# Verification of Imperative Programs in Theorema

Laura Ildikó Kovács, Nikolaj Popov, Tudor Jebelean[1]

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

**Abstract.** We present the design and the 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, but in addition we propose a novel method for analyzing loop constructs by aid of algebraic computations: combinatorial summation and equational elimination. The verification conditions and the termination term for programs containing loops and procedure calls 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.

## Introduction

Program verification (and verification of informational systems in general) will probably play an important role in the development of information technologies (IT), because, as IT products become more sophisticated, their reliability is more difficult to insure by artizanal means, and as IT products become more present in all aspects of human activity, their un-reliability has more dramatic negative effects.

In this paper we present our practical approach to program verification in the frame of the *Theorema* system.

The *Theorema* group is active since 10 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 [3]. In particular, the *Theorema* system offers several 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 metavariables, etc.).

[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.

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). However, the system also contains an analyzer and an interpretor for an imperative language containing the essential necessary constructs [7].

The *Theorema* system is particularly appropriate for program verification, because it allows the work with both logical formulae and with the description of algorithms in the same uniform frame, and also delivers the proofs in a natural language and using natural style inferences. Moreover, the system is implemented on top of the computer algebra system *Mathematica* [12], thus it has access to a wealth of powerful computing and solving algorithms.

The approach to program verification presented here is Hoare Logic [6, 9]. Program correctness is expressed by Hoare triples $\{P\}S\{Q\}$, where $S$ is a program and $P$ and $Q$ are assertions (logical formulae). The precondition $P$ has to be satisfied by the input values to the program (routine) and the postcondition $Q$ is satisfied by the input values and the result[s] of the program (output values). The program S is a finite sequence of statements. By using the so called "weakest precondition" 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 [7], by a more sophisticated handling of loops, and by extending the verification of function calls in order to handle more practical situations, including recursive calls. The main original contribution of our work is the use of algebraic algorithms for the analysis of algorithms: finding loop invariants, checking of termination, estimation of the time complexity. 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.[4] or [5] 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. [2, 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.

# 1 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 [10]. 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 the handling of 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:

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

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

Both the specification and the description of a routine indicate the 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 "=".

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.

## 2 Generation of the Verification Conditions

Continuing the example above, one may introduce the command:

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

and then one obtains:

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

*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:

$$Prove[Lemma[\text{“}Division'']]$$

The program *VCG* generates the verification conditions using Hoare Logic and the weakest precondition strategy in the classical way. For instance, the conditions for $WHILE$ are generated using the rule:
{P} `while C do S endwhile` {Q},
is correct if:
(Init) $\texttt{P} \Rightarrow \texttt{I}$
(WHILE.Term) $(\texttt{I} \wedge \texttt{C}) \Rightarrow \texttt{T} \geq 0$
(WHILE.Inv+Term) $\{\texttt{I} \wedge \texttt{C} \wedge (\texttt{T} = \texttt{T}_1)\}$ S $\{\texttt{I} \wedge (\texttt{T} < \texttt{T}_1)\}$
(WHILE.Final) $(\texttt{I} \wedge \neg\texttt{C}) \Rightarrow \texttt{Q}$

where I is a Loop-Invariant, T is a Termination Term and $T_1$ is a new variable.

Another interesting situation for program verification is the generation of verification conditions for *function calls*. Consider a simple example in *Theorema*

which uses the maximum of two numbers:

(*the specification of the function Max*)

$$Specification["Max", m = Max[\downarrow x, \downarrow y],$$
$$Pre \rightarrow (IsInteger[x] \wedge IsInteger[y]),$$
$$Post \rightarrow (m = x \wedge x \geq y) \bigvee (m = y \wedge y > x)]$$

(*the specification of the program and the source code*)

$$Specification["Calculus", Calc[\downarrow a, \downarrow b, \downarrow y, \updownarrow x],$$
$$Pre \rightarrow (IsInteger[a] \wedge IsInteger[b]),$$
$$Post \rightarrow \Big((x \geq (y + a)) \bigwedge (x \geq (y + b)\Big)]$$

$$Program["Calculus", Calc[\downarrow a, \downarrow b, \downarrow y, \updownarrow x],$$
$$x := y + Max[a, b]]$$

In a previous version of VCG, expressions containing function calls were not handled differently than other expressions in a program. Thus the function symbols occurring in the calls were simply inserted into verification conditions, as in the example:

```
VCG[Program["Calculus"],Specification["Calculus"]]
```

$$Lemma(Calculus):$$
$$forany: a, b, y, x$$
$$(Init)$$
$$IsInteger[a] \wedge IsInteger[b] \Rightarrow y + Max[a, b] \geq y + a \bigwedge y + Max[a, b] \geq y + b$$

An automatic prover will need additional information about the function Max (the specification of the function).

It may be better to use the following *verification rule for function calls*:

$$\{P_{X \leftarrow T,\ U \leftarrow A} \quad \bigwedge \quad \forall_{A,b}\Big(Q_{X \leftarrow T,\ U \leftarrow A,\ F(T,\,U) \leftarrow b} \Rightarrow V_{U' \leftarrow A,\ c \leftarrow b}\Big)\}$$

$$\texttt{c} = \texttt{F}(T, U')$$

$$\{\texttt{V}\}$$

where:

- $P$ and $Q$ are the pre- and post-condition of the function $F$,
- $X$ and $U$ denote the sequences of formal input and transient parameters respectively,
- $T$ and $U'$ denote the sequences of actual input terms and transient variables respectively,
- $A$ is a sequence of new variables (one for each transient parameter in $U$)
- $b$ is a new variable.

Thus, the information from the specification of the function is inserted into the lemmas. When a prover starts to prove the lemma it does not have to search in the knowledge base for additional information. For instance, the previous example will look as follows:

```
VCG[Program["Calculus"],Specification["Calculus"]]
```

$$
\begin{array}{l}
Lemma(Calculus): \\
for\ any:\ a, b, y, x \\
\quad (Init)\ IsInteger[a] \wedge IsInteger[b] \Rightarrow \\
\qquad \Big( IsInteger[a] \wedge IsInteger[b] \Big) \\
\wedge \forall_{x1} \Big( (x1 = a \wedge a \geq b) \vee (x1 = b \wedge b > a) \Rightarrow y + x1 \geq y + a \wedge y + x1 \geq y + b \Big)
\end{array}
$$

The verification of programs with procedure (function) calls is conceived hierarchically: the properties of the called objects have to be proven separately.

# 3. Generation of Loop Invariants

We are developing 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).

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. We are separating the specific information from the non-specific one by an analysis of the free variables and other characteristics which are easy to detect automatically.

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 recursion equations:

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

,

These recursive equations are solved by the Gosper-Zeilberger algorithm (see e. g. [8]). Namely, we use the Paule-Schorn [11] implementation in *Mathematica* which is already embedded in the *Theorema* system. We obtain the explicit equations:

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

. 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.

## 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 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 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.

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

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

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

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

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

[9] 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).

[10] K. Nakagawa. Logico-grafic symbols in theorema. In *LMCS'02 (Logic, Mathematics, and Computer Science: Interactions*, 2002. RISC-Linz technical report 02-60.

[11] P. Paule; M. Schorn. *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.

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