Practical Aspects of Algebraic Invariant Generation for Loops with Conditionals

Laura Ildikó Kovács*, Tudor Jebelean** [1], Adalbert Kovács***
*,**Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
Institute e-Austria Timişoara, Romania

***University "Politechnica" of Timişoara, Romania

Abstract.
We present a method that generates automatically algebraic invariant properties of a loop. The implementation and verification process is done in a prototype verification condition generator for imperative programs. This verification tool is integrated into the overall framework of the *Theorema* system, which is based on a version of higher order predicate logic and includes verification procedures for functional and rewrite algorithms but also for procedural programs. The main contribution of this paper is the algorithm that generates invariants for loops with conditionals. In the proposed algorithm program analysis is performed in order to transform the code into a form for which algebraic and combinatorial techniques (symbolic summation, variable elimination, polynomial algebra) can be applied to obtain an invariant property. The application of the method is demonstrated in few examples.

Keywords: Program Analysis and Verification, Loop Invariant Generation, Theorem Proving, Symbolic Summation

1 Introduction

Program (algorithm) verification [4] has a long research tradition but, so far, had relatively little impact on practical software development in the industry. A notable exception is the use of verification techniques in the test of computer processors. However, the design and implementation of reliable software still is an important issue and any progress in this area will be of outmost importance for the future development of the software industry. The logically deep parts of the code are characterized by (nested) loops or recursions. For these parts, formal program verification *is* an appropriate tool. Program verification is a systematic approach to proving the correctness of programs. Correctness means that the program satisfies its specification, given by two logical formulas: the precondition and the postcondition.

Approaching the problem of imperative program verification from a practical point of view has certain implications concerning: the style of specifications, the programming language which is used, the help provided to the user for finding appropriate loop invariants, the theoretical frame used for formal verification, the language used for expressing generated verification theorems as well as the database of necessary mathematical knowledge, and finally the proving power, style and language. The *Theorema* system (www. theorema.org) has certain capabilities which make it appropriate for such a practical approach. Our approach to imperative program verification in *Theorema* is based on the traditional method of inductive assertions, introduced by Floyd–Hoare–Dijkstra (using the weakest precondition strategy) [7], [10], [6], combined with a novel method for verification of programs that contain loops, namely a method based on recurrence equation solvers (the Gosper algorithm [9], the technique of generating functions [15], geometric series), variable elimination and polynomial algebra for generating necessary loop invariants.

[1]The program verification project is supported by BMBWK (Austrian Ministry of Education, Science, and Culture), BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research) in the frame of the e-Austria Timişoara project (www.ieat.ro). The *Theorema* system is supported by FWF (Austrian National Science Foundation) – SFB project F1302

The main contribution of the current paper is the algorithm that generates automatically polynomial invariant relations of loops that contain also conditional statements. This is done by program transformation of loops with conditionals into nested loops, and then systematic invariant generation (by recurrence solving) is performed, followed by variable elimination, invariance checking and Gröbner basis computation. The automatically obtained invariant relations are used in the verification process, that generates the verification conditions needed to be proven in order to ensure program correctness. The proofs can be performed by the available provers of the *Theorema* system.

The rest of the paper is organized as follows: In section 2 a short description of the overall framework of the *Theorema* system is given (see 2.1), followed by the presentation of the verification environment for imperative program verification (see 2.2). The invariant generation technique is described in section 3. The efficiency of our method is illustrated with few examples in section 4. Section 5 concludes with some ideas for the future work.

2 General Framework

2.1 Working Environment: *Theorema*

Theorema (`www.theorema.org`) is a project and a software system that aims at supporting the entire process of mathematical theory exploration: invention of mathematical concepts, invention and verification (proof) of propositions about concepts, invention of problems formulated in terms of concepts, and invention and verification (proof of correctness) of algorithms solving problems, and storage and retrieval of the formulae invented and verified during this process. This integral objective of supporting the entire mathematical invention and verification process was already formulated at the very beginning of the *Theorema* project, see e.g. [2].

Algorithms can be expressed in *Theorema* using the language of predicate logic with equalities interpreted as rewrite rules (which leads to an elegant functional programming style) and program verification is done by proving specifications based on definitions (both are logical formulae). However, the system also contains additionally an imperative language with interpreter and verifier, allowing program verification for imperative programs by generating and proving verification conditions depending on the program syntax [11].

The *Theorema* system is particularly appropriate for program verification, because it delivers the proofs in a natural language by using natural style inferences. The system is implemented on top of the computer algebra system *Mathematica* [16], thus it has access to a wealth of powerful computing and solving algorithms.

2.2 Verification Environment for Imperative Programs in *Theorema*

The implementation and verification process is done in a prototype verification condition generator for imperative programs. This verification tool is integrated into the overall framework of the *Theorema* system.

The user interface has few simple and intuitive commands ($Program$, $Specification$, VCG, $Execute$). Programs are annotated with pre- and postcondition, loop invariants and termination terms (invariants involving linear inequalities and modulo operations still have to be given by the users using the $Assert$ construct). We want to be able to represent a sequential imperative programming language, in which the programs are considered as procedures, without return values and with input, output and/or transient parameters. The commands of the programming language are (see [11], [14]): assignments (that can contain also function calls), blocks, conditionals, loops, procedure calls. Recursivity and mutually recursive procedure calls are not yet available.

For illustration, consider the following example:

Example 1.

$$
\begin{array}{l}
\text{Specification}[\text{“Fermat”, Fermat}[\downarrow N, \downarrow R, \uparrow a, \uparrow b] \\
\quad \text{Pre} \rightarrow \Big((\text{IsInteger}[N, R]) \wedge (N \geq 1) \wedge (R-1)^2 < N \wedge (N \leq R^2) \wedge (\text{Mod}[N, 2] = 1)\Big), \\
\quad \text{Post} \rightarrow \Big((\text{IsInteger}[a, b]) \wedge (N = a * b) \wedge (\text{LF}[a, N, R])\Big)] \\
\text{Program}[\text{“Fermat”, Fermat}[\downarrow N, \downarrow R, \uparrow a, \uparrow b] \\
\quad \text{Module}[u, v, r,
\end{array}
$$

$u := 2 * R + 1;\ v := 1;\ r := R * R - N;$
WHILE$[r \neq 0,$
IF$[r > 0,$
$r := r - v; v := v + 2,$
$r := r + u; u := u + 2],$
Assert $\rightarrow \big((\text{Mod}[u, 2] = 1) \wedge (\text{Mod}[v, 2] = 1)\big)];$
$a := (u - v)/2;\ b := (u + v - 2)/2]]$

Figure 1. Fermat's Algorithm for Integer Factorization

Figure 1 shows a program that factors a given number N (for practical reasons, it is assumed to be odd) in a product of two integer factors, $a,\ b$, such that a is the largest factor of N that is smaller or equal to R (R is the integer approximation of the square root of N). Fermat's algorithms shows how factoring can be done without using any division.

In the $Specification$, the pre- and postconditions of the program are given. The input, transient and output variables of a program are specified, respectively by $\downarrow,\ \updownarrow,\ \uparrow$.

The verification of programs contains two steps. First of all, the *Verification Condition Generator* (VCG), that takes an annotated program with pre– and postcondition (i.e. the program's specification) $\{P\}c\{Q\}$, and, working recursively bottom–to–front on the program syntax, produces, as output, a *Theorema* lemmata containing a collection of formulas (i.e. verification conditions) that must be satisfied in order to ensure the correctness of the program. The automated invariant generation (see section 3) is performed in this phase. Internally, VCG repeatedly modifies the postcondition using a predicate transformer, such that at the end the program c is completely processed and a purely logical proof obligation remains.

In the considered example 1, we will have:

$VCG[Program[\text{“}Fermat\text{”}], Specification[\text{“}Fermat\text{”}]]$

and obtain the lemma that contains all the formulae that have to be proven to ensure (total) correctness of the program.

Subsequently, as a second part of the verification, the generated verification conditions are given to the automated theorem provers of the *Theorema* system in order to check whether they hold. The obtained proofs are generated using natural style of inferences.

Another feature of this imperative programming environment is the $Execute$ constructor, that allows one to run (test) the program in specific input values. For example 1, one can call: $Execute[Fermat[21, 5, a, b]]$, and obtain the computed values of $\{x, y, a, b\} = \{3, 7\}$.

In this paper, we discuss and present practical issues of the first part of the verification process, namely the invariant generation process in the generation of verification conditions. For this, first, let us give some definitions that are needed in the next sections:

Definition 2.1. ***Algebraic Invariants.***

An assertion I is an algebraic invariant *iff:*

1). I is an invariant ([10], [7]);

2). I is an algebraic assertion, i.e. it is a conjunction of polynomial relations over the loop variables.

Details about algebraic notions that appear in the above definitions can be found in [5], [12].

3 Inferring Automatically Valid Invariant Properties

Verification of correctness of loops needs additional information, so-called annotations. In the case of for loops these annotations consist only in the invariants, but in the case of $while$ loops, beside the invariant, another annotation is a termination term, necessary for proving termination [13]. In most verification systems, these annotations are given by the user. It is generally agreed [8] that finding automatically such annotations is difficult. However, in most of the practical situations, finding invariant expression – or at least giving some useful hints – is quite feasible.

For practical applications this may be very helpful for the user. In this section we present our work–in–progress technique for automated invariant generation by combinatorial and algebraic methods.

So far in the automatic invariant generation we dealt with cases when a loop had only assignments (of certain type, namely with Gosper–summable recurrences, geometric series and mutual recurrences) [14]. However, more interesting situations arise in the case when a loop contains also conditional statements. Therefore, as a continuation of the combinatorial and algebraic approach for invariant generation for loops with only assignment, we develop a novel algorithm that allows to generate automatically *algebraic* invariant properties of loops with conditionals.

Let us denote by X the set of variables the loop operates on. For our technique, we assume that the assignment statements from the body of a loop are polynomial assignments of the form $x := p$ (where $x \in X$ and $p \in \Re[X]$), and they are either Gosper-summable recurrences, either geometric series or either mutual recursive with other assignment statement from the loop body. For the example 1, the set of variables is $X = \{r, u, v\}$.

The main idea of our algorithm is to simulate the execution of conditionals with recurrence equations. We have seen, that in the case of loops with only assignment statements the combinatorial manipulation of the statements from the loop's body is efficient for invariant generation. Starting from this base–case, as a first step, we transform our program into a system that contains no more conditional statements, only (nested) whiles. Furthermore, the execution of the nested whiles will be simulated with recurrence equations. The main steps of the method are synthesized as follows:

Invariant Generation Algorithm

- Step 1: Program Transformation;
- Step 2: Invariant generation for each system of nested loops by combinatorics and algebra;
  - Step 2.1: Indexing the inner loops;
  - Step 2.2: Statement and variable manipulation for the connected inner loops and recurrence solving for each inner loop;
  - Step 2.3: Recurrence-counter elimination;
- Step 3: Build the union of the obtained formulae for the two nested–loop subsystems;
- Step 4: Check invariance for generated formulae. Keep only those that are invariant;
- Step 5: Take the minimal set of the invariant properties, by using Gröbner basis w.r.t. to the loop variables;
- Step 6: The final invariant is the conjunction of the formulae from Step 5 and of the non–algebraic assertions (specified by the Assert construct).

In transforming the code at step 1, we use the following transformation rule:

Proposition 3.1. *Transformation Rule for while loops with conditionals*

$$\frac{\begin{array}{ll} \textit{WHILE}[b, & \textit{WHILE}[b, \\ \quad c1; c2; c4; & \quad c1; c3; c4; \\ \quad \textit{WHILE}[b \wedge b1, c1; c2; c4]; & \quad \textit{WHILE}[b \wedge \neg b1, c1; c3; c4]; \\ \quad \textit{WHILE}[b \wedge \neg b1, c1; c3; c4]] & \quad \textit{WHILE}[b \wedge b1, c1; c2; c4]] \end{array}}{\textit{WHILE}[b, c1; IF[b1, c2, c3]; c4]}$$

For example 1, after Step.1 we obtain two nested–loop subsystems, each with one outer-loop and two inner loops.

WHILE$[r \neq 0$	WHILE$[r \neq 0$
$r := r - v; v := v + 2;$	$r := r - v; v := v + 2;$
WHILE$[r \neq 0 \wedge r > 0$	WHILE$[r \neq 0 \wedge \neg(r > 0)$
$r := r - v; v := v + 2];$	$r := r + u; u := u + 2];$
WHILE$[r \neq 0 \wedge \neg(r > 0)$	WHILE$[r \neq 0 \wedge r > 0,$
$r := r + u; u := u + 2]];$	$r := r - v; v := v + 2]];$

Next, we continue with step 2. For step 2.1, we assign the counter j to the main loop, j_1 to the first inner loop and j_2 to the second inner loop.

The next step of the generation process is the statement– and variable–manipulation, i.e. step 2.2. Therefore, we proceed as follows:

1. for each nested–loop system, rewrite the recursive assignments using the proper indexes (loop–counters). For those variables from the set V of loop variables, that do not change in the specific part, consider the assignment that describes the constant property of them (i.e. $x_{j+1} := x_j$, where $x \in X$)
2. for the inner while, by the combinatorial methods for summation, generate closed forms for the recursive equations
3. replace the inner while loops with their system of closed forms and the assignments for the non–changed variables, taking care of the proper specification of the loop counter, and that the initial values of the variables of the first inner loop are given by the initial values of the outer loop's variables, the initial values of the variables of the second inner loop are given by the final values of the first inner loop's variables, etc.
4. solve – by repeated substitutions and counter elimination– the obtained system of recursive equations, obtaining the invariant for the outer loop.

For example 1, the intermediate steps are as follows:

- For the first inner loop, we obtain, by (Gosper) recurrence solving:

$$\begin{aligned} r_{j_1} &= r_j - j_1 * v_j - j_1 * (j_1 - 1) \\ v_{j_1} &= v_j + 2 * j_1 \\ u_{j_1} &= u_j \end{aligned}$$

  where $r_j,\ v_j,\ u_j$ are the values of $r,\ v,\ u$ before the first inner loop (i.e. the values from the beginning of the outer loop).

- For the second inner loop, we obtain, by (Gosper) recurrence solving:

$$\begin{aligned} r_{j_12} &= r_{j_1} + j_2 * u_{j_1} + j_2 * (j_2 - 1) \\ u_{j_2} &= u_{j_1} + 2 * j_2 \\ v_{j_2} &= v_{j_1} \end{aligned}$$

- By substituting the expressions for $r_{j_1},\ v_{j_1},\ u_{j_1}$, we obtain the following system of equations:

$$\begin{aligned} r_{j_2} &= r_j - j_1 * v_j - j_1 * (j_1 - 1) + j_2 * u_j + j_2 * (j_2 - 1) \\ u_{j_2} &= u_j + 2 * j_2 \\ v_{j_2} &= v_j + 2 * j_1 \end{aligned}$$

Furthermore, step 2.3. of the algorithm follows. Thus, we eliminate the inner-loop counters j_1 and j_2 and obtain the relation:

$$4 * r_{j_2} - 4 * r_j - 2 * u_j + u_j^2 - 2 * v_{j_2} + v_{j_2}^2 + 2 * v_j - v_j^2 = -2 * u_{j_2} + u_{j_2}^2.$$

Next, step 3 follows. For the second block of nested while loops we proceed in the same manner, and obtain also an invariant property (in example 1 it is the same as the one from step 2.3.). Taking the conjunction of the two formulas, we obtain a set of invariant properties of the while loop of the original problem.

In the next step (i.e. step 4) the invariance checking is performed, namely we have to check the inductiveness and initial properties ([7], [10]) of the generated assertions. The initial invariant–condition holds since the obtained formulae are closed forms generated by recurrence solvers, thus they satisfy the initial values. For the inductiveness property, one must perform an additional checking, since the variable elimination process can produce some intermediate formulae that are not true for each branching condition. The invariance property checking of a formula is done as follows:

1. Take the sequence of command $S1 = c1; c2; c4$ and $S2 = c1; c3; c4$. $S1$ and $S2$ represents one possible loop iteration.
2. Consider the assignments of $S1$ and $S2$ as rewrite rules, and apply them (separately) on each formula from step 3.
3. If a formula remains the same after the applications of the rewrite rules of $S1$ and $S2$, respectively, we can conclude that the formula holds before and after each iteration of the loop. Thus this formula represent an invariant property of the loop.

For example 1, the formula

$$4 * r_{j_2} - 4 * r_j - 2 * u_j + u_j^2 - 2 * v_{j_2} + v_{j_2}^2 + 2 * v_j - v_j^2 = -2 * u_{j_2} + u_{j_2}^2$$

satisfies the invariance property. Actually, it has a more intuitive form, namely:

$$4 * r_{j_2} + 2 * u_{j_2} - 2 * v_{j_2} - u_{j_2}^2 + v_{j_2}^2 = 4 * r_j + 2 * u_j - 2 * v_j - u_j^2 + v_j^2$$

thus a relation between the initial and final values of the variables of the outer loop after one iteration. Since this was also the formula obtained from the second nested–loop block, after step 4 we have the following set of invariant formulae:

$$\{4 * r_{j_2} + 2 * u_{j_2} - 2 * v_{j_2} - u_{j_2}^2 + v_{j_2}^2 = 4 * r_j + 2 * u_j - 2 * v_j - u_j^2 + v_j^2,$$
$$4 * r_{j_2} + 2 * u_{j_2} - 2 * v_{j_2} - u_{j_2}^2 + v_{j_2}^2 = 4 * r_j + 2 * u_j - 2 * v_j - u_j^2 + v_j^2\}$$

Finally, step 5 is performed. By application of Gröbner basis [1] w.r.t. to the set X of loop variables, the invariant property that was generated by our method is:

$$4 * r_{j_2} - 4 * r_j + 2 * u_{j_2} - u_{j_2}^2 - 2 * u_j + u_j^2 - 2 * v_{j_2} + v_{j_2}^2 + 2 * v_j - v_j^2 = 0$$

This relation establishes an invariant property of the loop, and, by initial values substitution (given by the assignments before the outer–loop), we obtain the invariant property:

$$4 * N + 4 * r + 2 * u - u^2 - 2 * v + v^2 = 0.$$

Applying the exit condition of the loop, $r = 0$, it yields $4 * N = u^2 - v^2 + 2 * v - 2 * u$. This property is then used for proving that the computed values of a, b, x, y satisfy the given postcondition of the program specification. However, some additional invariant property is also needed to prove (partial) correctness, namely: $(\text{Mod}[u, 2] = 1) \wedge (\text{Mod}[v, 2] = 1)$, which cannot be inferred with our method, thus, using the $Assert$ option, has to be given by the user.

Hence, the complete complete invariant property of the loop is:

$$(4 * N + 4 * r + 2 * u - u^2 - 2 * v + v^2 = 0) \wedge (\text{Mod}[u, 2] = 1) \wedge (\text{Mod}[v, 2] = 1)$$

For proving the (partial) correctness of the program, by calling VCG (see section 2.2), we obtain a lemmata in *Theorema* syntax:

Lemma(Fermat)

for any: $N, R, x, y, a, b, u, v, r$

(WHILE.Inv)

$(4 * N + 4 * r + 2 * u - u^2 - 2 * v + v^2 = 0) \wedge (\text{Mod}[u, 2] = 1) \wedge (\text{Mod}[v, 2] = 1) \wedge r \neq 0 \Rightarrow$

$\Big(r > 0 \Rightarrow (4 * N + 4 * (r - v) + 2 * u - u^2 - 2 * (v + 2) + (v + 2)^2 = 0) \wedge (\text{Mod}[u, 2] = 1) \wedge (\text{Mod}[v + 2, 2] = 1\Big)$

$\bigwedge$

$\Big(r \leq 0 \Rightarrow 4 * N + 4 * (r + u) + 2 * (u + 2) - (u + 2)^2 - 2 * v + v^2 = 0) \wedge (\text{Mod}[u + 2, 2] = 1) \wedge (\text{Mod}[v, 2] = 1\Big)$

(WHILE.Final)

$(4 * N + 4 * r + 2 * u - u^2 - 2 * v + v^2 = 0) \wedge (\text{Mod}[u, 2] = 1) \wedge (\text{Mod}[v, 2] = 1) \wedge \neg(r \neq 0) \Rightarrow$

$\text{IsInteger}\Big[\frac{u - v}{2}\Big] \bigwedge \text{IsInteger}\Big[\frac{u + v - 2}{2}\Big] \bigwedge \Big(N = \frac{1}{4} * (u - v) * (u + v - 2)\Big) \bigwedge \text{LF}\Big(\frac{u - v}{2}, N, R\Big)$

(Init)

$(\text{IsInteger}[N, R) \wedge (N \geq 1) \wedge (R - 1)^2 < N \wedge (N \leq R^2) \wedge (\text{Mod}[N, 2] = 1) \Rightarrow$

$(0 = 0) \wedge (\text{Mod}[1 + 2 * R, 2] = 1) \wedge (1 = 1)$

The proof of this lemma, using a specified knowledge base, can be done automatically by the PCS prover of the *Theorema* system [3], which uses quantifier elimination.

Note. The number of iteration step taken by the Fermat algorithm to find the factors $a,\ b$ of $n = a * b$ is essentially proportional to $(u + v - 2)/2 - \lfloor\sqrt{n}\rfloor = b - R$.

4 Examples

Example 4.1. *LCM-GCD Algorithm*

Specification[*"LCM-GCD Computation", LcmGcd*[$\downarrow a, \downarrow b, \uparrow L, \uparrow G$]

Pre $\rightarrow \Big((a > 0) \wedge (b > 0)\Big)$,

Post $\rightarrow \Big((L = LCM[a, b]) \wedge (G = GCD[a, b])\Big)$]

Program[*" LCM-GCD Computation", LcmGcd*[$\downarrow a, \downarrow b, \uparrow L, \uparrow G$]

Module[x, y, u, v,

$x := a;\ y := b;\ u := b;\ v := a;$

WHILE[$x \neq y$,

IF[$x > y$,

$x := x - y; v := v + u,$

$y := y - x; u := u + v$],

TeminationTerm $\rightarrow Abs[x - y]$]];

$L := (u + v)/2;\ G := x$]]

Figure 2. LCM-GCD Algorithm

Figure 2 shows a procedures that calculate simultaneously the lcm and gcd of integers a and b. The transformation of our program yields to two nested–loops, each of them with two inner–loops. The application of our technique produced invariant property obtained only by recurrence solving, simplification, variable elimination and usage of Gröbner basis. **The resulting invariant is** $u * x + v * y - 2 * a * b = 0$. Applying the exit condition

of $x = y$ yields $x * (u + v) = 2 * a * b$. Assuming that $x = G = GCD[a, b]$ and $\frac{u+v}{2} = L = LCM[a, b]$, the invariant states that: $GCD[a, b] * LCM[a, b] = a * b$.

Example 4.2. *Wensley's Algorithm for Real Division*

$$
\begin{array}{l}
\textit{Specification}[\text{“ReDiv”}, \textit{ReDiv}[\downarrow P, \downarrow Q, \downarrow Tol, \uparrow r] \\
\quad \textit{Pre} \rightarrow \Big((IsReal[P]) \wedge (IsReal[Q]) \wedge (IsReal[Tol])\Big) \wedge (Q > P) \\
\quad \wedge (P \geq 0) \wedge (Tol \geq 0)\Big), \\
\quad \textit{Post} \rightarrow \Big((P/Q < r + Tol) \wedge (r \leq P/Q)\Big)] \\
\textit{Program}[\text{“ReDiv”}, \textit{ReDiv}[\downarrow P, \downarrow Q, \downarrow Tol, \uparrow r] \\
\quad \textit{Module}[a, b, d, y, \\
\qquad a := 0;\ b := Q/2;\ d := 1;\ y := 0; \\
\qquad \textit{WHILE}[d \geq Tol, \\
\qquad\quad \textit{IF}[P < a + b, \\
\qquad\qquad b := b/2; d := d/2, \\
\qquad\qquad a := a + b; y := y + d/2; b := b/2; d := d/2], \\
\qquad\quad \textit{Assert} \rightarrow (y \leq P/Q < y + d) \wedge (0 < d \leq 1), \\
\qquad\quad \textit{TeminationTerm} \rightarrow d - Tol]; \\
\qquad r := y]]
\end{array}
$$

Figure 3. Real Division Algorithm

Figure 3 shows the Wensley's algorithm, that for two given real numbers P and Q, with $0 \leq P < Q \leq 1$, finds their quotient y with a tolerance Tol, that is finds y such that $y \leq P/Q < y + Tol$. The algorithm uses only addition and division by two. We introduce a new variable d, representing the tolerance of the current guess of the loop, thus d has to satisfy the invariant property $y \leq P/q < y + d \wedge 0 < d \leq 1$. This formula lies outside of the power of our method (i.e. it is not algebraic invariant), therefore one has to specify it manually, using the $Assert$ option.
Applying the automated generation technique for invariant properties, **the generated invariant in this case is** $(b = \frac{d*Q}{2}) \wedge (a * d = d * y * Q)$. This shows the second part of the specification of the property that d has to satisfy.

Example 4.3. *Extended Euclid Algorithm*

$$
\begin{array}{l}
\textit{Specification}[\text{“GCDExtend”}, \textit{GCDExt}[\downarrow x, \downarrow y, \uparrow G] \\
\quad \textit{Pre} \rightarrow \Big((x > 0) \wedge (y > 0)\Big) \\
\quad \textit{Post} \rightarrow (G = GCD[x, y])] \\
\textit{Program}[\text{“GCDExtend”}, \textit{GCDExt}[\downarrow x, \downarrow y, \uparrow G] \\
\quad \textit{Module}[a, b, p, q, r, s, \\
\qquad a := x;\ b := y;\ p := 1;\ q := 0;\ r := 0;\ s := 1; \\
\qquad \textit{WHILE}[a \neq b, \\
\qquad\quad \textit{IF}[a > b, \\
\qquad\qquad a := a - b;\ p := p - q;\ r := r - s, \\
\qquad\qquad b := b - a;\ q := q - p;\ s := s - r],
\end{array}
$$

$$\textit{Assert} \rightarrow \Big((\textit{GCD}[a,b] = \textit{GCD}[x,y]) \wedge (a > 0) \wedge (b \geq 0)\Big),$$
$$\textit{TeminationTerm} \rightarrow \textit{Abs}[a - b];$$
$$G := a]]$$

Figure 4. Extended GCD computation

The program in figure 4 shows Euclid's algorithm for extended GCD computation. **The generated invariant properties are** $\big((p*s-q*r=1)\wedge(b=q*x+s*y)\wedge(a=p*x+r*y)\wedge(x=a*s-b*r)\wedge(y=b*p-a*q)\big)$**.** Other invariant properties that are needed to be given by the user (since they are not polynomial relations) are: $\Big((\text{GCD}[a,b] = \text{GCD}[x,y]) \wedge (a > 0) \wedge (b \geq 0)\Big)$.

5 Conclusions and Future Work

Combined with a practically oriented version of the theoretical frame of Hoare–logic, *Theorema* provides readable arguments for the correctness of programs, as well as useful hints for debugging. Moreover, it is apparent that the use of program transformation, algebraic and combinatorial techniques (summation methods, variable elimination, polynomial algebra) is a promising approach to analysis of loops, namely for generation of (algebraic) invariants.

Regarding the verifier and the programming language, our work plans in the near future are following:

- enrich the invariant generation technique with treatment of other type of recurrences and solving techniques;
- develop and integrate in the verifier a technique for mechanically inferring loop invariants that are linear inequalities or non–algebraic;
- continue our previous work on generation of termination terms (see [14]).

References

[1] B. Buchberger. *Groebner-Bases: An Algorithmic Method in Polynomial Ideal Theory*, chapter 6, pages 184–232. Reidel Publishing Company, Dordrecht, Netherlands, 1985.

[2] B. Buchberger. Symbolic Computation: Computer Algebra and Logic. In *Frontiers of Combining Systems*, volume 3 of *Applied Logic Series*, pages 193–219. Kluwer Academic Publishers, 1996.

[3] B. Buchberger. The PCS Prover in Theorema. In *Proceedings of EUROCAST 2001*, Las Palmas de Gran Canaria, 2001. Springer. Lecture Notes in Computer Science 2178.

[4] B. Buchberger. Practical and Theoretical Aspects of Program Verification. In *Computer Aided Verification of Information Systems (www.ieat.ro)*, Timişoara, Romania, February 2003.

[5] D. Cox, J. Little, and D. O'Shea. *Ideal, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, second edition, 1998.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentince-Hall, 1976.

[7] R.W. Floyd. Assigning Meanings to Programs. In *Proc. Symphosia in Applied Mathematics 19*, pages 19–37, 1967.

[8] G. Futschek. *Programmentwicklung und Verifikation*. 1989.

[9] R. W. Gosper. Decision Procedures for Indefinite Hypergeometric Summation. *Proc. of the National Academy of Science, USA*, 75(5–6):40–42, 1978.

[10] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.

[11] M. Kirchner. Program Verification with the Mathematical Software System *Theorema*. Technical Report 99–16, RISC, Johannes Kepler University, Linz, Austria, 1999.

[12] A. Kovacs. *Special Chapters of Mathemtics (in romanian)*. Editura Politehnica, Timisoara, Romania, 2002.

[13] L. Kovacs. *Program Verification using Hoare Logic*, February 2003. Contributed talk at Computer Aided Verification of Information Systems (CAVIS-04), Timisoara, Romania.

[14] L. Kovacs and T. Jebelean. Automated Generation of Loop Invarinats by Recurrence Solving in *Theorema*. In *Proc. of SYNASC'04*, Timişoara, Romania, 2004. Mirton.

[15] R. P. Stanley. Differentiably finite power series. *European Journal of Combinatorics*, 1(2):175–188, 1980.

[16] S. Wolfram. *The Mathematica Book, 3rd ed.* Wolfram Media / Cambridge University Press, 1996.

Authors' address:

*,** Research Institute for Symbolic Computation,
Johannes Kepler University, A-4040 Linz, Austria
Institute e-Austria Timişoara, Romania
{lkovacs, jebelean}@risc.uni-linz.ac.at

*** University "Politechnica" Timişoara, Department of Mathematics,
Ro- 300223 Timişoara, Piaţa Regina Maria Nr.1,
profdrkovacs@yahoo.com