Practical Aspects of Imperative Program Verification using Theorema

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

Approaching the problem of imperative program verification from a practical point of view has certain implications concerning [4]: 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) [1] has certain capabilities which make it appropriate for such a practical approach: the logic language of the system is higher-order predicate logic expressed in natural style; the procedural language is simple and intuitive, yet sufficiently expressive and fully integrated in the logical frame of the system; the language and the style of the proofs are natural, similar to those used by humans; and finally the proving power of Theorema is enhanced by using specific provers for special domains, which are integrated with sophisticated mathematical algorithms.

Our approach for imperative program verification in Theorema is based on the so-called Hoare–Logic, which verification process is characterized by (for a tutorial introduction see also [7, 8]):

- an imperative program
- and a logical specification

*The program verification project is supported by BMBWK (Austrian Ministry of Education, Science and Culture) in the frame of the e-Austria Timişoara project, whih is supported additionally by BMWA (Austrian Ministry of Economy and Work) and by MEC (Romanian Ministry of Education and Research). The Theorema system is supported by FWF (Austrian National Science Foundation)–SFB project P1302.

- and (usually) some logical invariants
- are fed into a verification system,
- which generates a conjecture,
- which is fed into a prover,
- and one obtains the answer whether the program fulfills the specification.

The program is expressed in a programming language, the specification, the invariants, and the conjecture are expressed in a logic language, and the answer is Yes/Not or a proof [attempt] of the conjecture expressed in the above logical language and some proof language.

Programs written in functional style can be expressed directly into this language, thus the "compilation" step (and its possible errors) is avoided. However, for the 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 ([5]), with readable arguments for the correctness of programs, as well as with useful hints for debugging. This procedural language, called Verification Condition Generator (VCG), is composed of few simple and intuitive constructs:

- *Program*: for writing the source code of an imperative program. 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 $\updownarrow$);
- *Specification*: for writing the specification of a program;
- *Execute*: for executing a program. Some parts of program might have local variables which variables could appear is some other parts of the program, in other context. The execution of such a program should not be disturbed by previous values of the local variable, the local variable should be treated properly in the specific context. Also, previous usage of the output and input variables of a program should not effect the execution.

VCG which takes an annotated program with pre– and postcondition (i.e. specification) and produces as output a Theorema–`Lemma` with 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.

As a further development of our verification system, one of our current goal is to develop a method based on recurrence equation solvers that provides the possibility of proving automatically correctness of programs which have loops, without asking the user to give necessary annotations(i.e. invariants, termination terms). For proving the correctness of a `While` or `For` statement, one needs to have a logical invariant. Moreover, for proving the termination of a `While` loop, one also needs a termination term. It is generally agreed [3] that finding automatically such an invariant or term 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.

In our current work, the idea of the generation of 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.

Solving a given linear recurrence equation with polynomial coefficients – extracted from a While or For loop – we could obtain the invariant for the respective loop. To solve these recursive equations, we use the Gosper Algorithm, which is a suitable solution for indefinite hypergeometric summation in closed form summation problems [6]. In our examples, we deal with first order recursivity, therefor the problems that will occur could be solved by the implementation of the Gosper Algorithm (and its extension, namely the Zeilberger algorithm) in Mathematica. This implementation

is already embedded in the Theorema System (by Peter Paule and Markus Schorn [9]).

As future work, we would like to solve the treatment of transient variables. Usually transient parameters lead to the following problem: The postcondition has to relate to the initial values of the (modified) transient parameters. A solution would be to "copy" the initial values of the transient parameters into some new constants (so called ghost variables) which are not modified in the program code.

This solution could be applied for the execution of a problem which contains also transient variables, hence during the execution we will deal with two variables -an input and an output- generated for a transient variable. Both generated variables will be initialized with the value of the transient parameter and, at the end of the execution, the generated output variable will contain the result.

Another future goal would be the generation of the termination term of a While loop in order to be able to prove termination/non-termination of the loop. The idea is similar as the generation of invariants, from the obtained recursive equations, analyzing the condition of the loop, one might be able to collect the necessary informations for establishing the desired termination term.

References

[1] Bruno Buchberger. *The PCS Prover in Theorema*. 2001. pp. 469-478.

[2] Bruno Buchberger. *Practical and theoretical aspects of program verification*. In *Computer Aided Verification of Information Systemsm Romanian-Austrian Workshop*, 2003. Timisoara, Romania, 12 february 2003.

[3] Gerald Futschek. *Programmentwicklung und Verifikation*. 1989.

[4] Tudor Jebelean. *Imperative Program Verification with Theorema*. In *Computer Aided Verification of Information Systemsm Romanian-Austrian Workshop*, 2003. Timisoara, Romania, 12 february 2003.

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

[6] D.E. Knuth. *The Art of Computer Programming, volume 2 / Seminumerical Algorithms*. 1969.

[7] Bruno Buchberger; Franz Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics. (German.).* 1981. (First Edition 1980).

[8] Krzystof R. Apt; Ernst-Ruediger Olderog. *Verification of Sequential and Concurrent Programs.* 1997.

[9] Schorn M. Paule, P. *A Mathematica version of Zeilberger's algorithm for proving binomial coefficient identities - A description how to use it.* Technical Report 93-36, RISC-Linz Report Series, 1993.