Generation of Loop Invariants in Theorema by Combinatorial and Algebraic Methods

Laura Ildikó Kovács, Tudor Jebelean[1] [2]
Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
Institute e-Austria Timişoara, Romania

Abstract. When generating verification conditions for a program, one is faced with one major task, namely with the situation when some additional assertions are needed (e.g. loop invariants). These assertions have the property that either they are invariant during execution of the program, or they depend on some other invariant properties. Therefore, automated formal verification is sensitive to the automated generation of invariants. In this paper, we present an alternative to expecting programmers to fully annotate code with invariants, namely a method for automatic generation of invariants from the program itself, using combinatorial algorithms and equational elimination, as well as an ongoing research work with some results based on applications of polynomial ideal theory. The implementation and the verification process is done in a prototype verification condition generator for imperative programs, which is part of the *Theorema* system, a computer aided mathematical assistant that offers automated reasoning and computer algebra facilities for the working mathematician. The verification conditions for programs containing loops are generated fully automatically, in a form which can be immediately used by the automatic provers of *Theorema* in order to check whether they hold. We illustrate the effectiveness of our method by few examples.

1 Introduction

Computing invariants is the key issue in verifying a certain program, in particular for programs that contain combinations of loops. This interesting problem has a long research history, starting from the late sixties and early seventies, with the outstanding work of Floyd–Hoare–Dijkstra [9, 12, 6]. By their research, an inductive assertion method was introduced for program verification, using the so–called correctness formulas $\{Precondition\}\ statement\ \{Postcondition\}$ (which meaning is that after the execution of the statement, its precondition applies the postcondition), combined with loop invariants and termination terms. Since invariants can protect a programmer from making changes that violate assumptions upon which the program correctness depends, the absence of explicit invariants in programs makes it easy for the programmers to introduce errors when changing the program–code. The main importance of the automatic invariant generation for the code–analysis and verification of programs is that invariant assertions can be used directly to establish properties of the respective program, or can be used indirectly to obtain lemmas for proving other relevant properties. Therefore, our purpose is to develop a method that automatically generates invariants.

In this paper we present our practical approach to program verification using *backward propagation*, most specifically verification of while loops, in the frame of the *Theorema* (`www.theorema.org`) system.

Our method uses powerful algorithms from combinatorics, namely algorithms for solving recursive equations. The main idea is to detect the recursive equations from the body of a loop. Actually, these equations will contain a major part of the invariance properties of changing variables during the execution of the loop. Solving these recursive equations by the *Gosper–Zeilberger algorithm* [22, 24], one can generate a closed–form that will be embedded in the invariant of the loop. A further approach for solving higher–order recurrences is the technique of *generating functions* [25].

Nevertheless, only using recurrence solvers, one cannot detect all the invariance properties in general. Some properties, such as inequations, non–linear constraints, etc., might be also necessary for the verification process.

[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 Timisoara project.
[2]This work has been partially supported by the "Spezialforschungsbereich for Numerical and Symbolic Scientific Computing" (SFB F013) at the University of Linz and the european union "CALCULEMUS Project" (HPRN-CT-2000-00102).

For this purpose, one can analyze the postcondition of a loop to extract relevant information. Our ongoing work in this direction is to generate non–linear polynomial loop–invariants using Gröbner bases [3, 4], reducing this way the invariant generation method to a non-linear constraint solving problem. Then, by various constraint-solving techniques [26, 5], one can obtain the invariance properties. Thus, the main idea is to compute a polynomial ideal that represents the weakest precondition for the validity of the given polynomial expression at a certain point of the program. This computation can be done effectively by Hilbert's Basis Theorem and Buchberger's Algorithm [28].

This paper is organized as follows: in Section 2 we give a brief introduction of the working environment, *Theorema*, followed by a short presentation of our verification tool, *Verification Condition Generator (VCG)*, in Section 3. Then, in Section 4, we present the invariant generation methods by combinatorial algorithms. The novel method of polynomial ideal theory's application for invariant generation is discussed in Section 5. Finally, in Section 6, we conclude with some ideas for future work.

2 The Working Environment: *Theorema*

The *Theorema* group is active since 1994 in the area of computer aided mathematics, with main emphasis on developing automated reasoning. It is the *Theorema* system (`www.theorema.org`), an integrated environment for mathematical explorations [8]. In particular, the *Theorema* system offers support for computing, proving and solving mathematical expressions using specified knowledge bases, by applying several simplifiers, solvers and provers, which imitate the style used by human when proving mathematical statements. The *Theorema* system tries to combine proving, computing, and solving, use of computer algebra, special sequent calculus, domain specific provers, induction, use of meta–variables, etc.

Moreover, *Theorema* offers the possibility of composing, structuring and manipulating arbitrary complex mathematical texts consisting of formal mathematical expressions together with structural information like labels or keywords such as "Definition, "Theorem, "Proposition, "Algorithm, etc.

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 an imperative language with interpreter and verifier, allowing program verification for imperative programs by generating and proving verification conditions depending on the program text [14].

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* [27], thus it has access to a wealth of powerful computing and solving algorithms.

Our approach to program verification is based on the so–called *weakest precondition strategy*, using Hoare Logic's correctness triples $\{P\}S\{Q\}$ [12, 17] (S is a program, P and Q are the precondition and the postcondition of the program, respectively). By using this strategy, one starts backwards from the postcondition and generates at each statement the weakest logical formula which is necessary for the postcondition to hold (some additional verification conditions may be generated on the way – e.g. for While–loops).

We improved the verification condition generator and the interpreter implemented in *Theorema* in [14], by a more sophisticated handling of loops by using algebraic methods for the analysis of programs in order to find the necessary loop invariants. This is of course limited to programs which operate over certain domains (e.g. numbers), but the invariant generation is completely automatic, and these type of programs are very interesting in practice. Current attempts for solving these problems are based on a logical approach (see e. g.[10] or [11] Chapt. 16 for some heuristics), which is much more difficult, although more general.

Our latest contribution and a work in progress is the integration of algebraic concepts and methods, such as Noetherian rings, confluent reductions, ideals, Gröbner bases, to generate automatically linear and non–linear polynomial loop invariants.

Our approach is practical and experimental. Of course there are (and have been) many systems which solve (partially) the verification problem (see e.g. [2, 1], the PVS Specification and Verification System `http://pvs.csl.sri.com/` or the Sunrise verification condition generator – `http://www.cis.upenn.edu/`). The purpose of our work is to have a practical system for experiments, which, in conjunction with the rest of the *Theorema* system allows us to examine test cases and to obtain more insight into the problem.

Programs written in functional style can be expressed directly in the *Theorema* language, thus the "compilation" step (and its possible errors) is avoided. However, for users which are more comfortable with the imperative style, *Theorema* features a procedural language based on a practically oriented version of the theoretical frame of Hoare–Logic, namely on the Weakest Precondition Strategy ([14]), with readable arguments for the correctness of programs, as well as with useful hints for debugging. The user interface has few simple and intuitive commands ($Program$, $Specification$, VCG, $Execute$). The programs are considered as procedures, without return values and with input, output and/or transient parameters (input parameters are specified by $\uparrow$, output parameters by $\downarrow$ and transient parameters by $\updownarrow$). The source code of a program contains a sequence of the following constructs [14, 21]:

- assignments (separated by ;, and may contain also function calls);
- conditional statements: $IF[cond, THEN - branch, ELSE - branch]$
- WHILE loops: $WHILE[cond, body, Invariant, TerminationTerm]$
- FOR loops: $FOR[counter,\ lowerBound,\ upperBound,\ step, body, Invariant]$;
- procedure calls

In this imperative language, for the WHILE and FOR statements we allow additional arguments, namely the Invariant and TerminationTerm in the case of WHILE loop, and Invariant in the case of FOR loop. These optional arguments are relevant in the verification process of our program.

The $VerificationConditionGenerator$ (VCG) takes an annotated program with pre– and postcondition (i.e. the program's specification) and produces as output a *Theorema*–`Lemma` containing a collection of formulas, i.e. the verification conditions. Thus, the verification condition generator is a translator based on a list of inference rules. It is recursive on the structure of the code and works back–to–front statement by statement. Internally, it repeatedly modifies the postcondition using a predicate transformer such that at the end the result is a list of verification conditions in the *Theorema* synthax.

Subsequently, the generated formulae can be used by the *Theorema* system, for instance, one can call a *Theorema* prover in order to check whether they hold.

3 Generation of Loop Invariants

Verification of correctness of loops needs additional information, so-called annotations. In the case of FOR loops these annotations consist in the invariants only, but in the case of $WHILE$ loops, beside the invariant, another annotation is a termination term necessary for proving termination [16].

In most verification systems, these annotations are given by the user.

Our purpose is to generate these annotations in *Theorema*, in order to prove program correctness. It is generally agreed [10] that finding automatically such annotations is in general very difficult. However, in most of the practical situations finding the expression – or at least giving some useful hints – is quite feasible. For practical applications this may be very helpful to the user.

3.1 Solving First Order Recurrences

Analyzing the code of a loop, we can generate recursive equations that contain those variables which are modified during the execution of the loop. These variables are called *critical* variables.

Our main idea is to generate these (linear) recursive equations, and then to eliminate the variable which refers to the current iteration of the loop. The resulting equation(s) contain the information that have to embedded in the invariant of the loop.

For illustration, consider the "Division" program of two natural numbers, where the user does not specify a loop–invariant:

$$
\begin{array}{l}
Specification[\text{"}Division\text{"},\ Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo], \\
\quad Pre \rightarrow ((x \geq 0) \wedge (y > 0)), \\
\quad Post \rightarrow ((quo * y + rem = x) \wedge (0 \leq rem < y))]
\end{array}
$$

$$
\begin{array}{l}
Program[\text{"}Division\text{"},\ Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo], \\
\quad quo := 0; \\
\quad rem := x; \\
\quad WHILE[y \leq rem, \\
\quad\quad rem := rem - y; \\
\quad\quad quo := quo + 1 \\
\quad\quad Specification \rightarrow Specification[\text{"}Division\text{"}]]
\end{array}
$$

The automated generation of the invariant proceeds as follows:
From the body of the loop, we obtain the following recursive equations:

$$
\begin{array}{l}
quo_0 = 0;\ \ quo_{k+1} - quo_k = 1 \\
rem_0 = x;\ rem_{k+1} - rem_k = -y.
\end{array}
$$

where quo and rem are the critical variables and k is a new variable representing the current iteration of the loop.

These recursive equations are solved by the Gosper-Zeilberger algorithm (see e.g. [15, 22]). Namely, we use the Paule-Schorn implementation in *Mathematica* [24] which is already embedded in the *Theorema* system, in order to produce a closed–form for the expressions of quo_k and rem_k.

Hence, we obtain the explicit equations:

$$
\begin{array}{rcl}
quo_k & = & 0 + k \\
rem_k & = & x - k * y
\end{array}
$$

From these equations we eliminate k by calling the appropriate routine from *Mathematica*, and we obtain the equation:

$$rem = x - quo * y.$$

Some additional information which should be embedded in the loop invariant, namely conditions on the output parameters is extracted from the condition and the postcondition of the loop in the following manner:

- from the body of the loop we determine the recursive equations and the critical variables;
- using this list of critical variables, from the postcondition of the loop we extract those logical formulae that contain at least one of the critical variables;
- in the case of the WHILE loops, comparing these extracted logical formulae with the condition of the loop, we keep only those formulae which are not the same as the loop condition or they are not contradictory to the loop condition. These finally extracted formulae will be those which, together with the equations generated by the Gosper Algorithm, will form the invariant of our loop.

Hence, the complete invariant for this example will be:

$$Invariant \equiv (quo * y + rem = x) \wedge 0 \leq rem$$

In the case of the WHILE loop, one is also interested to be able to prove termination, i.e. to have an automatic generation of Termination Term. Knowing that the TerminationTerm must be positive [12], we transform the given loop-condition Φ using specific heuristics (algebraic manipulations) until we obtain a term T such that $t \geq 0 \ \Leftrightarrow\ \Phi$.

In the example above, the TerminationTerm will be: $rem - y$

In the case of FOR loop, the generation of the loop invariant is done in the same manner, but we use additionally the explicit equation for the counter of the For loop:

$$counter_k := counter_0 + k * step.$$

(where $counter$ is the counter of the loop and $step$ is the iteration step, by default it is 1).

In the above example, we worked with recursive equations whose behavior do not depend on other equations. For the case when a critical variable of a recursive statement is influenced also by some other recursive statements form the loop's body is problem, our method is still applicable. In this situation, first we work with that equation (critical variable) which do not depend on other equations, generate the closed–form of it, and then substitute this expression in the other recursive equations, each of them becoming in this way of one recurrence order less. Proceeding in a following manner for the other recursive equations, we will solve again first–order recursive equations by the Gosper–algorithm. This strategy works only if we do not have mutual recurrence(s) in the loop body.

3.2 Mutual Recurrences

In most cases, beside first–order recurrences, a program–code contains higher order recurrences. An interesting case of this type of recurrence is the case of *mutual* reccurence, where, for instance, two equations are mutually depending on each other. For solving such a problem, we use the technique of *generating functions* [25] from combinatorics ([25] is an original work about P-finite and D-finite sequences, namely about closure properties).

Consider the example of the well-known program for computing the Fibonacci numbers, which has its specification and source code, as follows:

$$\begin{array}{l} Specification[\text{"}Fibonacci\text{"},\ FibonacciProcSpec[\downarrow n, \uparrow F], \\ \quad Pre \rightarrow (n \geq 0), \\ \quad Post \rightarrow (F = Fib_exp[n])] \\ Program[\text{"}Fibonacci\text{"},\ FibonacciProc[\downarrow n, \uparrow F], \\ \quad Module[\{H,\ i\}, \\ \quad\quad i\ :=\ n; \\ \quad\quad F\ :=\ 1; \\ \quad\quad H\ :=\ 1; \\ \quad\quad WHILE[i\ >\ 1, \\ \quad\quad\quad H\ :=\ H\ +\ F; \\ \quad\quad\quad F\ :=\ H\ -\ F; \\ \quad\quad\quad i\ :=\ i\ -\ 1]] \end{array}$$

Note: $FibExp[n]$ denotes the term: $F_k\ =\ \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi}$ is its conjugate.

We can merely set up the recurrence:

$$H_n\ =\ H_{n-1}\ +\ F_{n-1}\ (n \geq 1),\ H_1 = 1$$

$$F_n\ =\ H_n\ +\ F_{n-1}\ (n \geq 1),\ F_1 = 1$$

For solving this problem, we apply the technique of *generating functions*, namely, given a sequence (g_n) that satisfies a given recurrence (in our example, the sequences are (F_n) and (H_n)), we seek a closed form for g_n in term of n. For our computations we have used Mallinger's Mathematica package GeneratingFunctions [18], which was developed in the Combinatoric Group of RISC. In order to apply this package, first we rewrite the equations of express H_n and F_n in such a way that they are valid for all integers n, assuming that $H_0\ =\ H_{negative}\ =\ 0$ and $F_0\ =\ F_{negative}\ =\ 0$.

Hence we obtain:

$$H_n \;=\; H_{n-1} \;+\; F_{n-1} \;+\; [n=1]\,(n \in Z)$$
$$F_n \;=\; H_n \;+\; F_{n-1}\,(n \in Z)$$

(where the meaning of $[n=1]$ is that it adds 1 (i.e. H_1) when $n=1$, and it makes no change when $n \neq 1$.

Then, by applying Mallinger's implementation of the *generating functions* technique (and solving a system of two equations with two unknowns), we obtain the generating functions and also the closed form of their coefficients, namely:

$$F(z) \;=\; \frac{z}{1-z-z^2} \qquad F_n \;=\; \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$
$$H(z) \;=\; \frac{z(1+z)}{1-z-z^2} \qquad H_n \;=\; \frac{\phi^{n+1} - \hat{\phi}^{n+1}}{\sqrt{5}}$$

Hence, at the k^{th} iteration, we have as invariance properties:

$$F_k \;=\; \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}}$$
$$H_k \;=\; \frac{\phi^{k+1} - \hat{\phi}^{k+1}}{\sqrt{5}}$$

From the third recurrence equation of the loop, i.e. $i_{k+1} = i_k - 1, i_0 = n$, by the Gosper algorithm, we obtain the closed form: $i_k = n - (k-1)$.

In the following steps, we proceed as in the previous section, namely we eliminate the loop's counter variable k from the three equations, thus we obtain:

$$\left(F \;=\; \frac{\phi^{n-i+1} - \hat{\phi}^{n-i+1}}{\sqrt{5}}\right) \bigwedge \left(H \;=\; \frac{\phi^{n-i+2} - \hat{\phi}^{n-i+2}}{\sqrt{5}}\right)$$

One notes that these are exactly the expressions of the Fibonacci numbers [20].

Summarizing, the technique of *generating functions* turns out to be practical and effective for solving higher order recurrences, thus for generating invariance properties for more complex programs.

The theoretical details of this methos are described in [23].

Conclusions and Further 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 algebraic computations (summation methods, variable elimination) is a promising approach to analysis of loops.

The presented combinatorial approach is effective and practical, but limited to the type of statements that are present in the body of the loop. In the case when the body of a loop contains only assignments, the combinatorial methods are applicable. A further work of us is to consider loops that contain also conditional statements ($IF - THEN - ELSE$). In this case, we generate by combinatorial methods the closed form of the recurrences for each branch. Then, the obtained expressions have to be combined in such a way that they describe the invariance property of the loop. An efficient method for solving this problem is the application of Gröbner Bases[3, 4], namely to generate the Gröbner bases of the already obtained closed–forms. Doing so, we rely on the work of E. Rodriguez–Carbonell [7], Müller–Olm and Seidl[13] or Zohar Manna[19], namely a method built up polynomial ideal theory and properties. By their approach, the invariant generation problem will be translated in this way to a (linear or non–linear) constraint solving problem (where the contraints describe properties of the coefficients of the polynomial loop invariant).

Another necessary continuation of our work is the analysis of programs containing recursive calls. We are currently investigating the theoretical framework and we are designing the methods for extracting the verification conditions of this type of programs.

References

[1] ***. *PStndfor Pascal Verifier - User Manual*. Computer Science Department, Stanford University, 1979.

[2] J. Barnes. *High Integrity Software - The Spark Approach to Safety and Security*. Addison-Wesley, 2003.

[3] B. Buchberger. *Ein algorithmisches Kriterium fr die Lsbarkeit eines algebraischen Gleichungssystems (An Algorithmical Criterion for the Solvability of Algebraic Systems of Equations)*. Aequationes mathematicae 4/3, pp. 374-383. (English translation in: B. Buchberger, F. Winkler (eds.), Gröbner Bases and Applications, Proceedings of the International Conference "33 Years of Gröbner Bases", 1998, RISC, Austria, London Mathematical Society Lecture Note Series, Vol. 251, Cambridge University Press, 1998, pp. 535 -545.), 1970.

[4] B. Buchberger. *Introduction to Gröbner Bases*. In: Gröbner Bases and Applications (B. Buchberger, F. Winkler, eds.), London Mathematical Society Lecture Notes Series 251, Cambridge University Press, pp.3-31., 1998.

[5] C.Ballarin and M.Kauers. *Solving Parametric Linear Systems: an Experiment with Constraint Algebraic Programming*. In *Eighth Rhine Workshop on Computer Algebra*, 2002. pp. 101–114.

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

[7] E.Rodriguez-Carbonell; D.Kapur. *Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations*. In *Proc. of the International Symphosium on Symbolic and Algebraic Computation (ISSAC 04)*, 2004. July 4-7, University of Cantabria, Santander, Spain.

[8] B. Buchberger et al. *The Theorema Project: A Progress Report*. In M. Kerber and M. Kohlhase, editors, *Calculemus 2000: Integration of Symbolic Computation and Mechanized Reasoning*. A. K. Peters, Natick, Massatchussets, 2000.

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

[10] G. Futschek. *Programmentwicklung und Verifikation*. Springer, 1989.

[11] D. Gries. *The Science of Programming*. Springer, 1981.

[12] C. A. R. Hoare. *An axiomatic basis for computer programming*. *Comm. ACM*, 12, 1969.

[13] M.Müller-Olm; H.Seidl. *Polynomial Constants are Decidable*. In *Static Analysis Symposium (SAS 2002), vol.2477 of LNCS*, 2002. pp. 4-19.

[14] M. Kirchner. *Program verification with the mathematical software system Theorema*. Technical Report 99-16, RISC-Linz, Austria, 1999. PhD Thesis.

[15] D. E. Knuth. *The Art of Computer Programming, volume 2 / Seminumerical Algorithms*. Addison-Wesley, 2nd edition, 1969.

[16] L. Kovács. *Program Verification using Hoare Logic*. In *Computer Aided Verification of Information Systemsm Romanian-Austrian Workshop*, 2003. Timisoara, Romania, February 2003.

[17] B. Buchberger; F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics. (German.)*. Springer, Berlin, Heidelberg, New York, 315 pages, 2nd edition, 1981. (First Edition 1980).

[18] C. Mallinger. Algorithmic manipulations and transformations of univariate holonomic functions and sequences. Master's thesis, RISC, J. Kepler University, Linz, August 1996.

[19] S.Sankaranaryanan; B.S.Henry; Z. Manna. *Non-Linear Loop Invariant Generation using Gröbner Bases*. In *ACM Principles of Programming Languages (POPL'04)*, 2004. January 14-16, Venice, Italy.

[20] R.L. Graham; D.E. Knuth; O. Patashnik. *Concrete Mathematics, 2nd ed.* Addison-Wesley Publishing Company, 1989. pg. 306-330.

[21] L. Kovács; N. Popov. *Procedural Program Verification in Theorema*. In *Omega-Theorema Workshop*, May 2003. Hagenberg, Austria.

[22] R.W.Gosper. *Decision procedures for indefinite hypergeometric summation*. 75:40–42, 1978.

[23] B. Salvy and P. Zimmermann. Gfun: A package for the manipulation of generating and holonomic functions in one variable. *ACM Trans. Math. Software*, 20:163–177, 1994.

[24] P. Paule; M. Schorn. *A Mathematica version of Zeilberger's algorithm for proving binomial coefficient identities*. 20(5-6):673–698, 1995.

[25] R.P. Stanley. Differentiably finite power series. 1:175–188, 1980.

[26] V.Weispfenning. *Quantifier Elimination for real algebra – the quadratic case and beyond*. In *Applied Algebra and Error-Correcting Codes (AAECC) 8*, 1997. pp. 85-101.

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

[28] W.Windsteiger and B.Buchberger. *A Library for Computing Gröbner Bases based on Saclib*. Technical report, 1993. Tech.Report, RISC-Linz.

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