, 1999.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL 1977*, p. 238–252, 1977.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL 1978*, p. 84–97, 1978.

[CLO98] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer-Verlag, 1998.

[Coh90] Edward Cohen. *Programming in the 1990s.* Springer-Verlag, 1990.

[CP93] R. Chadha and D. A. Plaisted. On the Mechanical Derivation of Loop Invariants. *Journal of Symbolic Computation*, 15(5-6):705–744, 1993.

[CSS03] M. A. Colón, S. Sankaranarayanan, and H.B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *CAV 2003*, volume 2725 of *LNCS*, p. 420–432. Springer-Verlag, 2003.

[Dij76] E. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Kal90] A. Kaldewaij. *Programming. The Derivation of Algorithms.* Prentice-Hall, 1990.

[Kap03] D. Kapur. Automatically Generating Loop Invariants using Quantifier Elimination. Technical Report TR-CS-2003-58, Department of Computer Science, UNM, 2003. Also in *10th International IMACS Conference on Applications of Computer Algebra (ACA 2004)*, Lamar, TX, July 2004.

[Kar76] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[Knu69] D. E. Knuth. *The Art of Computer Programming. Volume 2, Seminumerical Algorithms.* Addison-Wesley, 1969.

[MOS03] M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. Technical report, Fernuni Hagen, 2003. Num. 310. To appear in *IPL*.

[MOS04] M. Müller-Olm and H. Seidl. Computing Interprocedurally Valid Relations in Affine Programs. In *POPL 2004*, p. 330–341, 2004.

[RCK] E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. `www.lsi.upc.es/~erodri`. To appear in *SAS'04*.

[RCK04] E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *ISSAC'04*, 2004.

[SSM04] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear Loop Invariant Generation Using Gröbner Bases. In *POPL 2004*, p. 318–329, 2004.

[Tar51] A. Tarski. A Decision Method for Elementary Algebra and Geometry. University of California Press, 1951.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS version 2.0. In *CADE-18*, volume 2392 of *LNAI*, p. 275–279, 2002. Springer-Verlag.

[Weg75] B. Wegbreit. Property Extraction in Well-founded Property Sets. *IEEE Transactions on Software Engineering*, 1(3):270–285, September 1975.