Automatically Generating Loop Invariants Using Quantifier Elimination*† –Preliminary Report–

Deepak Kapur

Abstract. An approach for automatically generating loop invariants using quantifier-elimination is proposed. An invariant of a loop is hypothesized as a *parameterized* formula. Parameters in the invariant are discovered by generating constraints on the parameters by ensuring that the formula is indeed preserved by the execution path corresponding to every basic cycle of the loop. The parameterized formula can be successively refined by considering execution paths one by one; heuristics can be developed for determining the order in which the paths are considered. Initialization of program variables as well as the precondition and postcondition of the loop, if available, can also be used to further refine the hypothesized invariant. Constraints on parameters generated in this way are solved for possible values of parameters. If no solution is possible, this means that an invariant of the hypothesized form does not exist for the loop. Otherwise, if the parametric constraints are solvable, then under certain conditions on methods for generating these constraints, the strongest possible invariant of the hypothesized form can be generated from most general solutions of the parametric constraints. The approach is illustrated using the first-order theory of polynomial equations as well as Presburger arithmetic.

1. Introduction

Since Floyd-Hoare-Dijkstra's inductive assertion method, using pre/postconditions and loop invariants, was proposed in the late sixties for verifying properties of imperative programs, discovering invariants of loops automatically has been viewed as a considerable technical challenge. There have been many attempts to address this problem since the pioneering work of [**8, 10, 18**]; the success however has been limited. As a result, research in program verification has suffered considerably. Recently, there appears to be a revival of research activities relating to mechanically discovering loop invariants, especially using abstract interpretations and the associated widening operators [**4, 5, 20, 21, 24**].

This paper explores the use of quantifier elimination methods for generating loop invariants. A loop invariant is hypothesized to be a parameterized formula in a first-order theory in which certain free variables in a formula are called *parameters*, much in the spirit of the work on deducing subsidiary conditions in the mid 80's on

* This report discusses the method presented in a seminar given by the author on November 20, 2003 in the Computer Science Department Colloquium Series at the University of New Mexico.

† This research was partially supported by an NSF ITR award CCR-0113611.

geometry theorem proving by Wu Wen-Tsun [**30, 31**], Chou [**1**] and the author [**12, 13**].[1] By considering every possible basic cycle arising from the execution of a given loop, the goal is to successively deduce constraints on these parameters such that for values of parameters satisfying these constraints, the instantiated formula is indeed a loop invariant. Once all the basic cycles have been considered, the constraints on parameters generated through this process are solved. For every assignment of parameter values for which the associated constraints hold, the instantiated formula is shown to be a loop invariant.

If a method for generating constraints on parameters (for example, quantifier elimination) is powerful enough that the constraints on parameters produced by the method are logically equivalent to the verification condition obtained by considering all basic cycles of a given loop, then even stronger claims can be made. It can be shown that if there is no solution for parameters from the constraints generated (or there is only a trivial solution), then an invariant of the hypothesized form does not exist for the given loop. Further, if there is a finite description of all solutions of constraints on parameters, then the hypothesized formula can be instantiated using this finite description to get the strongest possible invariant (of the hypothesized form) of the loop. Such a claim cannot be made, however, in case of approximations made either in generating a verification condition (e.g., abstracting boolean expressions and tests appearing in a program in case they cannot be expressed in the language in which invariants are specified) or the algorithm for generating constraints on parameters from the verification conditions does not preserve equivalence. Other kinds of approximations using abstract interpretations may be possible as well.

Constraints on parameters appearing in a parameterized formula hypothesized as an invariant can be successively generated by considering every basic cycle through the program location (typically the beginning of the loop) to which the invariant is associated. If there is more than one basic cycle in a given program, order in which different basic cycles are considered can be arbitrary; heuristics can be developed to consider basic cycles in various orders so as to reduce the work involved in generating constraints for subsequent basic cycles.

We identify conditions on a first-order theory for it to be useful for writing invariants in parameterized form so that the proposed method is applicable. It is shown that Presburger arithmetic, the theory of polynomial equations as well as Tarski's theory of real closed fields satisfy these conditions. The proposed method is illustrated using examples. We have found the method to be simple enough to be taught in undergraduate courses for discovering loop invariants. The examples discussed in this paper as well as many other examples have been done by hand.

In the next subsection, some recent related work on automatically generating loop invariants is discussed. After discussing some preliminaries in Section 2, the proposed approach is illustrated in detail using an example in Section 3; a polynomial equation is hypothesized to be an invariant. Section 4 gives the main steps of

[1] In my opinion, Wu Wen-Tsun proposed a new perspective for theorem proving by proposing the following questions: Given a finite set of hypotheses, does a conclusion follow? And, if it does not follow, under what conditions on certain variables in the hypotheses and conclusions (which he called *parameters*), does the conclusion follow? He formulated this as a *generic* consequence relation using the concept of genericity in elimination theory and algebraic geometry since the objective was to ensure that a given conjecture is true for all values of variables and almost all values of parameters.

the proposed approach. Section 5 discusses three logical theories with the desired properties so that formulas in the associated languages can be used for assertions about programs. Section 5 discusses an example that illustrates how the proposed approach can generate multiple independent invariants of a loop in a program. Section 6 discusses another example illustrating the behavior of the proposed approach if the loop in a program does not have an invariant of the hypothesized form. Section 7 concludes with some remarks for future work. The presentation in the paper is deliberately kept informal with lots of illustration. A more comprehensive technical treatment will be presented in a subsequent paper.

1.1. Related Work. In the seventies, Wegbreit and his colleagues [**27, 28, 10**], Elpas et al [**8**], Manna and Katz [**18**] proposed writing the semantics of assignment statements in the body of a loop as recurrence equations (also called *difference equations*), finding an explicit expression for the value of each program variable as a function of the number s of loop iterations and then eliminating the variable s to obtain invariant predicates. They were however unable to demonstrate the effectiveness of their techniques for nontrivial loop programs.

Karr [**17**] gave an algorithm for finding *linear equalities* as loop invariants. This work was extended by Cousot and Halbwachs [**5**], who applied the concept of abstract interpretation [**4**] to finding *linear inequalities* as invariants. Another extension of Karr's work has been recently proposed by Müller-Olm and Seidl [**21**], who generate using linear algebra techniques, polynomial equations of bounded degree as invariants in programs with affine assignments. In [**3**], Colón et al. have used Farakka's lemma and non-linear constraint solving for finding invariant linear inequalities. This paper is inspired by [**3**].

Recently, Rodriguez-Carbonell and Kapur [**23, 22**] have proposed a general abstract framework for computing loop invariants using fixed point computation on formulas generated from the body of a given loop. They showed how this framework can be instantiated to give a sound and complete algorithm for computing conjunction of polynomial equations as loop invariants. They demonstrated the use of Gröbner basis algorithms for computing such loop invariants on a family of examples. The main advantage of that approach over the proposed method is that it does not assume any a priori bound on the degree of a polynomial as an invariant. Most interestingly, if there is an invariant of the loop that can be expressed as a conjunction of polynomial equations, their approach will find it without having to know their maximum degree; if such an invariant does not exist, their approach can find that as well, whereas the proposed approach can only determine that the invariant of the hypothesized form does not exist.

The proposed approach is, however, simpler to understand and implement. It is somewhat related to an approach the author was made aware of in November/December 2003 as reported in [**25**]; polynomial invariants whose form is a priori determined (called *templates*) are computed using an extended Gröbner basis algorithm over templates. They also suggested comprehensive Gröbner basis algorithms as well as the theory of real closed fields for quantifier-elimination, but opined that such techniques for constraint solving are intractable even for simple programs. They recommended a collection of weaker techniques under the heading "a practical alternative" and heuristics for solving constraints. A more detailed comparison of their heuristics with the proposed approach is needed since we have succeeded in solving many problems by hand using the proposed approach. Also, the proposed

approach gives different results than theirs on many examples, which suggests that our approach is more widely applicable and generates stronger invariants.

2. Preliminaries

We will assume a simple imperative programming language that supports multiple assignment statement, conditional statement, and loop; further, expression evaluation is side-effect free and there are no pointers allowed.

An invariant at any program point is a formula relating program variables such that whenever control passes through the point during the execution of the program, the formula is true. A loop invariant of an imperative program is typically a property of program variables that holds at the beginning of every iteration of a loop in the program. A loop invariant can be shown to follow from an inductive assertion requiring that for every possible execution path corresponding to any basic cycle during the execution of the body of the loop, the assertion is preserved by the path. Loop invariant generation is thus typically inductive assertion generation [**3**]. We will abuse the terminology a bit, and often use loop invariant and inductive assertion interchangeably.

A path in a program is assumed to be a sequence of assignments, possibly interspersed with boolean tests arising due to conditional statements and loops.

For a given loop **L**, let $p_1, \cdots, p_k$ be all possible basic cycles[2], consisting of boolean tests and assignment statements, each of which starts at the beginning of the body of the loop and ends at the same point. For a formula $I(\bar{x})$ to serve as a loop invariant, it must be the case that the Hoare triple

$$\{I(\bar{x})\}\ p_i\ \{I(\bar{x})\},$$

where p_i stands for the code fragment corresponding to the path p_i, and $\bar{x}$ is the tuple of all of the variables in the program.

2.1. Axiomatic Semantics of Assignment. Below we review the axiomatic semantics of an assignment statement. Given a multiple assignment statement and the associated Hoare triple, where P, Q are respectively precondition and postcondition of the statement, we have:

$$\{P\}\ \langle x_1, \ldots, x_l\rangle := \langle e_1, \ldots, e_l\rangle\ \{Q\} \text{ if and only if } (P \Rightarrow Q|^{\langle e_1,\ldots,e_l\rangle}_{\langle x_1,\ldots,x_l\rangle}),$$

where $Q|^{\langle e_1,\ldots,e_l\rangle}_{\langle x_1,\ldots,x_l\rangle}$ stands for the formula obtained from Q by simultaneously replacing all free occurrences of $x_1, \ldots, x_l$ by $e_1, \ldots, e_l$, respectively; x_i's are assumed to be distinct.[3] This is the backward propagation semantics.

An equivalent formulation of the semantics of an assignment statement is based on forward propagation; however new variables must be introduced to distinguish between the values of program variables before and after the assignment. A multiple assignment statement can be viewed as a finite set of rewrite rules of the form

[2]A basic cycle of a program is a path in which the beginning point and the end point are the same and the rest of the path is free of any cycles.

[3]If Q is not quantifier-free, one has to be careful in performing substitutions; the free variables appearing in e_i's should not get captured by the scope of quantifiers in Q; to avoid that, quantified variables may need be renamed. For simplicity, we will assume that Q is quantifier-free so that we do not have to worry about this requirement.

$x'_1 \rightarrow e_1, \ldots, x'_l \rightarrow e_l, \ldots$, where $x'_1, \ldots, x'_l$ are the new variables introduced to stand for the values of $x_1, \ldots, x_l$ after the assignment.

For an instance, consider a path p_i arising from the body of a loop,
$\langle x_1, \ldots, x_l \rangle := \langle e_1, \ldots, e_l \rangle;\ \langle x_1, \ldots, x_l \rangle := \langle f_1, \ldots, f_l \rangle;\ b_1;$
$\langle x_1, \ldots, x_l \rangle := \langle g_1, \ldots, g_l \rangle;\ b_2$
consisting of two assignment statements followed by a test followed by another assignment statement followed by another test. Given a Hoare triple $\{I\}\ p_i\ \{J\}$, there are two ways to generate a verification condition: (i) using backward propagation, we get:

$$I \Rightarrow ((b_1 \Rightarrow (b_2 \Rightarrow J)|^{\langle g_1,\ldots,g_l\rangle}_{\langle x_1,\ldots,x_l\rangle})|^{\langle f_1,\ldots,f_l\rangle}_{\langle x_1,\ldots,x_l\rangle})|^{\langle e_1,\ldots,e_l\rangle}_{\langle x_1,\ldots,x_l\rangle}).$$

The above formula is expressed in terms of $x_1, \ldots, x_l$, standing for the values of the variables at the beginning of the path.

Using new variables introduced to refer to the values of $x_1, \ldots, x_l$ after each assignment statement and by forward propagation, we get:

$$I(x_1 \ldots, x_l) \Rightarrow (b_1(x''_1, \ldots, x''_l) \Rightarrow (b_2(x'''_1 \ldots, x'''_l) \Rightarrow J(x'''_1, \ldots, x'''_l))),$$

where the rewrite rules $x'_1 \rightarrow e_1(x_1, \ldots, x_l), \ldots, x'_l \rightarrow e_l(x_1, \ldots, x_l), x''_1 \rightarrow f_1(x'_1, \ldots, x'_l), \ldots, x''_l \rightarrow f_l(x'_1, \ldots, x'_l), x'''_1 \rightarrow g_1(x''_1, \ldots, x''_l), \ldots, x'''_l \rightarrow g_l(x''_1, \ldots, x''_l)$ relate old variables to new variables after every assignment.

It can be shown that the above two formulations are equivalent.

3. A Detailed Illustration of the Proposed Approach

In this section, we give an informal overview of an automatic method for generating loop invariants; no assumption is made about the availability of pre or post conditions surrounding a loop. If they are available, then they can be made use of; otherwise, just the code is used for generating invariants.

The main idea of the proposed approach is to start with a quantifier-free formula $I(\bar{x}, \bar{u})$[4] where $\bar{x}$ are the variables standing for program variables which are possibly changing in a program, and $\bar{u}$ are called *parameters* whose values need to be determined for I to serve as an invariant. Constraints on parameters $\bar{u}$ are generated so that for all possible values of $\bar{x}$, I is preserved by every execution path of a given loop; this is done by considering all basic cycles of the loop.

Using Floyd-Hoare's inductive assertion method, a verification condition Φ is generated for all values of program variables $\bar{x}$ with the parameters $\bar{u}$ as its only free variables. If $\exists \bar{u} \forall \bar{x}\ \Phi$ is valid, it can be shown that there indeed exists an invariant of the hypothesized form $I(\bar{x}, \bar{u})$ with $\bar{u}$ taking specific values; otherwise, such an invariant does not exist. Further, if an equivalent quantifier-free formula, say P, on parameters $\bar{u}$ can be computed from $\forall \bar{x}\ \Phi$, then every assignment of parameter values that makes P true gives an invariant I' obtained from I after instantiating the parameters in it. Further, if a finite description of all assigments making P true can be computed, then the strongest invariant (under certain assumptions) can be generated. This is illustrated below using an example.

[4]Throughout the paper, we will consider quantifier-free formulas only as candidates for loop invariants in order to keep the presentation simple. We believe that the proposed method can be used to generate arbitrary formulas (with quantifiers) as invariants as well.

3.1. Example. Consider the following simple program to compute the product of two inputs.

function product (X, Y: integer) **returns** z: integer
var x, y: integer **end var**
$\langle x,\ y,\ z\rangle$:=$\langle X,\ Y,\ 0\rangle$;
while $y \neq 0$ **do**
if y `mod` $2 = 1$ **then** $\langle x, y, z\rangle$:=$\langle 2x, (y-1)$ `div` $2, x+z\rangle$;
[] y `mod` $2 = 0$ **then** $\langle x, y, z\rangle$:=$\langle 2x, y$ `div` $2, z\rangle$;
end if
end while

3.1.1. *Listing All Basic Cycles of a Loop and Generating the associated Hoare Triples:* There are two basic cycles for the loop:

(1) p_1 consisting of two tests $y \neq 0$; $y\ mod\ 2 = 1$; followed by the multiple assignment statement $\langle x, y, z\rangle$:=$\langle 2x, (y-1)$ `div` $2, x+z\rangle$;
(2) p_2 consisting of two tests $y \neq 0$; $y\ mod\ 2 = 0$; followed by the multiple assignment statement $\langle x, y, z\rangle$:=$\langle 2x, y$ `div` $2, z\rangle$.

For a formula $I(x, y, z)$ to be an invariant, it must be the case that
$\{I(x,y,z) \wedge y \neq 0 \wedge y\ mod\ 2 = 1\}\ \langle x, y, z\rangle$:=$\langle 2x, (y-1)$ `div` $2, x+z\rangle\ \{I(x,y,z)\}$
and
$\{I(x,y,z) \wedge y \neq 0 \wedge y\ mod\ 2 = 0\}\ \langle x, y, z\rangle$:=$\langle 2x, y$ `div` $2, z\rangle\ \{I(x,y,z)\}$.

Consider the second Hoare triple since its manipulation is easier. Using the backward propagation semantics of the multiple assignment statement, we get:

$$(I(x,y,z) \wedge y \neq 0 \wedge y\ mod\ 2 = 0) \Rightarrow I(2x, y \texttt{ div } 2, z),$$

where $I(2x, y$ `div` $2, z)$ stands for the formula obtained by replacing x, y, z by $2x, y$ `div` $2, z$, respectively in I (this is yet another notation for $I|^{\langle 2x, y \texttt{ div } 2, z\rangle}_{\langle x,y,z\rangle}$).[5]

3.1.2. *Hypothesizing I:* Suppose that $I(x, y, z)$ is a polynomial equation in x, y, z in which the degree of every variable is at most 1. In other words,
$I(x,y,z) = (A\ xyz + B\ xy + C\ xz + D\ yz + E\ x + F\ y + G\ z + H = 0)$,
where A, B, C, D, E, F, G, H are parameters whose values must be determined.

3.1.3. *Generating Constraints from Hoare Triples one by one: Path 2:* Substituting for I in the above formula obtained from the second Hoare triple, gives:
$((A\ xyz + B\ xy + C\ xz + D\ yz + E\ x + F\ y + G\ z + H = 0) \wedge y \neq 0 \wedge y\ mod\ 2 = 0)$
$\Rightarrow (A\ 2x(y$ `div`$2)\ z + B\ 2x(y$ `div` $2) + C\ 2xz + D\ (y$ `div` $2)z$
$+E\ 2x + F\ (y$ `div` $2) + G\ z + H = 0)$.
Since $y\ mod\ 2 = 0$, we can replace y by $2u$:
$((A\ 2xuz + B\ 2xu + C\ xz + D\ 2uz + E\ x + F\ 2u + G\ z + H = 0) \wedge u \neq 0 \Rightarrow$
$(A\ 2xuz + B\ 2xu + C\ 2xz + D\ uz + E\ 2x + F\ u + G\ z + H = 0)$.

The goal is to find values of parameters A, B, C, D, E, F, G, H such that the above formula is valid for all values of x, y, z. In general, if a theory admits quantifier-elimination, then we need to find a quantifier-free formula equivalent to the above formula universally quantified over x, u, z; this formula is only in A, B, C, D, E, F, G, H. In case, the hypothesis is a conjunction of polynomial equations and the conclusion is a polynomial equation, then quantifier-elimination can

[5] An interested reader can easily verify that the forward semantics of the assignment statement will give an equivalent result.

be done using *parametric* Gröbner basis construction [**14**] (also called comprehensive Gröbner basis in [**29**]). Crudely speaking, the simplification is done by case analysis on the coefficients of the terms appearing in a formula, e.g, whether $A = 0$ or not, $B = 0$ or not, etc; more details can be found in [**14, 29**].

The above formula can easily be simplified to give:

$((A\ 2xuz + B\ 2xu + C\ xz + D\ 2uz + E\ x + F\ 2u + G\ z + H = 0) \wedge u \neq 0 \Rightarrow$
$(C\ xz - D\ uz + E\ x - F\ u = 0)$.

The conclusion cannot be simplified any further. One possibility for the above formula to be valid for all values of x, u, z is that the polynomial $(C\ xz - D\ uz + E\ x - F\ u)$ is identically equal to 0; this implies that the coefficient of every term in the polynomial is 0: $C = D = E = F = 0$. There are other possibilities as well for the above formula to be valid but we will not discuss them here.

Path 1: To consider the first basic cycle, it is not necessary to start with the original hypothesized formula. Instead, we can use I to be:$(A\ xyz+B\ xy+G\ z+H = 0)$ after noting that $C = D = E = F = 0$ from the constraints generated while considering the second path. Using the backward semantics of the assignment statement, we get

$((A\ xyz + B\ xy + G\ z + H = 0) \wedge y \neq 0 \wedge y\ mod\ 2 = 1) \Rightarrow$
$(A\ 2x((y-1)\texttt{div2})(x+z) + B\ 2x((y-1)\texttt{div2}) + G\ (x+z) + H = 0)$.

Since $y\ mod\ 2 = 1$, replacing y by $2u + 1$ gives:

$(A\ x(2u+1)z + B\ x(2u+1) + G\ z + H = 0) \Rightarrow$
$(A\ 2xu(x+z) + B\ 2xu + G\ (x+z) + H = 0)$.

This formula can be simplified to give:

$(A\ x(2u+1)z+B\ x(2u+1)+G\ z+H = 0) \Rightarrow (2A\ x^2u-A\ xz+(G-B)\ x = 0)$.

One possibility for the above formula to be valid for all values x, y, z is that if the coefficient of every term in in the polynomial $(2A\ x^2u - A\ xz + (G - B)\ x = 0)$ is 0, implying that $A = 0$ and $G - B = 0$.

3.1.4. *Solving Constraints from All Paths and Initial Values to Generate Invariant:* After considering both the paths of the loop in the above program, we have:

$$I(x, y, z) = (B\ xy + B\ z + H = 0).$$

This formula can be verified to be an invariant of the above loop.

Using the initial values of the variables when the loop is entered, we get an additional constraint as follows: $I(X, Y, 0) = (B\ XY + H = 0)$, which gives $H = -B\ XY$. Substituting for H in I above:

$$I(x, y, z) = (B\ xy + B\ z - B\ XY = 0).$$

B can be factored out, thus giving

$$I(x, y, z) = (xy + z - XY = 0)$$

as an invariant of the above loop.

The reader can verify that $I(x, y, z) = (xy+z-XY = 0)$ is indeed an invariant of the loop. Furthermore, it can also be shown that this is the strongest possible invariant expressed as a polynomial equation in which the degree of every variable is at most 1.

If we had hypothesized that $I(x, y, z)$ to be a polynomial in x, y, z in which the degree of every variable was at most 3, even then the above method would have resulted in a polynomial equation as an invariant in which $xy + z - XY$ is a factor.

This behavior of the proposed approach is illustrated using another example in Section 6.

4. An Automatic Method for Generating Invariants

In this section, we present the method illustrated above for automatically generating invariants (inductive assertions) for imperative programs. The first step is to fix a logical language to be used for specifying invariants. The language chosen for the above example is the conjunction of polynomial equations. Later, we discuss other languages and associated theories. Parametric forms of formulas serving as invariants must then be identified. The theory associated with these parametric forms should have the property that the verification condition generated from imperative programs involving these parametric invariants can be manipulated to get constraints on parameters. To achieve that, some assumptions about the nature of assignment statements and tests allowed will have to be made or alternatively, approximations/abstractions will have to be used.

Given a program consisting of many loops (including nested loops and function/procedure calls), an assertion can be attached to the entry of each loop as well as the entry and exit of every function definition (alternatively, every control point in a program can be attached an assertion as in [**3**], where this labeling is called an *assertion map*). There should be sufficiently many assertions to cover all execution paths of a given program. For programs with simple loops (no nested loops or function calls), it suffices to consider basic cycles. Here are the steps of the method for mechanically generating inductive assertions.

(1) Generate all possible paths from one assertion to another assertion (including itself); a path is either free of cycles or is a basic cycle. For each such path, hypothesize with the end points, parameterized assertions, which are quantifier-free formulas involving program variables and parameters. Often, the same parameterized assertion can be used for the end points and multiple paths, as was the case in the above example.
(2) For each path,
    (a) formulate its Hoare triple. Get a formula from the Hoare triple (that does not involve any code). It may not be possible to generate a formula in the chosen language because boolean conditions serving as tests in conditional statements and loops as well as assignments may not be expressible in the language. In that case, a formula equivalent to the Hoare triple cannot be genearted; instead approximate and generate a formula weaker than the Hoare triple.
    (b) From the formula, which is of the form $\forall \bar{x}\ \phi(\bar{x}, \bar{u})$, generate a constraint $P(\bar{u})$ on the parameters $\bar{u}$, which is a quantifier-free formula over $\bar{u}$. Depending upon whether the logical language admits quantifier-elimination as well as assumptions made in the procedure for generating constraints on parameters, $P(\bar{u})$ is equivalent to or implies $\forall \bar{x}\ \phi(\bar{x}, \bar{u})$.
(3) Do a conjunction of all constraints on parameters generated for each path. If the conjunction is not satisfiable, implying there is no assignment of parameters which satisfies all the constraints, then inductive assertions of the hypothesized parameterized form can be proved not to exist for the program (assuming no approximations were made in generating formulas

equivalent to Hoare triples for the program as well as the constraint $P(\bar{u})$ on parameters is equivalent to $\forall \bar{x}\ \phi(\bar{x}, \bar{u})$). If the conjunction is satisfiable, then solve for the parameters, i.e., generate an assignment, say α of parameters $\bar{u}$, satisfying each constraint in the conjunction. Instantiate the parameterized assertions by assigning parameters in them by α; the resulting assertions are then invariants for the program. Every such assignment of parameters leads to inductive assertions for the program.

In case there are multiple solutions of the constraints on parameters, find a finite description of all the solutions, if possible. Use this finite description to instantiate parameters in the hypothesized inductive assertions. An example below illustrates this.

As the reader would notice, for each path, the constraint on parameters can be generated in parallel, independent of generation of other constraints. However, if done sequentially for paths, it is often helpful to incrementally simplify parameterized assertions and use these simplified forms for generating constraints on parameters from other paths, as illustrated in the above example. Particularly if some parameters can be instantiated, do so to get simpler assertions possibly with fewer parameters.

The following properties serve as a basis of correctness of the above method.

LEMMA 1. *Given a parameterized formula $I(\bar{x}, \bar{u})$ such that $\forall \bar{x}\ (I \Rightarrow I^{(e_1,\ldots,e_l)}_{(x_1,\ldots,x_l)})$ is equivalent to a formula $P(\bar{u})$ in parameters only, then for an assignment α of parameter values, $\alpha(P(\bar{u}))$ is true if and only if the instance $J(\bar{x}) = \alpha(I(\bar{x}, \bar{u}))$ has the property that $\{J(\bar{x})\}\ (x_1, \ldots, x_l) := (e_1, \ldots, e_l)\{J(\bar{x})\}$.*

In case, $\forall \bar{x}\ (I \Rightarrow I^{(e_1,\ldots,e_l)}_{(x_1,\ldots,x_l)})$ is not equivalent to $P(\bar{u})$ but is implied by $P(\bar{u})$, then for parameters values β that make $P(\bar{u})$ to be false, the instance $J'(\bar{x}) = \beta(I(\bar{x}, \bar{u}))$ may or may not satisfy $\{J'(\bar{x})\}\ (x_1, \ldots, x_l) := (e_1, \ldots, e_l)\ \{J'(\bar{x})\}$.

COROLLARY 1. *Given a constraint P on parameters $\bar{u}$ as defined above in Lemma 1 and two parameters assignments α_1 and α_2 making P true, $J(\bar{x}) = \alpha_1(I(\bar{x}, \bar{u})) \wedge \alpha_2(I(\bar{x}, \bar{u}))$ also satisfies $\{J(\bar{x})\}\ (x_1, \ldots, x_l) := (e_1, \ldots, e_l)\ \{J(\bar{x})\}$.*

Under the assumption that boolean tests and assignments in a program are not approximated so that the formula $\forall \bar{x}\ (I \Rightarrow I^{(e_1,\ldots,e_l)}_{(x_1,\ldots,x_l)})$ equivalent to each Hoare triple can be computed, and further, the constraint P on parameters is equivalent to $\forall \bar{x}\ (I \Rightarrow I^{(e_1,\ldots,e_l)}_{(x_1,\ldots,x_l)})$, it can be proved that $J(\bar{x}) = \alpha_1(I(\bar{x}, \bar{u})) \wedge \ldots \wedge \alpha_m(I(\bar{x}, \bar{u}))$ is the strongest invariant. where $\alpha_1 \ldots \alpha_m$ characterizes all assignments representing solutions to the conjunction of constraints P.

5. Admissible Theories

The proposed method requires that a theory used to express an inductive assertion as a parameterized formula should have the following property: Given a formula $\forall \bar{x}\ \phi(\bar{x}, \bar{u})$, generate a formula $P(\bar{u})$ such that $P(\bar{u}) \Rightarrow \forall \bar{x}\ \phi(\bar{x}, \bar{u})$. Even better, if the theory has a quantifier elimination algorithm for such formulas, i.e., there is an algorithm to generate $\forall \bar{x}\ \phi(\bar{x}, \bar{u}) \iff P(\bar{u})$, then stronger claims can be made about the results produced by the proposed method. That is, it can be shown that under certain restrictions on assignment expressions and boolean tests,

if there are inductive assertions of the hypothesized form possible for a given program, the proposed approach will find them; furthermore, such assertions will be the strongest possible ones.

Below, we discuss three theories over numbers: (i) Presburger arithmetic in which only linear polynomials related using $=, \leq$ relations (on reals, rationals or integers) can be expressed; (ii) polynomial equations with solutions over an algebraic closed field; and (iii) theory of real closed field, which generalizes Presburger arithmetic and the theory of polynomial equations to include $=, \leq$ relations on polynomials over the field of reals. Each of these theories admits quantifier-elimination. Furthermore, many heuristics can be employed to get constraints on parameters from verification conditions.

5.1. Presburger Arithmetic. Consider the theory of Prebsurger arithmetic over the integers with $\leq, =, 0, s, +$.[6] It is well known that this theory admits quantifier-elimination; Enderton's book [**9**] gives one such method. Fourier-Motzkin's algorithm for projection (elimination of vaiables) as implemented in our theorem prover *Rewrite Rule Laboratory* (*RRL*) [**16, 15**] can also be used; this is illustrated below.

Without any loss of generality, it can be assumed that the formula from which a variable x has to be eliminated is a conjunction of inequalities of the form $a_i x \leq p_i$ and $q_j \leq b_j x$, $1 \leq i \leq k_1, 1 \leq j \leq k_2$ (equalities can be replaced by inequalities as well).[7] For every pair of inequalities in which x appears with opposite signs, i.e., $ax \leq p$ and $bx \leq q$, where p, q do not have any occurrences of x, a is *positive* and b is negative, the inequality $0 \leq k_1 p + k_2 q$ is generated, where $lcm(a, b) = k_1 a = k_2 b$. After all such pairs of inequalities have been considered, the resulting inequalities do not have any occurrence of x. In the process, if any inequality of the form $c \leq 0$, where c is positive, is generated, then the formula is unsatisfiable.[8] Otherwise, if the unsatisfiability of a formula is not detected, the resulting formula after elimination of variables may or may not be equivalent to the original formula depending upon whether the eliminated variables are ranging over integers, rationals or reals.

Using the example below, it is shown how Fourier-Motzkin's algorithm can be generalized to do quantifier elimination from parameterized inequalities expreseed using $0, 1, +, \leq$. The reader should have noticed that a parameterized inequality is not expressible in Presburger arithmetic.

Consider the following example taken from [**4, 5**] and also discussed in [**3**].

var i, j: integer **end var**
$\langle i, j \rangle := \langle 2, 0 \rangle$;

[6]The proposed approach works for Presburger arithmetic over the naturals, rationals as well as the reals.

[7]In the case of rationals or reals, however, strict inequality of the form $a_i x < p_i$ and inequality $a_i x \leq p_i$ have to be distinguished as one cannot be transformed into the other, unlike in the case of naturals and integers. In the case of rationals and reals, the coefficient of x, the variable being eliminated, can be assumed to be 1. Further, if there is an equality relating a variable to other variables, that can be used to eliminate one of the variables. Over the rationals as well as reals, Fourier-Motzkin's algorithm is complete in the sense that linear inequalities are satisfiable if and if no inequality of the form $c \leq 0$, where c is a positive number is generated. In the case of integers, additional tests have to be performed.

[8]It is well-known that Fourier-Motzkin's algorithm can be bad in the worst case. Other algorithms including integer programming or linear programming in case of rationals can be used as well; see, for example, [**19**].

while *true* **do**
  if *true* → $\langle i, j\rangle := \langle i+4, j\rangle$;
  [] *true* → $\langle i, j\rangle := \langle i+2, j+1\rangle$;
  end if
end while

If the invariant for the above loop is hypothesized to be a polynomial equation, it can be shown that there does not exist such an invariant (e.g., if $I(i,j) = (c_1 i + c_2 j + d = 0)$, then after considering both the paths, I simplifies to $d = 0$; to make I satisfiable, d must be made 0, implying that I is `true`).

Suppose the invariant $I(i,j)$ is an inequality of the form $c_1 i + c_2 j + d \leq 0$, where c_1, c_2, d are unknown. This leads to two verification conditions:
$(c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 4c_1 \leq 0)$ and
$(c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 2c_1 + c_2 \leq 0)$.
And, from the initial values, the condition is $2c_1 + d \leq 0$.

For this invariant of the above form to exist, $\Phi = \exists c_1, c_2, d[(2c_1 + d \leq 0) \wedge (\forall i, j, (c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 4c_1 \leq 0))$
$\wedge(\forall i, j, (c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 2c_1 + c_2 \leq 0))]$
is valid over the integers, which is indeed the case. Had Φ not been valid, the invariant of the form $c_1 i + c_2 j + d \leq 0$ does not exist.

Let $\Phi'(c_1, c_2, d)$ be such that $\Phi = \exists c_1, c_2, d\ \Phi'(c_1, c_2, d)$. To generate an invariant, values of c_1, c_2, d that make $\Phi'(c_1, c_2, d)$ valid need to be computed. To get the strongest possible invariant of this form, we are interested in finding all possible values of c_1, c_2, d. The reader should note that the above formula falls outside the language of Presburger arithmetic because of subformulas of the form $c_1 i + c_2 j + d \leq 0$, where c_1, c_2 are not numbers.

Below, we give a generalization of Fourier-Motzkin's algorithm to get a formula equivalent to Φ'. Consider first the subformula
$\phi(c_1, c_2, d) = \forall i, j, (c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 4c_1 \leq 0)$.
We will negate ϕ and find a quantifier-free formula equivalent to $\neg\phi$, and then negate the result back to get the quantifier-free formula equivalent to ϕ.

Negating ϕ gives the conjunction of the two literals[9]

$$\neg\phi = (c_1 i + c_2 j + d \leq 0 \wedge -c_1 i - c_2 j - d - 4c_1 + 1 \leq 0).$$

The above pair of inequalities has the coefficient of i to be both c_1 and $-c_1$. Thus, for eliminating i, two cases must be considered:

(1) $c_1 = 0$: now, the inequalities do not have i any more since the above literals become $c_2 j + d \leq 0$ and $-c_2 j - d + 1 \leq 0$. To eliminate j from it, there are additional two subcases:
  (a) $c_2 = 0$: $d \leq 0 \wedge -d + 1 \leq 0$ which is unsatisfiable.
  (b) $c_2 \neq 0$, in which case j can be eliminated by adding the two inequalities giving $1 \leq 0$ implying this case is not possible either.
(2) $c_1 \neq 0$: i can be eliminated by adding the two inequalities giving $-4c_1 + 1 \leq 0$.

[9]We are abusing the notation by using i, j to stand for Skolem constants in $\neg\ \phi$.

This gives the formula $[\neg(c_1 = 0 \wedge c_2 = 0) \wedge \neg(c_1 = 0 \wedge c_2 \neq 0) \wedge (c_1 \neq 0 \Rightarrow -4c_1 + 1 \leq 0)]$, equivalent to $\neg\phi$, which can be simplified. After negating it, we get $\phi = (c_1 = 0 \vee c_1 < 0)$.
Similarly, for $\forall i, j, ((c_1 i + c_2 j + d \leq 0) \Rightarrow ((c_1 i + c_2 j + d) + 2c_1 + c_2 \leq 0))$, an equivalent formula obtained using the above steps is:

$$(c_1 = 0 \wedge c_2 \leq 0) \vee (c_1 \neq 0 \wedge (2c_1 + 2c_2) \leq 0).$$

Combining all these subformulas together, the equivalent formula for Φ above is:

$$[(2c_1 + d \leq 0) \wedge (c_1 = 0 \vee c_1 < 0) \wedge ((c_1 = 0 \wedge c_2 \leq 0) \vee (c_1 \neq 0 \wedge (2c_1 + 2c_2) \leq 0))],$$

which simplifies to:

$$P = [(d \leq 0 \wedge c_1 = 0 \wedge c_2 \leq 0) \vee (2c_1 + d \leq 0 \wedge c_1 < 0 \wedge 2c_1 + c_2 \leq 0)].$$

For any values c_1, c_2, d that satisfy the above formula, $c_1 i + c_2 j + d \leq 0$ is an invariant.

All solutions of P can be written in terms of the generator set consisting of $\langle c_1 = 0, c_2 = -1, d = 0\rangle$, $\langle c_1 = -1, c_2 = 2, d = 2\rangle$ [**26**]. Corresponding to each generator is an invariant. The conjunction of the invariants corresponding to these generators is $(-j \leq 0 \wedge -i + 2j + 2 \leq 0)$, which can be shown to be the strongest invariant expressed as a conjunction of linear inequalities for the above loop.

As the reader would have noticed, one easy way to generalize Fourier-Motzkin algorithm for quantifier-elimination of parameterized inequalities is to perform case analysis. For a variable x to be eliminated, if the coefficient of x in an inequality is a paremetric expression, three cases are considered making the coefficient negative, zero or positive. These constraints can be used to simplify other inequalities in further processing. For an efficient use of constraints and other sophisticated heuristics, see [**19**].

5.2. Quantifier-free Theory of Conjunctively Closed Polynomial Equations. If assignments in a program are polynomial expressions, its invariants expressed as polynomial equations can be generated using Gröbner basis construction as was illustrated above in the first example in Section 3. A typical verification condition in that case is $(p_1 = 0 \wedge \ldots \wedge p_k = 0) \Rightarrow q = 0$, where $p_1, \cdots, p_k$ as well as q are polynomials in variables $\bar{x}$ and parameters $\bar{u}$.

In [**14**], a method for computing parametric Gröbner basis from a finite set of parametric polynomials is given; comprehensive Gröbner bases introduced by Weispfenning can also be used [**29**]. These methods can be adapted to perform quantifier-elimination from parametric formulas of the form $(p_1 = 0 \wedge \ldots \wedge p_k = 0) \Rightarrow q = 0$ to get an equivalent formula over the parameters. Parameters $\bar{u}$ are made lower than variables $\bar{x}$ to be eliminated in term orderings used for computation; a block ordering in which the parameters $\bar{u}$ as a block are lower than the variables $\bar{x}$ as a block often works well. For a conclusion polynomial equation $q = 0$ to follow from the conjunction of polynomial equations, q must be in the radical ideal of the hypothesis polynomials $\{p_1, \ldots, p_k\}$. It often suffices to compute the Gröbner basis of the hypothesis polynomials, and use it to check whether the conclusion polynomial reduces to zero. In that case, the above formula is valid for all values of parameters; otherwise, if the normal form of the conclusion polynomial q is not 0, then a good heuristic is to require that in the normal form, the coefficient of every term built using $\bar{x}$, the variables to be eliminated, must be identically equal to 0, which gives constraints on parameters. For completeness, it may be necessary

to compute the Gröbner basis of the radical ideal of the hypothesis polynomials and use it to reduce the conclusion polynomial. (For a related approach based on refutational completeness for proving Euclidean geometry theorems as well as for deducing subsidiary conditions as well as missing hypotheses of geometry problem formulations, see [**13**].) Parametric constraints thus generated can themselves be solved using linear algebra, if they are linear, or using Gröbner basis construction if they are nonlinear.

The construction of a parametric Gröbner basis can be expensive due to branching based on the parametric coefficient of the leading term of a polynomial in a basis. The following heuristic often works very well: (i) assume that the leading coefficient of each polynomial p_i is nonzero while computing their Gröbner basis (insofar as this can be done consistently); (ii) once their Gröbner basis is computed, normalize the conclusion polynomial q again assuming the leading coefficient of every polynomial in the Gröbner basis is nonzero; (iii) if the normal form of q is 0, then under the assumptions, the polynomial follows from the hypotheses; otherwise, assert the coefficient of each term in the normal form of q to be 0. Collect all the constraints and solve for them to generate an invariant. This was the approach used in the above example as well as in the example in Section 6. While incomplete, the heuristic has been found to work well on many examples; it is quite efficient in contrast to having to compute a parametric (or comprehensive) Gröbner basis.

5.3. Theory of Real Closed Fields. The theory of real closed fields, whose decidability was shown by Tarski in 1930's, is the most expressive theory for specifying polynomial constraints as invariants. In this theory, both $\leq$ as well as $=$ can be used as relations over polynomial expressions. Further, solutions are sought over the field of real numbers, unlike the treatment in the previous subsection where the solutions are over an algebraically closed field (e.g., the field of complex numbers).

The theory of real closed field admits quantifier-elimination. Decision procedures for the theory has been extensively studied and implemented over the last 40 years; see [**2**]. Specialized implementation of decision procedures along with the ones for low degree polynomial constraints are available–software packages REDLOG [**6**] and QEPCAD [**11**] implemented on top of computer algebra systems REDUCE and Maple, respectively, are particularly promising. Unlike in the cases of the theory of polynomial equations and Presburger arithmetic, nothing special has to be done to consider parametric formulas since even parameterized formulas are in the theory; it suffices to make a distinction between variables and parameters. Finding constraints on parameters thus amounts to eliminating universally quantified variables (standing for program variables) from the verification conditions generated from considering all execution paths of a given loop. Once constraints on parameters are generated, their general solution can be obtained using also the decision procedure for the theory.

The main drawback of the decision procedures for the theory of real closed field is that even though their worst case complexity is doubly exponential in the number of variables, essentially the same as that of decision procedures for Presburger arithmetic as well as for polynomial equations over an algebraically closed field, these decision procedures in practice work only on toy small problems, unlike other methods which work reasonably well on many interesting problems; an interested reader may consult [**7**] for more details. We have done preliminary investigations and the results are not encouraging, primarily because of the number of variables

to be eliminated from verification conditions. Special heuristics need to be investigated that can exploit the special structure of the formulas serving as verification conditions. We omit the details here because of lack of space. In a forthcoming paper, we will show how an existing implementation of a decision procedure for this theory can be used to generate invariants of loop programs.

6. Conjunction of Formulas as an Invariant: Multiple Solutions for Constraints on Parameters

Constraints on parameters generated from different execution paths in a loop may not, in general, lead to the parameters getting totally determined. Instead, the parameter constraints can have multiple solutions. The proposed method in such a case produces a conjunction of formulas as an invariant, in which each formula corresponds to an independent solution of parameter constraints. Under certain conditions, such a conjunction of formulas can be shown to be the strongest possible invariant of the loop of the hypothesized form. The example below illustrates this as well as what happens if different parametric forms are hypothesized as assertions of the same program.

Consider the following simple loop for computing the floor of the square root of a natural number.

$\langle a,\ s,\ t\rangle$:=$\langle 0,\ 1,\ 1\rangle$;
while $s \leq N$ **do**
    $\langle a,\ s,\ t\rangle$:=$\langle a+1,\ s+t+2,\ t+2\rangle$;
end while

There is only one path in this loop consisting of the loop test and the assignment statement: $s \leq N$; $\langle a,\ s,\ t\rangle$:=$\langle a+1,\ s+t+2,\ t+2\rangle$.

6.1. Linear Equation as an Invariant. Let us hypothesize the invariant $I(a, s, t)$ to be a linear polynomial equation of the form: $A\ a + B\ s + C\ t + D = 0$. The verification condition generated is:
$(A\ a + B\ s + C\ t + D = 0 \wedge s \leq N) \Rightarrow A\ (a+1) + B\ (s+t+2) + C\ (t+2) + D = 0$.
Simplifying it leads to: $(A\ a + B\ s + C\ t + D = 0) \wedge s \leq N \Rightarrow (A + B\ (t+2) + 2C = B\ t + (A + 2B + 2C) = 0)$. This formula is valid if $B\ t + (A + 2B + 2C) = 0$ for any t, implying that $B = 0 \wedge A + 2B + 2C = 0$, which simplifies to $A = -2C, B = 0$. So the invariant becomes $-2C\ a + C\ t + D = 0$. Using the initial values, we get: $C + D = 0$ implying that $D = -C$. The invariant gets further refined to: $-2C\ a + C\ t - C = 0$, which after removing C as a factor gives:

$$-2a + t - 1 = 0$$

as the invariant. It can be proved to be the strongest linear polynomial invariant of the above loop.

6.2. Nonlinear Polynomial Equation as Invariant. Suppose the invariant is hypothesized to be a polynomial equation of degree 2. That is,
$I(a, s, t) = u_1\ a^2 + u_2\ s^2 + u_3\ t^2 + u_4\ as + u_5\ at + u_6\ st + u_7 a + u_8 s + u_9 t + u_{10} = 0$,
where $u_1, \cdots, u_{10}$ are parameters. After simplifying the verification condition, the following constraints on parameters are generated:

1. $u_1 = -u_5$, 2. $u_7 = -2u_3 - u_5 + 2u_{10}$, 3. $u_8 = -4u_3 - u_5$, 4. $u_9 = 3u_3 + u_5 - u_{10}$.

The above set of constraints has infinitely many solutions. Each solution can be obtained from an independent set of 3 solutions obtained by making exactly one of the independent parameters, u_3, u_5 and u_{10}, to be 1 and assigning each of u_2, u_4, u_6 to be 0. Values of other parameters are then determined.

(1) Assigning $u_{10} = 1$, gives $u_7 = 2, u_9 = -1$; all other parameters are 0; the polynomial $2a - t + 1 = 0$ is an invariant; recall that this is also the invariant generated when I is hypothesized earlier to be a linear polynomial equation.
(2) Assigning $u_5 = 1$ gives $u_1 = u_7 = u_8 = -1, u_9 = 1$; all other parameters are 0; this gives $-a^2 + at - a - s + t = 0$ as another invariant.
(3) Assigning $u_3 = 1$ gives $u_7 = -2, u_8 = -4, u_9 = 3$; all other parameters are 0; one gets the polynomial $t^2 - 2a - 4s + 3t = 0$ as the third invariant.

Consider the conjunction of the invariants corresponding to all independent solutions of the above constraint set. This formula can be shown to be the strongest invariant of the above loop, expressed in the polynomial equation form in which the degree of every variable is at most 2. Every other such invariant can be obtained from these three invariants; this is so because every solution of the above constraint set can be expressed in terms of the above independent solutions.

7. Hypothesizing an Invariant of Unsuitable Form

We show how the proposed method works in case a formula that cannot serve as an invariant is hypothesized to be an invariant since this is not known a priori.

Consider again the example introduced in Section 3. In contrast to a polynomial equation in which every variable has at most degree 1, suppose $I(x, y, z)$ is hypothesized as a linear equation in x, y, z. As we shall see below, the proposed method would discover that no linear equation can serve as an invariant;.

Let $I(x, y, z) = (A\ x + B\ y + C\ Z + D = 0)$.
Considering the path p_2, the verification condition generated is:

$$(A\ x + B\ y + C\ z + D = 0 \wedge y \neq 0 \wedge y\ mod\ 2 = 0) \Rightarrow$$
$$(A\ 2x + B\ (y\ \texttt{div}\ 2) + C\ z + D = 0).$$

After substituting $2u$ for y, the conclusion above becomes: $2A\ x + B\ u + C\ z + D = 0$. The formula is not valid if both $A \neq 0$ and $B \neq 0$; the formula equivalent to it, in terms of A, B, C, D, is $(A = 0 \wedge B = 0) \vee (A \neq 0 \wedge B = 0 \wedge C = 0 \wedge D = 0) \vee (A = 0 \wedge B \neq 0 \wedge C = 0 \wedge D = 0)$. The reader can easily verify that for the values of parameters satisfying these constraints, the above formula is indeed preserved by path p_2.

Considering the path p_1 using $I(x, y, z)$ with the above constraint, we get: $(A = 0 \wedge B = 0 \wedge C = 0) \vee (A \neq 0 \wedge B = 0 \wedge C = 0 \wedge D = 0)$. First makes $D = 0$, giving the trivial invariant *true*. From the second condition, the invariant is $Ax = 0$. From the initial values, we get $A = 0$, again giving the trivial invariant. This analysis implies that the loop in the program does not have a linear equation as an invariant as hypothesizing an invariant of this form trivializes to *true*.

As this example illustrates, the proposed approach will not generate an incorrect invariant.

8. Concluding Remarks

We have presented an approach for automatically generating invariants of loop programs using quantifier-elimination techniques. The method uses parameterized formulas for specifying invariants in which parameters are determined based on the constraints generated from requiring these formulas to serve as inductive assertions of the program. We have illustrated the approach using programs for which assertions can be expressed in three related theories–(i) linear inequalities, (ii) polynomial equalities and finally, (iii) polynomial inequalities and equalities which subsume (i) and (ii); each of these theories admits quantifier elimination. Under certain conditions on programs and based on the power of quantifier-elimination techniques, the approach can generate the strongest possible invariants of pre-determined forms; it can also deduce if invariants of such forms do not exist for a given program. If assertions are hypothesized to be parameterized polynomial equations, a heuristic for generating constraints on parameters from verification conditions is given using Gröbner basis techniques . The main advantage of the approach is that many example programs can be handled by hand; thus, it can be used in undergraduate courses on programming and algorithms to teach loop invariants.

The approach is currently being implemented and its effectiveness needs to be further investigated. It will be worth exploring how quantifier-elimination techniques can exploit the special structure of the verification conditions generated from programs. Most importantly, there is a need to investigate richer logical languages in which properties of complex data structures, including arrays, records, etc. can be expressed, to which the proposes approach can be extended.

Acknowledegments. The author would like to thank Stan Lee for posing a question in my seminar course on program verification and theorem proving offered at the UNM Department of Computer Science in Spring/Fall 1999, which led me to think about invariant generation as a quantifier elimination problem. Thanks also to Enric Rodriguez-Carbonell for collaboration on the topic of automatic generation of invariants and many discussions which resulted in this paper.

References

[1] S.-C. Chou. *Mechanical Geometry Theorem Proving*. D. Reidel Publishing Company, Dordrecht, Netherlands., 1988.

[2] G. Collins and H. Hong. Partial Cylindrical Algebraic Decomposition for Quantifier Elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991.

[3] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *Computer-Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 2003.

[4] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[5] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[6] A. Doltzmann and T. Sturm. REDLOG: Computer Algebra meets Computer Logic. *Technical Report, University of Passau, MIP-9603*, Sep. 1996.

[7] T. Doltzmann, A. Sturm and V. Weispfenning. Real Quantifier Elimination in Practice. *Technical Report, University of Passau, MIP-9720*, Dec. 1997.

[8] B. Elspas, M. W. Green, K. N. Levitt, and R. J. Waldinger. Research in Interactive Program-Proving Techniques. Technical report, Stanford Research Institute, Menlo Park, California, USA, May 1972.
[9] H. Enderton. *Mathematical Logic, An Introduction*. Academic Press, 1992.
[10] S. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, 1975.
[11] H. Hong et al. `http://www.cs.usna.edu/ qepcad/B/WhatisQEPCAD.html`.
[12] D. Kapur. Geometry theorem proving using Hilbert's Nullstellensatz. In *Proc. 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC 86)*, pages 202–208, 1986.
[13] D. Kapur. A Refutational Approach to Geometry Theorem Proving. *Artificial Intelligence*, 37:61–93, 1988.
[14] D. Kapur. An approach for solving systems of parametric polynomial equations. In Saraswat and V. Hentenryck, editors, *Principles and Practices of Constraint Programming*, pages 217–244. MIT Press, 1995.
[15] D. Kapur and X. Nie. Reasoning about Numbers in Tecton. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems (ISMIS'94)*, pages 57–70, 1994.
[16] D. Kapur and H. Zhang. An Overview of Rewrite Rule Laboratory (RRL). *Journal of Computer and Mathematics with Applications*, 29:91–114, 1995.
[17] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
[18] S. Katz and Z. Manna. Logical Analysis of Programs. *Communications of the ACM*, 19(4):188–206, Apr 1976.
[19] R. Loos and V. Weispfenning. Applying linear quantifier elimination. *Computer Journal*, 1993.
[20] M. Müller-Olm and H. Seidl. Polynomial Constants are Decidable. In *9th Static Analysis Symposium (SAS)*, LNCS 2477. Springer-Verlag, 2002.
[21] M. Müller-Olm and H. Seidl. Computing Interprocedurally Valid Relations in Affine Programs. *Symposium on Principles of Programming Languages*, 2004.
[22] E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations, 2003. `www.lsi.upc.es/~erodri`; submitted to ISSAC-2004.
[23] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants for imperative programs, 2003. `www.lsi.upc.es/~erodri`.
[24] E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants, 2004. `www.lsi.upc.es/~erodri`: submitted to SAS-2004.
[25] H. Sankaranarayanan, S. Sipma and M. Z. Non-linear Loop Invariant Generation using Gröbner Bases. *Symp. on Principles of Programming Languages*, 2004.
[26] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley, 1998.
[27] B. Wegbreit. The Synthesis of Loop Predicates. *Communications of the ACM*, 17(2):102–112, Feb 1974.
[28] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, 1(3):270–285, Sep 1975.
[29] V. Weispfenning. A Comprehensive Gröbner Basis Algorithm. *Journal of Symbolic Computation*, 1990.
[30] W. Wu. On the decision problem and the elementary geometry. *Scientia Sinica*, 21:150–172, 1978.
[31] W. Wu. Basic principles of mechanical theorem proving in geometries. *J. of Automated Reasoning*, 2:221–252, 1986.

Department of Computer Science,, University of New Mexico,, Albuquerque,USA
E-mail address: `kapur@cs.unm.edu`