Institute e-Austria in Timisoara

IeAT Report Series

Generation of Invariants in Theorema

Laura Ildiko KOVACS
Tudor JEBELEAN

2001

Generation of Invariants in Theorema

Laura Ildikó Kovács, Tudor Jebelean[1]
Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
Institute e-Austria Timişoara, Romania
{lkovacs, jebelean}@risc.uni-linz.ac.at

Abstract. Explicitly stated program invariants can help programmers by identifying program properties that must be preserved when modifying code. In practice, in most of the cases, however, these invariants are usually implicit. In this paper we present an alternative to expecting programmers to fully annotate code with invariants, namely a method for automatically generation of invariants from the program itself, using an implementation of a prototype verification condition generator for imperative programs. The generator is part of the *Theorema* system, a computer aided mathematical assistant which offers automated reasoning and computer algebra facilities. We use Hoare Logic and the weakest precondition strategy, and we propose a novel method for analyzing loop constructs by aid of algebraic computations: combinatorial summation and equational elimination. 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.

MSC(2000):33F10; 68W30; 68N30; 68Q60

1 Introduction

Invariants play a central role in program development. They 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 while making changes. Therefore, our purpose is to develop a method that automatically infers invariants, without any additional interactions with users. In this paper we present our practical approach to program verification, most specifically verification of while loops, in the frame of the *Theorema* system.

The *Theorema* group is active since 1994 years in the area of computer aided mathematics, with main emphasis on automated reasoning, and it is building the *Theorema* system (`www.theorema.org`), an integrated environment for mathematical explorations [5]. In particular, the *Theorema* system offers support for computing, proving and solving mathemati-

[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. The *Theorema* system is supported by FWF (Austrian National Science Foundation) – SFB project P1302.

cal expressions using specified knowledge bases, by applying several simplifiers, solvers and provers in natural style, which imitate the heuristics used by human provers (combining 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 informations 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 additionally an imperative language with interpretor and verifier, allowing program verification for imperative programs by generating and proving verification conditions depending on the program syntax [9].

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

Our approach to program verification is the so-called weakest precondition strategy, based on Hoare Logic's correctness triples

$$\{P\}S\{Q\}$$

[8, 12], where S is a program (sequence of statements) and P and Q are, respectively, the precondition and the postcondition of the program. 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 conditions may be generated on the way – e.g. for While loops).

We improved the verification condition generator and the interpretor implemented in Theorema by [9], by a more sophisticated handling of loops. Doing this, we have used algebraic methods for the analysis of algorithms in order to find the necessary loop invariants. This is of course limited to programs which operate over certain domains (e.g. numbers), but they are completely automatic, and these type of programs are very interesting in practice. Current attempts at solving these problems are based on a logical approach (see e. g.[6] or [7] Chapt. 16 for some heuristics), which is much more difficult, although more general.

Our approach is practical and experimental. Of course there are (and have been) many systems which solve [partially] this problem (see e.g. [3, 1], the PVS Specification and Verification System – http://pvs.csl.sri.com/, 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.

Basic Language Constructs in *Theorema*

Specifications, invariants, and conjectures can be expressed in the logical language of Theorema, which is practically identical to the mathematical language used by mathematicians and engineers: higher-order predicate logic, including the two-dimensional notations. Moreover, Theorema allows the introduction of ones own notations and symbols and even creating new graphical symbols [14]. The system provides few simple and intuitive commands for creating and manipulating mathematical knowledge, including its organization into mathematical theories, as well as proving, computing, and solving with respect to a certain theory. During the last few years, such domain specific knowledge bases have been developed in the system.

For handling imperative programs, the system provides the commands Program, Specification, and Execute. We illustrate these and the syntax of the imperative language through a simple example:

$$
\begin{aligned}
&Specification[\text{"}Division\text{"},\ Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo],\\
&\quad Pre \rightarrow ((x \geq 0) \wedge (y > 0)),\\
&\quad Post \rightarrow ((quo * y + rem = x) \wedge (0 \leq rem < y))]\\
&Program[\text{"}Division\text{"},\ Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo],\\
&\quad quo := 0;\\
&\quad rem := x;\\
&\quad WHILE[y \leq rem,\\
&\qquad rem := rem - y;\\
&\qquad quo := quo + 1,\\
&\qquad Invariant \rightarrow ((quo * y + rem = x) \wedge (0 \leq rem)),\\
&\qquad TerminationTerm \rightarrow rem],\\
&\qquad Specification \rightarrow Specification[\text{"}Division\text{"}]]
\end{aligned}
$$

Both the specification and the description of a routine indicate the parameters and their nature (input $\downarrow$, output $\uparrow$, input-output $\updownarrow$). The argument $Specification$ of $Program$ is optional, as are the arguments $Invariant$ and $TerminationTerm$ of $WHILE$. In the program text we differentiate between assignment ":=" and logical equality "=". 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.

After entering the previous commands into the Theorema system, one can enter:

$$Execute[Div[20,\ 3,\ rem,\ quo]],$$

which will have the effect of assigning to rem and quo the appropriate values 2 and 6.

Generation of the Verification Conditions

Continuing the example above, one may introduce the command:

$$VCG[Program[\text{"}Division\text{"}]]$$

and then one obtains:

$$
\begin{aligned}
&Lemma(Division) : for\ any :\ x,\ y,\ rem,\ quo \\
&(WHILE.Inv + Term) \\
&((quo * y + rem = x) \wedge 0 \leq rem) \wedge y \leq rem \wedge (rem = T1) \\
&\Rightarrow \left(\left((quo + 1) * y + (rem - y) = x \right) \wedge 0 \leq (rem - y) \right) \wedge (rem - y) < T1 \\
&(WHILE.Final) \\
&((quo * y + rem = x) \wedge 0 \leq rem) \wedge (y \not\leq rem) \Rightarrow \\
&(quo * y + rem = x) \wedge 0 \leq rem \wedge rem < y \\
&(WHILE.Term) \left((quo * y + rem = x) \wedge 0 \leq rem \right) \wedge y \leq rem \Rightarrow rem \geq 0 \\
&(Init) x \geq 0 \wedge y > 0 \Rightarrow (0 * y + x = x) \wedge 0 \leq x
\end{aligned}
$$

$Division$ is the label of the lemma, and $WHILE.Term$, etc. are the labels of the individual formulae. Using this labels one can further make reference to these formulae, for instance one can call a Theorema prover in order to check whether they hold:

$$Prove[Lemma[" Division"]]$$

The program VCG generates the verification conditions using Hoare Logic and the weakest precondition strategy in the classical way [8, 12, 17].

Generation of Loop Invariants

Verification of correctness of loops needs additional information, so-called annotations. In the case of For loops these annotations are only the invariants; in the case of While loops, beside the invariant, another necessary annotation is a termination term for proving total correctness [11].

This annotations, so far, were considered to be given by the user.

Our purpose is to generate this annotations in Theorema, in order to prove program correctness. It is generally agreed [6] that finding automatically such annotations is in general impractical – thus most systems will just ask the user for the appropriate expression. 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.

A "hidden" problem in the theoretical treatment of the invariant is the fact that in most practical situations it will also contain information about other parts of the program, which is not related to the respective loop. This may make the task of finding the invariant more difficult, however it may be relatively easy to separate the specific information from the non-specific one by an analysis of the free variables and other characteristics which are easy to detect automatically. This could also provide useful hints to the user.

Our current goal of is to develop a method that provides the possibility of proving automatically correctness of programs which have loops, without asking the user to give necessary annotations.

Other Approaches for Invariant Generation

There are two main approaches of invariant generation, 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. The accuracy of the inferred invariant depends in part on the quality and completeness of the test cases; additional test cases might provide new data from which more accurate invariants can be inferred. Such a system is DAIKON [15], a prototype invariant detector that instruments the source program to trace the variables of interest and runs the instrumented program over a set of test cases in order to infer invariants over the instrumented variables and over derived variables that are not manifest in the original program. Another system, ANDREW [2], generates invariants by comparing actual behavior of the program against of a user-defined model and indicating divergences between the two.

The static approach of invariant generation operates on the program text, not on the test runs, therefore has the advantage that the reported properties are true for any program run. Theoretically, they can detect sound invariants. Some formal proof systems generate intermediate assertions for help in proving a given formula by propagating known invariants forward or backward in the program [21, 4]. ReForm [20] semi-automatically transforms, by provable correctness steps, a program into a specification. The Maintainer's Assistant [13] uses program transformation techniques to prove equivalence of two programs (if they can be transformed to the same specification or to one another).

In our work, in the Theorema system, we apply the statical approach.

Generating Invariants of For and While Loops in Theorema

Annotating a program is often non–trivial and needs a good understanding of how the algorithm works. The idea of the invariants is mostly identical to the basic design idea. That is why programming is more effective if one thinks about the invariant before coding a loop and also gives heuristics for developing invariants.

Analyzing the code of loops, we can generate recursive equations that contain those terms which occur in the condition of the loops.

Our main idea, is to generate these (linear) recursive equations, and by eliminating the variable which refers to the current step of the loop, we would obtain the necessary informations that have to embedded in the invariant of the loops.

Thus, we are developing a method based on recurrence equation solvers that provides the possibility of proving automatically correctness of programs which have loops.

Generating Invariant from Dependent or Independent Recursive Equations

Consider the "Division" program presented in section 1. If the user does not specify the loop invariant, then we find it as follows:
From the body of the loop, we obtain the following recursive equations:

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

These recursive equations are solved by the Gosper-Zeilberger algorithm (see e.g. [10, 18]). Namely, we use the Paule-Schorn [19] implementation in Mathematica which is already embedded in the Theorema system, namely the Gosper function, in order to produce a closed-form for sums. In our example, we use: $Gosper[1, i, 0, k]$ and $Gosper[-y, i, 0, k]$.

Hence, we obtain the explicit equations:

$$quo_0 := 0;\ \ quo_k := quo_0 + k$$
$$rem_0 := x;\ rem_k := rem_0 - k * y$$

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

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

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.

Hence, for the considered example, the produced verification conditions – using the generated loop invariant – are exactly the same as the ones presented in section 2.

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:

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

Note also that by using the explicit expressions of the recursively modified variables, it is relatively easy to analyze the termination of the loop. For instance, in the division example, checking whether the loop terminates reduces to solving the inequality:

$$y > rem_k$$

that is:

$$y > (x - ky)$$

which gives:

$$k \geq \lfloor x/y \rfloor.$$

This shows that the loop terminates, but also gives the number of iterations.

In the above example, we worked with independent recursive equations. However, this situation does not happen in practice very often, at least one of the detected equations depends on the other equations. For this problem, our method is still applicable, by taking into consideration the already generated explicit equations of the recursive terms that appear in the equations.

Generating Invariant from Mutual Recursive Equations

A more interesting problem shows up in the case of mutual recursivity, where, for instance, two equations are mutually depending on each other. For solving such a problem, we use the technique of generating functions from combinatorics.

Consider the example of $3 \times n$ domino tilings [16]. If we want to know only the total number of ways, U_n, to cover a $3 \times n$ rectangle with dominoes, without breaking this number down into vertical dominoes versus horizontal dominoes, we need not go into many details. We can merely set up the recurrence:

$$U_n = 2V_{n-1} + U_{n-2}(n \geq 2), U_0 = 1, U_1 = 0$$

$$V_n = U_{n-1} + V_{n-2}(n \geq 2), V_0 = 0, V_1 = 1$$

where V_n is the number of ways to cover a $3 \times n$ rectangle-minus-corner, using $(3n-1)/2$ dominoes. Solving this problem, we process in the following steps:
We have:

$$U_n = 2V_{n-1} + U_{n-2} + U_0, V_n = U_{n-1} + V_{n-2}(n \geq 0).$$

Hence, we can write:

$$U(z) = 2zV(z) + z^2U(z) + 1, V(z) = zU(z) + z^2V(z).$$

Now, we have to solve two equations in two unknowns; but these are easy, since the second yields $V(z) = zU(z)/(1-z^2)$. Thus we find, by using the Generating Functions package of the Combinatorics group from RISC :

$$U(z) = \frac{1-z^2}{1-4z^2+z^4}; V(z) = \frac{z}{1-4z^2+z^4}$$

which can be embedded in the invariant.

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.

Another necessary continuation of this 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.H. Andrews. Testing using log file analysis: tools, methods and issues. In 13th Annual International Conference on Automated Software Engineering (ASE'98), October 1998. Honolulu, Hawaii.

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

[4] S.M.German; B.Wegbreit. A synthesizer of inductive assertions . In IEEE Transactions on Software Engineering, March 1975. 1(1):68-75.

[5] 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.

[6] G. Futschek. Programmentwicklung und Verifikation. Springer, 1989.

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

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

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

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

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

[12] 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).

[13] M. Ward; F.W. Callis; M. Munro. The maintainer's assistant. In Proceedings of the International Conference on Software Maintenance 1989, pages 307–315, 1989. Miami, Florida.

[14] K. Nakagawa. Logico-Grafic Symbols in Theorema. In LMCS'02 (Logic, Mathematics, and Computer Science: Interactions, 2002. RISC-Linz technical report 02-60.

[15] M.D. Ernst; J. Cockrell; W.G.Griswold; D. Notkin. Dynamically discovering likely program invariants to support program evaluation. Technical report, 2000. April 24.

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

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

[18] R.W.Gosper. decision procedures for indefinite hypergeometric summation. 75:40–42, 1978.

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

[20] M.P. Ward. Program analysis by formal transformation. 39:598–618, 1996.

[21] B. Wegbreit. The synthesis of loop predicates. In Communication of the ACM, February 1974. 17(2):102-112.

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