An Algorithm for Automated Generation of Invariants for Loops with Conditionals

Laura Ildikó Kovács and Tudor Jebelean
Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria,
Institute e–Austria, Timişoara, Romania
Email: {kovacs, jebelean}@risc.uni-linz.ac.at

***Abstract*— We present an algorithm that generates automatically (algebraic) invariant properties of a loop 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 can be applied to obtain invariant properties. These invariants are then used for verifying partial correctness of imperative programs in the *Theorema* system (www.theorema.org). The application of the method is demonstrated in few examples.**

AMS Subject Classification: **33F10, 65G20, 68N30, 68Q60, 68W30**

Keywords and phrases: **program analysis and verification, invariant generation, theorem proving, symbolic summation**

I. Introduction

The main goal of our work is to develop a suitable imperative programming language model and to support imperative program verification in the automated theorem prover system *Theorema*. The design of a framework for program verification in an expressive logic like *Theorema* is driven by two main reasons. On the one hand, we want to develop a method that generates verification conditions, thus proof obligations, in the *Theorema* syntax. This process is based on the traditional method of inductive assertions, introduced by Floyd–Hoare–Dijkstra (using the weakest precondition strategy) [9], [11], [7], combined with a novel method for automated invariant generation for loops, namely a method based on recurrence equation solvers (the Gosper algorithm [10], the technique of generating functions [21], geometric series), variable elimination and polynomial algebra. On the other hand, we want to apply the *Theorema* provers to prove these verification conditions, producing in this way useful case studies for the development of the existing *Theorema* provers.

The current paper extends an earlier conference paper [16] in a number of respects:

• A modest generalization of the programming environment. A new loop option, `Assert`, is introduced to allow the user in specifying non-algebraic invariants (such as inequalities, modulo operation, etc.);

• We treat geometric series recurrences;

• Most importantly, we are now able to generate 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 then in the verification process.

II. General Framework

A. 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, and invention and verification of algorithms [4]. The *Theorema* system is particularly appropriate for functional and imperative program verification [12], 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* [23], thus it has access to a wealth of powerful computing and solving algorithms.

B. Programming Model for Imperative Program Verification in Theorema

1) Abstract Syntax: The basic model of the programming language is quite general. 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 ([12]): assignments, blocks, conditional statements, loops (with optional arguments for loop assertions), procedure calls. Recursivity and mutually recursive procedure calls are not yet available.

2) Semantics: We use an axiomatic semantics for the programming language, by using the so-called Hoare triple [11]. The Hoare logic rules are defined in a *weakest precondition style* [7], [9], and they were already presented in some of our previous conference papers (see, e.g. [14]). In this chapter we state only the semantic rule for the partial correctness of a while loop, but first let us give three definitions.

Definition 2.1: Algebraic Assertions.
An assertion is *algebraic*, iff it is a conjunction of polynomial equalities (polynomials over a ring of numbers, with program variable indeterminates).

Definition 2.2: Invariants(Inductive Assertions)[9]
An assertion I is an *invariant* for $\{P\}\text{While}[b, c]\{Q\}$ iff it satisfies the following conditions:

(1). Initial condition: $P \Rightarrow I$;
(2). Iterative condition: $\{I \wedge b\}c\{I\}$;
(3). Exit condition: $(I \wedge \neg b) \Rightarrow Q$.

Definition 2.3: Algebraic Invariants.
An assertion I is an *algebraic invariant* iff I is an invariant and I is an algebraic assertion.
Now, the semantic rule for the *partial correctness of a while loop* is as follows:

$$\frac{P \Rightarrow I \qquad I \wedge b \Rightarrow I' \qquad (I \wedge \neg b) \Rightarrow Q}{\{P\}\ \text{While}[b, c]\ \{Q\}},$$

where I' denotes the computed weakest precondition of the loop body c with respect to the postconditions I (I is a loop invariant).

C. Verification Environment

The implementation and verification process is done in a prototype verification condition generator for imperative programs, 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. For illustration, consider the following example:

Example 2.1: Wensley's Algorithm for Real Division

`Specification`["ReDiv", ReDiv$[\downarrow P, \downarrow Q, \downarrow Tol, \uparrow r]$,
  `Pre` $\rightarrow (IsReal[P, Q, Tol]) \wedge (Q > P \geq 0) \wedge (Tol \geq 0)$,
  `Post` $\rightarrow (P/Q < r + Tol) \wedge (r \leq P/Q)$]
`Program`["ReDiv", ReDiv$[\downarrow P, \downarrow Q, \downarrow Tol, \uparrow r]$,
  `Module`$[\{a, b, d, y\}$,
    $a := 0;\ b := Q/2;\ d := 1;\ y := 0;$
    `While`$[d \geq Tol$,
      `If`$[P < a + b$,
        $b := b/2; d := d/2,$
        $a := a + b; y := y + d/2; b := b/2; d := d/2]];$
    $r := y]]$

The *Verification Condition Generator* (`VCG`) takes an annotated program with pre- and postcondition, and, working recursively bottom–up on the program syntax, produces, as output, a *Theorema* lemma 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 III) is performed in this phase. These verification conditions are then given to the automated theorem provers of *Theorema* in order to check whether they hold. The obtained proofs are generated using natural style of inferences.

III. Inferring Automatically Valid Invariant Properties

We present our work–in–progress technique for automated *algebraic invariant generation* for loops with conditionals. This is done by transforming the loop by a certain rule (see below) into two nested loops. The resulting program can then be treated with algebraic and combinatorial techniques. Not–algebraic invariants, e.g. involving linear inequalities, modulo operations, still have to be given by the users (using the `Assert` option).

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 $x := p$ ($x \in X$ and $p \in \Re[X]$, $\Re$ is a ring of numbers), and they are Gosper-summable recurrences, geometric series or mutual recursive with other assignment statement from the loop body. (For example 2.1, $X = \{a, b, d, y\}$.)

Algorithm for Invariant Generation
Step 1: Transformation of loops with conditionals into nested loops with assignments only (see prop. 3.1);
Step 2: Generation of possible invariants 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 property 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` option).

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

Proposition 3.1: Transformation Rule for while loops with conditionals

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

where all the loops have the invariant I, and $b1'$ denotes the modified formula $b1$ after the assignement-statement(s) $c1$.

Proof: The proof is done by applying the semantic rules for while-, conditional- and compositional statements [11], [14], together with some reasoning about propositional formulae. A step-by-step prof is available in [15]. ■

We illustrate now the method by applying it to example 2.1:
Step.1: We obtain two nested-loop subsystems, each with one outer-loop and two inner loops.
Step 2.1: We proceed with simulating the execution of nested loops by assigning the counter j to the main loop, j_1 to the first inner loop and j_2 to the second inner loop.
Step 2.2: For each nested–loop system, rewrite the recursive assignments using the proper indexes (loop–counters). For those variables from the set X 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

$x \in X$). For the inner whiles, by the combinatorial methods for summation, generate closed forms for the recursive equations [16]. Thus, for the inner loops, by (Gosper and geometric series) recurrence solving, we obtain:

$$\begin{array}{lcllcl} a_{j_1} & = & a_j & a_{j_2} & = & a_{j_1} + 2 * b_{j_1} - \frac{b_{j_1}}{2^{j_2-1}} \\ b_{j_1} & = & \frac{b_j}{2^{j_1}} & b_{j_2} & = & \frac{b_{j_1}}{2^{j_2}} \\ d_{j_1} & = & \frac{d_j}{2^{j_1}} & d_{j_2} & = & \frac{d_{j_1}}{2^{j_2}} \\ y_{j_1} & = & y_j & y_{j_2} & = & y_{j_1} + d_{j_1} - \frac{d_{j_1}}{2^{j_2}}, \end{array}$$

where $a_j,\ b_j,\ d_j,\ y_j$ are the values of $a,\ b,\ d,\ y$ before the first inner loop (i.e. the values from the beginning of the outer loop).

Finally, we replace the inner loops with their system of closed forms and the assignments for the non–changed variables, using 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.

$$\begin{array}{lcl} a_{j_2} & = & a_j + \frac{b_j}{2^{j_1-1}}\left(1 - \frac{1}{2^{j_2}}\right) \\ b_{j_2} & = & \frac{b_j}{2^{j_1+j_2}} \\ d_{j_2} & = & \frac{d_j}{2^{j_1+j_2}} \\ y_{j_2} & = & y_j + \frac{d_j}{2^{j_1}}\left(1 - \frac{1}{2^{j_2}}\right). \end{array}$$

Step 2.3: We eliminate the inner-loop counters j_1 and j_2, and obtain the equations between the initial and final values of the loop variables, after an iteration. Writing respectively, $a,\ b,\ d,\ y$ instead of $a_{j_2},\ b_{j_2},\ d_{j_2},\ y_{j_2}$ (i.e. final values of loop variables), and $a_0,\ b_0,\ d_0,\ y_0$ instead of $a_j,\ b_j,\ d_j,\ y_j$ (i.e. initial values of loop variables) the possible invariant properties are:

$$\begin{array}{lcl} -b + \frac{b_0 * d}{d_0} & = & 0 \\ a * d_0 - a_0 * d_0 - 2b_0 * y + 2b_0 * y_0 & = & 0 \\ -b_0 d + b d_0 & = & 0 \\ -(a - a_0) * d + b * (-2y_0 + 2y) & = & 0 \\ -2b_0 + \frac{(a - a_0 + 2b) * d_0}{d} + y - y_0 & = & 0. \end{array} \quad (1)$$

For the second block of nested while loops we proceed in the same manner, and obtain also a set of possible invariant properties.

$$\begin{array}{lcl} a * d - a_0 * d - 2by + 2by_0 & = & 0 \\ \frac{(a - a_0) * d}{d_0} - b_0 - 2 * d_0^{-1} * d_0 + y_0 - y & = & 0 \\ -b + \frac{b_0 * d}{d_0} & = & 0 \\ a * d_0 - a_0 * d_0 - 2b_0 * y + 2b_0 * y_0 & = & 0 \\ -b_0 * d + b * d_0 & = & 0 \\ \frac{d * (y - y_0)}{d_0} - d_0 - d_0^{-1} * d_0 + y_0 - y & = & 0 \\ 2b_0 + \frac{(a - 2b_0 - a_0) * d_0}{d_0 + y_0 - y} & = & 0. \end{array} \quad (2)$$

Step 3: Taking the conjunction of the two set of formulas obtained from the nested loop subsystems, i.e. (1) and (2), we obtain a set of possible invariant properties of the while loop from the original problem. For this particular example we have a system with 12 polynomial equations.

Step 4: For the obtained polynomial equations we have to check the conditions from definition 2.2. Condition (1) of definition 2.2 holds since the obtained formulae are closed forms generated by recurrence solvers using the initial values given by the initial values of the loop variables before the loop execution. For condition (2) of definition 2.2 one must perform an additional checking, since the variable elimination process may produce some intermediate formulae that are not true for each branching condition. This additional check is done as follows:

• Take the sequence of command $S1 = c1; c2; c4$ and $S2 = c1; c3; c4$. $S1$ and $S2$ represents one possible loop iteration;
• Consider the assignments of $S1$ and $S2$ as rewrite rules, and apply them (separately) on each formula from step 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 represents an invariant property of the loop.

After performing these steps, the set of invariant formulas has 6 polynomial equations, namely:

$$\begin{array}{lcl} -b + \frac{b_0 * d}{d_0} & = & 0 \\ -b_0 * d + b * d_0 & = & 0 \\ -(a - a_0) * d + b * (-2y_0 + 2y) & = & 0 \\ \frac{d * (y - y_0)}{d_0} - d_0 - d_0^{-1} * d_0 + y_0 - y & = & 0 \\ \frac{(a - a_0) * d}{d_0} - b_0 - 2d_0^{-1} * d_0 + y_0 - y & = & 0 \\ a * d - a_0 * d - 2b * y + 2 * b * y_0 & = & 0. \end{array} \quad (3)$$

Step 5: By application of Gröbner basis [3] on (3) w.r.t. to X, the invariant property that was generated by our method is:

$$-b + \frac{b_0 * d}{d_0} = 0 \wedge a * d - a_0 * d - 2b_0 * d * d_0^{-1} * y + \frac{2 * b_0 * d * y_0}{d_0} = 0.$$

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:

$$-b + \frac{1}{2} * d * Q = 0 \ \wedge \ a * d - d * y * Q = 0.$$

Step 6: However, some additional invariant property is also needed to prove (partial) correctness, namely: $y \leq P/q < y + d \wedge 0 < d \leq 1$. This formula, required by condition (3) of definition 2.2, lies outside of the power of our method (i.e. it is not algebraic), therefore one has to specify it manually, using the `Assert` option. The complete invariant will be the conjunction of the automatically generated invariant by our method and the user-asserted one.

For proving the partial correctness of the program, by calling `VCG` (see section II-C), we obtain a *Theorema* lemma that contains the verification conditions. The proof of this lemma, using a specified knowledge base, can be done automatically by the PCS prover of the *Theorema* system [5], which uses quantifier elimination.

IV. Related Work on Invariant Generation

There are two main approaches, namely static and dynamic techniques for invariant discovery. The *dynamic method* executes a program on a collection of inputs and infers invariants

from captured variable traces [8], [1], [6]. Since our method performs static invariant generation, it is not possible yet for us to make comparison with these techniques. The *static approach* of invariant generation operates on the program text, not on test runs, therefore has the advantage that the reported properties hold for any program run. There are several research directions:

Abstract interpretations, widening and narrowing. In [22], [2] linear invariants are generated by computing under- and over-approximations of the reachable state set, using refined widening and narrowing and quantifier elimination. Generation of non–linear invariants involving multiplication becomes difficult with these methods. Our method, with restriction to certain classes of recurrences and algebraic properties, can generate invariants involving multiplications.

Using Gröbner basis. As an alternative to the iterative strategy are the approaches from [20], [18], [19], namely a method built upon polynomial ideal theory, by Gröbner bases computation, in order to generate polynomial imvariants by using: numerical constraint solvers [20]; fixed point computation [19]; weakest precondition computation of a generic polynomial relation [18]. So far, the usage of these methods has been limited to linear invariants, and they need to fix apriori the degree of a generic polynomial template, which is not the case in our technique. Moreover, as our recent practical experiments show, we are able also to obtain non–polynomial invariant properties (e.g. factorial or exponential expressions).

V. Further Examples

We have tested our algorithm with a number of examples (see e.g. [15]). For instance, in the case of *Fermat's algorithm for integer factorization* [13] the generated invariant is $4*N+4*r+2*u-u^2-2*v+v^2=0$; for *LCM computation* [7] the obtained invariant is $u*x+v*y-2*a*b=0$; for *Extended Euclid Algorithm* [13] the generated invariant property is $\big((p*s-q*r=1)\wedge(b=q*x+s*y)\wedge(a=p*x+r*y)\wedge(x=a*s-b*r)\wedge(y=b*p-a*q)\big)$; for a nested loop-system, such as *M*anna's Hardware Integer division algorithm [17], the algorithm succeeds with the generated invariant property $(x2*y3-y2=0)\wedge(x1*y3=y3*(y1+x2*y4)$ for the loop with two conditionals, whereas for the loops with only assignments the generated invariant is $x2*y3-y2=0$.

VI. 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 [16].

Acknowledgment

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 Timisoara project. The *Theorema* system is supported by FWF (Austrian National Science Foundation) – SFB project F1302.

References

[1] J. H. Andrews. Testing Using Log File Analysis: Tools, Methods and Issues. In *13th Annual Int. Conference on Automated Software Engineering (ASE'98)*, 1998.

[2] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful Techniques for the Automatic Generation of Invariants. In *Static Analysis Symposium*, volume 1102 of *LNCS*, pages 323–335, 1996.

[3] B. Buchberger. Groebner-Bases: An Algorithmic Method in Polynomial Ideal Theory. In *Multidimensional Systems Theory - Progress, Directions and Open Problems in Multidimensional Systems*, pages 184–232, 1985.

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

[5] B. Buchberger. The PCS Prover in Theorema. In *Proceedings of EUROCAST 2001*, 2001. Lecture Notes in Computer Science 2178.

[6] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[7] E. W. Dijkstra. *A Discipline of Programming*. 1976.

[8] M. D. Ernst et al. Dynamically Discovering Likely Program Invariants to Support Program Evaluation. Technical report, University of Washington, 2000.

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

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

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

[12] T. Jebelean, L. Kovacs, and N. Popov. Large Experimental Program Verification in the Theorema System. In *Proceedings ISOLA 2004, Cyprus*, pages 92–99, 2004.

[13] D. E. Knuth. *The Art of Computer Programming*, volume 2. 1969.

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

[15] L. Kovacs. Using Combinatorial and Algebraic Techniques for Automatic Generation of Loop Invariants. Technical Report 05–09, RISC-Linz, Austria, 2005.

[16] L. Kovacs and T. Jebelean. Automated Generation of Loop Invarinats by Recurrence Solving in *Theorema*. In *Proc. of SYNASC'04*, pages 451–464, Romania, 2004.

[17] Z. Manna. *Mathematical Theory of Computation*. 1974.

[18] M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Information Processing Letters*, 91(5):233–244, 2004.

[19] E. Rodriguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *Proc. of ISSAC'04*, pages 266–273, Spain, 2004.

[20] S. Sankaranaryanan, B.S. Henry, and Z. Manna. Nonlinear Loop Invariant Generation using Groebner Bases. In *ACM Principles of Programming Languages (POPL'04)*, Italy, 2004.

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

[22] A. Tiwari, H. Ruess, H. Saidi, and N. Shankar. A Technique for Invariant Generation. In *TACAS 2001*, volume 2031 of *LNCS*, pages 113–127, 2001.

[23] S. Wolfram. *The Mathematica Book, 3rd ed.* 1996.